Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут телекомунікаційних систем
(повна назва інституту/факультету)

Кафедра телекомунікацій
(повна назва кафедри)

«На правах рукопису»
УДК ______________

**До захисту допущено**
**В.о. завідувача кафедри**

__________ Явіся В.С.___
(підпис) (ініціали, прізвище)

“___”_____________2019_р.

# Магістерська дисертація

**на здобуття освітнього ступеня «магістр»**

Спеціальність 172 Телекомунікації та радіотехніка,
(код і назва)

За освітньо-професійною програмою Інженерія та програмування інфокомунікацій.

на тему: Дослідження можливості використання сучасних хмарних технологій для ресурсоємних наукових розрахунків.

Виконав: студент _2_ курсу, групи ТЗ -81мп
(шифр групи)

Буфін Гліб Германович
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник професор, д.т.н. академік НАНУ Ільченко М. Ю.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант III к.т.н., с.н.с. Міночкін Д. А.
(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент д.т.н., доц. Скуліш М. А.
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________
(підпис)

Київ – 2019 рік

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут телекомунікаційних систем
( повна назва )

Кафедра телекомунікацій
( повна назва )

Спеціальність 172 Телекомунікації та радіотехніка
(код і назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою Інженерія та програмування інфокомунікацій.

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри

__________ Явіся В.С.
(підпис) (ініціали, прізвище)

«___»_____________2019 р.

**ЗАВДАННЯ**
**на магістерську дисертацію студенту**
Буфіну Глібу Германовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації "Дослідження можливості використання сучасних хмарних технологій для ресурсоємних наукових розрахунків."
науковий керівник дисертації Ільченко М. Ю., д.т.н., академік НАНУ
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_07_» «_11_» 2019р. № _3840-с_

2. Строк подання студентом дисертації 05.12.2019 р.

3. Об'єкт дослідження: системи виконання ресурсоємних наукових розрахунків.

4. Предмет дослідження: використання хмарних технологій

5. Перелік завдань, які потрібно розробити: 1) Аналіз хмарних технологій; 2) Дослідження безсерверної архітектури; 3) Розробка безсерверної системи для виконання ресурсоємних розрахунків.

6. Орієнтовний перелік ілюстративного матеріалу: 1) Застосування хмарних технологій; 2) Моделі сервісу хмарних обчислень; 3) Недоліки хмарних обчислень; 4) Безсерверна архітектура; 5) Веб-додаток на основі безсерверної архітектури.

7. Орієнтовний перелік публікацій ________________________________

________________________________________________________________

________________________________________________________________

8. Консультанти розділів дисертації

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| III | Міночкін Д. А. | 09.04.2019 | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

9. Дата видачі завдання: 08.10.2018 р.

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Строк виконання етапів магістерської дисертації | Примітка |
|---|---|---|---|
| 1 | Аналіз літератури за темою дисертації | 14.01.2019 | |
| 2 | Дослідження хмарних технологій | 08.04.2019 | |
| 3 | Дослідження безсерверної архітектури | 22.07.2019 | |
| 4 | Розробка системи для ресурсоємних | 18.11.2019 | |
| | розрахунків | | |
| | | | |
| | | | |
| | | | |

Студент ____________ Буфін Г. Г.
(підпис) (ініціали, прізвище)

Науковий керівник дисертації ____________ Ільченко М. Ю.
(підпис) (ініціали, прізвище)

# Пояснювальна записка
# до магістерської дисертації

на тему: “Дослідження можливості використання сучасних хмарних технологій для ресурсоємних наукових розрахунків”

Київ – 2019 року

# ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ........................................................................... 3
ВСТУП ................................................................................................. 5
1. ГЛОБАЛЬНІ АСПЕКТИ ХМАРНИХ ТЕХНОЛОГІЙ ................................. 8
1.1. Детальний огляд хмарних технологій............................................... 8
1.2. Сервісні моделі ............................................................................ 13
1.2.1. Модель IaaS (Infrastructure as a Service). ............................. 17
1.2.2. Модель SaaS (Software as a Service)..................................... 20
1.2.3. Модель PaaS (Platform as a Service). .................................... 23
1.3. Пояснення недоліків хмарних обчислень......................................... 27
2. SERVERLESS ARCHITECTURE......................................................... 34
2.1. Що таке безсерверна архітектура.................................................... 34
2.2. Програмна модель serverless ......................................................... 41
2.3. Обробка подій ............................................................................. 42
2.3.1. Склад API ........................................................................... 43
2.3.2. Агрегація API для зменшення викликів API....................... 44
2.4. Контроль потоку для відстеження проблем.................................... 45
2.5. Serverless навантаження ............................................................... 46

| З | Л | № докум. | Підп | Д | КПІ ім.Ігоря Сікорського _3840_-с 01.ТЗ- | | | |
|---|---|---|---|---|---|---|---|---|
| Розроб. | | | | | Дослідження можливості використання сучасних хмарних технологій для ресурсоємних наукових розрахунків | Літ. | Арк. | Акрушів |
| Перевір. | | | | | | | | |
| Реценз. | | | | | | | | |
| Н. | | Петрова | | | | | | |
| Затверд. | | Явіся | | | | | | |

3. **БЕЗСЕРВЕРНИЙ РОБОЧИЙ ПРОЦЕС TENSORFLOW** ........ 47
3.1. Машинне навчання ........ 47
3.2. Принципи машинного навчання ........ 48
3.3. Реалізація TensorFlow з автоматичною підготовкою EC2 ........ 52
3.3.1. Налаштування навколишнього середовища. ........ 53
3.3.2. Налаштування Lambda ........ 66
3.3.3. Тестування повного робочого процесу ........ 78
ВИСНОВКИ ........ 80
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ........ 81

## ПЕРЕЛІК СКОРОЧЕНЬ

RJE Remote Job Entry - Віддалений ввід завдання

IBM Корпорація International Business Machines

DEC Digital Equipment Corporation

GE General Electric

UNIX Багатозадачна операційна система комп'ютера

VPN Virtual private network — Віртуальна приватна мережа

IT Інформаційні технології

CRM Customer relationship management - Управління відносинами з клієнтами

NIST National Institute of Standards and Technology - Національний інститут стандартів і технологій

IaaS Infrastructure as a Service — інфраструктура як сервіс

PaaS Platform as a Service - Платформа як сервіс

SaaS Software as a service — Програмне забезпечення як сервіс

AWS Amazon Web Services

XML Extensible Markup Language - Розширювана мова розмітки

SOA service-oriented architecture - Сервісна орієнтована архітектура

HMAC Hash-based message authentication code - механізм перевірки цілісності інформації, що передається або зберігається в ненадійному середовищі

HTML HyperText Markup Language – стандарт намова розмітки веб-сторінок в Інтернеті

IMS IP Multimedia Subsystem -мультимедійна підсистемана основ і протоколу IP

ISP Internet Service Provider – Інтернет-провайдер

LAN LocalAreaNetwork - об'єднання певного числа комп'ютерів на відносно невеликій території

SOAP Simple Object Access Protocol - Простий протокол доступу до об'єктів

REST Representational State Transfer – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів

HTTP Hypertext Transfer Protocol - Протокол передачі гіпертексту

SMTP Simple Mail Transfer Protocol – простий протокол передачі почти

FFTP Fast file transfer protocol - Швидкий протокол передачі файлів

JSON JavaScript Object Notation — об'єктний запис JavaScript

PAN Personal Area Network - мережа, побудована «навкруг» людини

POSIX Portable Operating System Interface for uniX - набірстандартів, які описують інтерфейси між операційною системою та прикладною програмою

QoS Quality of service - якість обслуговування

S3 Simple Storage Service - Проста служба зберігання даних

SNS Simple Notification Service - Проста служба сповіщень

TCP/IP Transmission Control Protocol/ Internet Protocol - набір протоколів мережі Інтернет

UDP User Datagram Protocol – один із протоколів в стеку TCP/IP. Від протоколу TCP він відрізняється тим, що працює без встановлення з'єднання

WSN Wireless sensor network - бездротова сенсорна мережа

ОС Операційна система

**Дослідження можливості використання сучасних хмарних технологій для ресурсоємних наукових розрахунків.**

# ВСТУП

Чергова замітка цілком і повністю буде присвячена досить цікавою і перспективною технологією (так би мовити «золотій жилі» IT-індустрії), що ховається під псевдонімом - cloud technologies.

Буде розгдянута сама концепція хмарних обчислень, наведено найрізноманітніші приклади її втілення (на рівні рішень для звичайних користувачів), а саме, буде розглянут  теоретичний підхід, потім плавний перехід до практики. Таким чином, мета даної роботи - систематизувати основні відомості, пов'язані з даною тематикою та разробити цілістну структуру легкого сприйняття інформації.

За останні роки, ця тема, стала однією з найбільш популярних в IT-сфері, про неї написано чимало статей, проведено ще більшу кількість конференцій, а скільки рішень вже існує на ринку (і на всю використовується нами в повсякденному житті, часом навіть несвідомо), так і взагалі не злічити. Однак, як завжди, є одне "але", а саме, велика частина користувачів, як і раніше не мали можливості дізнатися, що є саме хмарні технології і який непомірно великий вплив та можливості вони надають суспільству нашого часу.

Хмарні обчислення (cloud computing) - це технологія розподіленої обробки даних в якій комп'ютерні ресурси і потужності надаються користувачеві як інтернет-сервіс. Якщо пояснити доступною мовою, то - це Ваш, в деякому сенсі робочий майданчик в інтернеті, а точніше на

віддаленому сервері. Давайте розглянемо приклад, щоб переконається, що практично кожен з нас, так чи інакше, вже стикався з цим рішенням.

У вас є електронна пошта (e-mail)? Звісно є. Так ось, якщо Ви працюєте з поштою на якомусь сайті-сервісі (наприклад, gmail), який цю пошту дозволяє використовувати, то це і є ніщо інше як хмарний сервіс, що є частиною хмарних технологій. Або, наприклад, обробка зображень. Якщо ви зменшуєте розмір, перевертаєте свою фотографію в Photoshop або іншої спеціальної програми, то до хмарної технології Ви не маєте ніякого відношення, - все відбувається і обробляється локально на Вашому комп'ютері. А ось, якщо, завантаживши зображення, наприклад, через сервіс Picasa, Ви його обробляєте по ту сторону, а сваме в браузері, то це і є та саме "хмара".

Актуальність теми даної роботи обумовлена тим, що хмарні технології набули легкої доступності, відносної дешевизни і незначного початкового капіталу – як знань для підтримки і розгортання інфраструктури, так і фінансового характеру. Основу архітектури складають мікросервіси, або функції (lambda), що виконують певне завдання і запускаються на логічних контейнерах, захованих від сторонніх очей. Тобто кінцевому користувачеві дано тільки інтерфейс завантаження коду функції (сервісу) і можливість підключення до цієї функції джерел подій (events).

Дана робота складається з трьох основних частин: в першій частині розглядаються опис та дедальный огляд хмарних технологій, описана їх архітектура та складові елементи. Перелічено основні моделі обчислень, їх відмінності та особливості. Виведено як глобальні переваги так і недоліки хмарних технологій.

Наступна частина даної роботи описує безсерверну архітектуру, а саме - serverless architecture. Галузі в яких найліпше буде реалізовано застосування даної архітектури. Опис сервісу та обслуговування, рівні розвитку та архітектурні стилі SOAP та REST.

Третя частина надає до розгляду приклад застосування технології serverless. Було реалізовано безсерверний робочий процес TensorFlow з автоматичною підготовкою EC2. Завдання для машинного навчання інтегровано у автоматизований робочий процес у режимі реального часу.

# 1. ГЛОБАЛЬНІ АСПЕКТИ ХМАРНИХ ТЕХНОЛОГІЙ

## 1.1. Детальний огляд хмарних технологій

Хмарні обчислення - це наявність за потребою ресурсів комп'ютерної системи, особливо для зберігання даних та обчислювальної потужності, без безпосереднього активного управління користувачем. Термін зазвичай використовується для опису центрів обробки даних, доступних багатьом користувачам через Інтернет. Великі хмари, що переважають сьогодні, часто мають функції, розподілені по декількох місцях від центральних серверів. Якщо з'єднання з користувачем є відносно близьким, він може бути призначений крайовим сервером. Хмари можуть бути обмежені однією організацією (хмари підприємств) або бути доступними багатьом організаціям (публічна хмара). Хмарні обчислення покладаються на обмін ресурсами для досягнення узгодженості та економії масштабу.

Прихильники громадських та гібридних хмар відзначають, що хмарні обчислення дозволяють компаніям уникати або мінімізувати авансові витрати на ІТ-інфраструктуру. Прихильники також стверджують, що хмарні обчислення дозволяють підприємствам швидше запускати свої програми та працювати з покращеною керованістю та меншим обслуговуванням, а також дає можливість ІТ-командам швидше налагоджувати ресурси для задоволення коливаючих та непередбачуваних потреб. Хмарні провайдери, як правило, використовують модель "окупайся", що може призвести до несподіваних операційних витрат, якщо адміністратори не ознайомлені з хмарними ціноутворюючими моделями.

Наявність мереж високої ємності, недорогих комп'ютерів та пристроїв зберігання даних, а також широке впровадження апаратної віртуалізації, орієнтованої на сервісну архітектуру автономних та корисних обчислень

призвело до зростання хмарних обчислень. До 2019 року Linux була найбільш широко використовуваною операційною системою, включаючи пропозиції Microsoft, і тому вона описується як домінуюча. Хмарний постачальник послуг (CSP) буде відображати, зберігати та збирати дані про брандмауери, ідентифікацію вторгнень та / чи рам протидії дії та потоку інформації всередині мережі.

Перемикаючи трафік, оскільки компанії вважали за потрібне збалансувати використання серверів, вони могли б ефективніше використовувати загальну смугу пропускання мережі. У 90-х роках телекомунікаційні компанії, які раніше пропонували в основному спеціалізовані схеми передачі даних за схемою “точка-точка”, почали пропонувати послуги віртуальної приватної мережі (VPN) зі стабільною якістю послуг, але за меншу вартість. Вони почали використовувати символ хмар, щоб позначити точку демаркації між тим, за що відповідав постачальник, і за що користувачі несуть відповідальність. По мірі того, як комп'ютери стали більш розповсюдженими, вчені та інженери вивчали способи отримання великомасштабної обчислювальної потужності для більшої кількості користувачів за допомогою спільного використання часу. Хмарні обчислення розширили цю межу для охоплення всіх серверів, а також мережеву інфраструктуру. Вони експериментували з алгоритмами для оптимізації інфраструктури, платформ та програм, щоб визначити пріоритет процесорів та підвищити ефективність для кінцевих користувачів.

Хмарні технології (обчислення) - це метод надання послуг інформаційних технологій (IT), в яких ресурси завантажуються з Інтернету через веб-інструменти та програми, а не через безпосереднє підключення до сервера.

Хмарні обчислення були популяризовані Amazon.com, випустивши свій продукт Elastic Compute Cloud у 2006 році. Посилання на фразу "хмарні обчислення" з'явилися ще в 1996 році, з першою відомою згадкою у внутрішньому документі Compaq. Хмарний символ використовувався для представлення мереж обчислювальної техніки в оригінальній ARPANET ще в

1977 р. та CSNET до 1981 р. - як попередники самого Інтернету. Слово хмара використовувалося як метафора для Інтернету, а для позначення мережі на телефонній схемі використовується стандартизована хмарна форма. Під цим спрощенням розуміється, що специфіка з'єднання кінцевих точок мережі не є актуальною для розуміння діаграми.

Поки електронний пристрій має доступ до Інтернету, він має доступ до даних та  програм для його запуску. Замість того, щоб зберігати файли на власному жорсткому диску чи на локальному пристрої зберігання даних, хмарне сховище дає змогу зберігати їх у віддаленій базі даних. Термін хмара використовувався для позначення платформ для розподілених обчислень ще в 1993 році, коли Apple відокремила General Magic і AT&T використовувала його для опису своїх (парних) технологій Telescript і PersonaLink. У квітковій програмі Wired у квітні 1994 року "Чудова пригода Білла та Енді Енді" Енді Герцфельд прокоментував розповсюджену мову програмування General Magic Telescript:

"Краса Telescript ... полягає в тому, що зараз, замість того, щоб просто програмувати пристрій, у нас зараз є вся Хмара, де одна програма може перейти до багатьох різних джерел інформації та створити якусь віртуальну. Ніхто раніше цього не здогадувався. Приклад Джим Уайт (дизайнер Telescript, X.400 та ASN.1) використовує зараз - це служба встановлення дат, коли програмний агент ходить до квіткового магазину і замовляє квіти, а потім йде до магазину квитків і отримує квитки на виставу, і все повідомляється обом сторонам ".

Рис. 1.1. Застосування хмарних технологій

Компанії, що надають хмарне обслуговування, дозволяють користувачам зберігати файли та програми на віддалених серверах, а потім отримувати доступ до всіх даних через Інтернет. Це називається хмарним обчисленням, оскільки інформація, до якої звертаються, знаходиться в «хмарі», і не вимагає, щоб користувач знаходився у певному місці, щоб отримати доступ до нього. Цей тип системи дозволяє співробітникам працювати дистанційно.

Мета хмарних обчислень - дозволити користувачам скористатись усіма цими технологіями, не потребуючи глибоких знань про можливості та досвід роботи з кожною з них. Хмара має на меті скоротити витрати і допомагає користувачам зосередитися на основному бізнесі, а не перешкоджати ІТ-перешкодам. За допомогою хмарних обчислень ви вилучаєте ті головні болі, які пов'язані зі збереженням власних даних, оскільки ви не керуєте апаратним та програмним забезпеченням - це відповідальність досвідченого

постачальника, як Salesforce. Основна технологія хмарних обчислень - віртуалізація. Спільна інфраструктура означає, що вона працює як утиліта: ви платите лише за те, що вам потрібно, автоматичні оновлення, і простота масштабування вгору або вниз. Програмне забезпечення для віртуалізації відокремлює фізичний обчислювальний пристрій на один або декілька "віртуальних" пристроїв, кожен з яких може бути легко використаний та керований для виконання обчислювальних завдань. Додатки на базі Cloud Functions можуть працювати протягом декількох днів чи тижнів, і вони коштують дешево. Завдяки віртуалізації на рівні операційної системи, що створює масштабовану систему з декількох незалежних обчислювальних пристроїв, простої обчислювальні ресурси можуть бути розподілені та використані більш ефективно. Віртуалізація забезпечує спритність, необхідну для прискорення IT-операцій, і зменшує витрати за рахунок збільшення використання інфраструктури. За допомогою додатка для хмар ви просто відкриваєте веб-переглядач, входите в систему, налаштовуєте додаток і починаєте його використовувати. Автономні обчислення автоматизують процес, за допомогою якого користувач може надавати ресурси на вимогу. За рахунок мінімізації участі користувачів автоматизація прискорює процес, зменшує затрати праці та зменшує можливість помилок людини.

Деякі найбільші в світі компанії перенесли свої додатки в хмару з Salesforce після ретельного тестування на безпеку та надійність інфраструктури. Підприємства працюють із різними додатками в хмарі, як-от управління відносинами з клієнтами (CRM), HR, бухгалтерія та багато іншого. Завжди треба поглиблюватися, оцінюючи пропозиції хмари, і мати на увазі, якщо вам потрібно купувати та управляти обладнанням та програмним забезпеченням, те, що ви шукаєте, насправді може бути не хмарним обчисленням, а фальшивою хмарою. Оскільки хмарне обчислення зростає з високою популярністю, тисячі компаній просто ребрендингували свої продукти і послуг як «хмарні обчислення».

### 1.2. Сервісні моделі

Хоча архітектура, орієнтована на сервіс, виступає за "Все як послуга" (з абревіатурами EaaS або XaaS, або просто ASA), постачальники хмарних обчислень пропонують свої "послуги" за різними моделями, з яких три стандартні моделі в NIST це інфраструктура як послуга (IaaS), платформа як послуга (PaaS) та програмне забезпечення як послуга (SaaS). Мабуть, вся концепція з'явилася поза необхідністю мати доступ до програмного забезпечення на ходу. Ці моделі пропонують зростаючу абстракцію; Таким чином, вони часто зображуються як шари в стеку: інфраструктура, платформа та програмне забезпечення як послуга, але вони не повинні бути пов'язані між собою. Наприклад, можна надати SaaS, реалізований на фізичних машинах (голий метал), не використовуючи підстилаючі шари PaaS або IaaS, і навпаки, можна запускати програму на IaaS та отримувати доступ до неї безпосередньо, не загортаючи її як SaaS. Необхідність мобільної доступності програмного забезпечення зробила революцію у світі можливого обчислення. Незліченна кількість людей відчула комфорт та легкість використання сервісу на платформі хмар. Хмарні обчислення - це модель для надання повсюдного, зручного доступу на вимогу мережі до спільного пулу налаштованих обчислювальних ресурсів (наприклад, до мереж, серверів, сховищ даних, додатків та служб), які можна швидко забезпечити та випустити з мінімальними зусиллями управління або взаємодія постачальника послуг.Детальніше про деякі основні переваги хмарних обчислень для вашого бізнесу.

Згідно з останнім опитуванням, відсоток компаній, які використовують публічну хмару, очікується, що зросте до 51% у 2016 році. За оцінками, сервери, що доставляють до публічної хмари, зростуть на 60% CAGR (Річна швидкість зростання сполук), тоді як Витрати на сервер сайту зменшаться на 8,6% протягом наступних двох років.

Більшість з нас усвідомлюють переваги, надані хмарними службами, - гнучкість використання та економічність, оскільки не потрібно встановлювати програмне забезпечення, інвестувати в сервіси, сервери та

платформи. Незважаючи на те, що хмарні обчислення еволюціонували з плином часу, вони були в основному розділені на три широкі категорії послуг: інфраструктура як послуга (IAAS), платформа як послуга (PAAS) та програмне забезпечення як послуга (SAAS), які широко обговорюються нижче.

Майбутнє обчислень - у хмарі. Це означає, що ви адаптуєте свій бізнес, щоб він відповідав хмарній моделі. Хмарне обчислення є моделлю для забезпечення повсюдного, зручного та легкого доступу до спільного пулу обчислювальних ресурсів (наприклад, мереж, серверів, пам'яті, програм та служб) які можуть бути швидко забезпечені та випущені з мінімальними зусиллями керування або взаємодії з постачальниками послуг. Протилежне також справедливо, оскільки ваш бізнес може залишитися позаду, якщо ця нова технологія буде недостатньо використана.

Однією з найбільших переваг хмари є автоматизація. Після підписки на моделі хмарних служб Saas, IaaS, Paas, ви можете використовувати її більш широкі можливості, щоб забезпечити гнучкість та ефективність, що підштовхує зростання вашого бізнесу. Коли ваша організація правильно використовує хмари, ви видаляєте рівні керування. Ви не будете нести відповідальність за фізичні центри обробки даних, операційні системи та обладнання.

Видаляючи ці рівні, ви можете скоротити витрати та мінімізувати час, витрачений на загальноприйняті завдання IT. Замість того, щоб підтримувати апаратне забезпечення, ви можете зосередитися на інноваціях та масштабувати свої програми на вимогу. Більше підприємств звертаються до хмари, щоб винаходити свої програми та скористатися новими стратегіями розширення.

Протягом багатьох років хмарні служби спостерігали експоненційний ріст у всьому світі. Хмарні сервіси дозволяють організаціям керувати та використовувати обчислювальні послуги в безпечному та контрольованому режимі, оскільки хмарна інфраструктура використовує протипожежну стіну. Прогноз Gartner щодо глобальних доходів від публічних послуг у сфері

хмарних служб оцінює, що лише 2018 рік отримає 305,8 мільярда доларів, при прогнозі зростання в 2020 році 411,4 мільярда доларів. Деякі постачальники послуг спеціально спеціалізуються на обслуговуванні виключно підприємств. Через мережу корпоративна хмара забезпечує необхідну платформу, програмне забезпечення або необхідні послуги інфраструктуриПровідні організації у сфері охорони здоров'я, державні установи, які документують та керують конфіденційними даними, воліють вибирати хмарні служби підприємства, а не загальнодоступну хмару. Ці звіти означають постійне прийняття хмарних служб бізнесу по всьому світу для вирішення всього спектру операцій, які вони роблять. Тим часом численні провідні гравці в галузі інформаційних технологій зараз змагаються за надання гнучких хмарних послуг як для населення, так і для підприємств.

Зростаюча конкуренція означала кращу доставку послуг та нововведень, що може глибоко принести користь вашому масштабуванню бізнесу. Отже, зараз настав час, коли ви розмістите хмарну модель у своїй бізнес-інфраструктурі. Хмарні обчислення підтримують різні потреби користувачів комп'ютера, його архітектура розроблена з різними пристосованими функціями.

Моделі сервісу хмарних обчислень розташовані на рівнях у стеку. Ці моделі пропонують збільшення абстракції; таким чином вони часто зображуються як рівні у стеку: інфраструктура, платформа та програмне забезпечення як служби, але вони не повинні бути пов'язаними. Хоча сервіс-орієнтована архітектура виступає за політику «все як сервіс», постачальники хмарних обчислень пропонують свої «послуги» за різними моделями, з яких три стандартні моделі для NIST (National Institute of Standards and Technology) - це інфраструктура як служба (IaaS), платформа як служба (PaaS ) та програмне забезпечення як сервіс (SaaS).

Рис. 1.2. Моделі сервісу хмарних обчислень та їх призначення

Яка модель обслуговування хмарних обчислень підходить для вашого бізнесу? Наприклад, можна встановити SaaS, реалізований на фізичних машинах (без металу), не використовуючи основні шари PaaS або IaaS, і навпаки, можна запустити програму на IaaS і отримати доступ до неї безпосередньо, не обернувши її як SaaS. Підключення та доступність мають вирішальне значення в сучасному світі бізнесу, а хмарні обчислення дозволяють вашому бізнесу працювати, де завгодно, коли завгодно. Однак хмарні обчислення не так просто, як може здатися. Всередині світу хмарних обчислень є три основні моделі обслуговування. Порівнюючи три різні моделі, ви зможете визначити, яка модель обслуговування хмарних обчислень підходить для вашого бізнесу. Три типи хмарних моделей - це інфраструктура як послуга, платформа як послуга та програмне забезпечення як послуга. Їх основні функції можна узагальнити у фразах "Хост", "Побудувати" та "Спожити". Кожен пропонує різний рівень гнучкості та контролю над тим продуктом, який «купує» ваш бізнес. Кожен з них також залежить від вашої існуючої IT-інфраструктури. Через великі розбіжності

між трьома, важливо визначити, яка модель буде відповідати потребам вашого бізнесу найкраще.

### 1.2.1. Модель IaaS (Infrastructure as a Service)

IaaS, як найбільш гнучка з хмарних моделей, дозволяє вашому бізнесу мати повний масштабований контроль над управлінням та налаштуванням вашої інфраструктури.

У моделі IaaS хмарний провайдер розміщує ваші інфраструктурні компоненти, які традиційно будуть наявні в локальному центрі обробки даних (наприклад, сервери, обладнання для зберігання даних та мережеве обладнання). Однак ваш бізнес підтримуватиме контроль над операційними системами, сховищем, розгорнутими програмами та, можливо, обмеженим контролем над окремими мережевими компонентами (наприклад, брандмауерами хостів). IaaS допомагає важливим інфраструктурним функціям, таким як зберігання, встановлення ефективних брандмауерів та мереж. Інфраструктура як сервіс (IaaS) - це модель самообслуговування, створена спеціально для надання послуг Cloud Services, яка допомагає отримувати, контролювати та керувати інфраструктурою віддаленого центру обробки даних.

Користувачі IaaS спеціально отримують перевагу, маючи можливість встановлювати будь-яку платформу по інфраструктурі. Крім того, кілька постачальників IaaS пропонують послуги, пов'язані з наданням черг повідомлень та баз даних. Одним з важливих факторів є той факт, що за допомогою IaaS користувачі керують додатками та програмним забезпеченням. Постачальники послуг обробляють складні деталі, що обертаються навколо серверів, зберігання, мереж та віртуалізації.

Інфраструктура як сервіс включає в себе метод надання послуг від операційних систем до серверів та зберігання через підключення через IP як частину послуги за запитом. IaaS - це модель обслуговування, яка забезпечує комп'ютерну інфраструктуру на підході до зовнішніх джерел для підтримки

операцій підприємства. Однак з IaaS користувачі повинні дбати про оновлення програмного забезпечення, якщо випускаються нові оновлені версії програмного забезпечення. Клієнти можуть уникнути необхідності придбати програмне забезпечення або сервери, а замість цього закуповувати ці ресурси на замовленій службі за запитом. Як правило, IaaS забезпечує простір апаратного забезпечення, пам'яті, серверів та центру обробки даних або мережеві компоненти; це може також включати програмне забезпечення.

Служба, як правило, функціонує за допомогою моделі «оплати за вами», що дозволяє можливість росту та гнучкості. Інфраструктура як служба (IaaS) також відома як апаратне забезпечення як служба (HaaS). Вона дозволяє уникнути довгострокових інвестицій у обладнання та забезпечити незалежність від місця розташування. Її використовують команди розробників та системні адміністратори як альтернативу внутрішнім серверам перед масштабуванням. Замість цього апаратне забезпечення розміщується в декількох центрах обробки даних і керується постачальником.

Переваги IaaS:

Усуває капітальні витрати. Використання хмарної інфраструктури виключає капітальні витрати на використання власного обладнання та програмного забезпечення. Крім того, IaaS, як правило, пропонується як платна модель, з оплатою, заснованою або в часі, або в кількості використовуваного простору віртуальної машини.

Підтримує гнучкість. IaaS корисний для підтримки робочих навантажень, які є тимчасовими, можуть змінюватися несподівано або експериментально. Як і всі робочі навантаження, ці навантаження потребують інфраструктури для їх підтримки, однак дорого зобов’язати додаткову постійну внутрішньобудинкову інфраструктуру для тимчасової потреби. Хмарна інфраструктура відповідає потребі у гнучкості.

Просте розгортання. Вашому постачальнику хмари набагато простіше розгортати ваші сервери, обробку, зберігання та мережу в моделі IaaS, ніж

вам для розгортання цих елементів вдома, без попереднього відсутня база для створення. Як результат, ваш час роботи збільшиться, оскільки ваші системи будуть доступні для швидшого використання.

Недоліки IaaS:

В полі зору. Оскільки вся ваша інфраструктура підтримується та контролюється постачальником послуг IaaS, рідко вам будуть надані деталі її конфігурації та продуктивності. У свою чергу це може ускладнити управління та моніторинг систем для вашої компанії.

Змінність стійкості. Наявність та ефективність робочого навантаження сильно залежить від постачальника. Якщо постачальники послуг IaaS відчувають внутрішній або зовнішній простої, це також вплине на ваші навантаження.

Дорого. Моделі IaaS, як правило, значно дорожчі, ніж моделі PaaS та SaaS, оскільки вони пропонують набагато більше підтримки для вашого бізнесу, ніж інші дві хмарні моделі. Однак вони все ще можуть бути рентабельними, виходячи зі своєї корисності для вашого бізнесу.

Клієнти зазвичай платять за користування обчислювальною або комунальною базою. Провайдер IaaS надає послуги, що базуються на політиці, і відповідає за житло, експлуатацію та обслуговування обладнання, яке він надає клієнту.

Характеристики IaaS включають:

- Автоматизовані адміністративні завдання
- Динамічне масштабування
- Віртуалізація платформи
- Інтернет-з'єднання

Треті сторони розміщують елементи інфраструктури, такі як апаратне забезпечення, програмне забезпечення, сервери та зберігання, а також забезпечує резервне копіювання, безпеку та технічне обслуговування.

### 1.2.2. Модель SaaS (Software as a Service)

Програмне забезпечення як послуга (SAAS) - це модель розповсюдження програмного забезпечення, в якій програми розміщуються постачальником або постачальником послуг та надаються клієнтам через мережу, як правило, через Інтернет. Служба додатків у хмарі або Програмне забезпечення як сервіс (SaaS) є найбільшим «хмарним ринком» і зростає швидкими темпами, щоб утримати попит. SAAS стає все більш поширеною моделлю доставки, оскільки основні технології, що підтримують веб-сервіси та архітектуру, орієнтовану на сервіс (SOA), стають популярними та нові підходи до розвитку, такі як Ajax. Кілька додатків, розміщених у вигляді SaaS, досить популярні у газузях охорони здоров'я, електронної пошти та співпраці тощо. SAAS тісно пов'язаний з ASP (постачальником послуг) та моделями доставки обчислювальних програм на вимогу. За допомогою SaaS користувачу не потрібно турбуватися про встановлення або завантаження, хоча для деяких додатків може знадобитися плагін. IDC виділяє дві дещо різні моделі доставки для SAAS, а саме модель розміщеної програми та модель розробки програмного забезпечення.

Той факт, що більшість додатків можна отримати за допомогою веб-браузера, робить SaaS набагато популярнішим. SaaS дозволяє кінцевому користувачеві користуватися доступом до сервісів через Інтернет і взаємодіяти з клієнтами через інтерфейс. Компанії легко спрощують їх технічне обслуговування та підтримку за допомогою SaaS, оскільки постачальники можуть керувати всією інформацією, підтримувати та забезпечувати плавність роботи програми.

Деякі великі компанії, які традиційно не визнаються постачальниками програмного забезпечення, почали будувати SaaS як додаткове джерело доходу для отримання конкурентної переваги. Щоб йти в ногу з ринком і отримати масове зчеплення, зараз підприємства охоплюють SaaS для отримання додаткових доходів, навіть якщо підприємства не працюють переважно як постачальники програмного забезпечення.

У SaaS постачальник послуг розміщує програму в своєму центрі обробки даних, а користувач отримує доступ до нього за допомогою стандартного веб-переглядача.

Модель SaaS дозволяє вашому бізнесу швидко отримувати доступ до хмарних веб-додатків, не зобов'язуючись встановлювати нову інфраструктуру. Програмне забезпечення як сервіс (SaaS) - це модель для розподілу програмного забезпечення, де клієнти отримують доступ до програмного забезпечення через Інтернет. Програми працюють на хмарі постачальника, яку вони, звичайно, контролюють і підтримують. Програмне забезпечення як сервіс замінює традиційне програмне забезпечення на пристрої за допомогою програмного забезпечення, яке ліцензовано на підписці. Програми доступні для використання з платною ліцензованою підпискою або безкоштовно з обмеженим доступом. SaaS не вимагає будь-яких установок або завантажень у існуючій інфраструктурі, що, в свою чергу, виключає необхідність встановлення, обслуговування та оновлення програм на кожному з ваших комп'ютерів.

За допомогою хмари програмне забезпечення, таке як Інтернет-браузер або програма, може стати корисним інструментом. Він розміщений у хмарі централізовано. Яскравим прикладом є Salesforce.com. Більшість застосувань SaaS можна отримати безпосередньо з веб-браузера без необхідності завантажувати або встановлювати. Однак для деяких додатків SaaS потрібні плагіни.

Переваги SaaS:

Доступна. Для цієї моделі не потрібно апаратне обладнання, що забезпечує низькі пов'язані з цим витрати. Малий бізнес може вважати цю хмарну платформу особливо привабливою.

Доступний скрізь. Хмарні програми доступні всюди, де є доступ до Інтернету. Таким чином, компанії, які потребують частої співпраці, вважають

платформи SaaS корисними, оскільки їх працівники можуть легко отримати доступ до необхідних їм програм.

Готовий до використання. З SaaS потрібні програми вже повністю розроблені та готові до використання. Час налаштування програм SaaS значно скорочується в порівнянні з іншими двома типами хмарних платформ.

Недоліки SaaS:

Відсутність контролю. З SaaS постачальник має контроль над програмами, якими користується ваша компанія. Якщо вам не комфортно передавати контроль над своїми критичними бізнес-додатками іншій стороні, можливо, SaaS - це не найкращий варіант для вашого бізнесу.

Повільніші швидкості. Спираючись на доступ до Інтернету до функції, програми SaaS, як правило, повільніше, ніж програми клієнт / сервер. Однак ці програми, як правило, швидкі, хоча і не миттєві.

Змінні функції та особливості. У багатьох випадках хмарні програми SaaS мають меншу функціональність та особливості, ніж їхні клієнтські / серверні аналоги. Однак цей недолік може бути недійсним, якщо для вашого бізнесу потрібні лише функції, запропоновані у версії SaaS.

Є кілька основних характеристик, які застосовуються для більшості постачальників SaaS:

- Оновлення застосовуються автоматично без втручання клієнта
- Послуга придбана на підписці
- Жодне обладнання не вимагає встановлення замовником

Стара модель отримання фізичних DVD-дисків та встановлення на локальних серверах була єдиним реалістичним рішенням на багато років. SaaS також відомий як розміщене програмне забезпечення або програмне забезпечення під замовлення. Справді, для багатьох сценаріїв модель клієнт-сервер потрібна. SaaS - це природна еволюція програмного забезпечення. Одним з факторів є пропускна здатність; Інтернет став швидше, ніж десять років тому. Тим не менш, останніми роками ряд подій дозволив SaaS стати

основним. Інші основні фактори включають еволюцію як віртуалізації, так і інструментів у великих даних. Software as a Service використовується у низці спільних бізнес-напрямків, включаючи управління взаємовідносинами з клієнтами (CRM), управління документами, бухгалтерський облік, управління людськими ресурсами (HR), управління сервісними службами, управління контентом та співпраця. Всі ці досягнення зробили постачальникам набагато простіше масштабувати та керувати своєю власною інфраструктурою, а отже, надавати рішення SaaS. SaaS тісно пов'язана з платформою як служба (PaaS) та інфраструктурою як служба (IaaS). Він підпадає під дію більшої категорії хмарних обчислень, хоча багато людей розглядають терміни як синоніми.

Є буквально тисячі постачальників SaaS, але Salesforce.com, мабуть, є найвідомішим прикладом, оскільки він є одним з перших постачальників, який значно руйнує традиційну версію програмного забезпечення.

### 1.2.3. Модель PaaS (Platform as a Service)

Платформа як послуга (PAAS) - це модель хмарних обчислень, яка доставляє програми через Інтернет. PaaS - це фреймворк, який найбільш бажаний розробниками, оскільки вони можуть налаштовувати та розробляти програми. У моделі PAAS хмарний постачальник послуг постачає користувачеві та програмному тарифу, як правило, необхідні для розробки додатків, своїм користувачам як послугу. Платформа як служба (PaaS) або служба Cloud Platform є досить популярною, і вона використовується для розробки додатків та надання хмарних компонентів для програмного забезпечення. Постачальник ПААС розміщує обладнання та програмне забезпечення на власній інфраструктурі. Вибравши роботу з обласною платформою, багато питань логістики ставляться до уваги - розробка, тестування додатків та розгортання є досить швидкими, простими та рентабельними. Як результат, PAAS звільняє користувачів від необхідності

встановлювати внутрішнє обладнання та програмне забезпечення для розробки або запуску нового додатка.

Підприємство PaaS забезпечує ефективне маневрування та допомагає управляти обчислювальною інфраструктурою з централізованого IT-центру за допомогою платформ, що присутні на апаратному забезпеченні. PAAS не замінює всю інфраструктуру бізнесу, але натомість бізнес покладається на постачальників PAAS для ключових послуг, таких як розробка Java або розміщення програм. Програми, що використовують PaaS, успадковують характеристики, унікальні для хмарної інфраструктури. Наприклад, у програмах будуть відображатися відомі хмарні функції, такі як багаторазовий прокат, розширення можливостей PaaS, масштабованість тощо. Однак постачальник PAAS підтримує всі основні обчислювальні програми та програмне забезпечення; Користувачам потрібно лише увійти та почати користуватися платформою, як правило, через інтерфейс веб-браузера. Потім провайдери ПААС платять за цей доступ щомісяця. Багато переваг, які пропонує PaaS для підприємств, - це забезпечує величезний перепочинок з широкого кодування, дозволяє автоматизувати бізнес-політику, а також забезпечує легкий перехід додатків на гібридні моделі Користувачі можуть ефективно створювати індивідуальні програми за допомогою програмних компонентів, які вбудовані в PaaS.

Відгалуження хмарних обчислень, що дозволяє користувачам розробляти, запускати та керувати програмами без необхідності потрапляти в код. Існує кілька типів PaaS. PaaS можна класифікувати в залежності від того, відкрите чи закрите джерело, незалежно від того, який з них сумісний із мобільним пристроєм (mPaaS), і які типи бізнесу він обслуговує. Кожна опція PaaS - це загальнодоступна, приватна або гібридна комбінація з двох. Приватна PaaS, з іншого боку, розміщується на локальних серверах або приватних мережах і підтримується користувачем. Громадський PaaS розміщений у хмарі, а його інфраструктура керується постачальником. Гібридний PaaS використовує елементи як публічних, так і приватних, і здатний виконувати програми з різних хмарних інфраструктур.

За допомогою цієї моделі сторонній постачальник надає вашому бізнесу платформу, на якій ваш бізнес може розвивати та запускати програми. Вибираючи рішення PaaS, найважливіші міркування, крім того, як він розміщується, полягають в тому, наскільки добре він інтегрується з існуючими інформаційними системами, якими мовами програмування він підтримується, які інструменти для створення додатків він пропонує, і наскільки ефективно це підтримується провайдером.

Оскільки постачальник розміщує хмарну інфраструктуру, яка підтримує платформу, PaaS позбавляє вас від необхідності встановлювати внутрішнє обладнання або програмне забезпечення. Підприємства використовують нові можливості PaaS для подальшого передавання сторонніх завдань, які могли б інакше покладатися на місцеві рішення. Все це стало можливим завдяки прогресу в області хмарних обчислень. Ваш бізнес не керував би і не контролював базову хмарну інфраструктуру, але ви б підтримували контроль над розгорнутими програмами (на відміну від SaaS).

Переваги PaaS:

Швидкий час на ринок. PaaS спрощує управління додатками, усуваючи необхідність підтримувати та контролювати базову інфраструктуру. В результаті програми можна розробити та розгортати швидше.

Економічний розвиток. Хмарна платформа надає вашому бізнесу базу, на якій можна будувати свої додатки, а не будувати з нічого, тим самим різко знижуючи пов'язані з цим витрати.

Масштабованість. Хмарні платформи пропонують код багаторазового використання, що, звичайно, полегшує розробку та розгортання додатків, а також пропонує підвищену масштабованість.

Недоліки PaaS:

Блокування для постачальників. Важко перенести багато послуг, що надаються одним продуктом PaaS, на конкуруючий продукт, що ускладнює переключення постачальників PaaS. Перед тим, як перейти від одного постачальника PaaS до іншого, швидше за все, виникають простої та додаткові витрати.

Безпека та відповідність. У моделі PaaS постачальник зберігатиме більшість або навіть усі дані програми. Таким чином, необхідно обов'язково оцінити заходи безпеки постачальника. Це, однак, часто виявляється важким, оскільки постачальник може зберігати свої бази даних через третю сторону, тим самим залишаючи вас неінформованими про безпеку ваших даних.

Відсутність сумісності. Можливо, що ваша поточна інфраструктура може бути не сумісна з хмарною платформою. Якщо деякі елементи не можуть бути включені в хмару, можливо, вам доведеться перейти від своїх поточних додатків і програм до сумісних з хмарою аналогів, щоб повністю інтегруватися. Крім того, вам може знадобитися залишити ці елементи поза хмарою та в межах вашої поточної інфраструктури.

Традиційні бізнес-додатки завжди були дуже складними та дорогими. Оскільки цифрові технології стають все більш потужними та доступними, додатки та обласні платформи стають майже загально поширеними.  Сума та різноманітність обладнання та програмного забезпечення, необхідного для їх використання, є складними. Потрібна ціла команда експертів для встановлення, налаштування, тестування, запуску, захисту та оновлення.

### 1.3. Пояснення недоліків хмарних обчислень

Якщо ви хочете доставляти цифрові послуги будь-якого типу, вам потрібно оцінити всі типи ресурсів, не останньою з яких є процесор, пам'ять, сховище та мережеве з'єднання. Які ресурси ви обираєте для доставки - хмарна чи локальна - залежить від вас. Хмарні технології тепер дають змогу легко отримати доступ до програм та додатків з Інтернету. Вам потрібно зрозуміти плюси та мінуси хмарних обчислень та як контекстуалізувати будь-які наявні недоліки.

Раніше програмне забезпечення та додатки мали бути фізично встановленими, але не більше. З розвитком технологій та можливістю доступу до корисних додатків через Інтернет підприємства отримують величезні переваги від хмарних обчислень. Однак, поряд із перевагами хмарного обчислення, недоліки є досить очевидними.

Після тривалих досліджень було виділено наступні недоліки хмарних технологій:

1) Час простою

Час простою часто називають одним з найбільших недоліків хмарних обчислень. Оскільки системи хмарних обчислень базуються на Інтернеті, відключення послуг - це завжди невдала можливість і може статися з будь-якої причини.

Чи може ваш бізнес дозволити собі наслідки відключення або уповільнення? Відключення веб-сервісів Amazon в 2017 році коштувало компаніям, що торгуються на публічному ринку, до 150 мільйонів доларів. На жаль, жодна організація не застрахована, особливо коли критичні бізнес-процеси не можуть дозволити собі переривати. У червні та липні 2019 року цілий ряд компаній та послуг потрапив у відключення, включаючи Cloudflare (провідний постачальник веб-послуг), Google, Amazon, Shopify, Reddit, Verizon та Spectrum.

Кращі практики для мінімізації запланованих простоїв у хмарному середовищі:

- Дизайнерські послуги з високою доступністю та відновлення після аварій. Використовуйте зони інфраструктури, що надаються постачальниками хмар, у вашій інфраструктурі.
- Якщо ваші служби мають низьку толерантність до відмов, розгляньте багаторегіональні розгортання з автоматичною відмовою, щоб забезпечити найкращу безперервність бізнесу.
- Визначте та застосуйте план відновлення після катастроф відповідно до цілей вашого бізнесу, які забезпечують найменший час відновлення (RTO) та цілі точки відновлення (RPO).
- Подумайте про те, як застосувати спеціальні підключення, такі як AWS Direct Connect, Azure ExpressRoute або виділений взаємозв'язок Google або партнерський взаємозв’язок Google Cloud. Ці сервіси забезпечують виділений мережевий зв’язок між вами та точкою присутності хмарних служб. Це може зменшити вплив ризику переривання бізнесу через загальнодоступний Інтернет.
- Прочитайте тонкий шрифт угоди про рівень обслуговування (SLA). Ви гарантуєте 99,9% часу роботи або навіть краще? Цей 0,1% простою становить приблизно 45 хвилин на місяць або близько восьми годин на рік.

2) Безпека та конфіденційність

Хоча постачальники хмарних послуг застосовують найкращі стандарти безпеки та галузеві сертифікати, зберігання даних та важливих файлів для зовнішніх постачальників послуг завжди відкриває ризики. Будь-яка дискусія, що включає дані, повинна стосуватися безпеки та конфіденційності, особливо якщо мова йде про управління конфіденційними даними. Ми не повинні забувати, що сталося в Code Space та злому їх консолі AWS EC2, що призвело до видалення даних та можливої зупинки компанії. Їх залежність від віддаленої хмарної інфраструктури означала взяти на себе ризик аутсорсингу всього.

Звичайно, очікується, що будь-який постачальник хмарних послуг буде керувати та захищати базову апаратну інфраструктуру розгортання. Однак ваші обов'язки лежать у царині управління доступом користувачів, і ви ретельно зважуйте всі сценарії ризику.

Незважаючи на те, що нещодавні порушення даних щодо кредитних карток та облікових даних для входу користувачів все ще свіжі у свідомості громадськості, вжито заходів для забезпечення безпеки даних. Одним із таких прикладів є Загальне правило захисту даних (GDPR), яке нещодавно було прийнято в Європейському Союзі, щоб забезпечити користувачам більше контролю над своїми даними. Тим не менш, вам все одно потрібно усвідомлювати свої обов'язки та дотримуватися кращих практик.

Найкращі практики мінімізації ризиків для безпеки та конфіденційності:

- Це важливо: зрозумійте модель спільної відповідальності вашого постачальника послуг у хмарі. Ви все ще несете відповідальність за те, що відбувається у вашій мережі та у вашому продукті.
- Забезпечуйте безпеку на кожному рівні вашого розгортання.
- Знайте, хто повинен мати доступ до кожного ресурсу та послуги, і обмежте доступ до найменших привілеїв. Якщо працівник переходить на шахрайство і отримує доступ до вашого розгортання, ви хочете, щоб їх вплив був на найменшій площі.
- Переконайтеся, що навички вашої команди досконалі. Фахівці з кібербезпеки повинні знати як пом'якшити проблеми безпеки та конфіденційності в хмарі.
- Використовуйте підхід, орієнтований на ризик, щодо забезпечення активів, що використовуються у хмарі, та розширення безпеки пристроїв.
- Реалізуйте багатофакторну автентифікацію для всіх облікових записів, що мають доступ до конфіденційних даних або систем.

- Шифрування, шифрування, шифрування. Увімкніть шифрування, де можна - легкі виграші на сховищах об'єктів, таких як Amazon S3 або Azure Blob Storage, де дані клієнтів часто перебувають. Простий акт увімкнення шифрування на S3 міг уникнути порушення даних Capital One у липні 2019 року, що викрило інформацію про 100 мільйонів користувачів.

3) Вразливість до нападу

У хмарних обчисленнях кожен компонент є в Інтернеті, що розкриває потенційні вразливості. Навіть найкращі команди час від часу зазнають серйозних атак та порушень безпеки. Оскільки хмарні обчислення побудовані як державна служба, її легко запустити, перш ніж навчитися ходити. Зрештою, ніхто у продавця хмари не перевіряє ваші навички адміністрування, перш ніж надавати вам рахунок: все, що потрібно, щоб почати роботу - це, як правило, дійсна кредитна картка.

Кращі практики, які допоможуть вам зменшити хмарні атаки:

- Зробіть безпеку основним аспектом усіх IT-операцій.
- Слідкуйте за тим, щоб ВСІ ваші команди були в курсі кращих практик хмарної безпеки.
- Забезпечити регулярну перевірку та перегляд політик та процедур безпеки.
- Активно класифікуйте інформацію та застосовуйте контроль доступу.
- Використовуйте хмарні сервіси, такі як AWS Inspector, AWS CloudWatch, AWS CloudTrail та AWS Config для автоматизації контролю за дотриманням правил.
- Запобігання ексфільтрації даних.
- Інтегруйте стратегії запобігання та реагування в операції з безпеки.
- Відкрийте для себе ревізійні проекти.
- Видаліть доступ до пароля з облікових записів, для яких не потрібно входити в сервіси.

- Переглядайте та обертайте ключі доступу та облікові дані.
- Слідкуйте за блогами та оголошеннями про безпеку, щоб бути в курсі відомих нападів.
- Застосовуйте найкращі практики безпеки для будь-якого програмного забезпечення з відкритим кодом, яке ви використовуєте.
- Знову ж таки, використовуйте шифрування, коли і де це можливо.
- Ці практики допоможуть вашій організації контролювати вплив та рух важливих даних, захищати важливі системи від нападу та компрометації, а також автентифікувати доступ до інфраструктури та даних для захисту від подальших ризиків.

4). Обмежений контроль та гнучкість

Оскільки хмарна інфраструктура повністю належить, управляється та контролюється постачальником послуг, вона передає мінімальний контроль на клієнта.

У різній мірі (залежно від конкретної послуги) користувачі хмари можуть виявити, що вони мають менший контроль над функцією та виконанням послуг у хмарній інфраструктурі. Ліцензійна угода постачальника хмарних послуг (EULA) та політика управління можуть встановлювати обмеження щодо того, що клієнти можуть робити з їх розгортанням. Клієнти зберігають контроль над своїми програмами, даними та послугами, але можуть не мати однакового рівня контролю над їхньою інфраструктурою.

Кращі практики збереження контролю та гнучкості:

- Спробуйте скористатися партнером хмарного постачальника послуг, щоб допомогти у впровадженні, запуску та підтримці хмарних служб.
- Розумійте свої обов'язки та відповідальність постачальника хмарних продуктів у моделі спільної відповідальності, щоб зменшити ймовірність упущення чи помилки.

- Знайдіть час, щоб зрозуміти базовий рівень підтримки вашого постачальника хмарних послуг. Чи відповідає цей рівень обслуговування вашим вимогам підтримки? Більшість постачальників хмарних технологій пропонують додаткові рівні підтримки понад базову підтримку за додаткову ціну.
- Переконайтеся, що ви розумієте угоду про домовленості щодо інфраструктури та послуг, які збираєтесь використовувати, і як це вплине на ваші угоди з клієнтами.

5). Блокування провайдера

Блокування для постачальників - ще один сприйнятий недолік хмарних обчислень. Легке перемикання між хмарними службами - це послуга, яка ще не розвинулася повністю, і організаціям може бути важко перенести свої послуги від одного постачальника до іншого. Відмінності між платформами постачальників можуть створювати труднощі при переході від однієї хмарної платформи до іншої, що може прирівнюватися до додаткових витрат та складності конфігурації. Прогалини або компроміси, здійснені під час міграції, також можуть піддавати вашим даним додаткові вразливості безпеки та конфіденційності.

Найкращі практики зменшення залежності:

- Дизайн із найкращими практиками архітектури хмари. Усі хмарні сервіси надають можливість покращити доступність та продуктивність, розв’язати шари та зменшити вузькі місця. Якщо ви створили свої сервіси, використовуючи кращі методи хмарної архітектури, у вас менше шансів виникнути проблеми з перенесенням з однієї хмарної платформи на іншу.
- Правильно зрозумійте, що продають ваші постачальники, щоб уникнути проблем із блокуванням.
- Використовуйте мульти хмарну стратегію, щоб уникнути блокування постачальника. Хоча це може додати як розробки, так і оперативні

складності вашим розгортанням, це не повинно бути розбивачем угод. Навчання може допомогти підготувати команди архітектора та вибрати найкращі послуги та технології.

- Побудувати гнучкість як питання стратегії при розробці програм для забезпечення переносимості зараз та в майбутньому.
- Створіть свої програми за допомогою служб, які пропонують перші хмарні переваги, такі як модульність та портативність мікросервісів та коду. Подумайте про контейнери та Кубернети.

6). Побоювання щодо вартості

Прийняття хмарних рішень у малому масштабі та для короткострокових проектів може сприйматися як дороге. Однак найважливіша вигода в хмарних обчисленнях полягає в економії витрат на ІТ. Хмарні послуги з оплатою за проходом можуть забезпечити більшу гнучкість та менші витрати на обладнання, але загальний цінник може виявитися вищим, ніж ви очікували. Поки ви не будете впевнені в тому, що найкраще підійде для вас, корисно експериментувати з різноманітними пропозиціями. Ви також можете скористатися калькуляторами витрат, які надаються такими постачальниками, як веб-сервіси Amazon та веб-платформа Google Cloud.

Найкращі практики зменшення витрат:

- Намагайтеся не надмірно надавати свої послуги, а скоріше вивчіть використання послуг автоматичного масштабування.
- Переконайтеся, що у вас є можливість масштабування вниз та вгору.
- Попередньо сплачуйте та скористайтеся зарезервованими екземплярами, якщо у вас відомий мінімальний обсяг використання.
- Автоматизуйте процес, щоб запустити / зупинити ваші екземпляри, щоб заощадити гроші, коли вони не використовуються.
- Створіть сповіщення для відстеження витрат у хмарі.

# 2. SERVERLESS ARCHITECTURE

## 2.1. Що таке безсерверна архітектура ?

Протягом останніх 4 років розробки Frameworkless Server і тісно співпрацюючи з багатьма тисячами організацій, які використовують її для створення своїх додатків, я мав щастя спостерігати за тим, як рух значно розвивається. У перші дні було знайдено незліченних розробників-піонерів, які використовували цю нову архітектуру для створення неймовірних речей, незважаючи на значні виклики та відносно грубі інструменти, які існували. Новітні розробники також працював з багатьма з цих перших піонерів, щоб переконати їхні організації перейти на все без сервера, незважаючи на відсутність успішних кейсів та перевірених справжніх найкращих практик - часто грунтуючись просто на внутрішній POC, який обіцяв коротший цикл розвитку і нижчу загальну вартість власності.

Визначити термін без сервера може бути досить складно, оскільки визначення буде перетинатися з іншими термінами, такими як PaaS та Software-as-a-Service (SaaS). Один із способів пояснити без сервера - розглянути різні рівні контролю розробників над хмарною інфраструктурою. Модель IaaS Інфраструктура як послуга (IaaS), саме там розробник має найбільший контроль як над кодом програми, так і над операційною інфраструктурою в хмарі. Тут розробник несе відповідальність за забезпечення апаратними чи віртуальними машинами та може налаштувати всі аспекти того, як програма розгортається та виконується. На протилежній крайності знаходяться PaaS і SaaS моделі, де розробник не знає жодної інфраструктури і, отже, більше не має контролю над інфраструктурою. Натомість розробник має доступ до розфасованих компонентів або повних програм. Тут розробнику дозволяється розміщувати код хосту, хоча цей код може бути щільно пов'язаний з платформою.

Тут розробник має контроль над кодом, який вони розгортають у хмару, хоча цей код повинен бути записаний у вигляді функцій без стану. Розробник не турбується про експлуатаційні аспекти розгортання та обслуговування цього коду, і він очікує, що він буде стійким до відмов та автоматичного масштабування. Зокрема, код може бути зменшений до нуля там, де фактично не працює жоден сервер, коли код функції користувача не використовується, і користувачеві це не коштує. Це на відміну від рішень PaaS, де користувач часто стягується навіть під час простою.

На початку подорожі без сервера, коли спочатку розробляли безсерверну основу (в ті часи, відомі як JAWS), увага зосереджувалася на розробці та впровадженні. По мірі того, як інструмент розвивався, і нагромаджувались тематичні дослідження, ці ранні розробники без сервера підробили нову назву, яка набуває популярності в організаціях, - серверного архітектора без сервера.

Було зрозуміло, що цей новий фрагмент інфраструктури під назвою Lambda має деякі дивовижні якості, але як розробники, насправді могли побудувати з ним щось значиме? Протягом багатьох років веб-додатки були розроблені на базі трирівневої архітектури, яка розглядає програму як три незалежних рівні, а саме презентацію, бізнес-логіку та дані.  І якщо бачити, як Lambda - це хмарний сервіс, питання, яке відбулося незабаром, було таке: як ми можемо реально розгорнути ці додатки з розумом? В принципі, архітектура визначає відокремлення проблем між зовнішнім інтерфейсом (відображення даних та взаємодія з користувачами), джерел даних (постійне зберігання даних програми) та бекенда (опосередкує між інтерфейсом і джерелами даних, що обробляє бізнес-логіку) [7].

Оскільки різні рішення цих проблем були розроблені та вдосконалені, фокус розробників, що будують додатки без сервера, розширився на весь життєвий цикл додатків, включаючи тестування, моніторинг та безпеку їх безсерверних додатків.

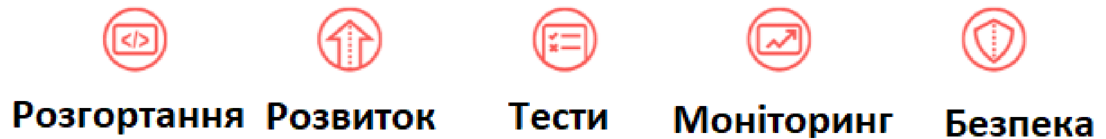


Рис. 2.1. Розширення безсерверної архітектури на весь життєвий цикл додатку

Рис. 2.2. Етапи розвитку веб-сервісів

Безсерверний архітектор - це розробник, котрий зосереджується на цьому життєвому циклі, і часто особисто володіє принаймні частиною кожного етапу життєвого циклу безсерверних додатків. Концепція

роз'єднання дозволяє розробникам впроваджувати значні зміни в рівні, не впливаючи на інші рівні, а отже, на програмне забезпечення, яке легко підтримувати. Вони не просто записують функції - вони реалізують ділові результати, роздумуючи над тим, як буде розроблений, розгорнутий, протестований, перевірений та захищений код, який забезпечує ці результати.

Безсерверні архітектури посилаються на додатки, які значно залежать від сторонніх служб (відомих як Backend як послуга або "BaaS") або спеціального коду, який працює в ефемерних контейнерах (функція як послуга або "FaaS"), найвідомішого хоста постачальника з яких на даний момент є AWS Lambda. Останнім часом веб-сервіси публікують API (інтерфейси прикладного програмування), щоб навчити їх споживачам встановлювати зв'язок та обмінюватися даними.

| **Керівні принципи** | **Тези** |
|---|---|
| Дизайн як артефакт (результати дослідження) | - Безсерверна архітектура<br>- Практика розвитку |
| Проблема релевантності (вирішення проблеми) | - Налаштування інфраструктури: апаратне та програмне забезпечення; закупівля та розгортання<br>- Підтримка інфраструктури: робочі години та зусилля команди щоб забезпечити високу тривалість роботи (програмні патчі, оновлення або заміна обладнання)<br>- Масштабованість додатків<br>- Концентрація на написанні коду, тестування та розгортання |

Таблиця 2.1. Основні вказівки керування

Незважаючи на назву, він фактично не включає запуск коду без серверів. Назва "обчислення без сервера" використовується тому, що підприємству чи особі, яка є власником системи, не потрібно купувати, орендувати або надавати сервери або віртуальні машини для того, щоб працювати заднім кодом. Популярні реалізації доступу до веб-сервісів - це SOAP (Простий протокол доступу до об'єктів) і REST (Representational State Transfer).

Код без сервера можна використовувати разом з кодом, написаним у традиційному стилі сервера, наприклад мікросервісах. Веб-сервіс розгортається та обслуговується веб-сервером, що є програмним забезпеченням, який працює на фізичній або віртуальній машині. Наприклад, частина веб-програми може бути записана у вигляді мікросервісів, а інша частина - як безсерверний код. Веб-сервер забезпечує передачу даних через HTTP (протокол передачі гіпертексту), який слідкує та повертає запити та відповіді HTTP. Крім того, може бути написана програма, яка зовсім не використовує передбачені сервери, будучи повністю безсерверною.

FaaS надає платформу, що дозволяє розробникам виконувати код у відповідь на події без складності створення та підтримки інфраструктури. Додатки чи служби сторонніх розробників керуватимуть логікою та станом на стороні сервера.

Обчислення без сервера (або просто безсерверно) стає новим і переконливим парадигма розгортання хмарних програм, багато в чому пов'язана з недавньою зміною архітектури прикладних програм для контейнерів та мікросервісів. Про це свідчить зростання уваги, що обчислювачі без серверів були використані в галузевих виставках, зустрічах, блогах та спільнотах розвитку. Навпаки, увага в академічній громаді була обмеженою.

З точки зору замовника інфраструктури як послуга (IaaS), це зміна парадигми представляє і можливість, і ризик. З одного боку, він надає розробникам спрощену модель програмування для створення хмарних

додатків, які абстрагують більшість, якщо не всіх, операційних проблем; це знижує вартість розгортання хмарного коду шляхом стягнення на час виконання, а не на розподіл ресурсів; і це платформа для швидкого розгортання невеликих фрагментів програмного коду, що реагує на події, наприклад, для координації композицій мікросервісів, який інакше працюватиме на клієнті або на спеціальному програмному забезпеченні.

З іншої сторони, розгортання таких програм на безсерверній платформі складно та вимагає відмови від дизайнерських рішень платформи, які стосуються, серед іншого, властивості контролю якості (QoS) якості, масштабування та відмовостійкості. З точки зору постачальника хмарних технологій, обчислення без серверів надає додаткову можливість контролювати весь стек розробки, зменшувати експлуатаційні витрати за рахунок ефективної оптимізації та управління хмарними ресурсами, пропонувати платформу, яка заохочує використання додаткових сервісів у їх екосистемі та знижує зусилля необхідниі для створення та керування хмарними програмами.

Обчислення без серверів - це термін, придуманий галуззю для опису моделі програмування та архітектури, де невеликі фрагменти коду виконуються в хмарі без будь-якого контролю над ресурсами, на яких працює код. Це аж ніяк не вказівка на відсутність серверів, просто розробник повинен залишати більшість операційних проблем, таких як забезпечення ресурсами, моніторинг, обслуговування, масштабованість та відмовостійкість для хмарного постачальника.

З вище наведеної інформації можна поставити логічне запитання, чим це відрізняється від моделі "Платформа як послуга" (PaaS), яка також абстрагує управління серверами. Модель без сервера надає модель програмування «збита» на основі функцій без стану. На відміну від PaaS, розробники можуть писати довільний код і не обмежуються використанням попередньо упакованої програми. Версія без сервера, яка явно використовує функції як блок розгортання, також називається Function-as-a-Service (FaaS).

Безсерверні платформи обіцяють нові можливості, що дозволяють писати масштабовані мікросервіси простішими та економічно ефективнішими, позиціонуючи себе як наступний крок у розвитку архітектури хмарних обчислень. Більшість відомих постачальників хмарних обчислень, включаючи Amazon [2], IBM [16], Microsoft [23] та Google [14], нещодавно випустили серверні обчислювальні можливості без серверів. Також є декілька зусиль з відкритих джерел, включаючи проект OpenLambda [24].

Наразі обчислення без серверів знаходиться в зародковому стані, і науково-дослідницька спільнота наразі створила лише кілька публікацій, що належать до публікацій. OpenLambda [24] пропонує довідкову архітектуру для безсерверних платформ та описує проблеми в цьому просторі. Існує також декілька книг для практиків, які орієнтуються на розробників, зацікавлених у створенні програм із використанням безсерверних платформ [11, 28].

### 2.2. Програмна модель serverless

Обслуговування без серверів було використано для підтримки більш широкого кола програм. З точки зору функціональності безсерверні та традиційні архітектури можуть використовуватися взаємозамінно. На визначення часу використання сервера, ймовірно, впливатимуть інші нефункціональні вимоги, такі як обсяг контролю над необхідними операціями, вартість, а також характеристики робочого навантаження додатків.

З точки зору витрат, переваги архітектури без сервера є найбільш очевидними для проривних, обчислювальних інтенсивних навантажень. Великі навантаження добре справляються з тим, що розробник перевантажує еластичність функції на платформу, і так само важливо, що функція може масштабуватися до нуля, тому споживачі не мають витрат, коли система не працює. Обчислювальні інтенсивні навантаження підходять, оскільки на більшості платформ сьогодні ціна виклику функції пропорційна часу роботи функції. Отже, функції, пов'язані з входом / виводом, платять за

обчислювальні ресурси, якими вони не повністю користуються. У цьому випадку серверний додаток для багатьох орендарів, що вимагає мультиплексування запитів, може бути дешевшим для роботи.

З точки зору моделі програмування, стан без функцій безсерверних функцій піддається структурі додатків, аналогічній тій, що знаходиться у функціональному реактивному програмуванні [5]. Сюди входять додатки, які демонструють схеми обробки, керовані подіями та потоком.

### 2.3. Обробка подій

Один клас застосувань, який дуже підходить для програмування на основі подій [6, 29]. Найбільш базовий приклад, популяризований AWS Lambda, що став "Hello World" безсерверних обчислень, - це проста функція обробника зображень. Функція підключена до сховища даних, наприклад Amazon S3 [2], який випромінює події зміни. Кожен раз, коли новий файл зображень завантажується в папку в S3, подія генерується та передається до функції обробника подій, яка генерує ескіз зображення, яке зберігається в іншій папці S3.

Цей приклад добре працює для демонстрацій без серверів, оскільки функція є повністю без стану та безвідмовної, що має ту перевагу, що у випадку відмови (наприклад, мережеві проблеми з доступом до папки S3) функція може бути виконана знову без побічних ефектів. Це також зразковий випадок використання бурхливого, обчислювального інтенсивного навантаження, як описано вище.

Рис. 2.3. Приклад обробки подій

### 2.3.1. Склад API

Ще один клас додатків передбачає склад ряду API. У цьому випадку логіка програми складається з фільтрації та перетворення даних. Наприклад, мобільний додаток може викликати API геолокації, погоди та мови для перекладу мови, щоб відобразити прогноз погоди для поточного місцезнаходження користувача. Код клею для виклику цих API може бути записаний у короткій функції без сервера, як проілюстровано функцією Python на малюнку 2.4. Таким чином мобільний додаток дозволяє уникнути витрат на виклик декількох API через потенційно обмежене ресурсом з'єднання мобільної мережі, і вивантажує логіку фільтрування та агрегації до бекенда.

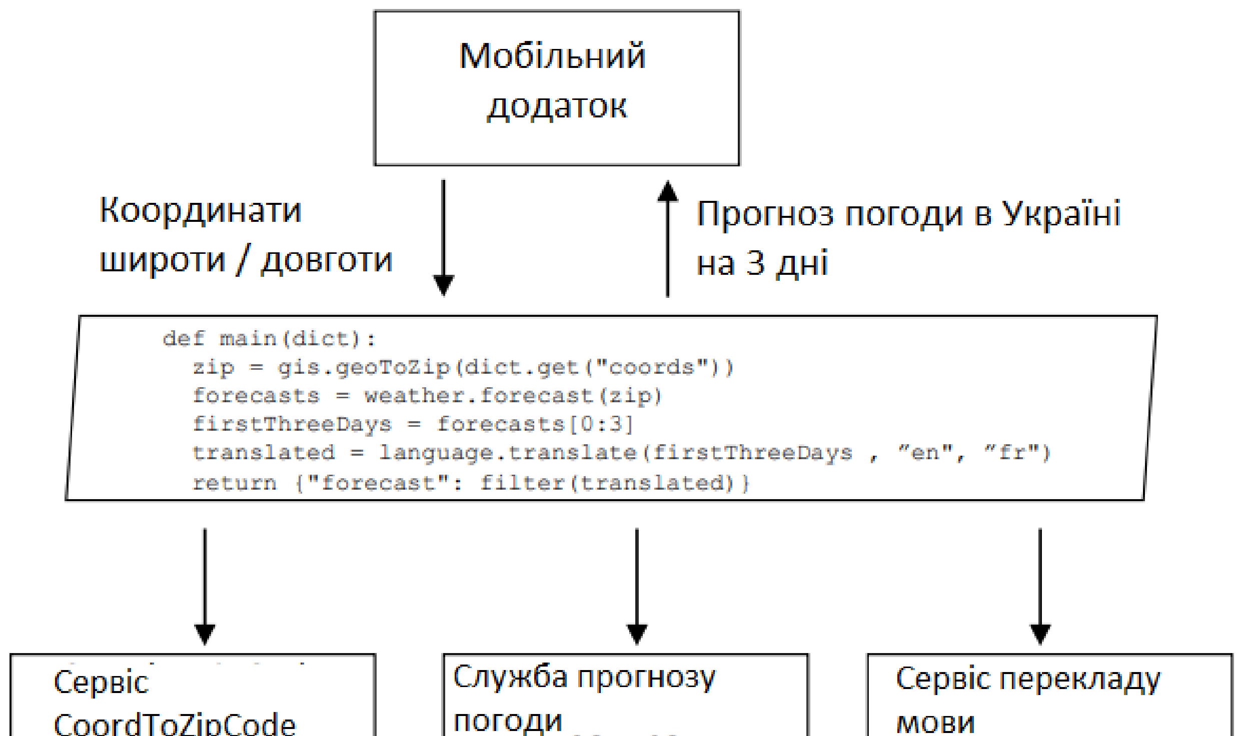

Рис. 2.4. Завантаження викликів API та логіки з мобільного додатка до бекенда

### 2.3.2. Агрегація API для зменшення викликів API

Агрегація API може працювати не тільки як механізм композиції, але і як засіб спрощення коду на стороні клієнта, який взаємодіє з агрегованим викликом. Наприклад, розглянемо мобільний додаток, який дозволяє адмініструвати екземпляр Open Stack. Виклики API у Open Stack [25] вимагають від клієнта спочатку отримати токен API, вирішити URL-адресу послуги, з якою потрібно поговорити, а потім зробити необхідний виклик API за цією URL-адресою за допомогою токена API. В ідеалі мобільний додаток заощадить енергію, зменшивши кількість необхідних викликів, необхідних для видачі команди екземпляру Open Stack. Рис. 2.5 ілюструє альтернативний підхід, коли три функції реалізують вищезазначений потік для забезпечення автентифікованих резервних копій у екземплярі Open Stack. Тепер мобільний клієнт робить один виклик, щоб викликати цю сукупну функцію. Сам потік відображається як єдиний виклик API. Зауважте, що авторизація виклику цього звернення може оброблятися зовнішньою службою авторизації, наприклад шлюз API.

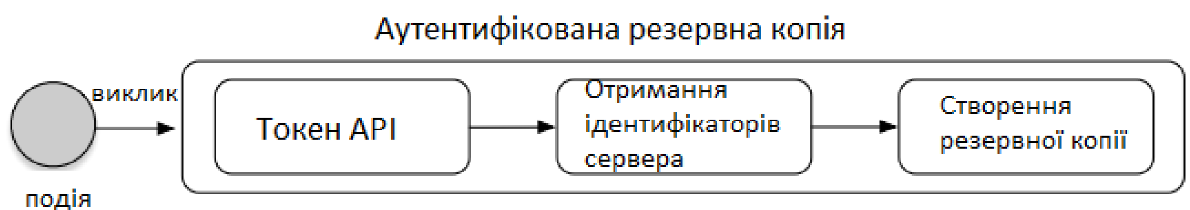


Рис. 2.5. Зменшення кількості викликів API, необхідних для мобільного клієнта

### 2.4. Контроль потоку для відстеження проблем

Склад функцій без сервера може використовуватися для управління потоком даних між двома службами. Наприклад, уявіть додаток, який дозволяє користувачам надсилати відгуки розробникам додатків у вигляді анотованих знімків екрана та тексту. На малюнку 2.6 додаток подає ці дані в бекенд, що складається з масштабованої бази даних та системи відстеження випусків за місцем. Останнє в основному використовується командою розробників і не розраховане на прийняття великого обсягу трафіку.

З іншого боку, перший здатний реагувати на великий обсяг трафіку. Ми розробляємо нашу систему для постановки всіх записів зворотного зв’язку в базу даних за допомогою функції без сервера, що виключає необхідність створення окремого сервера для обробки запитів зворотного зв’язку, але все ж дозволяє нам рівень опосередкованості між додатком і базовою базою даних. Після того, як ми наберемо достатню кількість оновлень, ми можемо об'єднати їх в одне оновлення, яке викликає функцію контрольованого надсилання проблем до трекера проблем. Цей потік буде працювати для масштабованої системи баз даних [7] та системи відстеження випусків, яка приймає пакетні входи [20].

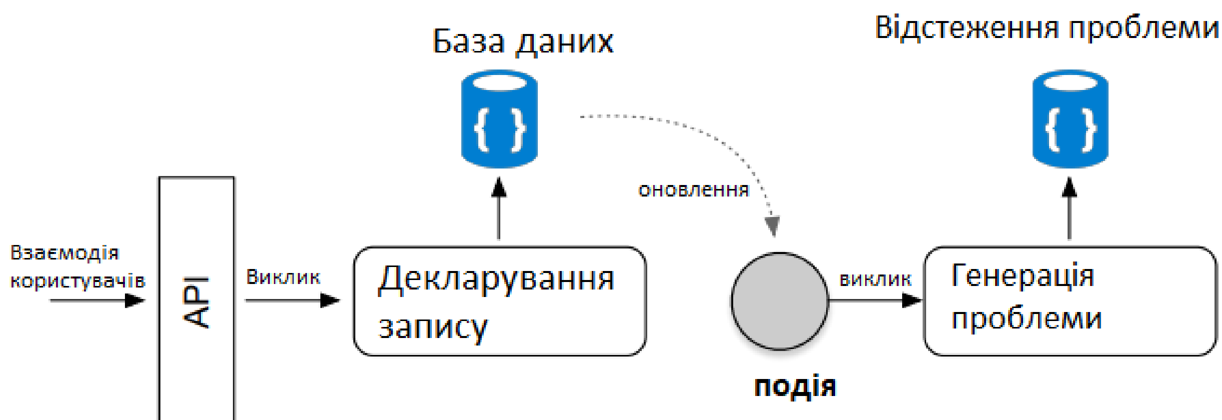

Рис. 2.6. Пакетна виклик для відстеження проблем

## 2.5. Serverless навантаження

Навантаження та його співвідношення з вартістю можуть допомогти визначити, чи підходить безсерверна фрхітектура. Нечасті, але бурхливі навантаження можуть краще обслуговуватися без серверів, що забезпечує горизонтальне масштабування без необхідності спеціальної інфраструктури, яка стягує час простою. Для більш стійких навантажень частота, з якою виконується функція, впливатиме на те, наскільки економним може бути кешування, що дозволяє набагато швидше виконати теплі контейнери, ніж виконувати з холодного контейнера.

Ці характеристики продуктивності можуть допомогти керівнику розробника при розгляді без серверів. Цікаво, що міркування щодо вартості можуть вплинути на структурування програми без сервера. Наприклад, функція без сервера, пов'язана з входом / виводом, може бути розкладена на декілька обчислених обчислень

# 3. БЕЗСЕРВЕРНИЙ РОБОЧИЙ ПРОЦЕС TENSORFLOW

Основна мета процесу проектування наукових досліджень полягає у вирішенні проблеми шляхом створення нових знань. Рішення виникають через розуміння проблеми, а також на основі існуючих знань. Вони набуваються та захоплюються у формі життєздатних артефактів. У своїх опублікованих статтях визначні вчені сьогодення наполягали на керівних принципах науково-дослідницьких робіт, що результат процесу повинен бути цілеспрямованим артефактом у вигляді конструкції, моделі, методу або інстанції. Крім того, артефакт заданий проблемою, що стосується проблемного домену.

## 3.1. Машинне навчання

Машинне навчання - це наука про те, щоб комп’ютери навчалися та діяли так, як це роблять люди, і вдосконалювати своє навчання з часом самостійно, подаючи їм дані та інформацію у вигляді спостережень та взаємодії в реальному світі.

Вищевикладене визначення охоплює ідеальну мету або кінцеву мету машинного навчання, як висловили багато дослідників у цій галузі. Мета цього розділу - надати користувачеві, орієнтованому на бізнес, експертний погляд на те, як визначено машинне навчання та як воно працює. Машинне навчання та штучний інтелект поділяють одне і те ж визначення у свідомості багатьох, проте користувачі повинні також визнати деякі відмінності. Список

літератури та пов'язані з ними інтерв'ю  містяться в кінці цієї дипломної роботі для подальшого пошуку.

Існує багато різних типів алгоритмів машинного навчання, сотні щодня публікуються, і вони, як правило, групуються за стилем навчання (тобто, під контролем навчання, непідконтрольним навчанням, напівконтрольним навчанням) або за подібністю за формою чи функцією (тобто класифікацією, регресія, дерево рішень, кластеризація, глибоке навчання тощо). Незалежно від стилю чи функції навчання, всі комбінації алгоритмів машинного навчання складаються з наступного:

- Представлення (набір класифікаторів або мова, яку розуміє комп'ютер)
- Оцінювання (мета / оцінка функції)
- Оптимізація (метод пошуку; наприклад, найчастіше класифікатор з найвищим балом; застосовуються як позаштатні, так і спеціальні методи оптимізації)

Основна мета алгоритмів машинного навчання полягає в узагальненні поза навчальними зразками, тобто успішно інтерпретувати дані, які раніше ніколи не бачили.

### 3.2. Принципи машинного навчання

Існують різні підходи до навчання машин, від використання дерев базових рішень до кластеризації до шарів штучних нейронних мереж (останній з яких поступився місцем глибокому навчанню), залежно від того, яке завдання ви намагаєтеся виконати, типу та кількість наявних у вас даних. Ця динаміка розглядається в програмах настільки ж різними, як медична діагностика або автошкола.

Хоча акцент робиться часто на виборі найкращого алгоритму навчання, дослідники виявили, що деякі з найцікавіших питань виникають із жодного з

доступних алгоритмів машинного навчання, які не відповідають рівню. Здебільшого це проблема з навчальними даними, але це також виникає при роботі з машинним навчанням у нових областях.

Дослідження, проведені під час роботи над реальними додатками, часто сприяють прогресу в цій галузі, а причини двоякі:

1. Схильність до виявлення меж та обмежень існуючих методів

2. Дослідники та розробники, що працюють з експертами в галузі домену та використовують час та досвід для підвищення продуктивності системи.

Іноді це також відбувається «випадково». Ми можемо розглядати модельні ансамблі або комбінації багатьох алгоритмів навчання для підвищення точності. Команди, які змагалися за ціну Netflix 2009 року, виявили, що вони отримали найкращі результати, поєднуючи своїх учнів з учнями іншої команди, в результаті чого вдосконалився алгоритм рекомендацій.

Один важливий момент (заснований на інтерв'ю та бесідах з експертами в цій галузі), що стосується застосування в бізнесі та інших місцях, - це те, що машинне навчання - це не просто, а то й навіть, автоматизація, часто неправильно зрозуміла концепція. Якщо ви думаєте таким чином, ви зобов’язані пропустити цінні уявлення, які можуть надати машини, і отримані можливості (переосмислення цілої бізнес-моделі, наприклад, як у галузях промисловості, таких як виробництво та сільське господарство).

Машини, які навчаються, корисні людям, оскільки, використовуючи всю потужність обробки, вони можуть швидше виділити або знайти зразки у великих (або інших) даних, які інакше люди не пропустили б. Машинне навчання - це інструмент, який можна використовувати для розширення здібностей людини до вирішення проблем та обґрунтованих висновків щодо широкого спектру проблем, починаючи від допомоги діагностиці захворювань до розробки глобальних змін клімату.

"Машинне навчання не може отримати щось із нічого ... те, що він робить, отримує більше від меншого". - Доктор Педро Домінго, Університет Вашингтона.

Дві найбільші, історичні (і тривалі) проблеми машинного навчання включали переозброєння (в якому модель виявляє упередженість щодо навчальних даних і не узагальнює нові дані та / або дисперсію, тобто вивчає випадкові речі під час навчання нових даних) та розмірність (алгоритми з більшою кількістю функцій працюють у вищих / декількох вимірах, що ускладнює розуміння даних). Наявність доступу до достатньо великого набору даних у деяких випадках також була основною проблемою.

Однією з найпоширеніших помилок серед початківців машинного навчання є успішна перевірка даних про навчання та наявність ілюзії успіху; Домінго (та інші) наголошують на важливості зберігання деяких наборів даних під час тестування моделей і лише використання цих зарезервованих даних для тестування обраної моделі з подальшим вивченням усього набору даних.

Коли алгоритм навчання (тобто, який навчається) не працює, часто швидший шлях до успіху полягає в тому, щоб подати машині більше даних, доступність яких на сьогоднішній день відома як головний рушій прогресу в алгоритмах машинного та глибокого навчання останнім часом років; однак це може призвести до проблем зі масштабованістю, за яких у нас є більше даних, але час, щоб дізнатися, що дані залишаються проблемою.

З точки зору мети, машинне навчання - це не мета і рішення саме по собі. Крім того, спроба використовувати його як покривне рішення, тобто "BLANK" не є корисною вправою; натомість, підходити до скоупу з проблемою або метою, найчастіше, керується більш конкретним запитанням - "BLANK".

Глибоке навчання передбачає вивчення та проектування машинних алгоритмів для вивчення хорошого представлення даних на кількох рівнях абстрагування (способи організації комп'ютерних систем). Нещодавнє оприлюднення глибокого навчання через DeepMind, Facebook та інші установи виділило його як "наступний кордон" машинного навчання.

Міжнародна конференція з машинного навчання (ICML) вважається однією з найважливіших у світі. Цьогоріч відбувся в червні в Нью-Йорку, і

він об’єднав дослідників з усього світу, які працюють над вирішенням сучасних проблем глибокого навчання:

1. Безперервне навчання в невеликих наборах даних
2. На основі імітаційного навчання та передачі в реальному світі

Системи глибокого навчання за останні десятиліття досягли значних успіхів у таких сферах, як виявлення та розпізнавання викидів, переклад тексту в мовлення, пошук інформації та інші. Зараз дослідження орієнтовані на розробку технологій машинного навчання з ефективними даними, тобто системи глибокого навчання, які дозволять навчатися ефективніше, з однаковою ефективністю за менший час та з меншою кількістю даних, у передових областях, таких як персональна охорона здоров'я, посилене навчання роботів, аналіз настроїв та ін.

Нижче наводиться підбірка найкращих практик та концепцій застосування машинного навчання, які ми зібрали з наших інтерв'ю для серій подкастів та з окремих джерел, цитованих у кінці цієї статті. Ми сподіваємось, що деякі з цих принципів пояснять, як застосовується ВР та як уникнути деяких загальних підводних каменів, до яких компанії та дослідники можуть бути вразливими, починаючи проект, пов’язаний з веденням МЛ.

- Напевно, найважливішим фактором успішних проектів машинного навчання є функції, які використовуються для опису даних (які залежать від домену), і наявність адекватних даних для тренування ваших моделей в першу чергу
- Більшість випадків, коли алгоритми не працюють добре, це пов’язано з проблемою з навчальними даними (тобто недостатньою кількістю / перекошеними даними; шумними даними або недостатньою характеристикою, що описує дані для прийняття рішень

- "Простота не передбачає точності" - немає (за Домінго) зв'язку між кількістю параметрів моделі та тенденцією до перевиконання.
- Отримання експериментальних даних (на відміну від даних спостережень, над якими ми не маємо контролю) слід робити, якщо це можливо (наприклад, дані, отримані від надсилання різних варіантів електронного листа до вибіркової вибірки випадкової аудиторії)
- Незалежно від того, чи ми позначаємо дані причинними чи співвідносними, важливішим моментом є прогнозування наслідків наших дій
- Завжди відкладайте частину вашого набору даних про навчання для перехресної перевірки; ви хочете, щоб обраний вами класифікатор або алгоритм навчання добре працював на нових даних

### 3.3. Реалізація TensorFlow з автоматичною підготовкою EC2

Завдання для машинного навчання часто вимагають великих витрат часу та ресурсів, тому інтегрувати цей процес у автоматизований робочий процес у режимі реального часу може бути складним завданням. У попередній дипломній роботі я продемонстрував прототип виконання завдань без сервера TensorFlow на AWS Lambda. Мотивуючий випадок використання для цього був тоді, коли мічені дані мають обмежений характер, і нові вхідні записи потрібно дуже швидко включати в модель.

Хоча в Lambda можливо запустити стандартну бібліотеку Python TensorFlow, ймовірно, що багато додатків незабаром зіткнуться з обмеженнями щодо розміру пакета розгортання та / або часу виконання або потребують додаткових параметрів обчислення. Цей пункт дипломної роботи описує як зберегти управління даними та прогнозування без серверів, але вивантажити навчальні завдання тимчасовим екземпляром EC2. Цей шаблон, наприклад, створює принцип для розробки економічно ефективної гіперпараметричної оптимізації в хмарі.

Збереження функціональності передбачення в Lambda означає, що все ще може бути обмеження розміру через завантаження TensorFlow. Щоб пом'якшити це, всі функції Lambda будуть записані для Node.js, що також дозволить нам використовувати TensorFlow.js замість стандартної бібліотеки Python.

TensorFlow.js постачається у веб-переглядачі та версії Node, остання включає C ++ прив'язки для поліпшення продуктивності. Версія Node здається очевидним вибором, проте вона розпаковується до 690 МБ (!), Що робить її негайно непридатною для Lambda. Зважаючи на те, що ми не будемо тренуватися в рамках функцій Lambda, показник продуктивності прийнятний для прогнозування, тому ми будемо використовувати версію браузера, яка розпаковується до 55 Мб.

Для основної моделі машинного навчання ми спробуємо передбачити рівень комфорту людини на основі таких вхідних параметрів:

- Температура (F)
- Відносна вологість (%)
- Ізоляція (в одиницях "clo")
- Швидкість вітру (м / с)

Фактична модель використовуватиме просту (неоптимізовану) нейронну мережу, побудовану за допомогою Keras API TensorFlow. Для зберігання даних ми створимо дві таблиці в DynamoDB:

- data - Зберігатиме марковані вхідні дані для навчання
- model - Зберігає метадані та показники з навчальних завдань

### 3.3.1. Налаштування навколишнього середовища

Оскільки наш проект буде змішаний з файлами Node Lambda та файлами Python EC2, ми розділимо їх у структурі папок, як показано нижче. Ми також будемо використовувати безсерверний фреймворк, який

залишатиметься на найвищому рівні, тоді як частини Node та Python будуть ініціалізовані у відповідних папках.

Для початку встановіть без сервера та ініціалізуйте новий проект за допомогою шаблону Node. Повинні з'явитися обробник пластини котла (handler.js) та файл конфігурації (serverless.yml).

```
$ npm install -g serverless
$ mkdir -p LambdaAutoTraining/{js,py}
$cdLambdaAutoTraining
$ serverless create --template aws-nodejs
```

Налаштування Node

Перейдіть до папки js, ініціалізуйте новий проект Node та встановіть Tensorflow.js (лише версія браузера!).

```
$ cd js
$ npm init
...follow prompts
$ npm install @tensorflow/tfjs
```

Далі, використовуючи діаграму архітектури в якості керівництва, створіть необхідні файли JavaScript, які відображатимуть кінцеві функції Lambda.

```
$ touch test.js upload.js train.js infer.js s3proxy.js
```

Нарешті, скопіюйте код котла з handler.js у кожен із цих файлів, а потім видаліть handler.js.

Настройка Python

Перейдите в папку py и создайте новую виртуальную среду. Для создания схемы обучения мы будем использовать записную книжку Jupyter, а также нам потребуется модуль tenorflowjs, чтобы мы могли преобразовать сохраненную модель в формат, понятный TensorFlow.js.

```
$ cd ../py
$ pyenv virtualenv 3.7.3 autotraining
$ pyenv activate autotraining
```

```
$ pip install tensorflow tensorflowjs jupyter
$ pip freeze > requirements.txt
```

Нам потрібно лише створити файл блокноту Юпітера та Dockerfile у цьому розділі. Файл Python буде створений як частина процесу складання Docker.

```
$ touch train.ipynb Dockerfile
```

Налаштування Serverless

Файл serverless.yml є основною конфігурацією для вашого проекту. Спочатку видаліть увесь текст із панелі котлів у файлі (ви можете звернутися до документів пізніше для всіх різних параметрів, якщо потрібно), а потім починайте розробляти розділ провайдера.

Одна з ключових відмінностей від більшості прикладів без сервера AWS полягає в тому, що ми будемо визначати власну роль IAM. Зазвичай роль буде замінена розділом iamRoleStatements, який дозволяє налаштувати політику, яка без сервера об'єднується зі своєю загальною роллю IAM. Однак нам потрібно включити EC2 як довірену сутність, яка недоступна як частина iamRoleStatements. Побудова цього відбуватиметься згодом у розділі ресурсів.

Розділ оточення надає нам доступ до змінних, що залежать від розгортання, всередині функцій Lambda. IAM_ROLE знадобиться для створення політики екземпляра EC2, а API_URL буде використовуватися як test.js, так і infer.js для здійснення викликів у наші кінцеві точки API шлюзу.

```
service: lambdaautotraining
provider:
  name: aws
  runtime: nodejs10.x
  stage: dev
  region: us-east-1
  role: IamRole
```

```
    environment:
      SERVICE: ${self:service}
      REGION: ${self:provider.region}
      BUCKET: ${self:service}-${opt:stage, self:provider.stage}
      STAGE: ${opt:stage, self:provider.stage}
      DYNAMODB_TABLE_MODELS: ${self:service}-models-${opt:stage, self:provider.stage}
      DYNAMODB_TABLE_DATA: ${self:service}-data-${opt:stage, self:provider.stage}
      FUNCTION_PREFIX: ${self:service}-${opt:stage, self:provider.stage}-
      IAM_ROLE: ${self:service}-${opt:stage, self:provider.stage}-${self:provider.region}-managedrole
      API_URL:
        Fn::Join:
          - ""
          - - "https://"
            - Ref: "ApiGatewayRestApi"
            - ".execute-api.${self:provider.region}.amazonaws.com/${self:provider.stage}"

  package:
    exclude:
      - py/**

  # ...
```

Далі визначте кожну з функцій, використовуючи діаграму та створені файли як керівництво. Для простоти, ім’я кожної функції обробника та кінцева точка API будуть такими ж, як і ім’я файлу. Завантаження, висновок та s3proxy буде викликано через шлюз API і, отже, матиме події http. Для

s3proxy ми будемо використовувати параметри шляху для визначення файлу, який запитується, тому що це папка у bucket S3.

Для функції train ми будемо використовувати тригер DynamoDB, який буде включений у розділ ресурсів. Ця подія буде ініційована, коли буде принаймні одна нова подія і буде дотримано будь-який з наступних обмежень:

- batchSize - максимальна кількість створених елементів
- batchWindow - Максимальна кількість часу після створення першого елемента

Оскільки train буде головним чином відповідати за запуск екземпляра EC2, ми також визначимо кілька додаткових змінних умов середовища. У цьому прикладі наші зображення Докера зберігатимуться в реєстрі AWS Docker (ECR), однак можуть бути використані інші.

- AMI_ID - у цьому прикладі ми будемо використовувати ami-0f812849f5bc97db5, оскільки він попередньо створений для Docker
- KEY_NAME - це ім'я файлу pem, необхідного для доступу SSH до примірника; будьте впевнені, що у вас є доступ до приватного ключа!
- INSTANCE_TYPE - Дійсні значення - це доступні аромати EC2 для цього зображення
- SPOT_DURATION - Мінімальний час в хвилинах до того, як екземпляр спот може бути перерваний
- VALID_HRS - Максимальний час, коли запит на спот триватиме, якщо його не буде виконано
- ECR_ID - має бути таким же, як ідентифікатор вашого облікового запису AWS
- ECR_REPO - Назва сховища та проекту ECR

Нарешті, test буде використовуватися лише для запуску вручну, тому не має пов'язаних подій.

# ...

```yaml
functions:
  upload:
    handler: js/upload.upload
    events:
      - http:
          path: upload
          method: post

  train:
    handler: js/train.train
    environment:
      AMI_ID: ami-0f812849f5bc97db5
      KEY_NAME: ec2testing
      INSTANCE_TYPE: t2.micro
      SPOT_DURATION: 60
      VALID_HRS: 4
      ECR_ID: 530583866435
      ECR_REPO: lambda-auto-training/lambda-auto-training-${opt:stage, self:provider.stage}
    events:
      - stream:
          type: dynamodb
          arn:
            Fn::GetAtt: [DataDynamoDbTable, StreamArn]
          batchSize: 100
          batchWindow: 300

  infer:
    handler: js/infer.infer
    events:
      - http:
```

```yaml
          path: infer
          method: post

  test:
    handler: js/test.test

  s3proxy:
    handler: js/s3proxy.s3proxy
    events:
      - http:
          path: s3proxy/{key}/{filename}
          method: get

# ...
```

Далі створіть bucket S3 та дві таблиці DynamoDB (з обмеженою умовою на цьому етапі). Зауважте, що таблиця даних також містить StreamSpecification, який буде використовуватися для запуску функції train.

```yaml
# ...

resources:
 Resources:
   Bucket:
     Type: AWS::S3::Bucket
     Properties:
       BucketName: ${self:provider.environment.BUCKET}
   ModelsDynamoDbTable:
     Type: 'AWS::DynamoDB::Table'
     DeletionPolicy: Retain
     Properties:
       AttributeDefinitions:
```

```
          -
            AttributeName: created
            AttributeType: N
        KeySchema:
          -
            AttributeName: created
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TableName:
${self:provider.environment.DYNAMODB_TABLE_MODELS}
    DataDynamoDbTable:
      Type: 'AWS::DynamoDB::Table'
      DeletionPolicy: Retain
      Properties:
        AttributeDefinitions:
          -
            AttributeName: created
            AttributeType: N
        KeySchema:
          -
            AttributeName: created
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TableName:
${self:provider.environment.DYNAMODB_TABLE_DATA}
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
```

```
# ...
```

Кінцевим ресурсом для створення є наша спеціальна роль IAM, яка буде використовуватися всіма функціями, а документи без сервера забезпечують хороший шаблон відправної точки. Щоб дозволити перенесення ролі від Lambda до EC2, нам знадобляться дві речі:

- Додати ec2.amazonaws.com до розділу AssumeRolePolicyDocument
- Додати дозвольну дію для iam: PassRole у розділі Політика

У розділі "Policies" спершу скопіюємо за замовчуванням політики без сервера для ведення журналів і bucket для розгортання S3 (зазвичай вони створюються автоматично). Далі ми додамо спеціальні оператори для bucket S3 та таблиць DynamoDB, визначених раніше. Зауважте, що при створенні власної політики, політика потоків DynamoDB не буде створена автоматично, тому нам потрібно чітко її визначити.

Додатково ми додамо політики, необхідні для створення екземпляра EC2:

- EC2 - Створіть і запустіть примірник.
- CloudWatch - Створіть, опишіть та увімкніть сигнал тривоги, щоб ми могли автоматично припинити екземпляр після закінчення навчання.
- ECR - Дозволяє витягувати зображення Докера (цим буде користуватися лише EC2, а не функцією Lambda).
- IAM - Отримайте, створіть та додайте роль до профілю екземпляра. Під час запуску екземпляра EC2 з консолі та вибору ролі IAM цей профіль автоматично створюється, але нам це потрібно робити вручну в межах нашої функції.

Примітка безпеки: Перед розгортанням у виробництві ці політики слід обмежувати лише необхідними ресурсами.

```
# ...

  IamRole:
```

```
    Type: AWS::IAM::Role
    Properties:
     Path: /
     RoleName: ${self:service}-${opt:stage, self:provider.stage}-${self:provider.region}-managedrole
     AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
       - Effect: Allow
        Principal:
         Service:
          - lambda.amazonaws.com
          - ec2.amazonaws.com # added for transfer in train.js
        Action: sts:AssumeRole
     Policies:
      - PolicyName: ${self:service}-${opt:stage, self:provider.stage}-${self:provider.region}-managedpolicy
       PolicyDocument:
        Version: '2012-10-17'
        Statement:
         # Begin default Serverless policies
         - Effect: Allow
          Action:
           - logs:CreateLogGroup
           - logs:CreateLogStream
           - logs:PutLogEvents
          Resource:
           - 'Fn::Join':
            - ':'
            -
             - 'arn:aws:logs'
```

```
        - Ref: 'AWS::Region'
        - Ref: 'AWS::AccountId'
        - 'log-group:/aws/lambda/*:*:*'
-  Effect: "Allow"
   Action:
     - "s3:PutObject"
   Resource:
     Fn::Join:
       - ""
       - - "arn:aws:s3:::"
         - "Ref" : "ServerlessDeploymentBucket"
# Begin custom policies
- Effect: Allow
  Action:
    - s3:*
  Resource:
   Fn::Join:
     - ""
     - - "arn:aws:s3:::"
       - ${self:provider.environment.BUCKET}
       - "/*"
- Effect: Allow
  Action:
    - dynamodb:Query
    - dynamodb:Scan
    - dynamodb:GetItem
    - dynamodb:PutItem
    - dynamodb:UpdateItem
    - dynamodb:DeleteItem
    - dynamodb:GetRecords
    - dynamodb:GetShardIterator
```

```
          - dynamodb:DescribeStream
          - dynamodb:ListStreams
          - dynamodb:BatchWriteItem
        Resource:
          -                                        "arn:aws:dynamodb:${opt:region,
self:provider.region}:*:table/${self:provider.environment.DYNAMODB_TABLE
_MODELS}"
          -                                        "arn:aws:dynamodb:${opt:region,
self:provider.region}:*:table/${self:provider.environment.DYNAMODB_TABLE
_DATA}"
          - { Fn::GetAtt: [ DataDynamoDbTable, Arn ] }
      # For custom role stream policy must be explicitly defined
      - Effect: Allow
        Action:
          - dynamodb:GetRecords
          - dynamodb:GetShardIterator
          - dynamodb:DescribeStream
          - dynamodb:ListStreams
        Resource:
          -                                        "arn:aws:dynamodb:${opt:region,
self:provider.region}:*:table/${self:provider.environment.DYNAMODB_TABLE
_DATA}/stream/*"
      - Effect: Allow
        Action:
          - ec2:RunInstances
          - ec2:TerminateInstances
          - cloudwatch:PutMetricAlarm
          - cloudwatch:DescribeAlarms
          - cloudwatch:EnableAlarmActions
        Resource: "*"
      # Allow for transferring role to EC2
```

```
- Effect: Allow
  Action:
    - iam:PassRole
    - iam:GetInstanceProfile
    - iam:CreateInstanceProfile
    - iam:AddRoleToInstanceProfile
  Resource: "*"
# Used only by EC2 when role is transfered
- Effect: Allow
  Action:
    - ecr:GetAuthorizationToken
    - ecr:GetDownloadUrlForLayer
    - ecr:BatchGetImage
    - ecr:BatchCheckLayerAvailability
  Resource:
```

Оскільки ви вже додали код bucket до кожної з функцій, тепер ви можете розгорнути і перевірити, чи все налаштовано належним чином.

```
$ serverless deploy --stage dev
...
$ curl -X POST "https://<api_id>.execute-api.<region>.amazonaws.com/dev/upload"
Go Serverless v1.0! Your function executed successfully!
```

Тепер ми готові створити додаток!

### 3.3.2. Налаштування Lambda

Функція `upload` буде приймати як вхід масив нещодавно маркованих даних і зберігати їх у таблиці DynamoDB. Потім це оновлення почне спрацьовувати потік для запуску функції `train` .

У upload.js спочатку імпортуйте та налаштуйте SDK AWS. Оскільки ця функція запускається від події HTTP, ми прочитаємо поле `body` , а потім побудуємо масив об'єктів, що представляють окремі елементи вставки DynamoDB. Зауважте, що хоча поля мають різні типи (наприклад, "N" або "S" для числа та рядка відповідно), фактичні значення потрібно передавати у вигляді рядків.

```
'use strict';

// Load the SDK for JavaScript
const AWS = require('aws-sdk');
AWS.config.update({region: process.env.REGION});
const ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});

module.exports.upload = (event, context, callback) => {
  var data = event.body
  if (typeof data == 'string') {
    data = JSON.parse(data)
  }

  // construct put request items
  var requestItems = [];
  data.forEach(item => {
    requestItems.push({
      PutRequest: {
        Item: {
          created: { "N": item.created.toString() },
          temp: { "N": item.temp.toString() },
```

```
          rh: { "N": item.rh.toString() },
          wind: { "N": item.wind.toString() },
          clo: { "N": item.clo.toString() },
          label: { "S": item.label.toString() },
          score: { "N": item.score.toString() },
        }
      }
    });
  });

  if (requestItems.length == 0) {
    callback(null, {
      statusCode: 200,
      headers: {'Content-Type': 'text/plain'},
      body: 'No records created since request body length was zero',
    });
    return;
  }

  // ....
```

Якщо є нові елементи для запису, ми побудуємо новий об'єкт, а потім використаємо batchWriteItem з DynamoDB AWS SDK для запису нових елементів. batchWriteItem є більш ефективним, ніж ряд запитів putItem, а також є атомним.

```
// ...

  var params = {
    RequestItems: {
```

```
      [process.env.DYNAMODB_TABLE_DATA]: requestItems
    }
  };

  console.log('params>>>', params);

  ddb.batchWriteItem(params, function(err, data) {
    if (err) {
      console.log("Error", err);
      callback(Error(err), {
        statusCode: 500,
        headers: {'Content-Type': 'text/plain'},
        body: 'Error encountered during record creation',
      });
    } else {
      console.log("Success", data);
      callback(null, {
        statusCode: 200,
        headers: {'Content-Type': 'text/plain'},
        body: requestItems.length.toString() + ' records successfully created',
      });
    }
  });
};
```

Тепер, коли ми створили функцію завантаження, ви також можете створити test.js для створення випадкових даних для тестування робочого процесу та заповнення бази даних. Докладніше див. Файл Github.

Повторно розгорніть на етап розробки і протестуйте кінцеву точку. У цей момент важливо почати заповнювати DynamoDB даними, що можна зробити через ручні дзвінки до функції test.js.

```
$ severless deploy --stage dev
...
$ curl -X POST "https://<api_id>.execute-api.<region>.amazonaws.com/dev/upload" -d '[{"created": 1570323309012, "temp": 75, "rh": 60, "wind": 0.6, "clo": 1.0, "label": "ok", "score": -1}]'
1 records created successfully
```

Незважаючи на те, що функція train.js не була розроблена, ви все одно повинні бачити її, коли буде досягнуто розміру партії. Нарешті, рядок потрібно кодувати base64 відповідно до вимог EC2.

```
'use strict';

const child_process = require("child_process");

const AWS = require('aws-sdk');
AWS.config.update({region: process.env.REGION});
const iam = new AWS.IAM();
const ec2 = new AWS.EC2({apiVersion: '2016-11-15'});
const cw = new AWS.CloudWatch({apiVersion: '2010-08-01'});

module.exports.train = async event => {
  // Create init script
  var userData = `#!/bin/bash
  sudo yum install -y unzip
  curl "https://s3.amazonaws.com/aws-cli/awscli-bundle.zip" -o "awscli-bundle.zip"
  unzip awscli-bundle.zip
  sudo ./awscli-bundle/install -i /usr/local/aws -b /usr/local/bin/aws
  $(aws ecr get-login --no-include-email --region <region>)
  docker pull <ecr_id>.dkr.ecr.<region>.amazonaws.com/<repo>:latest
  docker run --name keras-remote-training \
```

```
  -e "TABLE_MODELS=<table_models>" -e "TABLE_DATA=<table_data>" \
  -e "BUCKET=<bucket>" -e "REGION=<region>"\
  -d --rm <ecr_id>.dkr.ecr.<region>.amazonaws.com/<repo>:latest
`;

userData = userData.replace(/<region>/g, process.env.REGION);
userData = userData.replace(/<ecr_id>/g, process.env.ECR_ID);
userData = userData.replace(/<repo>/g, process.env.ECR_REPO);
userData = userData.replace(/<table_models>/g, process.env.DYNAMODB_TABLE_MODELS);
userData = userData.replace(/<table_data>/g, process.env.DYNAMODB_TABLE_DATA);
userData = userData.replace(/<bucket>/g, process.env.BUCKET);

console.log('Final userData>>> ', userData);

var userDataBuff = new Buffer(userData);
var userDataBase64 = userDataBuff.toString('base64');
console.log('Base64 encoded userData>>>', userDataBase64);
```

Потім завантажте профіль екземпляра, який визначає роль IAM, яку використовуватиме екземпляр EC2. Кожен з викликів, який потрібно блокувати, використовує форму обіцянки з ключовим словом очікування.

```
// Get instance profile
var profileParams = {
  InstanceProfileName: process.env.SERVICE+'-'+process.env.STAGE,
}

var profileData = null;
try {
  profileData = await iam.getInstanceProfile(profileParams).promise();
```

```
} catch (err) {
  if (err.statusCode == 404) {
    console.log('Instance profile does not exist, creating');
    profileData = await iam.createInstanceProfile(profileParams).promise();
    child_process.execSync("sleep 10"); // alllows new profile to propogate
    profileParams.RoleName = process.env.IAM_ROLE;
    var roleData = await
iam.addRoleToInstanceProfile(profileParams).promise();
    console.log('Added role to instance profile');
  } else {
    console.error('Unexpected error getting instance profile: ', err);
    return {
      statusCode: 500,
      body: JSON.stringify({error: "Error getting instance profile"}),
    };
  }
}
```

З використанням профілю екземпляра ми визначимо повний набір параметрів EC2 для точкового екземпляра. Альтернативним варіантом було б окремо створити шаблон і запустити його безпосередньо. Ми також припинимо примірник після вимкнення, і додатковою оптимізацією тут було б зупинити / запустити стійкий екземпляр за потребою.

```
// Set EC2 instance parameters
var instanceParams = {
  ImageId: process.env.AMI_ID,
  InstanceType: process.env.INSTANCE_TYPE,
  KeyName: process.env.KEY_NAME,
  MinCount: 1,
  MaxCount: 1,
  UserData: userDataBase64,
  InstanceInitiatedShutdownBehavior: 'terminate',
```

```
  InstanceMarketOptions: {
    MarketType: 'spot',
    SpotOptions: {
      BlockDurationMinutes: process.env.SPOT_DURATION,
      InstanceInterruptionBehavior: 'terminate',
      SpotInstanceType: 'one-time',
    }
  },
  IamInstanceProfile: {
    Arn: profileData.InstanceProfile.Arn,
  },
};
```

Зараз ми готові розпочати створення EC2. Після успіху ми створимо та включимо сигнал тривоги, який автоматично припинить екземпляр, коли CPU опуститься нижче певного порогу, який ми використовуємо як проксі для завершення навчання.

```
// Create EC2 instance
try {
  var instanceData = await ec2.runInstances(instanceParams).promise();
  var instanceId = instanceData.Instances[0].InstanceId;
  console.log("Created instance", instanceId);

  // Create alarm to auto-terminate
  var alarmParams = {
    AlarmName: process.env.SERVICE+'_CPU_Utilization_'+instanceId,
    ComparisonOperator: 'LessThanThreshold',
    EvaluationPeriods: 2,
    MetricName: 'CPUUtilization',
    Namespace: 'AWS/EC2',
    Period: 60*5,
    Statistic: 'Average',
```

```
  Threshold: 5.0,
  ActionsEnabled: true,
  AlarmActions:
['arn:aws:automate:'+process.env.REGION+':ec2:terminate'],
  AlarmDescription: 'Termination alarm for CPU below 5%',
  Dimensions: [
    {
      Name: 'InstanceId',
      Value: instanceId
    },
  ],
  Unit: 'Percent'
};

var alarmData = await cw.putMetricAlarm(alarmParams).promise();
console.log('Alarm created: ', alarmData);

// Enable action on alarm
var paramsEnableAlarmAction = {
  AlarmNames: [alarmParams.AlarmName]
};

var actionData = await
cw.enableAlarmActions(paramsEnableAlarmAction).promise();
console.log("Alarm action enabled", actionData);

return {
  statusCode: 200,
  body: JSON.stringify({result: "Instance created"}),
};
```

```
  } catch (err) {
    console.error('Error creating instance: ', err);
    return {
      statusCode: 500,
      body: JSON.stringify({error: "Error creating instance"}),
    };
  }
};
```

Після завершення повного робочого процесу навчання ми готові створити частину передбачення / висновку. Основною метою висновку буде завантажити модель, завантажити її в TensorFlow.js, а потім зробити прогнози на основі набору входів, що подаються на неї тригером HTTP. Функція очікує, що вхід представляє собою масив об'єктів з ключами, що представляють потрібні поля введення моделі.

У браузерній версії TensorFlow.js використовується збірка, яка не є стандартною для Node.js. Для вирішення цього питання ми встановимо node-fetch і використовуватимемо його замість цього в усьому світі.

Далі модель потрібно завантажити. Ще раз нам доведеться подолати той факт, що ми використовуємо версію браузера, яка не очікує доступу до стандартної локальної файлової системи. Ми можемо витягти необхідний модуль з tfjs-вузла в наш проект, але замість цього в цьому прикладі ми будемо використовувати опцію прямого завантаження HTTP у loadLayersModel. Однак, оскільки наше bucket для S3 не відкрито для світу, нам потрібно визначити, як дозволити цей доступ. Для доступу HTTP до S3 за допомогою підписаного URL-адреси є розумним варіантом, але на кроці завантаження TensorFlow насправді робить дві речі:

- Завантаження model.json - тут ми можемо передати підписаний URL
- Використання кореня URL для завантаження топології моделі - Підписаний URL з кроку 1 більше не працюватиме!

Щоб вирішити це, ми використовуватимемо окремий проксі-сервер, який буде приймати кожен запит і перенаправляти його на відповідний підписаний URL. Оновіть s3proxy, щоб підтримати це, як показано нижче:

```
'use strict';

const AWS = require('aws-sdk');
AWS.config.update({region: process.env.REGION});
const s3 = new AWS.S3();

module.exports.s3proxy = async event => {
  var filename = 'models/latest/model.json';
  console.log('Final filename: ', filename);

  var signedUrl = s3.getSignedUrl('getObject', {
      Bucket: process.env.BUCKET,
      Key: filename,
      Expires: 100 // seconds
  });

  console.log('Signed url: ', signedUrl);

  return {
    statusCode: 302,
    headers: {
      Location: signedUrl
    }
  };
```

Повернувшись до функції висновку, із завантаженою моделлю ми перетворимо вхід у 2D тензор і виконаємо прогнозування. arraySync перетворить результати в стандартні поплавці, при цьому кожен набір входів

переводиться на набір прогнозів у вихідних розмірах. Це передбачення перетворюється на просте відображення міток шляхом знаходження максимуму, а потім повертається в новому об'єкті JSON.

```
'use strict';
const tf = require('@tensorflow/tfjs');
// See https://github.com/tensorflow/tfjs/issues/2029
const nodeFetch = require('node-fetch');
global.fetch = nodeFetch;
const AWS = require('aws-sdk');
AWS.config.update({region: process.env.REGION});
const ddb = new AWS.DynamoDB({apiVersion: '2012-08-10'});
module.exports.infer = async event => {
  // Expecting to be array of json objects with keys matching inputs
  var data = event.body
  if (typeof data == 'string') {
    data = JSON.parse(data)
  }
  var input = [];
  for (var entry in data) {
    input.push([entry.temp, entry.clo, entry.rh, entry.wind]);
  }
  // Download and load the model
  var url = process.env.API_URL + '/s3proxy/latest/model.json';
  console.log('s3proxy url: ', url);
  const model = await tf.loadLayersModel(url);

  // Run inference with predict()
  var predictResult = model.predict(tf.tensor2d(input));
  var resultArray = predictResult.arraySync();
  // Construct output
```

```
  var output = [];
  for (var entry in resultArray) {
    var maxIndex = entry.indexOf(Math.max(...entry));
    var simpleResult = null;
    switch (maxIndex) {
      case 0:
        simpleResult = 'cold';
        break;
      case 1:
        simpleResult = 'ok';
        break;
      case 2:
        simpleResult = 'warm';
        break;
    }

    output.push({
      simpleResult: simpleResult,
      rawResult: entry,
    });
  }

  return {
    statusCode: 200,
    body: JSON.stringify(output),
  };

};
```

### 3.3.3. Тестування повного робочого процесу

Якщо ви створили тестову функцію, ви можете встановити завдання cron для виконання через визначений інтервал, який буде імітувати реальний трафік. Для цього нам потрібно додати триггер подій CloudWatch (за замовчуванням вимкнено) до нашої конфігурації serverless.yml:

```
  test:
handler:                                                    js/test.test
events:
-                                                              schedule:
rate:                       rate(2                              minutes)
enabled: false
```

Ручне включення тригера може бути трохи заплутаним, оскільки інтерфейс Lambda покаже його як "увімкнено", але вам потрібно перейти до CloudWatch, щоб активувати основну подію:

Проблема полягає в тому, що в інтерфейсі Lambda (який можна ввімкнути / вимкнути) є як AWS :: Events :: Правило (яке можна включити / вимкнути), так і тригер. Інтерфейс Lambda відображає стан тригера, який увімкнено. Однак насправді ми не можемо торкнутися цього за допомогою CloudFormation. AWS :: Events :: Правило встановлено у вимкненому стані, що ми встановлюємо за допомогою CloudFormation. Якщо або тригер, або правило вимкнено, воно не запустить вашу лямбда. Для сторони прогнозування ми можемо протестувати вручну, як і раніше, або розширити нашу стратегію тестових функцій, щоб також включити висновок.

Зважаючи на те, що це прототип, є багато аспектів, які слід врахувати, перш ніж розгорнути в реальному виробничому середовищі:

- Інтеграція домену для стійких кінцевих точок API (див. Плагін без сервера-домена).
- Входи HTTP події повинні бути перевірені і включати обробку помилок.
- Функції зігрівання можуть бути додані до кінцевих точок, орієнтованих на клієнта, щоб обмежити тривалий час виклику під час холодного запуску.

- Дозволи на ресурси IAM слід посилити. Інкапсуляція цього середовища в VPC була б хорошим варіантом, а також надає альтернативу проксі для дозволу HTTP доступу до S3.
- Тригер потокового передачі DynamoDB є відносно рудиментарним і може виявитися занадто агресивним у середовищі з великим об'ємом. Більш надійним рішенням може бути додавання нових подій у файл та обчислення нових подій окремо, що також може полегшити сканування всієї таблиці для кожного тренувального циклу.
- Якщо екземпляр EC2 припиняється після кожного запуску, з часом невикористані сигнали тривоги повинні бути очищені. Якщо використовується альтернативна схема зупинки / запуску одного екземпляра, сигнал тривоги також може бути використаний повторно.
- Для захисту у виробництві слід застосовувати порогові значення під час навчальної роботи, щоб погано працюючі моделі не були введені для прогнозування.

# ВИСНОВКИ

В даній роботі було максимально детально описано хмарні технології, без серверну архітектуру з її сучасними концепціями використання. Ось що пропонує архітектура без сервера - вона побудована на публічних хмарних сервісах нового покоління, які автоматично масштабують та заряджають лише при використанні. Коли масштаб, планування потужностей та управління витратами автоматизовані, результатом цього є програмне забезпечення, яке простіше побудувати, обслуговувати та часто до 99% дешевше.

Безсерверні архітектури є новими, тому потребують зміни в тому, як ми раніше думали про архітектури та робочі процеси. Було розроблено додаток для машинного навчання TensorFlow на базі автоматичної підготовки EC2. Підключичили та розробири алгоритм програмування люмбда функції для навчання машини. Таким чином, вона повинна включати в себе необхідне програмне забезпечення та інструменти для підтримки процесу розробки. Детально обговорюється технічна реалізація, структура проекту та методи кодування. Також пояснюється програмна залежність та бібліотека, необхідні для прототипу програми. Цілі використання, функціональні можливості, а також установка розглядаються.

Згодом четверте питання - використання платформи безперервної інтеграції (CI) для автоматичного створення та розгортання програми на AWS. В цілому, моделювання проекту спрямоване на створення веб-сервісу, його розгортання в середовищі хмарних технологій AWS, підтримання та моніторинг його роботи. Результатом є успішно створений додаток.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. A. Barros and M. Dumas, "The Rise of Web Service Ecosystems," IT Professional, vol. 8, no. 5, pp. 31-37, 2006.
2. Zappa: Serverless python web services. URL https://github.com/Miserlou/Zappa. Online; accessed December 1, 2016
3. Yan, M., Castro, P., Cheng, P., Ishakian, V.: Building a chatbot with serverless computing. In: First International Workshop on Mashups of Things, MOTA '16 (colocated with Middleware) (2016)
4. D. Fensel, H. Lausen, A. Polleres, J. d. Bruijn, M. Stollberg, D. Roman and J. Domingue, Enabling Semantic Web Services: The Web Service Modeling Ontology, Berlin: Springer, 2007.
5. Sbarski, P., Kroonenburg, S.: Serverless Architectures on AWS With Examples Using AWS Lambda. in preparation (2016). URL https://www.manning.com/books/serverless-architectures-on-aws
6. Parse Cloud Code Getting Started. URL https://parseplatform.github.io/docs/cloudcode/guide/. Online; accessed December 1, 2016
7. Google Apps Marketplace. URL https://developers.google.com/appsmarketplace/. Online; accessed December 1, 2016
8. Jira. URL https://www.openstack.org. Online; accessed December 5, 2016
9. Openlambda. URL https://open-lambda.org/. Online; accessed December 1, 2016
10. Jira. URL https://www.atlassian.com/software/jira. Online; accessed December 5, 2016
11. LeverOS. https://github.com/leveros/leveros. URL https://github.com/leveros/leveros. Online; accessed December 5, 2016
12. ServerlessConf, "ServerlessConf," 18 October 2016. Available: http://serverlessconf.io/ .

13. Openlambda. URL https://open-lambda.org/. Online; accessed December 1, 2016
14. NGINX Announces Results of 2016 Future of Application Development and Delivery Survey. URL https://www.nginx.com/press/nginx-announces-results-of2016-future-of-application-development-and-delivery-survey/. Online; accessed December 5, 2016
15. P. Brittenham, F. Cubera, D. Ehnebuske and S. Graham, "Understanding WSDL in a UDDI Registry," IBM, New York, 2001.

16. Azure functions. URL https://functions.azure.com/. Online; accessed December 1, 2016
17. Sharable, Open Source Workers for Scalable Processing. URL https://www.iron.io/
18. Introducing Lambda support on Iron.io. URL https://www.iron.io/introducingaws-lambda-support/. Online; accessed December 1, 2016
19. Cloud Foundry and Iron.io Deliver Serverless. URL https://www.iron.io/cloudfoundry-and-ironio-deliver-serverless/. Online; accessed December 1, 2016
20. Openwhisk. URL https://github.com/openwhisk/openwhisk. Online; accessed December 1, 2016
21. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016. (2016). URL https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson
22. Cloud functions. URL https://cloud.google.com/functions/, Online; accessed December 1, 2016
23. Galactic Fog Gestalt Framework. URL http://www.galacticfog.com/. Online; accessed December 1, 2016

24. OpenFog Consortium. URL http://www.openfogconsortium.org/. Online; accessed December 1, 2016
25. Fernandez, O.: Serverless: Patterns of Modern Application Design Using Microservices (Amazon Web Services Edition). in preparation (2016). URL https://leanpub.com/serverless

26. A. Barros and M. Dumas, "The Rise of Web Service Ecosystems," IT Professional, vol. 8, no. 5, pp. 31-37, 2006.
27. Aws lambda. URL https://aws.amazon.com/lambda/. Online; accessed December 1, 2016

28. A. D. Birell and B. J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, vol. 1, no. 2, pp. 39-59, 1984.
29. A. Handy, "Amazon Introduces Lambda, Containers at AWS re:Invent," BZ Media LLC, 14 November 2014. Available: http://sdtimes.com/amazon-introduceslambda-containers/.
30. A. O. Ramirez, "Three-Tier Architecture," Linux Journal, vol. 2000, no. 75es, July 2000.
31. A. R. Hevner, S. T. March, J. Park and S. Ram, "Design Science in Information Systems Research," Management Information Systems Quarterly, vol. 28, no. 1, pp. 75-105, March 2004.
32. A. Skonnard, "Understanding SOAP," March 2003. Available: https://msdn.microsoft.com/en-us/library/ms995800.aspx.
33. A. Watson, "OMG (Object Management Group) Architecture and CORBA (Common Object Request Broker Architecture) Specification," in IEE Colloquium on Distributed Object Management, London, 1994.
34. Amazon Web Services, Inc., "Amazon Web Services: Overview of Security Processes," Amazon Web Services, Inc., Seattle, 2016
35. Amazon Web Services, Inc., "AWS Lambda: Developer Guide," Amazon Web Services, Inc., Seattle, 2017.

36. Amazon Web Services, Inc., "AWS Lambda: How It Works," 19 11 2017. Available: http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html .
37. Amazon Web Services, Inc., "Overview of Amazon Web Services," Amazon Web Services, Inc., Seattle, 2017.
38. Atlassian, "Atlassian Company," Atlassian, 6 August 2017. Available: https://www.atlassian.com/company.
39. Cloud functions. URL https://cloud.google.com/functions/. Online; accessed December 1, 2016
40. Atlassian, "Bitbucket," Atlassian, 6 August 2017. Available: https://www.atlassian.com/software/bitbucket.
41. Openwhisk. URL https://github.com/openwhisk/openwhisk. Online; accessed December 1, 2016
42. BlazeMeter, "BlazeMeter," 6 December 2011. Available: https://www.blazemeter.com/.
43. BlazeMeter, "Understanding Your Reports - Part 4: How to Read Your Load Testing Reports on BlazeMeter," 19 September 2016. Available: https://www.blazemeter.com/blog/understanding-your-reports-part-4-how-read-yourload-testing-reports-blazemeter.
44. D. Cvetkovic, M. Mijatovic and B. Medic, "Web Service Model for Distance Learning Using Cloud Computing Technologies," in 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, 2017.
45. D. Krafzig, K. Banke and D. Slama, Enterprise SOA: Service-Oriented Architecture Best Practices, New Jersey: Prentice Hall PTR, 2004.
46. E. J. Weyuker and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study," IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1147-1156, December 2000.
47. Azure functions. URL https://functions.azure.com/. Online; accessed December 1, 2016

48. F. Abidi and V. Singh, "Cloud Servers vs. Dedicated Servers - A Survey," in Innovation and Technology in Education (MITE), 2013 IEEE International Conference in MOOC, Jaipur, 2013.
49. R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, Irvine, 2000.
50. S. Helal, J. Hammer, J. Zhang and A. Khushraj, "A Three-Tier Architecture for Ubiquitous Data Access," in Proceedings ACS/IEEE International Conference on Computer Systems and Applications, Beirut, 2001.
51. S. T. March and G. F. Smith, "Design and Natural Science Research on Information Technology," Decision Support Systems, vol. 15, no. 4, pp. 251-266, December 1995.
52. Ž. Stojanov and D. Dobrilović, "An Approach to Integration of Maintenance Services in Educational Web Portal," in IEEE 8th International Symposium on Intelligent Systems and Informatics, Subotica, 2010.
53. S. Vinoski, "Demystifying RESTful Data Coupling," IEEE Internet Computing, vol. 12, no. 2, pp. 87-90, 2008.
54. Serverless Inc., "Serverless Framework," Serverless Inc., 12 October 2015. Available: https://serverless.com/framework/ .
55. Z. Shao, H. Jin and D. Zhang, "A Performance Study of Web Server Based on Hardware-Assisted Virtual Machine," in Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference, Rabat, 2009.