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

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

«На правах рукопису»
УДК 004.75

До захисту допущено:

Завідувач кафедри

_____ Ігор ПАРХОМЕЙ

«___»_____2020 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інформаційне забезпечення робототехнічних систем»

зі спеціальності 126 «Інформаційні системи та технології»

на тему: «Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів»

Виконав:
студент ІІ курсу, групи ІК-91мп
Старовойтенко Олексій Володимирович _____

Керівник:
Доцент кафедри ТК, к.т.н., доцент,
Резніков Сергій Анатолійович _____

Консультант з нормоконтролю:
доцент, к.т.н., доц.,
Пасько Віктор Петрович _____

Рецензент:
доцент, к.т.н., доц.,
Баклан Ігор Всеволодович _____

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

Київ – 2020 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інформаційне забезпечення робототехнічних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Ігор ПАРХОМЕЙ

«___»_____2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Старовойтенку Олексію Володимировичу

1. Тема дисертації «Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів», науковий керівник дисертації Рєзніков Сергій Анатолійович, к.т.н., доцент затверджені наказом по університету від « 26 » жовтня 2020р. № 3132-с

2. Термін подання студентом дисертації __18.11.2020_____

3. Об'єкт дослідження – синхронізація даних групи мобільних роботів в реальному часі за допомогою хмарних технологій

4. Вихідні дані – сигнали, що надходять від мобільних роботів в уніфікованому форматі

5. Перелік завдань, які потрібно розробити – аналіз існуючих хмарних рішень, вибір необхідних технологій, проектування архітектури синхронізації даних, розробка тестових сервісів з метою моделювання роботи спроектованої архітектури синхронізації даних мобільних роботів в реальному часі.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – шість плакатів

7. Орієнтовний перелік публікацій – дві публікації

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Перевірка на співпадіння	доцент Лісовиченко О.І.		
Нормоконтроль	доцент Пасько В.П.		

9. Дата видачі завдання ___01.09.2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Формування проблематики	02.09.2020 – 08.09.2020	
2	Аналіз проблематики	09.09.2020 – 15.09.2020	
3.	Постановка задачі	16.09.2020 – 22.09.2020	
4.	Вибір технологій до вирішення задач	23.09.2020 – 29.09.2020	
5.	Проектування архітектури системи	30.09.2020 – 06.10.2020	
6.	Розробка ПЗ для апробації архітектури	07.10.2020 – 13.10.2020	
7.	Тестування ПЗ та покращення архітектури	14.10.2020 – 20.10.2020	
8.	Підготовка документації	21.10.2020 – 27.10.2020	
9.	Попередній захист	23.11.2020	
10.	Нормоконтроль	10.12.2020	
11.	Перевірка на співпадіння	09.12.2020	
12.	Захист	21.12.2020	

Студент Олексій Старовойтенко

Науковий керівник Сергій Рєзніков

АНОТАЦІЯ

У магістерській дисертації розглянуто проблему синхронізації великих об’ємів даних в реальному часі, що надходять від групи мобільних роботів. В якості рішення обрано хмарні технології.

У розділі аналізу проблематики та постановки задачі визначено основні проблеми, що виникають при синхронізації даних такі як затримки в обробці, відмови в роботі сервісів, втрата даних та інші. Поставлено задачу розробити рішення, яке дозволить гнучко масштабувати систему, здійснювати відтворення втраченої інформації під час обробки та мінімізує затримки при надсилання сигналів керування.

У розділі вибору технологій проаналізовано можливі технологічні підходи та сервіси хмарного провайдера Amazon Web Services (AWS). Визначено перелік сервісів, що є основними компонентами в архітектурі: AWS IoT, AWS DynamoDB, AWS Kinesis та інші.

У розділі проектування архітектури розроблено архітектурні концепції, що вирішують поставлено задачу. Описано сервіси агрегацій даних, що є складовими компонентами спроектованої архітектури.

У розділі маркетингового аналізу стартап-проекту здійснено аналіз поточної ситуацію на ринку, створено стратегії та маркетинговий плани для впровадження рішення.

Ключові слова: синхронізація, дані в реальному часі, мобільні роботи, хмарні сервіси, черга повідомлень. AWS, DynamoDB, Kinesis, AWS SQS.

Розмір пояснювальної записки – 136 аркушів, містить 57 ілюстрацій, 26 таблиць, 8 додатків.

ABSTRACT

The master's thesis deals with the problem of synchronizing large amounts of real-time data from a group of mobile robots. Cloud technologies were chosen as a solution.

In the section of the analysis of problems and statement of the problem the basic problems which arise at data synchronization such as delays in processing, failures in work of services, data loss and others are defined. The task is to develop a solution that will flexibly scale the system, reproduce lost information during processing and minimize delays in sending control signals.

The technology selection section analyzes possible technology approaches and services from the cloud provider Amazon Web Services (AWS). The list of services that are the main components in the architecture is defined: AWS IoT, AWS DynamoDB, AWS Kinesis and others.

In the section of architecture design, architectural concepts are developed that solve the problem. Data aggregation services that are components of the designed architecture are described.

In the section of marketing analysis of the startup project the analysis of a current situation in the market is carried out, strategies and marketing plans for implementation of the decision are created.

Keywords: synchronization, real-time data, mobile robots, cloud services, message queue. AWS, DynamoDB, Kinesis, AWS SQS..

Explanatory note size – 136 pages, contains 57 illustrations, 26 tables, 8 applications.

Пояснювальна записка
до магістерської дисертації

на тему: Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів

Київ – 2020 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних;

ABAC (attribute based access control) – керування доступів на основі атрибутів;

ACL (access control list) – список розмежувань доступів;

ACO (access control oriented) – принцип побудови безпеки, орієнтований на доступ;

API (application programming interface) – програмний інтерфейс застосунку;

AWS (Amazon Web Services) – веб-сервіси хмарного провайдера Amazon;

DBaaS (database as a service) – база даних як сервіс;

JSON (JavaScript Object Notation) – текстовий формат обміну даними;

IaaS (infrastructure as a service) – інфраструктура як сервіс;

IAM (identity and access management) – механізм керування обліковими даними;

IoT (internet of things) – інтернет речей;

MQTT (message queuing telemetry transport) – прокол обміну повідомленнями за принципом видавець-підписник;

NoSQL (not only SQL) – база даних з механізмом зберігання даних відміним від класичних реляційних баз;

PaaS (platform as a service) – платформа як сервіс;

RBAC (role based access control) – керування доступів на основі ролей;

RDMS (relational database management system) – реляційна система керування базами даних;

REST (representational state transfer) – підхід до мережевих протоколів;

RFID (radio frequency identification) – радіочастотна ідентифікація;

ROS (robot operating system) – вільна операційна система для програмування роботів;

SaaS (software as a service) – програмне забезпечення як сервіс;

SLAM (simultaneous localization and mapping) – одначасна локалізація та картографування;

SOAP (simple object access protocol) – протокол обміну структурованими повідомленнями в сервісній архітектурі;

QoS (quality of service) – якість обслуговування;

VM (virtual machine) – віртуальна машина;

VO (virtual object) – віртуальний об’єкт;

VPC (virtual private cloud) – віртуальна приватна мережа;

ВСТУП

Внаслідок збільшення кількості господарств різних галузей включаючи сферу послугу виникає необхідність в автоматизації/роботизації з метою зменшення рутинної участі працівників та досягнення ефективного стабільного виробництва. Важливу ніше в роботизації займають мобільні роботи-помічники, наприклад, дозволяють обслуговувати тепличне аграрне господарство чи виконувати небезпечні для здоров’я людини роботи. Крім мобільних роботів важливу нішу займають пристрої, що інтегруються в існуючі системи з метою збору аналітики чи віддаленого керування. Наприклад, пристрій збору даних з двигуна вантажівки, який дозволяє забезпечувати аналітику даних та впливати на управління. Зі збільшенням кількості мобільних роботів виникає гостра проблема синхронізації даних, адже необхідно передавати дані про локацію, навколишні умови, в результаті чого можна отримати повну картину щодо заданого середовища та виконати вплив на інші пристрої. Синхронізація даних вимагає єдиного джерела, яке забезпечує зчитування інформації з усіх пристроїв та оброблює наданий об’єм даних.

В такому процесі виникає ряд проблем та вимог: забезпечення високої пропускної здатності, консистентність та перезестивність інформації, уніфікація зчитування джерел даних, мінімальні затримки в обчисленні та аналізі. В системах реального часу затримка в обробці інформації може бути критичним фактором, адже некоректна обробка сигналів призводить до збою системи та вимагає додаткових повторних обчислень починаючи з точки збереження. Щоб виконати повторне обчислення необхідно зберігати вхідні дані після кожного етапу обробки, адже лише в цьому випадку можливе коректне відновлення роботоздатності системи. Звичайно деякі операції

виконуються безпесередньо локально роботом, проте комплексні показники різних роботів та прийняття рішень на основі цих показників вимагають централізованої ефективної обробки.

Продовж довго часу за допомогою класичної серверної обробки успішно досягались задані вимоги, тобто обчислювальної потужності серверів достатньо для здійснення операцій. А в разі збільшення кількості клієнтів – проводили горизонтальне чи вертикальне масштабування, впроваджуючи підходи до реплікацій даних та узгодження роботи програмного забезпечення. Проте наразі актуальними є хмарні рішення, що мають ряд переваг порівняна з класичними серверами, що є особливо важливим в тому числі для синхронізації даних в реальному часі. Так архітектура систем в контексті хмарних рішень має ряд таких переваг як велика кількість сервісів націлених для вирішення тієї чи іншої задачі, проста горизонтальна та вертикальна масштабованість сервісів за лічені хвилини без необхідності додаткових міграцій даних, плата за користування конкретними ресурсами а не за певний передбачений об’єм, доступність з різних частин світу за рахунок розміщення додаткових датацентрів хмарного провайдера, гнучка система конфігурування безпеки та наявність єдиної екосистеми. Варто зазначити, що наразі об’єм даних невпинно зростає, багато організацій здійснюють рефакторинг власних систем в пошуках рішення, яке дозволить на найближчий час забезпечити зростаючі об’єми даних.

Отже, проектування архітектури синхронізації даних в реальному часі на прикладі даних мобільних роботів за допомогою хмарної архітектури є актуальною задачею, що дозволить малим та середнім господарством впроваджувати роботизацію діяльності в умовах обмежених ресурсів та можливості стрімкого зростання.

РОЗДІЛ 1. АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ СИНХРОНІЗАЦІЇ ДАНИХ

1.1. Аналіз задачі синхронізації даних за допомогою хмарної робототехніки

На основі розвитку в галузі хмарної робототехніки, особливо останніх чотири роки хмарна робототехніка повинна мати більш широке визначення. Мотивація для цього більш широкого визначення та підхід до вирішення цієї проблеми випливає з наступних двох ключових знань/реалізацій:

Люди та процеси над технологією. Важливо дозволити мобільним роботам здійснювати максимальну автономію щодо прийняття рішень без стороннього втручання людини.

Синхронність роботи групи мобільних роботів. Необхідно синхронізувати дані датчиків для розуміння навколишнього середовища, виконавчих механізмів для виконання корисної фізичної роботи. Крім того, значна частина випадків використання включає сценарій з кількома роботами, тобто кілька наборів обчислень, датчиків та виконавчих механізмів, що працюють у синхронізації.

Варто виділити такі основні проблеми, що існують в робототехніці:

- Технічна складність: надзвичайно складний технічний характер робототехнічних рішень.
- Великі капітальні витрати: значний обсяг капіталу необхідний для впровадження рішень з робототехніки.
- Жорсткий дизайн: рішення робототехніки створені з цілком конкретною метою, що робить їх негнучкими до змін навколишнього середовища та процесів.
- Обмежений доступ: більшість рішень забезпечують лише фізичний доступ на місці, є оперативною та масштабованою проблемою.

- Запатентовані інтерфейси: нестандартизовані API між програмними / апаратними компонентами створюють фрагментацію та обмежують інновації.

До появи хмарних обчислень ціла команда експертів повинна встановити, налаштувати, протестувати, запустити, захистити та оновити сервери та додатки. Навіть найбільші компанії з найкращими IT-департаментами боролися на цьому шляху, а малий та середній бізнес не мав шансів. З метою полегшення запуску робототехнічних рішень розроблено хмарну робототехніку.

Хмарна робототехніка повинна вирішувати ряд проблем в області синхронізації та обробки даних:

- Самообслуговування на вимогу: масштабування здійснюється автоматично залежно від об’єму необхідних ресурсів.
- Спільний ресурс: наявність спільного пулу ресурсів.
- Швидка еластичність: вертикальне та горизонтальне масштабування залежно від навантажень.
- Повсюдний доступ: ресурси доступні через мережу Інтернет з будь-якого місця.
- Вимірювана послуга: використання ресурсів прозоро відслідковується (наприклад, зберігання, обробка, пропускна здатність та активні облікові записи користувачів), що дає можливість вимірюваних послуг (так звана "плата за використання"), оптимізації ресурсів та прогнозне планування.

Самообслуговування на замовлення.

Кінцеві користувачі забезпечують і відміняють роботи роботів за потреби. З огляду на неоднорідність роботизованих апаратних та програмних компонентів, етап підготовки може також включати етап складання, коли користувач складає необхідну конфігурацію з каталогу апаратних та

програмних компонентів без потреби експерта. Це створить відкритий ринок, що стимулює інновації та конкурентоспроможні ціни.[5]

Спільні ресурси.

Кінцеві користувачі не повинні володіти роботами, оскільки установи з великим капіталом будуть утримувати обладнання. Наявність загального керування всіма блоками певного типу обладнання призведе до економії масштабу апаратного забезпечення та зменшить витрати на технічне обслуговування.

Повсюдний доступ.

Кінцеві користувачі можуть отримати доступ до своїх роботів / парків роботів з будь-якого місця – навіть використовуючи тонкі клієнти – як браузер. Наприклад, дозволяють кінцевому користувачеві дистанційно запускати та керувати повітряним роботом за допомогою веб-браузера.

Виміряне обслуговування.

Використання ресурсів прозоро вимірюється для всіх кінцевих користувачів, постачальників рішень, розробників обладнання, розробників програмного забезпечення – наприклад, скільки часу працював робот, скільки разів викликали API глобального планування, яка пропускна здатність / пам’ять використовується для журналів/метрик, скільки обчислень витратив глобальний оптимізатор карт? Це вимірюване використання використано для забезпечення плати за використання, оптимізації та прогнозного планування, тим самим оптимізуючи витрати.

Хмарна архітектура дозволяє вирішити вищевказані проблеми наступним чином:

- Самообслуговування: кінцевий користувач не потребує експерта для налаштування та роботи з робототехнічним рішенням.
- Спільні ресурси: кінцевий користувач не володіє роботами. Натомість роботи здаються в оренду щомісяця.

- Еластичність: повторна конфігурація, щоб відповідати бізнес-процесу та забезпечувати масштабування вгору або вниз із мінімальними зусиллями.
- Повсюдний доступ: стан роботизованої системи та її підкомпонентів може спостерігатись та контролюватись з будь-якого місця та особами, що мають відповідний дозвіл.
- Відкриті інтерфейси: інтерфейси чітко визначені, що полегшує інтеграцію сторонніх розробників. Операції є прозорими для відповідних зацікавлених сторін. Наприклад, компанія, що займається технічним обслуговуванням, має повний доступ до експлуатаційних даних, обладнання.

1.2. Аналіз існуючих архітектурних рішень

Через велику різноманітність можливих послуг та різних аспектів обчислень, які можна надати за допомогою хмарних технологій, а також неможливості створити одну універсальну систему, хмарні моделі можна класифікувати на такі групи:

- SaaS – програма як сервіс;
- PaaS – платформа як сервіс;
- IaaS – інфраструктура як сервіс.

Архітектура SaaS надає користувачам доступ лише до хмарної інфраструктури. Клієнти не можуть виконувати будь-які зміни на платформі. Типовим прикладом хмарної служби SaaS є веб-поштовий клієнт. Друга архітектура, PaaS, дозволяє користувачеві отримувати доступ до програм та засобів розробки. Іноді постачальник послуги вимагає додаткових програм, встановлених на машині клієнта. Третій тип IaaS, надає користувачеві простір для зберігання, екземпляри віртуального сервера та інші ресурси. Типово, що для доступу до даних потрібна спеціальна програма.

Хоча в основному всі рішення знаходиться на початковій стадії розробки, можна знайти кілька проектів хмарної робототехніки з усього світу. Популярні ініціативи, як приватні, так і державні, включають такі:

- проект DAvinCi (розподілені агенти з колективним інтелектом);
- проект RoboEarth;
- проект Rapyuta.

DAvinCi розроблений Інститутом зберігання даних A * STAR Сінгапуру. Мета цього проекту – представити переваги використання хмарного підходу в важкому обчислювальному процесі локалізації та картографування великих площ. Робототехнічні агенти оснащені основними датчиками, а також різними датчиками локалізації. До них належать LIDARS, камери для збору зображень та інші точні датчики локалізації. Машини отримали доступ до популярних алгоритмів локалізації та відображення, що зберігаються всередині кластера Hadoop. На рис. 1. висвітлено основні елементи архітектури DAvinCi. На одному кінці встановлені роботи з програмним забезпеченням ROS, яким належить наступна функція – збирати та пересилати дані за допомогою Wi-Fi-з'єднання. Ця інформація завантажується в центральний контролер, де проводяться подальші обчислення. ROS Framework використовується у зв'язку між роботами та хмарним двигуном. Усередині кластера Hadoop зберігаються та виконуються дані, зібрані роботами та різними алгоритмами обчислень та аналізу даних. Сервер DAvinCi можна описати як постачальника послуг для роботів. Основна його робота – прив'язка роботів та кластера Hadoop за допомогою розподіленої файлової системи ROS та Hadoop.

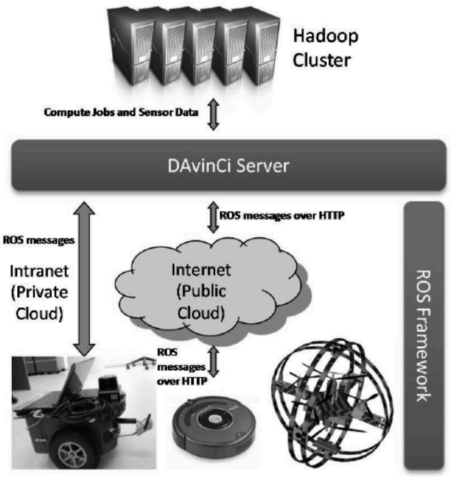


Рисунок 1. Огляд основних компонентів архітектури системи DAvinCi

Метою проекту «Захоплення роботів у хмарі» є створення хмарної системи для роботів для обміну даними про навчання та маркування для навчання. Розроблену систему можна розділити на дві фази: офлайн та онлайн. Під час першої фази фотографії об'єктів отримуються для того, щоб навчити сервер розпізнавання. Крім того, створюється 3D-модель САПР, а потім використовується для створення та аналізу набору схоплень для кожного елемента. Онлайн-фаза починається з того, що робот сфотографував об’єкт і надіслав його через мережу на згаданий вище сервер. Якщо екземпляр об'єкта знайдено на сервері, повертається інформація про[елемент. Робот використовує ці дані для виконання оцінки пози та виконання захоплення. Потім результати цієї операції зберігаються для довідки всередині хмари.

RoboEarth описується його творцями як Всесвітня павутина для роботів. Це онлайн-база даних, що містить інформацію про об’єкти, карти, компоненти програмного забезпечення та знання про завдання. Основна мета цього проекту – дозволити роботам вчитися один на одному досвіду та ділитися власними отриманими даними. Зрештою, це призведе не лише до розвитку робототехніки, але і до більш складної взаємодії людини і робота.

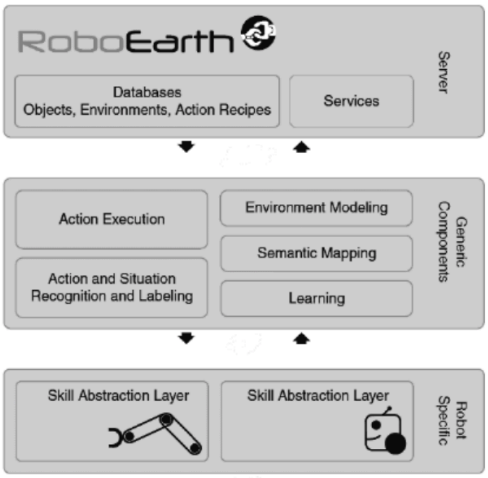


Рисунок 2. Огляд архітектури системи RoboEarth

Сервер зберігає таку інформацію, як об'єктні моделі, семантичні карти та інструкції до дії, а також інші послуги. Загальні компоненти – це апаратно-незалежні рівні, яким дано завдання інтерпретувати інструкції щодо дії, а також розширювати зондування, міркування і т. д. Спеціальний рівень робота залежний від апаратного забезпечення та забезпечує інтерфейс для специфічних функцій робота. Платформа RoboEarth не є уніфікованою програмою, але складається з шести програмних компонентів, які можна використовувати самостійно, залежно від потреб: Rapyuta (RoboEarth Cloud Engine), RoboEarth DB, KnowRob , WIRE та детектор об’єктів. Ці програми в значній мірі інтегровані з операційною системою Robot, більш відомою як ROS.[2]

Rapyuta – RoboEarth Cloud Engine – пакет з відкритим вихідним кодом, який намагається забезпечити простий інструмент для вирішення проблем робототехніки.

Основна ідея хмарної робототехніки полягає в тому, що робот зазвичай взаємодіє з навколишнім середовищем за допомогою своїх датчиків і виконавчих механізмів. Фактично це означає, що дані, отримані від датчиків, повинні бути оброблені та використані для прийняття рішень (як зворотного зв'язку) щодо наступної дії робота. Для завдань, які не вимагають роботи в режимі реального часу, обробку можна перенести в хмару. Це зменшує

обчислювальну потужність, необхідну роботу, що може збільшити тривалість його мобільної роботи (без підзарядки), а також вартість робота. Більшість робототехнічних завдань високого рівня, таких як картографування та планування руху, не вимагають роботи в режимі реального часу.

Отже, створення карти складається із збереження опису навколишнього середовища, щоб робот міг в майбутньому знаходити своє місцезнаходження на карті. Карта також використовується для планування траєкторії або вибору позиції для захоплення об'єкта. Крім того, обидва ці завдання можна виконати заздалегідь, і тому вони не можуть бути прив'язані до обмежень у реальному часі.

Перевага хмарних обчислень також може бути роботою спільних роботів. Наприклад, спільного використання даних для створення спільної карти або спільного планування завдання можна легше досягти за допомогою централізованої архітектури з потужними обчислювальними можливостями.

RoboEarth дозволяє роботам обмінюватися знаннями через централізовану базу знань. Такий підхід дозволяє уникнути дублювання. Однак для реалізації повного потенціалу цього підходу потрібно не тільки централізоване зберігання знань, але й централізоване міркування, що є хмарними обчисленнями для роботів.

Проект Rapyuta прагне заповнити цю прогалину, надаючи відсутню частину хмарної інфраструктури для роботів. Rapyuta пропонує хмарну платформу, на якій роботи можуть створювати комп'ютерне середовище для важких обчислень. Ці обчислювальні середовища можуть виступати або як окрема хмара для одного робота, або спільно використовуватися кількома роботами. Крім того, обчислювальні середовища мають високошвидкісний доступ до бази даних RoboEarth, завдяки чому процеси можуть швидко отримати доступ до знань.

Спрощено, рішення Rapyuta виглядає так:

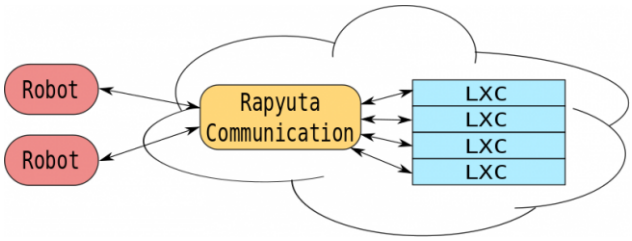


Рисунок 3. Огляд архітектури системи Rapyuta

Rapyuta подібний до Google App Engine, а також визначає протокол для зв'язку між програмами в хмарі. Однак, на відміну від веб-моделі Google, Rapyuta представляє додатковий рівень.

Це означає, що робот не спілкується безпосередньо з хмарою додатків, а використовує Rapyuta як посередника. Цей підхід пропонує більшу гнучкість і контроль над системою. Наприклад, Rapyuta може використовуватися для передачі зв'язку між роботами без створення комп’ютерного середовища. Крім того, необов’язковий рівень Rapyuta дозволяє запускати безліч процесів за допомогою сокетів для зв’язку між собою. Для роботи з Rapyuta сторона робота використовує стандартні протоколи, що дозволяють доступ до Інтернету через веб-браузер (rosbridge та ros.js). Rapyuta використовує WebSockets для постійних з'єднань та JSON (JavaScript Object Notation) для повідомлень.

Однак протоколи Rapyuta та Rosbridge несумісні, оскільки клієнт з Rapyuta може взаємодіяти з більш ніж одним середовищем ROS, що призводить до більш складного протоколу.

На стороні хмарних додатків використовується система обміну повідомленнями, визначена Робочою операційною системою (ROS). Це дозволяє розгортати та запускати більшість із 3000+ існуючих ROS-пакетів із мінімальними змінами в RoboEarth Cloud Engine.

Для відокремлення ROS-вузлів для різних користувачів створені спеціальні комп’ютерні середовища для реалізації хмарних додатків. Таким чином, один робот може мати кілька обчислювальних середовищ, а одне обчислювальне середовище може використовуватися багатьма роботами.

Контейнери Linux (LXC) використовуються для реалізації обчислювального середовища, за допомогою якого можливо надійно контролювати всі операційні аспекти, такі як зв'язок та використання процесорних ресурсів, оперативної пам'яті та місця для зберігання.

Крім того, Rapyuta дозволяє різним комп’ютерним середовищам спілкуватися за допомогою системи обміну повідомленнями ROS.

Rapyuta написана на Python і є відкритим кодом. Альфа-версія Rapyuta доступна на GitHub під ліцензією Apache 2.0.

Основними недоліками представлених сервісів є:

- Відсутність уніфікованих механізмів інтеграції зі стороннімі рішеннями.
- Обмеженість наданих ресурсів в межах наданого оплаченого плану.
- Відсутність гнучкості при використанні ресурсів, фіксована плата на відміну від плати за використовуваний ресурс.
- Зав’язка на конкретних типах пристроїв та підтримуваних даних.
- Складність побудови синхронізації даних про навколишнє середовище та наявність значних затримок щодо обробки інформації – критична характеристика у випадку обробки даних в реальному часі.

1.3. Використання хмарних SAAS рішень на основі AWS (Amazon Web Services)

Amazon Web Services (AWS) пропонує платформу хмарних обчислень для оренди приватним особам, компаніям та урядам за окрему плату. Віртуальні машини AWS поділяють більшість атрибутів реального комп’ютера,

включаючи апаратні пристрої (процесор, відеокарта, локальна та оперативна пам’ять, жорсткий диск або SSD). додаткова операційна система, мережа, попередньо встановлені програми. Кожна система AWS також віртуалізує консоль вводу-виводу (клавіатура, дисплей та миша), щоб користувачі AWS мали змогу підключатися до своєї системи AWS через браузер. Браузер діє як вікно у віртуальну машину і дозволяє користувачеві входити, налаштовувати та використовувати свої віртуальні системи, як справжній фізичний комп’ютер.[1]

Технологія AWS забезпечена роботою серверних кластерах по всьому світу.

Плата за використання базується на поєднанні використання апаратного забезпечення, операційної системи, програмного забезпечення та мережевих функцій, вибраних користувачем, а також вимог щодо доступності, надмірності, безпеки та додаткових параметрів. Залежно від того, що потрібно і за що платить користувач, вони можуть зарезервувати віртуальну машину (VM), кластер віртуальних машин (кластер VM), фізичний (реальний) комп'ютер (сервер), призначений виключно для використання, або навіть кластери серверів. Amazon прагне керувати та оновлювати програмне та апаратне забезпечення, щоб відповідати необхідним стандартам безпеки. AWS працює в багатьох географічних регіонах, включаючи Канаду, Німеччину, Ірландію, Сінгапур, Токіо, Сідней, Пекін, Лондон тощо.

У 2016 році AWS розгорнула понад 70 служб, що охоплюють широкий спектр, включаючи обчислення та зберігання, мережу, аналітику, мобільні додатки, інструменти для розробників тощо. Найпопулярніші з них – Amazon Elastic Compute Cloud (EC2). та Amazon Simple Storage Service (S3). Більшість послуг не надаються безпосередньо кінцевим користувачам, а надають функціональність через API, які розробники можуть використовувати у своїх додатках. Пропозиції веб-служб Amazon доступні через HTTP з використанням архітектури REST та SOAP.[1]

Amazon просуває AWS як спосіб отримати обчислювальну потужність, яка масштабується швидше і дешевше, ніж створення власного фізичного кластера серверів. Усі послуги оплачуються залежно від використання.

Для керування мобільними роботами слід використовувати рішення AWS IoT, яке пропонує:

- Розширена уніфікована інтеграція зі штучним інтелектом.
- Багаторівнева безпека з можливістю контролю на різних рівнях інтеграції та шифрування даних.
- Можливість інтеграції зі сторонніми послугами.

AWS IoT включає в себе такі сервіси.

Програмне забезпечення для пристроїв:[3]

- FreeRTOS – операційна система для мікроконтролерів, що спрощує програмування, розгортання, забезпечення безпеки, підключення малопотужних пристроїв та керування ними.
- IoT GreenGrass – програмне рішення для безпечного виконання таких задач як локальне обчислення, передача повідомлень, синхронізація та кешування даних, формування висновків на основі алгоритмів машинного навчання.

Сервіси для підключення та управління:

- IoT Core – керування мережевими пристроями та проста взаємодія з хмарними застосунками.
- IoT Device Defender – неперервний моніторинг та аудит конфігурацій, щоб не допускати відхилення від рекомендацій з безпеки.
- IoT Device Management – спрощена безпечна реєстрація, організація, моніторинг підключених мобільних роботів в будь-якому масштабі.

Сервіси аналітики:

- IoT Analytics – аналіз складного аналізу великих об’ємів даних.

- IoT SiteWise – систематизація та аналіз промислових даних в великих масштабах.
- IoT Events – детектування датчиків та реагування на них.
- IoT Things Graph – підключення різних пристроїв.

Розглянемо детальніше AWS IoT Core.

AWS IoT Core – це керований хмарний сервіс, який дозволяє підключеним пристроям просто і безпечно взаємодіяти з хмарними додатками та іншими пристроями. AWS IoT Core підтримує роботу з мільярдами пристроїв і трильйонами повідомлень, дозволяючи надійно і безпечно обробляти і направляти ці повідомлення до адрес AWS і інших пристроїв. З AWS IoT Core додатки можуть постійно відстежувати всі використовувані пристрої і взаємодіяти з ними, навіть коли ці пристрої знаходяться в автономному режимі.

AWS IoT Core спрощує роботу з такими сервісами AWS і Amazon, як AWS Lambda, Amazon S3, Amazon Kinesis, Amazon DynamoDB, Amazon CloudWatch і Alexa Voice Service. Крім того, AWS IoT Core дозволяє додаткової програмної розробки створювати додатки для збору, обробки та аналізу даних, що генеруються підключеними пристроями, а також для виконання дій на основі цих даних без необхідності управління будь-якої інфраструктурою.

Принцип роботи:

Підключення пристроїв та керування ними.

AWS IoT Core дозволяє просто підключати будь-яку кількість пристроїв до хмари або один до одного. AWS IoT Core підтримує протоколи HTTP, WebSockets і спрощений протокол зв'язку MQTT, спеціально спроектований для підтримки нестабільних підключень, скорочення обсягу коду, переданого пристрою, і для роботи в мережах з низькою пропускною здатністю. AWS IoT Core також підтримує інші стандартні і спеціальні протоколи, забезпечуючи взаємодію між пристроями навіть в разі використання ними різних протоколів.

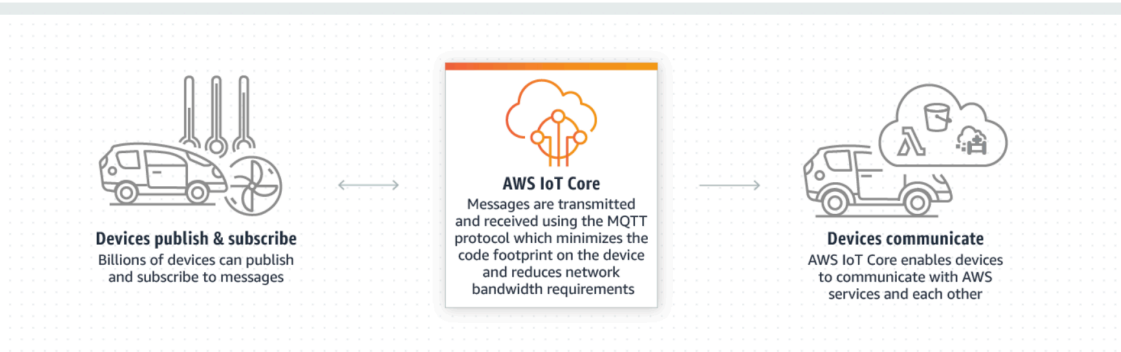


Рисунок 4. Принцип підключення пристроїв

Безпека підключення та даних

AWS IoT Core забезпечує автоматичну конфігурацію і автентифікацію при першому підключенні пристрою до AWS IoT Core, а також наскрізне шифрування по всіх точках підключення. Таким чином, будь-який обмін даними між пристроями і AWS IoT Core відбувається тільки після перевірки ідентифікації. Крім того, можна забезпечити безпечний доступ до своїх пристроїв і додатків, застосовуючи політики з точним настроюванням дозволів.

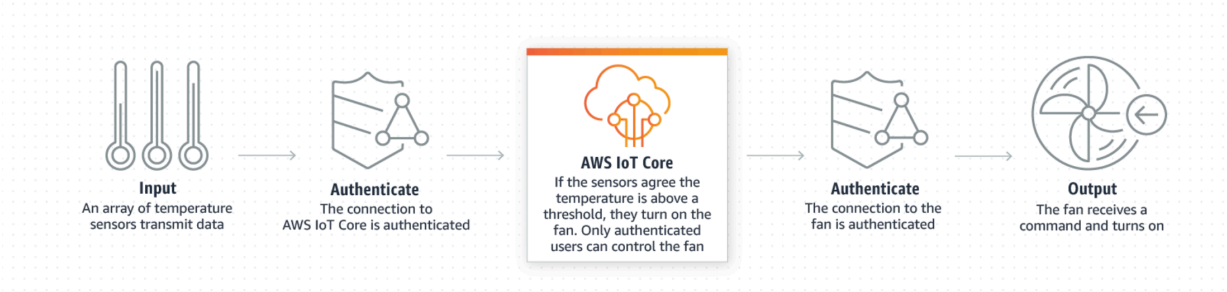


Рисунок 5. Забезпечення безпечної передачі даних

Обробка даних пристроїв та пов’язаних дій

За допомогою AWS IoT Core можна фільтрувати дані пристроїв, перетворювати їх на льоту і виконувати різні пов'язані дії на основі певних користувачем правил роботи. Ці правила можна оновити в будь-який момент з урахуванням нових пристроїв або можливостей програми. AWS IoT Core дозволяє з легкістю створювати ще більш потужні IoT-додатки з використанням сервісів AWS.

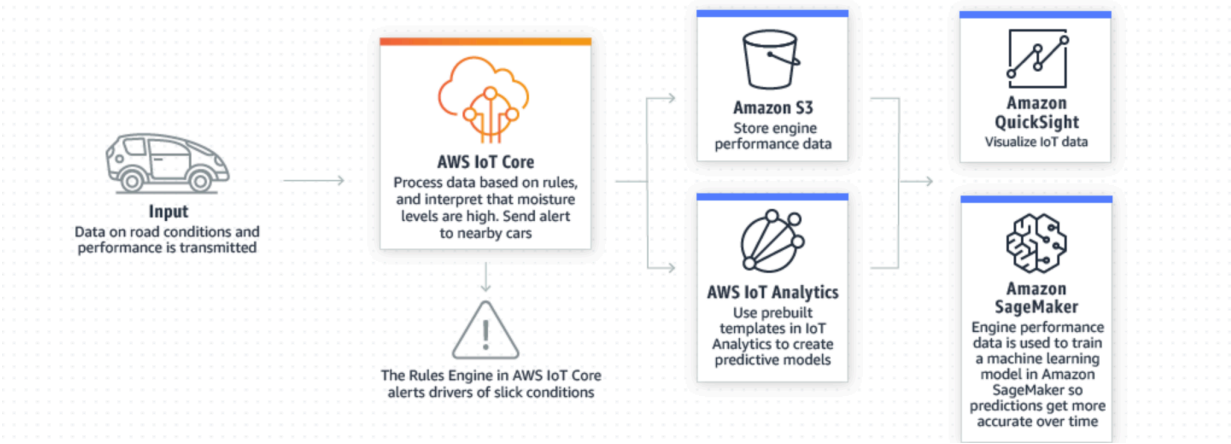


Рисунок 6. Архітектура побудови обробників даних

Зчитування на налаштування пристроїв в будь-яких момент

AWS IoT Core зберігає останній стан підключеного пристрою, що дозволяє в будь-який момент отримати або задати цей стан. Завдяки цьому програми працюють з пристроєм так, як ніби він постійно підключений. Як наслідок, додаток може зчитувати інформацію про стан пристрою, навіть якщо пристрій працює автономно, або може задавати стан пристрою, який буде застосовано при наступному його підключенні.

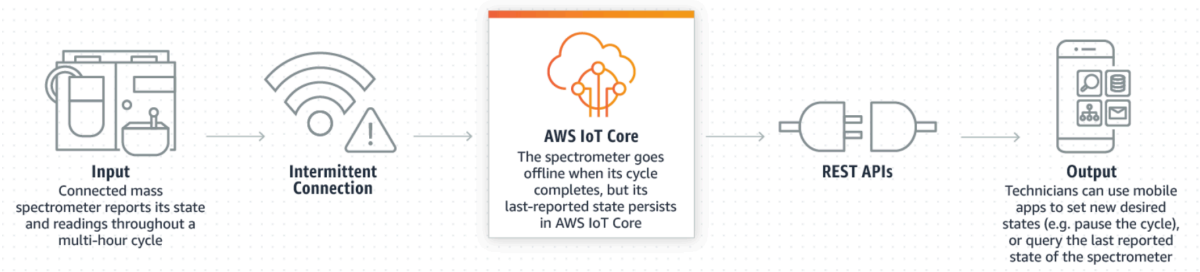


Рисунок 7. Обробка даних в умовах втраченого сигналу

Масштабування до сотень мільйонів пристроїв без зайвих витрат

Alexa Voice Service (AVS) Integration для AWS IoT Core представляє новий віртуальний пристрій Alexa Built-in в хмарі. Для використання сервісу AVS Integration клієнтам необхідний новий набір тем MQTT, зарезервований для AWS IoT, для передачі звукових повідомлень на основі протоколу MQTT між пристроями, підключеними до AWS IoT Core, і новим віртуальним пристроєм

Alexa Built-in. Завдяки цьому клієнти можуть відправляти і отримувати звукові повідомлення з використанням зарезервованих тим MQTT, встановлювати зв'язок з мікрофоном і динаміком пристрою і керувати станом з боку пристрою, використовуючи єдине безпечне підключення IoT Core.

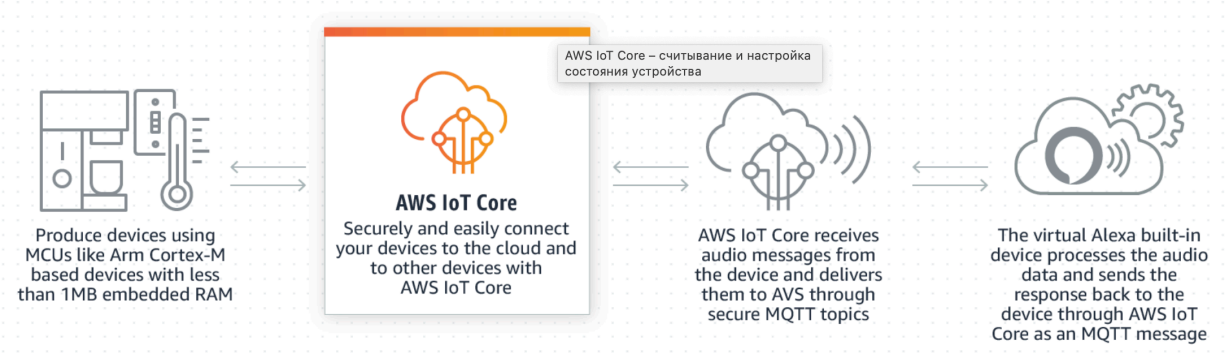


Рисунок 8. Обробка звукових повідомлень

Висновок до розділу

Поставлена задача ефективної синхронізації великих об’ємів даних в реальному часі групи мобільних роботів вимагає вирішення ряду проблем, що присутні в усталених формах програмного забезпечення роботизованих систем. До них відносять вирішення проблеми технічних складності системи, обмежений доступ пристроїв, жорсткий архітектурний дизайн, що потребує значних грошових та часових затрат. Основними особливостями хмарних систем, що дозволяють вирішити вказані проблеми є самообслуговування на вимогу, спільний ресурс, повсюдний доступ, вимірювана послуга та самообслуговування на вимогу.

Серед основних аналогів існуючих систем виділено такі як DavinCi, RoboEarth, RoboEarth. Представлені системи надають ряд переваг таких як наявність власної системи обміну повідомлення, централізована база знань, можливість обміну великих об’ємів інформації між мобільними роботами, власні механізми ми взаємодії з пристроями, гнучкий вибір планів залежно від

заданих вхідних навантажень Проте виділено такі недоліки як відсутність уніфікованих механізмів інтеграції зі сторонніми рішеннями, обмежене надання ресурсів в межах поточного плану, складність побудови системи синхронізації внаслідок використання специфічних рішень, що надано провайдером послуг.

З метою вирішення вказаних проблем та недоліків конкурентних систем обрано хмарний провайдер послуг Amazon Web Services як підґрунтя до побудови архітектури. Зазначений провайдер містить значну кількість сервісів для роботи з роботозиваними системами: AWS IoT Core, AWS IoT Device Defender, AWS IoT Device Management та інші. Доступні сервіси дозволяють ефективно вирішити проблеми уніфікованого інтерфейсу підключень, забезпечити високі показники швидкодії в умовах пікового навантаження, гнучко налаштовувати підсистеми безпеки та конфігурувати аналітичні системи з метою контролю за роботами сервісів.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ПІДХОДІВ ДО ПОБУДОВИ ХМАРНОЇ АРХІТЕКТУРИ ТА ВИБІР ОСНОВНИХ КОМПОНЕНТІВ

2.1. Огляд рішень в сфері хмарних СУБД

Хмарна база даних або DBaaS (База даних як послуга) – це база даних, яка надається як сервісний підхід через підписку чи хмарний сервіс на платформі. Тим не менш, DBaaS є одним із особливих випадків PaaS. За допомогою PaaS (Platform as a Service) клієнт отримує попередньо встановлене та налаштоване програмне забезпечення для розробки та тестування або надання додатків.

Клієнт у режимі самообслуговування може створити попередньо налаштовану базу даних у хмарі. DBaaS – це спеціальні елементи інфраструктури, які можна створити лише декількома клацаннями миші на порталі самообслуговування або викликавши інтерфейс програми. До них немає доступу до SSH, і насправді доступ до операційної системи неможливий. Користувач бачить лише інтерфейс для взаємодії з базою даних. Розгортати бази даних у контейнерах та на віртуальних машинах практично не потрібно, якщо правильно використовувати підхід Cloud Native. Власний хмарний підхід вимагає використання інтегрованих сервісів для роботи з даними.

За допомогою DBaaS клієнт може отримати доступ до бази даних того чи іншого типу, коли це необхідно, і швидко розгорнути базу даних на необхідній апаратно-програмній платформі (операційній системі). У цьому випадку оплата стягується на основі потужності та інших споживаних IT-ресурсів, а також функцій бази даних та адміністративних інструментів.

Реляційні хмарні бази даних (далі – БД) можуть включати:

Amazon RDS є однією з "ідеальних" хмарних баз даних. База даних пов’язана з API управління AWS і сумісна з низкою інших хмарних служб на підприємстві. Якщо раніше RDS давала можливість керувати базами даних, які є лише на AWS, тепер користувачі можуть використовувати інтерфейс

програмування для управління іншими базами даних у Microsoft SQL Server та Oracle Database.

Google Cloud SQL – може бути легко інтегрований з усіма іншими сервісами Google Cloud. Cloud SQL географічно розподілений для максимальної доступності. Однак він підтримує лише мови програмування Java та Python і обмежений 10 гігабайтами.

База даних Microsoft SQL – Може працювати як незалежна база даних хмари та пропонує загальний користувальницький інтерфейс Microsoft SQL Server, а також можливість обміну даними через SQL Server. Також можливо синхронізувати з іншими базами даних.

На ринку хмарних послуг існує безліч реляційних хмарних баз даних, за допомогою яких замовник може зробити для себе найбільш оптимальний вибір.

Хмарні бази даних NoSQL.

Бази даних NoSQL добре підходять для багатьох сучасних додатків, таких як мобільні, ігрові та інтернет-програми, які можуть потребувати таких функцій, як масштабованість, висока продуктивність, функціональність та простота використання. У той же час, залежно від призначення бази даних NoSQL, замовник може вибрати конкретний тип побудови бази даних:

БД на основі пар ключ-значення. Вони в основному використовуються в сферах маркетингу або IoT. Amazon DynamoDB є яскравим прикладом такої бази даних. Ця база даних пропонує затримку менше 10 мс і може обробляти пікові навантаження до 20 мільйонів запитів в секунду. Основна перевага полягає в тому, що немає необхідності налаштовувати масштабованість та стійкість до несправностей. Amazon пропонує готовий кінцевий продукт як послугу.[2]

База даних документів – У коді програми дані часто представляються як об’єкт або документ у форматі, подібному до JSON, оскільки це ефективна та інтуїтивна модель даних для розробників. Бази даних документів дозволяють

розробникам зберігати та запитувати дані в базі даних, використовуючи ту саму модель документа, яку вони використовують у своєму коді програми. Amazon DocumentDB (сумісний з MongoDB) та MongoDB – це популярні бази даних, що забезпечують функціональні та інтуїтивно зрозумілі API для гнучкої розробки.

Графічні бази даних використовуються для спрощення розробки та запуску програм, що працюють зі складними наборами даних. Приклади використання баз даних діаграм: соціальні мережі, системи виявлення шахрайства, діаграми знань. Видатним представником графічної бази даних є Amazon Neptune – повністю керована служба графічних баз даних.

БД в пам'яті. Таблиці лідерів та аналітика в режимі реального часу широко використовуються в ігрових та рекламних додатках. Такі функції вимагають відповіді протягом декількох мікросекунд, і збільшення трафіку можливе в будь-який час. Amazon ElasticCache пропонує агрегацію ріщень Memcached та Redis для обробки високопродуктивних робочих навантажень із низькою затримкою, що не досягається використанням дискового простору.

Пошук в базах даних. Багато програм генерують журнали, які розробники можуть використовувати для діагностики та усунення неполадок. Служба пошуку Amazon ElasticSearch (Amazon ES) – це спеціально розроблена послуга для візуалізації та аналізу автоматично сформованих потоків даних майже в режимі реального часу шляхом індексації, агрегування та пошуку напів структурованих журналів та метрик. Amazon ES – це також потужна повнотекстова пошукова система.[1]

Реляційна служба баз даних Amazon: як одна з перших хмарних баз даних, RDS від Amazon Web Services також є однією з найповніших хмарних баз даних. Як і більшість продуктів AWS, він прив'язаний до інтерфейсу управління AWS і сумісний з безліччю інших хмарних сервісів AWS. Якщо раніше RDS пропонувала можливість керувати базами даних, які є лише на хостингу AWS, користувачі тепер можуть теж обирати Microsoft SQL Server та Oracle Database.

База даних Clustrix як послуга: Clustrix лише нещодавно вийшов на хмарний ринок, але відразу створив враження сильного гравця. Послуга, яка працює в Rackspace Cloud, пропонує користувачеві потужне апаратне забезпечення та масштабовані функції MySQL. Компанія стверджує, що її обладнання підходить як для OLTP, так і для OLAP-програм, і вона одночасно контролює стан системи за 2500 показниками.

Хмарна база даних EnterpriseDB Postgres Plus: база даних PostgreSQL та її хмарна версія Postgres Plus. Postgres Plus Cloud зосереджується більше на корпоративних розробниках, ніж на хакерах-самоучках, і пропонує такі функції, як висока доступність кластерів, висока підключеність та сумісність із середовищами Oracle.

Heroku Postgres: Ця база даних надійна та безпечна – Heroku стверджує, що 99,99% часу розробки надійний на 99,99%. – і намагається передати свій досвід тим розробникам, які не можуть використовувати свої PaaS. Однією з найцікавіших особливостей цієї бази даних є кліпи даних, які дозволяють користувачам надсилати результати запитів SQL комусь іншому через URL-адресу.

База даних Microsoft SQL: База даних SQL є важливим компонентом нової гібридної хмарної стратегії Microsoft. Може виступати як окрема хмарна база даних, але також пропонує загальний користувальницький інтерфейс Microsoft SQL Server та можливість спілкування через SQL Server. Також є можливість синхронізувати кілька баз даних.

Хмарна служба Oracle Database: Ця послуга призначена лише для користувачів баз даних Oracle, які хочуть спробувати хмарний хостинг. Крім того, Oracle Database Cloud Service пропонує всі функції Oracle Database 11g Release 2, яких насправді дуже багато. Цінова політика не дуже чітка, але зрозуміло, що вона базується на розмірі бази даних.

Amazon DynamoDB: DynamoDB – це сервіс NoSQL, керований AWS, заснований на оригінальній системі ключ-значення Dynamo, яка розроблена власноруч кілька років тому. DynamoDB розроблений для Інтернету або великих додатків і має потенціал для масштабування. DynamoDB базується на архітектурі SD і автоматично масштабується при додаванні інформації до системи.

Amazon ElastiCache: Технічно кажучи, ElastiCache не є службою NoSQL, але доповнює систему хорошим кешуванням, щоб забезпечити якнайшвидший доступ користувачів.

Cloudant: Хоча Cloudant базується на CouchDB, це не служба NoSQL, а рівень даних. Він базується на декількох хмарних базах даних і пропонує не тільки масштабоване сховище даних NoSQL, але й механізм аналізу MapReduce.

MongoHQ / MongoLab: MongoDB – найпопулярніша база даних NoSQL, але вона може бути занадто великою для розміщення в хмарі. Як результат, існує кілька служб MongoDB, хоча MongoHQ та MongoLab більш відомі. Переваги обох простих: швидке розгортання, точний моніторинг та надійність. Обидві служби намагаються охопити широке коло користувачів, пропонуючи як загальні, так і спеціальні пропозиції.

Великі дані повинні зберігатися та керуватися більшими організаціями, які не мають належних систем зберігання, оскільки жодна з них не розроблена. Мета – знайти рішення цієї зростаючої проблеми шляхом рішення протиставлення підходів управління базами даних Big Data до баз даних NoSQL. BigTable, DynamoDB та Cassandra. Продуктивність системи аналізується на основі її послідовності, доступності та несумісності розділів. У результаті отримано висновок, що BigTable від Google і Amazon DynamoDB також самі по собі є критично важливими та ефективними, і що поєднання двох систем призвело до розвитку Cassandra. Сьогодні Cassandra знаходиться в центрі уваги численних компаній для розробки різних додатків. Крім того, усі три системи є

системами зберігання даних NoSQL та BigTable і, на відміну від DynamoDB та Cassandra, базуються на підході головного вузла та дотримуються однорангової системи. Однак BigTable сприяв розвитку Cassandra завдяки деяким додатковим функціям від DynamoDB, що є основою для різноманітних сучасних додатків.

Google BigTable

Підхід BigTable – це стовпчаста система зберігання для високопродуктивних програм. Як розподілена система зберігання, BigTable може обробляти "структуровані" дані. Слід враховувати різні системні вимоги, такі як масштабованість, продуктивність та доступність систем, досягнуті продуктами та проектами BigTable. Google розробив систему управління даними на базі NoSQL для внутрішнього використання своєї компанії під назвою "BigTable".

BigTable – це розподілена система зберігання, яка управляє "структурованими даними". Ця система використовує масштабованість великих даних для деяких програм Google.

Завдяки використанню "багатовимірної відсортованої карти", BigTable є дуже гнучким щодо багатьох аспектів, таких як модель даних. Файлова система Google (GFS) – це платформа для зберігання BigTable, дуже масштабована. Файли поділяються на фрагменти (коренасті файли), що копіюються на декількох комп'ютерах та беруть участь у всій системі. Метою цього процесу є підвищення доступності записів та надійності системи в режимі реального часу. Карта індексується унікальним «ключем рядка», «ключем стовпця» та «позначкою часу».

AWS DynamoDB

Amazon використовує Dynamo Databases, повністю керовану базу даних NoSQL, що пропонує передбачувану та нашвидку продуктивність з рівномірною масштабованістю. Це забезпечує базову платформу зберігання в системі Amazon. На платформі Amazon існують різні сервіси, які забезпечують

високі вимоги до надійності та вимагають доступу до сховища даних лише за допомогою первинного ключа. Однак обмежена масштабованість та вибір узгодженості порівняно з доступністю можуть бути не ідеальними. На відміну від RDMS, DynamoDB спроектовано таким чином, що в кінцевому підсумку є стабільним, але при цьому дуже масштабованим.

Основними особливостями є:

- однорангові системи структуровані та неструктуровані;
- розподілена файлова система;
- масштабована та децентралізована;
- підтримують лише API ключ-значення (відсутні ієрархічні простори імен або реляційна схема);
- ефективні затримки (відсутність мульти-стрибків);
- доступна система зберігання ключових значень;
- доступність, послідовність, продуктивність;
- розподіл даних за допомогою послідовного хешування;
- узгодженість за допомогою версій об’єктів.

Apache Cassandra.

Apache Cassandra – одна з найбільш масштабованих NoSQL. Підхід Cassandra дозволяє компаніям аналізувати дані певним чином, оскільки вони здатні обробляти дані, створені BigData. Cassandra з відкритим кодом заснована в 2009 році для управління великими даними компаній, таких як Google, Facebook та Amazon. Нещодавно Cassandra використовують деякі сучасні організації для обробки гострої "інфраструктури даних". На відміну від BigTable та DynamoDB, Cassandra є першим вибором для професіоналів, оскільки пропонує більшу гнучкість в обробці помилок та управлінні великими обсягами даних. Це робиться без шкоди для продуктивності системи . Cassandra використовує поєднання методів, що використовуються іншими базами даних

NoSQL, такими як Big Table та DynamoDB. Це дозволяє Cassandra зберігати доступність та масштабованість. [2]

Основними функціями цієї системи є:

- структуровані та неструктуровані однорангові системи;
- децентралізована система зберігання;
- симетрична орієнтація системи;
- ефективні затримки.

Система реального часу вимагає певних стилів на основі доступності, масштабованості та довговічності баз даних, які ретельно вивчені. Файлова система Google BigTable, на відміну від Amazon DynamoDB, яка використовує однорангові системи зберігання, управляє "ієрархічним простором імен". GFS використовується для продуктів та проектів Google Inc. Архітектура містить основний (головний) сервер для обробки наборів даних. Ці записи даних зберігаються у вигляді фрагментів на підлеглих серверах . [6]

З іншого боку, головний сервер GFS називають "товстою" абстракцією через його відмовостійкість. Amazon використовує DynamoDB для різних проектів та для численних ігрових додатків. Amazon використовує цю систему зберігання для зберігання та зчитування / отримання інформації про користувача. Різні конфлікти вирішують різні підходи, які використовуються для обробки оновлених конфліктів та мережевих розділів. Це дозволяє DynamoDB читати / писати в будь-який час.

Традиційна СУБД головним чином зосереджена на поліпшенні системної послідовності, але стає можливою за рахунок доступності та масштабованості системи. Оскільки вони лише гарантують та забезпечують високу узгодженість, ці системи зазвичай виходять з ладу під час обробки розділів мережі. Однорангове з’єднання та алгоритм, що базується на «Мережі», дозволяють кожному вузлу системи підтримувати інформацію вузла. Це структуроване перекриття з максимальною маршрутизацією запитів з одним стрибком.

BigTable пропонує як розподіл даних, так і структуровану систему, але залежить від доступності розподіленої файлової системи.

Подібно до BigTable, Cassandra – це "розподілена багатовимірна система карт", яка індексується за допомогою певного ключа. Рівень зберігання Cassandra, на відміну від розподіленої файлової системи BigTable та підключення бази даних Dynamo, є локальною файловою системою. Операції, що виконуються під кожним ключем, є атомними для кожної реплікації, незалежно від кількості прочитаних / записаних стовпців. Крім того, групи стовпців (сімейства стовпців) вирівнюються так само, як і BigTable. Запит на читання або запис – це ключ, який зростає і переадресовується на будь-який вузол системного кластера.

Таблиця 1. Основні властивості BigTable, DynamoDB, Cassandra

Властивості	BigTable	DynamoDB	Cassandra
Data Management	Structured	Structured & unstructured	Structured &unstructured
Data Source	Strings, Graphs	Binary, string, number	All data types
Data Model	Multidimensional sorted map	Key value	Column family (BigTable)
System Orientation	Sparse	Symmetric	Symmetric
Data Storage	Column stores (SSTables)	Binary-value objects	SSTable Disk Storage
Load distribution	Load Balancing	Load sharing (heterogeneous)	–
Scalability	High	Incremental	Linear scalability
Consistency	High	Eventual	Eventual (DynamoDB)

CAP Теорема дає інформацію про бажані особливості RDMS. Відповідно до теореми CAP кожна сучасна база даних може виявляти лише дві властивості з таких: узгодженість (C), доступність (A) та толерантність до розділів (P). BigTable – це розподілена система зберігання, яка може керувати “Структурованими даними і є орієнтованою системою баз даних. Як вже

сказано, Amazon є високо доступною системою, яка використовує "ключ-значення" для доступу до "структурованих" та "неструктурованих" даних сховища. Реплікація даних здійснюється на кожному вузлі системи, на відміну від BigTable, який включає один головний вузол. Це дозволяє системі уникнути несправностей та недоступності.

Впорядковані дані на розподіленій карті та індексованому пошуку показують, що цей підхід має форму Cassandra. Незважаючи на те, що Cassandra є комбінацією BigTable та DynamoDB, вони в основному є орієнтацією на з іншими функціями, які мають властивості BigTable від Google.

Виявлено, що NoSQL, BigTable, DynamoDB та Cassandra застосовуються при керуванні великими даними. Контраст та порівняння цих даних також демонструють, що різні особливості цих систем потенційно застосовні до цих систем із відкритих джерел. Спостереження, засновані на даних, взяті з повідомлень, які чітко показують, що Cassandra в основному базується на системній архітектурі та моделі даних BigTable.

Cassandra спочатку використовувалась у пошуку вхідних повідомлень Facebook, однак зараз вона використовується у деяких соціальних та приватних додатках. BigTable базується на підході одного головного вузла, на відміну від BigTable, Dynamo та Cassandra, дотримуючись системи Peer-to-Peer.

2.2. Потокове передавання даних у режимі реального часу за допомогою Kinesis Firehose

Рисунок 9. Взаємодія з Kinesis Data Firehouse

Elasticsearch – це рішення з відкритим кодом, яке використовується багатьма компаніями по всьому світу для аналітики. За визначенням, Elasticsearch – це рішення з відкритим кодом, RESTful, розподіленим, індексованим пошуком та аналітикою.

Перша частина визначення полягає в тому, що це рішення з відкритим вихідним кодом, яке керується спільнотою, і воно вільне для широкого використання. Далі, це RESTful тому всі зв'язки та конфігурації можна здійснити за допомогою простих викликів REST HTTP API. Elasticsearch розробив багатофункціональну структуру REST API для використання клієнтами для споживання. Elasticsearch диференціюється із загальним пошуковим рішенням, яке застосовується поверх існуючих баз даних. Elasticsearch масштабований, так як функціонал та дані, що зберігаються, використовують кілька ресурсів, і всі ці ресурси використовуються для виконання необхідних функцій, що робить їх дуже ефективними, забезпечуючи при цьому високу доступність. Elasticsearch використовує Apache Lucene для індексації, що є надзвичайно швидким рішенням для індексування та забезпечує надзвичайно швидкий пошук або аналітику.

Amazon Kinesis – це послуга, що надається AWS для обробки даних у режимі реального часу.[4] Дані надходять до Kinesis як потік, де Kinesis виконує всі види функціональних можливостей на основі заданих вимог. У спільноті також є багато інших рішень для обробки потоків даних: Apache Kafka, Apache Spark. Основна причина вибору Kinesis полягає в тому, що рішення Elasticseach також створено як послугу AWS, тому легко зосередитись на результатах, а не керувати інфраструктура.

Потоки даних – це дані, які генеруються та передаються безперервно з джерела даних. Це джерело даних називають виробниками даних у потоковому світі. Головна особливість, яка відрізняє потоки даних від інших форм джерел даних, полягає в тому, що вони генеруються постійно. Отож замість того, щоб

надсилати дані групами, у потоковому режимі, кожного разу дані будуть передаватися нам за допомогою потоку.

Таким чином, використання потоків даних, дозволяє виконувати аналітичні задачі в режимі реального часу і надавати аналітичні дані як вихідні дані чи зберігати ці дані та робити аналітику поверх даних. Перший сценарій використовується в багатьох рішеннях, де потрібна аналітика в режимі реального часу, така як виявлення заторів на трафіку, виявлення різних моделей тощо. Для другого сценарію дані зберігаються на аналітичній платформі.[7]

Потоки даних Kinesis – використовуються для збору та обробки великих потоків записів даних у режимі реального часу:

- Kinesis Data Firehose – використовується для доставки потокових даних у режимі реального часу до таких сервісів, як Amazon S3, Redshift, Elasticsearch тощо.
- Kineses Data Analytics – використовується для обробки та аналізу потокових даних за допомогою стандартного SQL.
- Kinesis Video Streams – використовується для повного управління службами, які використовують потокове відео з пристроїв.

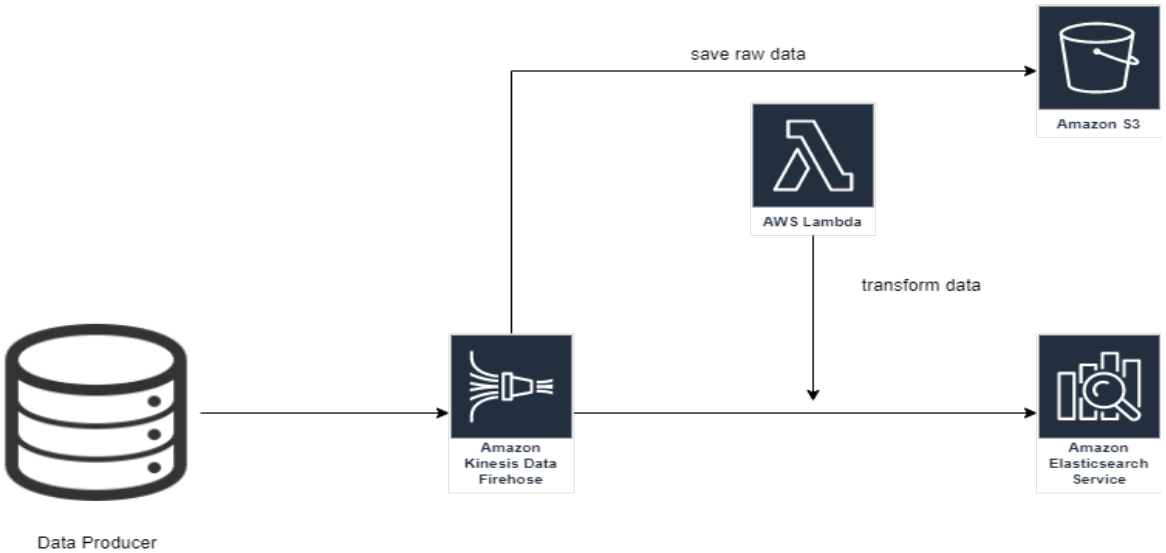


Рисунок 10. Схема Kinesis Data Firehose

Отже, з запропонованих рішень слід використовувати Kinesis Data Firehose.

Kinesis Data Firehose – одне з чотирьох рішень, які надає сервіс AWS Kinesis. Отож як служба Kinesis, вона також повністю управляється AWS, тому не потрібно турбуватися про розгортання, доступність та безпеку на кожному рівні. Kinesis Firehose використовується для доставки потокових даних у реальному часі до заздалегідь визначених пунктів обслуговування. Доступні насутпні кінцеві обробники даних:

- Amazon S3 – простий у використанні об'єкт для зберігання.
- Amazon Redshift – сховище даних у масштабі петабайт.
- Splunk – інтелектуальний операційний інструмент для аналізу машинних даних.

Перш ніж впроваджувати та розробляти чи вирішувати, спершу розглянемо деякі основні концепції Kinesis Firehose.

Потік доставки Kinesis Data Firehose – основний компонент потоку доставки, який відправляється до пунктів призначення.

Виробник даних – організація, яка надсилає записи даних до Kinesis Data Firehose. Це основне джерело даних.

Запис – дані, які виробник даних надсилає потоку доставки Kinesis Firehose. У потоці даних ці записи будуть надсилатися безперервно. Зазвичай ці записи будуть дуже малі з максимальним значенням 1000 КБ.

Розмір буфера та інтервал буфера – конфігурації, які визначають, скільки буферизації потрібно перед доставкою їх до пунктів призначення.

Для обробки потоку даних виробники даних повинні постійно надсилати дані до Firehose. Отже, тепер питання повинно бути таким, як ці виробники даних надсилають дані до нашого Firehose. Існує декілька способів, як виробники даних можуть надсилати дані до Firehose.

Kinesis Data Firehose PUT API – API PutRecord () або PutRecordBatch () для надсилання вихідних записів у потік доставки.

Amazon Kinesis Agent – Kinesis Agent – це окремий додаток Java, який пропонує простий спосіб збору та надсилання вихідних записів. Для того, щоб завантажувати дані в агент Kinesis, цей агент повинен бути доступним у системі отримання даних.

AWS IoT – правила AWS IoT, які надсилають дані з повідомлень MQTT.

Журнали CloudWatch – фільтри підписки для передачі потоку журнальних подій у реальному часі.

У представленому рішенні використовується демонстраційний потік даних, наданий Kinesis Firehose. Відповідно до наведеної вище схеми перед надсиланням даних до служби Elasticsearch, функція лямбда трансформує дані відповідно до вимог. Тим часом необроблені дані перед перетворенням будуть надіслані у сегмент AWS S3. Там, використовуючи правила життєвого циклу їх можна перенести або в AWS Galcier, або в інші категорії S3 відповідно до вимог.

Створення служби Elasticsearch здійснюється з метою розробки та навчання, тому конфігурації, які обрано для домену, не будуть найбільш безпечними конфігураціями і не повинні використовуватись для корпоративного рішення.

Необхідно до служби AWS Elasticservice та створити новий домен. Після чого обрати тип розгортання як розробка та тестування.

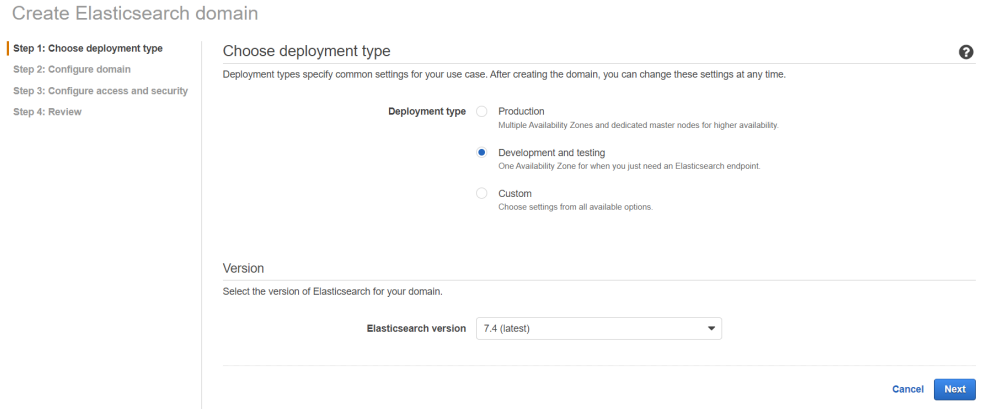


Рисунок 11. Створення ElasticSearch

Далі вказати доменне ім’я і після цього обрати тип екземпляра – t2.small.elasticsearch, оскільки це єдиний тип екземпляра, доступний для вільного рівня. Інші параметри слід залишити як значення за замовчуванням без будь-яких змін.

Конфігурування безпеки здійснюється наступним чином. Рекомендується використовувати доступ до VPC там, де домен буде знаходитись у приватній мережі, лише з екземплярами в мережі VPC, які матимуть доступ.

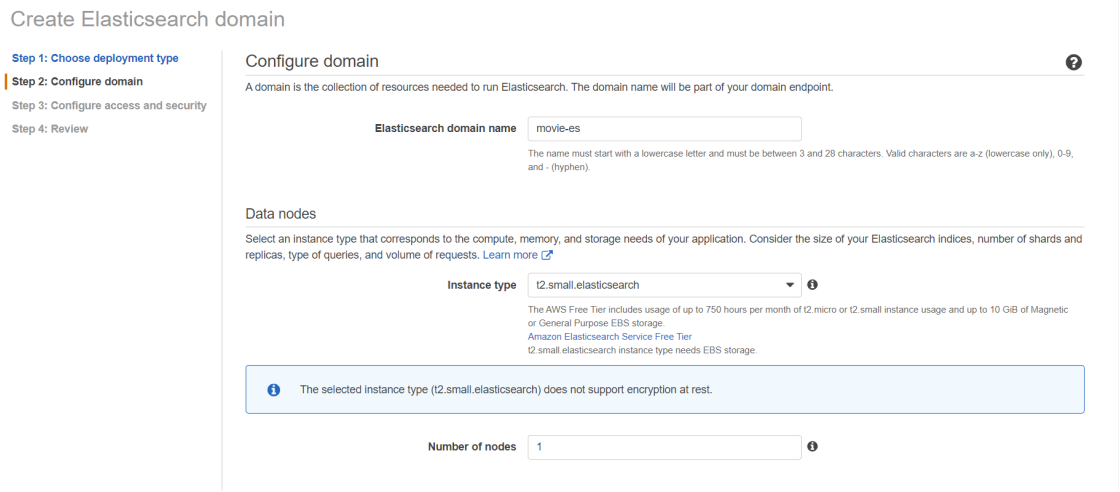


Рисунок 12. Конфігурація ElasticSearch

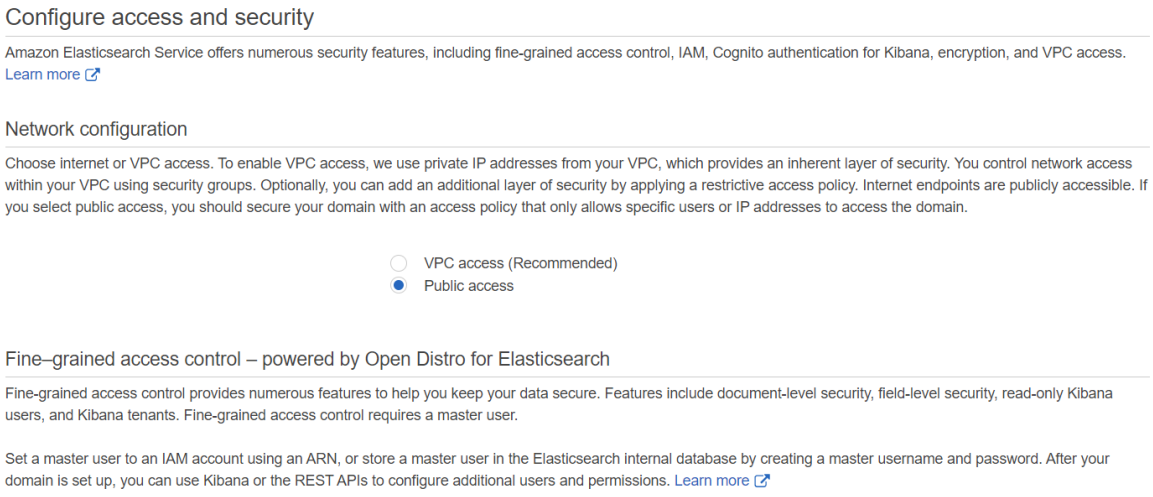


Рисунок 13. Налаштування безпки ElasticSearch

Необхідно визначити безпеку політики доступу. Оскільки це змодельоване рішення, слід дозволити відкритий доступ до домену, який зробить доступ до будь-якої IP. Вказується або роль IAM, або конкретні облікові записи користувачів AWS.

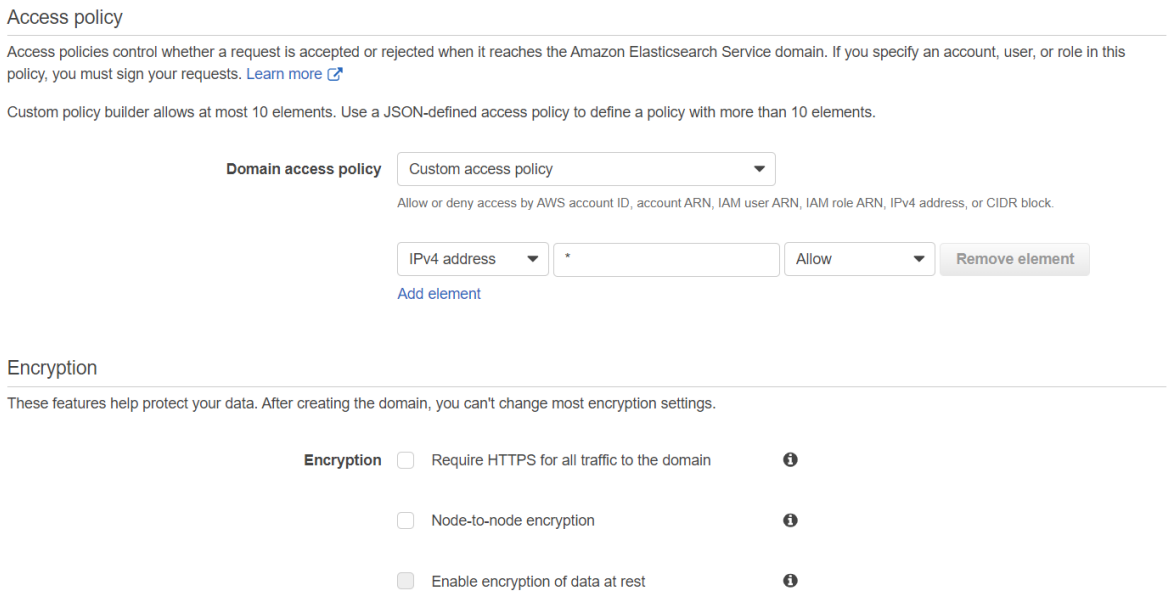


Рисунок 14. Налаштування безпеки ElasticSearch

Отже, створення нашої Elasticsearch завершено.

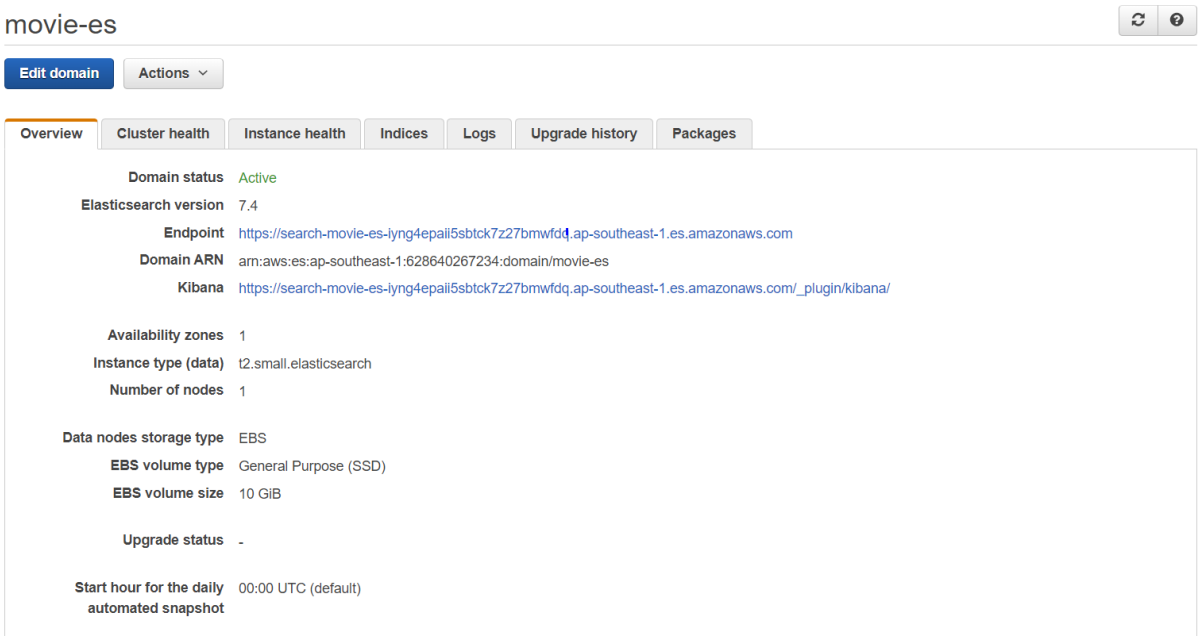


Рисунок 15. Результат створення ElasticSearch

Створення Kinesis Data Firehose

Необхідно перейти до служби AWS Kinesis та обрати Kinesis Firehose, створивши потік доставки. На наступному екрані вказати назву потоку та обрати джерело як Direct PUT або інші джерела. Іншим варіантом тут є вибір джерела як потоку даних Kinesis. У змодельованому сценарії у немає потоку даних Kinesis, і дані надходять до Firehose безпосередньо від виробника.

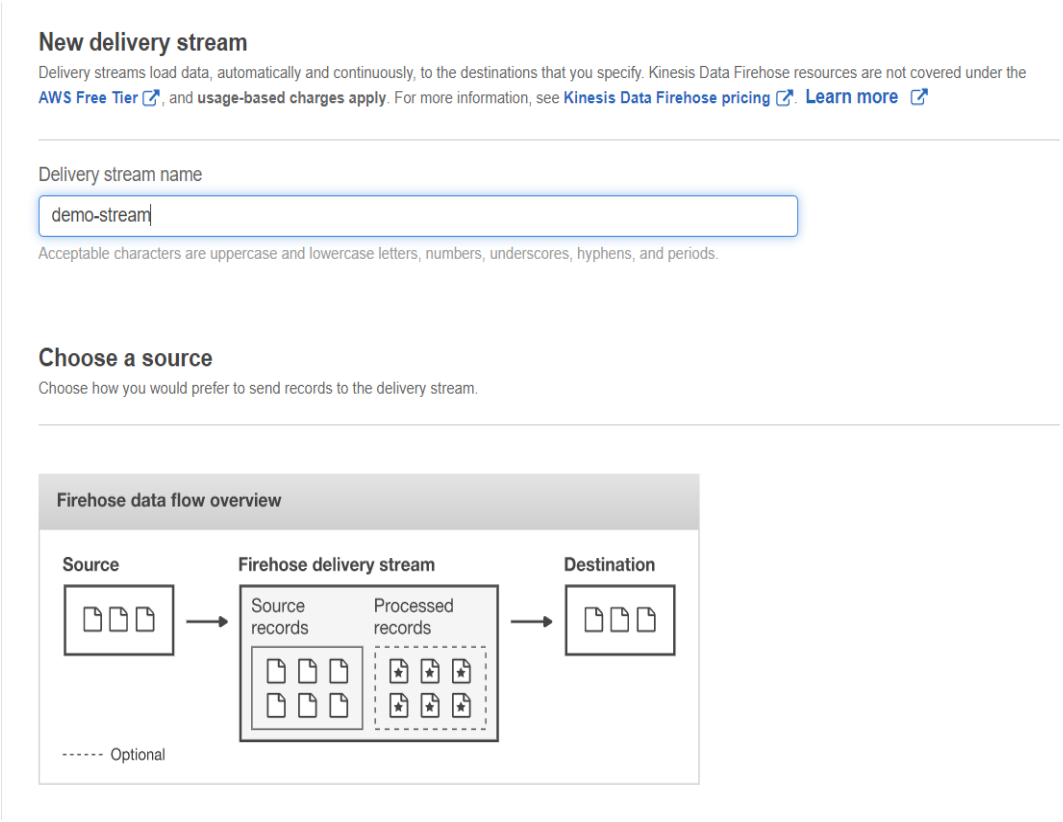


Рисунок 16. Створення Kinesis Data Firehose

Є можливість увімкнути шифрування на стороні сервера для даних, які передаються. В даному випадку слід увімкнути шифрувати дані та надати необхідний ключ шифрування.

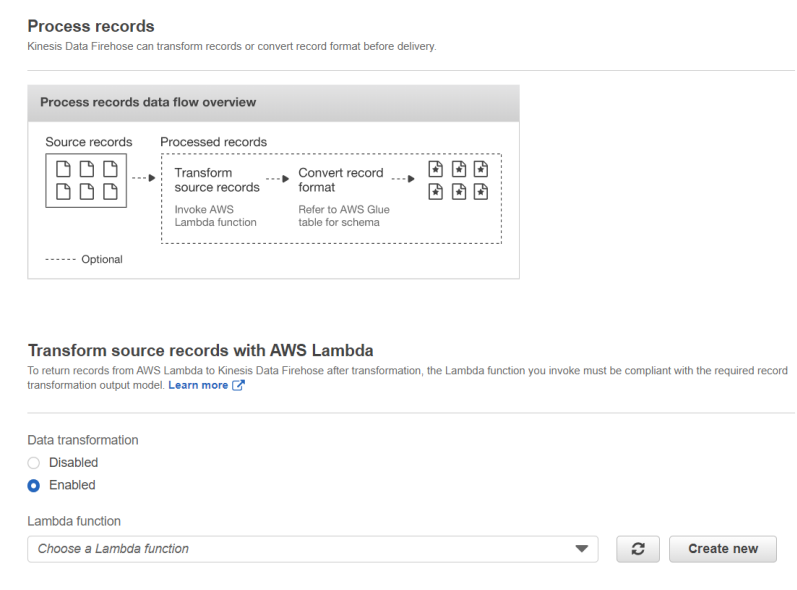


Рисунок 17. Конфігурація Kinesis Data Firehose

В конфігурації трансформування вихідних записів необхідно увімкнути перетворення даних. Після чого слід обрати створення нової Lambda функції на основі шаблону General Kinesis Data Firehose Processing.

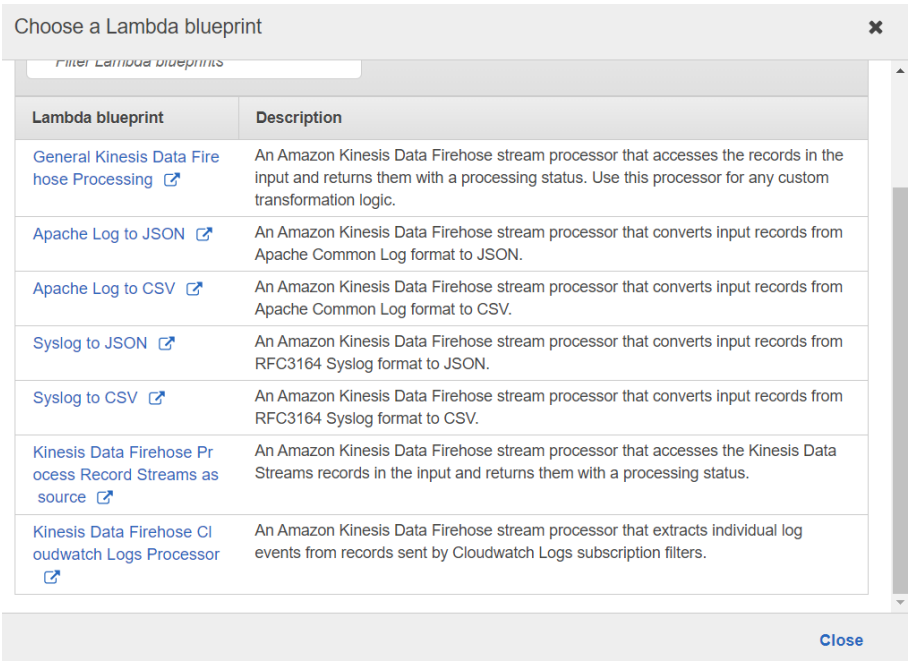


Рисунок 18. Лямбда Конфігурація Kinesis Data Firehose

Щоб відбулось успішне створення Lambda функції потрібно конфігурувати роль IAM для лямбда-функції з дозволу на послуги AWS Kinesis.

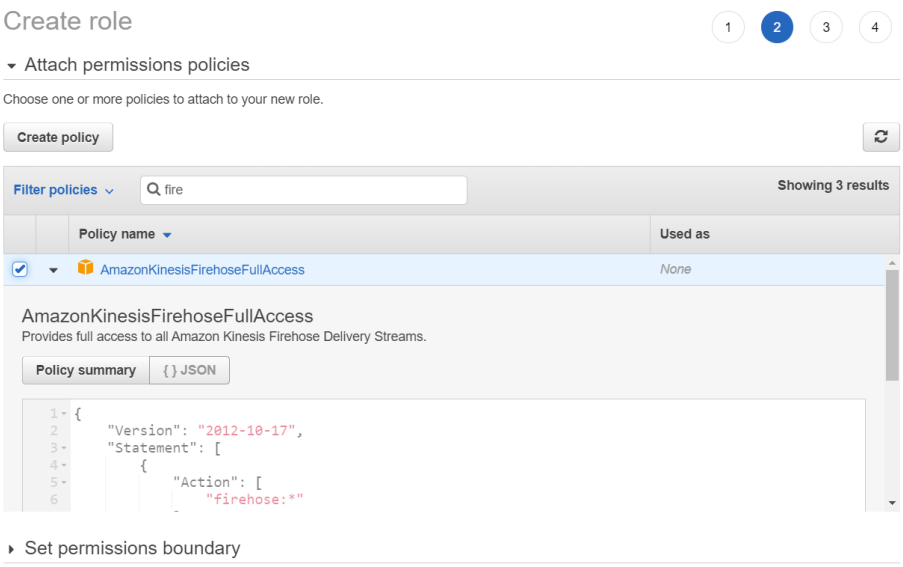


Рисунок 19. Налаштування безпеки Kinesis Data Firehose

Слід надати повний доступ до Firehose та окремо дозвіл AWSLambdaExecute, який забезпечить можливість виконання створеної функції, а також дозвіл на створення журналів для служби CloudWatch.

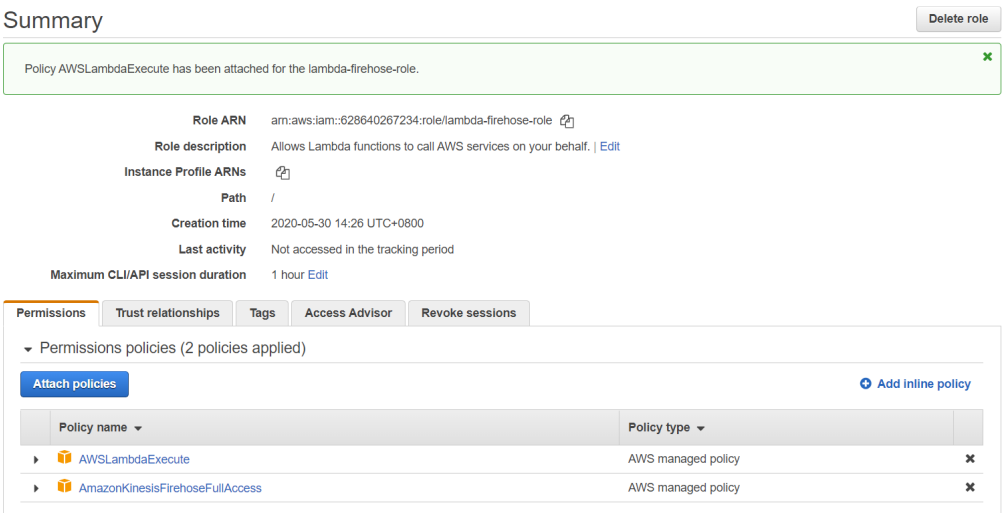


Рисунок 20. Опис безпеки Kinesis Data Firehose

Після створення ролі необхідно перейти до сторінки створення функції лямбда. Дала додати функції назву і обрати раніше створену роль IAM.

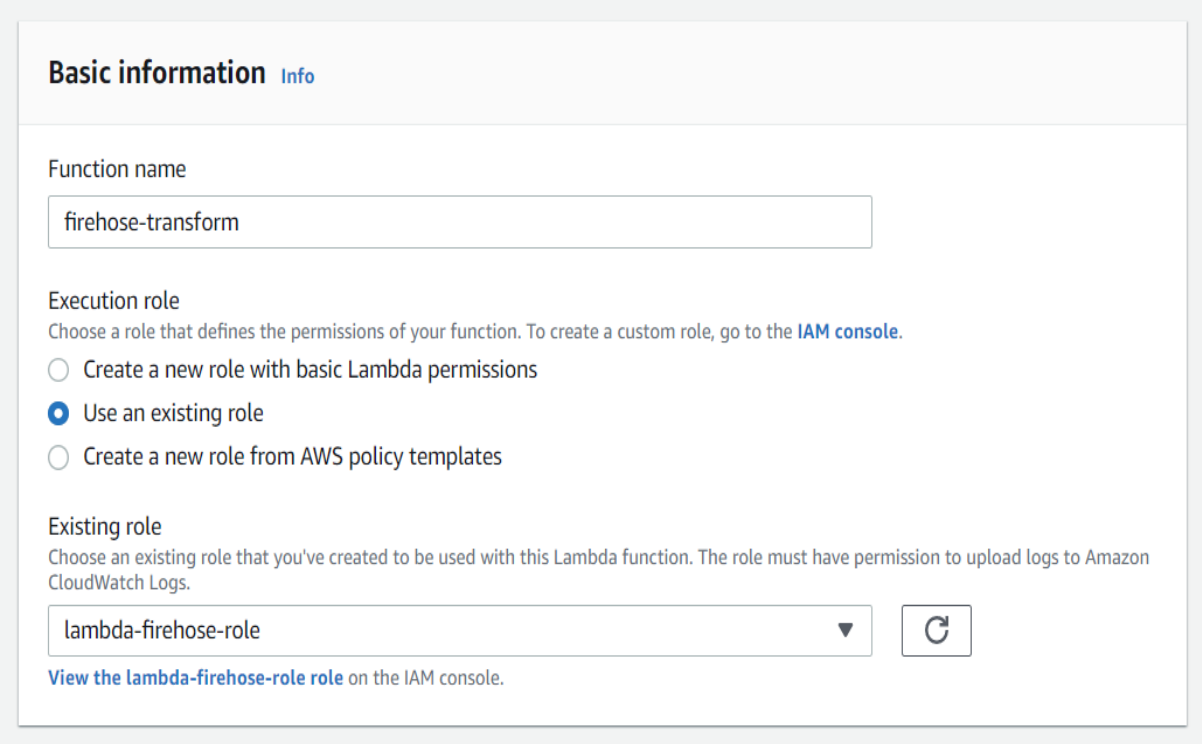


Рисунок 21. Створення ролі Kinesis Data Firehose

Формат даних:

{"ticker_symbol":"QXZ","sector":"HEALTHCARE", "change":-0.05, "price":84.51}

Припустимо, що потрібно зберігати властивість зміни в службі Elasticsearch. Нижче наведена лямбда-функція, створена за допомогою node.js для того, щоб виконати перетворення. Lambda підтримує будь-яку мову для створення своєї функції.

Як уже згадувалося вище, здійснюється проста трансформацію перейменування властивості та видалення небажаної властивості. Слід обрати створену функцію Lambda.

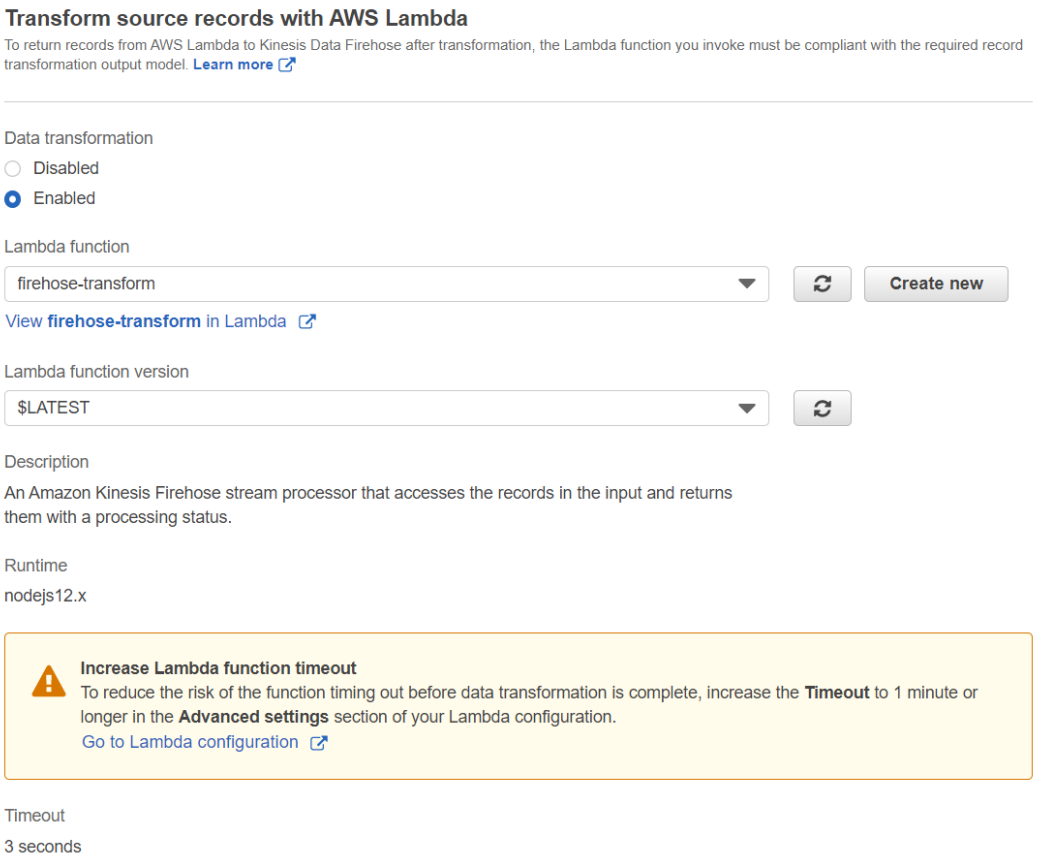


Рисунок 22. Опис зміни даних за допомогою Kinesis Data Firehose

В результаті отримаємо попередження про те, що час очікування функції лямбда становить лише 3 секунди і слід збільшити його принаймні на 1 хвилину. Оскільки здійснюється обробка потокових даних, для виконання та завершення функції може знадобитися деякий час, і тайм-ауту за замовчуванням 3 секунди недостатньо. Lambda підтримує тайм-аут до 5 хвилин.

Після цього є можливість перетворити формат запису у формат Apache Parquet для формату Apache ORC, а не у формат JSON. Це перетворення здійснюється за допомогою служби AWS Glue, визначаючи там схеми. Але в змодельованому сценарії така функціональність нам не потрібна.

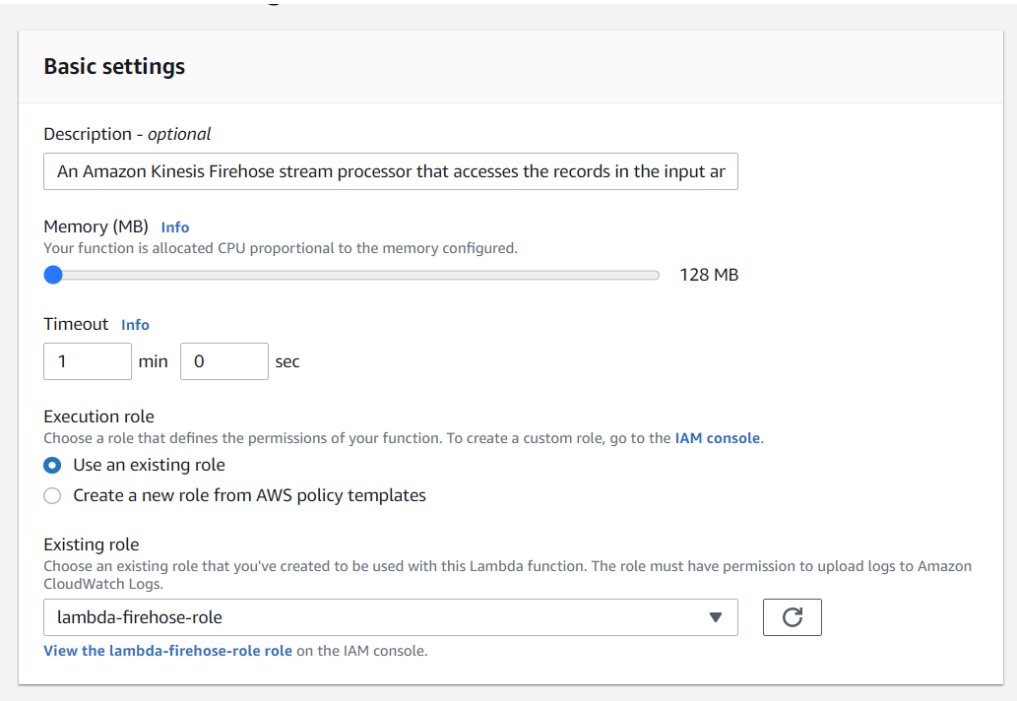


Рисунок 23. Конфігурація змін даних за допомогою Kinesis Firehose

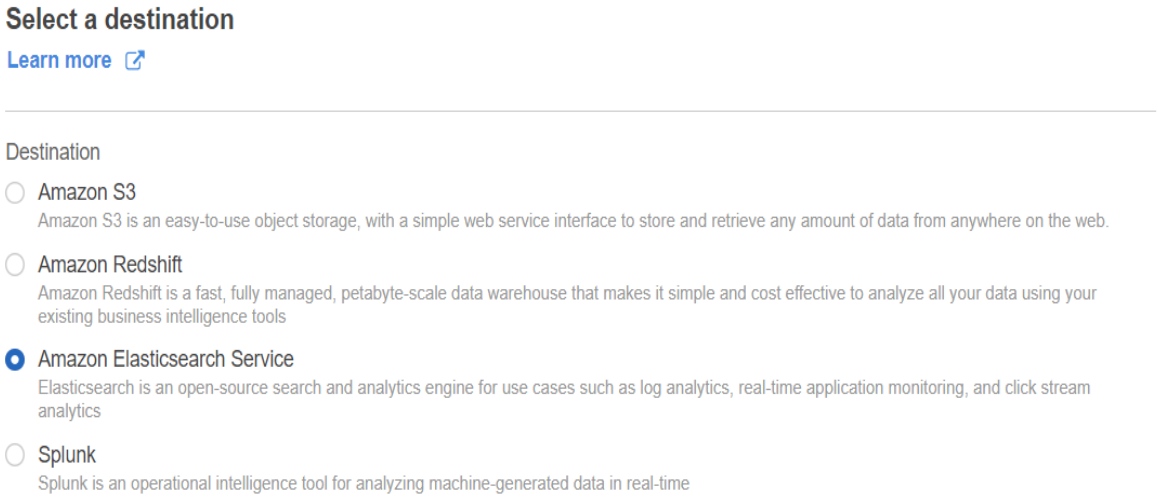


Рисунок 24. Вибір джерела даних за допомогою Kinesis Data Firehose

Обрати призначенням потоку даних Elasticsearch та домен, який створено раніше.

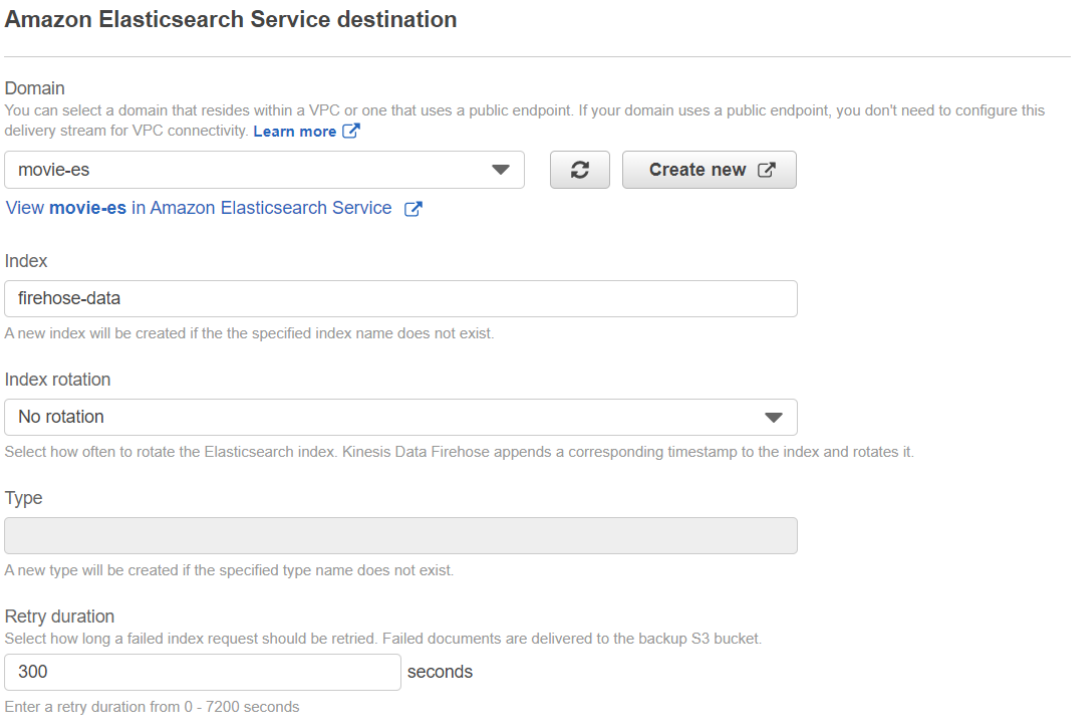


Рисунок 25. Інформація про сервіс Elasticsearch

Далі наявна можливість робити резервні копії необроблених даних. Можна вибрати для резервного копіювання лише невдалі записи або всі записи для майбутніх цілей. Потрібно вибрати сегмент S3 чи створити новий. Слід додати префікс до потоку даних, який буде збережено на сегменті S3, щоб забезпечити легку класифікацію збережених даних всередині сегмента S3. Kinesis Data Firehose автоматично додає префікс UTC “РРРР / ММ / ДД / ЧЧ /” до доставлених файлів S3.

Можливо змініити конфігурацію Elasticsearch . Перший пункт – це умови буфера Elasticsearch. Firehose буферезує дані перед тим, як відправити їх на зберігання в Elasticsearch, можнв визначити цей буфер, використовуючи дві метрики: розмір буфера та інтервал буфера. Тож, коли будь-яка з цих умов виконана, Kinesis Firehose передасть дані Elasticsearch. Оскільки використовуються функції Lamda для трансформації потоку даних, потрібно дотримуватися обмеження корисного навантаження функції AWS Lambda, яке становить 6 МБ.

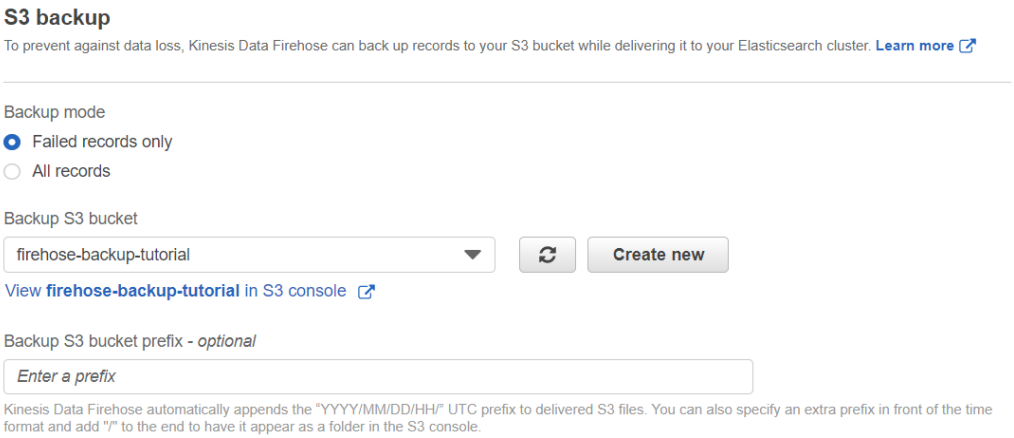


Рисунок 26. Збереження даних S3

Наступна конфігурація – стиснення та шифрування S3. Ці конфігурації пов'язані із резервним сегментом S3, який використовується для необроблених даних. Тут можна або стиснути дані, щоб зробити їх меншими, або зробити їх безпечними, зашифрувавши дані в S3. Наступні реєстрації помилок будуть ввімкнені за замовчуванням, що реєструватиме помилки в Cloudwatch, нарешті, потрібно створити нову роль IAM для служби Firehose.

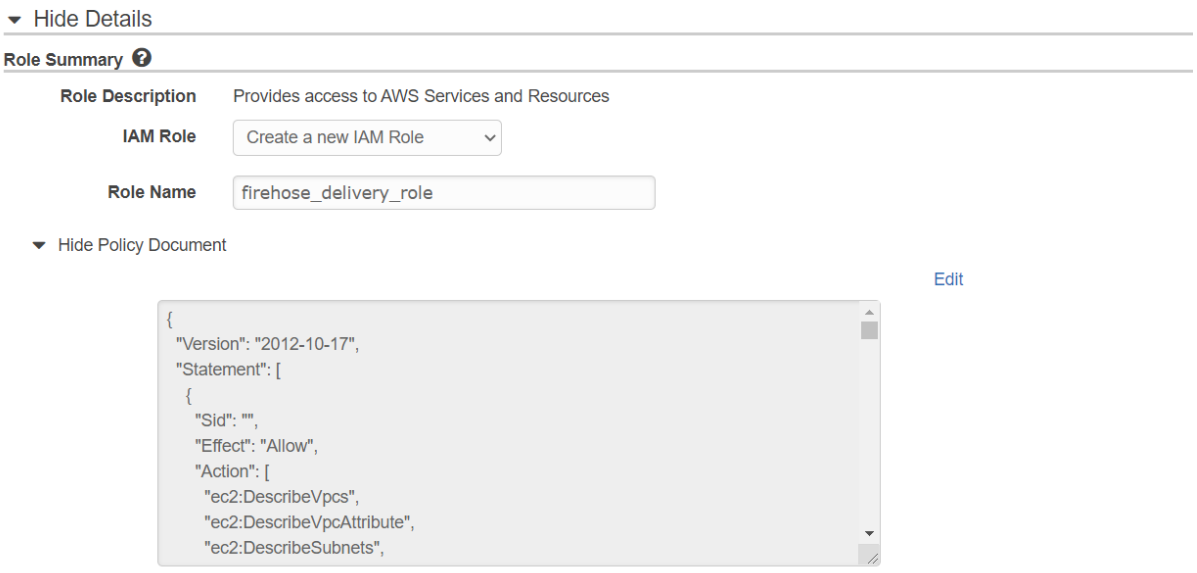


Рисунок 27. Результат конфігурації ролі

Після задання всіх необхідних конфігурацій потрібно створити потік Firehouse, Статус потоку стане активним, коли потік буде повністю створений.

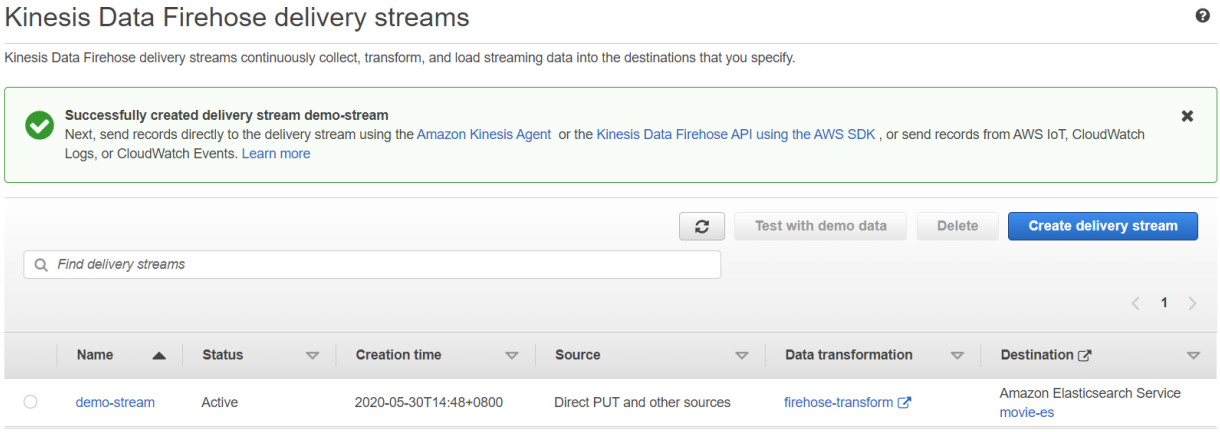


Рисунок 28. Результат конфігурації ролі

Тепер для того, щоб протестувати систему, яка працює за заданою моделлю, слід перейти до потоку та протестувати з демонстраційними даними.

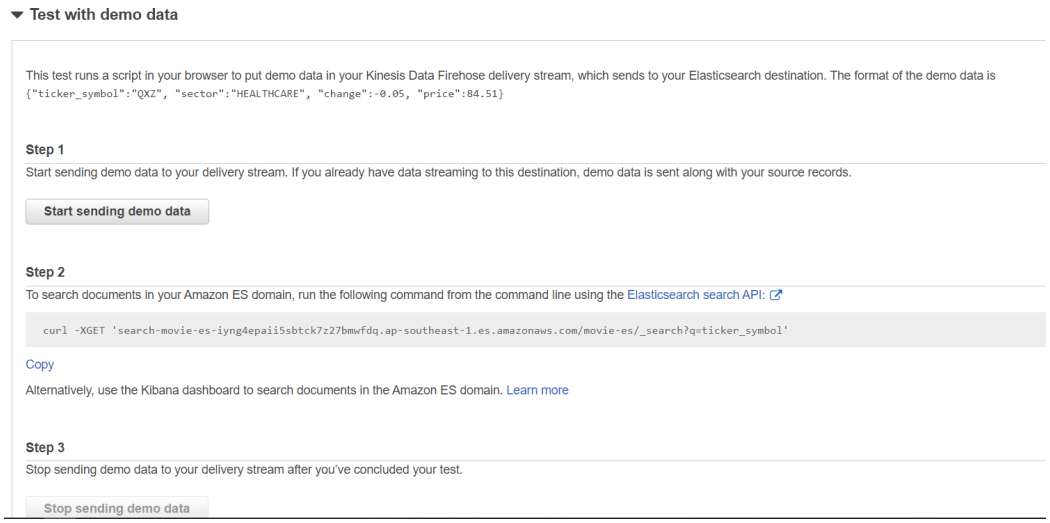


Рисунок 29. Схема обробки даних AWS Kinesis

Висновок до розділу

Хмарна архітектура включає в себе ряд доступних сервісів зберігання даних, передачі потоку інформації, забезпечення комунікації між окремими частинами системи. Хмарні СУБД на відміну від класичних містять переваг у вигляді простої вертикальної та горизинтальної масштабованість в лічені хвилини без необхідності здійсння міграції даних, оплату лише за час використання враховуючи кількість операцій зчитування/запиту. Кожна окрема хмарна СУБД розроблена з метою забезпечення вузьких функціональних вимог. До хмарних сховищ які варто використовувати для збереження даних роботизованизх систем віднесено Google BigTable, Amazon DynamoDB, Amazon Neptune, Apache Cassandra. Враховуючи такі особливості хмарного сховища Amazon DynamoDB як забезпечення високих вимог до надійності, передбачувана та надшвидка масштабованість дану СУБД обрано як основну в проектування архітектури. Amazon DynamoDB базується на однорангових структурованих та неструктурвоаних системах, є розподіленою файловою системою, масштабована та централізована, забезпечує відсутність мульти стрибків, узгодженість даних основана на версіях обєктів. Варто зазначити, що ефективність пошуку даних в AWS DynamoDB базується на основі використання індексів, що складаються з первинного ключа та ключа сортувань.

Зменшення кількості операцій зчитувань та запису досягнуто за рахунок сервісу хешування Redis, що забезпечує швидкість операцій вставки, зчитування на рівні операцій зі звичайним об’єктом. Сервіс Redis дозволяє здійснювати горизонтальне масштабування сервісів за рахунок агрегування даних з екземплярів сервісів. Гнучкість даних досягнуто за рахунок великого різноманіття структур даних від простого виду пари ключ-значення до карт.

AWS Kinesis Data Firehouse обрано в якості рішення для обробки потоків даних. На відміну від свої конкурентів – Apache Kafka, Apache Spark забезпечує

потокову передачу даних від різних джерел, а не пересилку даних групами. Пропускна здатність сервісу досягає десятка тисяч записів в секунду з можливістю горизонтального масштабування. Гнучка конфігурація вхід та вихідних потоків інформації дозволяє автоматично синхронізувати дані між різними зв’язаними сервісами, наприклад вихідні дані потрапляють відразу на обробку в без серверне середовище AWS Lambda, логуються в AWS S3 та аналізуються сервісом AWS CloudWatch.

Більшість сервісів хмарної системи синхронізації даних групи мобільних роботів створено на основі без серверного середовища AWS Lambda та хмарних серверів типу AWS EC2. Архітектура, що побудована на зазначеному підході забезпечує як горизонтальну так і вертикальну масштабованість.[7]

РОЗДІЛ 3. ПРОЕКТУВАННЯ АРХІТЕКТУРНИХ РІШЕНЬ

3.1. Побудова хмарної архітектури на основі доменних моделей (Domain Drive Development)

Деякі системи обробки контексту засновані на мережі, в той час як інші мають свої тісно пов'язані компоненти. Однак можна визначити основні завдання обробки контексту для досягнення обізнаності щодо контексту, наступним чином:

- Зондування та вибір контексту.
- Фільтрування та агрегування контексту.
- Зберігання контексту.
- Контекстне міркування.
- Використання контексту.

Взагалі, архітектурний дизайн для контекстно-значущих додатків розпочався із повністю централізованим підходом, що стосується контексту одного користувача та одного додатка, з обмеженими можливостями управління контекстом, повністю вбудованими в програму. Пізніше управління контекстом абстраговано від логіки додатків із виділеним контекстним сервером обробки, що дозволяє краще використовувати при запуску декількох локальних додатків. Поступово управління контекстом переросло у розподілену архітектуру, де контекстно-відомі додатки використовували той самий віддалений централізований сервер обробки контексту, що дозволяє багатьом віддаленим користувачам користуватися персоналізованими послугами.

Однак за останні 5 років досягнуто порівняно невеликого прогресу в контекстно-обізнаних розробках щодо того, як зробити управління контекстом більш ефективним та результативним шляхом розподілу відповідних завдань та загального навантаження. Робота, представлена в цьому розділі, намагається забезпечити нову фазу розробки додатків з контекстом, використовуючи

парадигму хмарних обчислень для кращої масштабованості та багаторазового використання управління контекстом, впроваджуючи слабозв'язані компоненти, що підтримують контекст, для досягнення вищезазначеного контексту опрацювання завдань.[4]

Сервісно-орієнтований підхід до програм, що усвідомлюють контекст.

Розроблено ряд архітектур, що підтримують контекстно-орієнтовані додатки, але особливої вимоги та обмеження мобільних роботів приділяється мало. Багато сучасних централізованих моделей розгортання, специфічних для додатків комунікаційних протоколів та представлення контекстних даних непросто застосувати в хмарному середовищі, коли спільне використання контексту між взаємопов'язаними службами породжує нові завдання представлення контекстних даних, зберігання, обробка та розподіл сервісно орієнтованим способом. Переваги та необхідність сервісно-орієнтованої абстракції інфраструктури для надійних контекстних систем.

Парадигма хмарних обчислень – це екосистема, побудована поверх веб-сервісів. Для гнучкого управління його обробкою та складом допоміжні компоненти обробки контексту повинні надаватися як вільно пов'язані веб-служби. Ця реалізація у формі веб-сервісів вимагає представлення та агрегування контекстної інформації у структурованому та стандартному форматі. У сучасних контекстно-обізнаних системах мови, засновані на XML, широко використовуються для моделювання та представлення контекстної інформації. Мова загального опису профілю (PPDL) – це мова, заснована на XML, яка представляє ситуативний контекст. Існують схеми XML для визначення контекстної інформації мобільних мереж, тобто стану пристрою та доступності.

Структура опису ресурсу (RDF) та мова веб-онтології (OWL) також широко прийняті для представлення контексту. Складений профіль можливостей / налаштувань (CC / PP) можна використовувати для опису

можливостей та вподобань користувачів. CC / PP базується на RDF. CC / PP є широко розповсюдженим прийнятим стандартом W3C, але він не вказує, як контекстна інформація може зберігатися. [8]

В даний час не існує систем обробки контексту, спрямованих на усвідомлення контексту, спеціально для мобільних роботів, заснованих на хмарних послугах. Ці системи націлені на декілька атрибутів контексту та допоміжні компоненти обробки контексту. У більшості згаданих систем мобільність, зберігання контексту та міркування за контекстом не забезпечуються.

Поінформованість про контекст – це механізм адаптації послуги. Адаптація послуги – це також складний процес, який передбачає швидкий аналіз різних контекстних даних та налаштування відповідних послуг. [13]

Проблеми перенесення обізнаності про контекст у хмару.

Мобільні хмарні обчислення – це модель прозорого еластичного збільшення можливостей мобільних пристроїв за допомогою повсюдного бездротового доступу до хмарних сховищ та обчислювальних ресурсів. Налаштування контекстного динамічного розвантаження здійснюється з урахуванням змін робочих умов, зберігаючи доступні можливості зондування та інтерактивності мобільних пристроїв. Мобільні пристрої мають вбудовані датчики, щоб відчувати реальну ситуацію користувачів. Таким чином, вони важливі для отримання контекстних даних, але розмір отриманих контекстних даних варіюється залежно від цілей програми. Цей обсяг даних може бути значним, напр. тривісний акселерометр смартфона, що використовується для розпізнавання активності, такого як підрахунок кроків або виявлення падіння, може генерувати 100 зразків в секунду, що даватиме близько 200 кБ в хвилину (одне значення типу double для кожної осі x, y, z та позначка часу, отримуючи 32 байти на зразок).

Надсилання всіх цих даних у хмару для вилучення та вибору статистичних ознак може бути надто незручним.. Часто не сама посада становить безпосередньо корисний контекст, а додаткова інформація, яку можна зробити з місця розташування, наприклад, користувач у залі засідань передбачає, що він перебуває на нараді. Акселерометри та гіроскопи надають інформацію про рух та орієнтацію. Камери аналогічним чином забезпечують доступ до потенційно багатої інформації, яку можна отримати за допомогою вилучення функцій та відеоаналізу для спостереження та сценаріїв життя.

Хоча позиційний контекст, контекст руху та візуальний контекст є потужними для збільшення пристроїв з певним усвідомленням їх ситуації, вони також мають чіткі недоліки. Позиція – це статичний опис середовища і не охоплює динамічні аспекти ситуації. Його корисність як контекст також значною мірою залежить від заздалегідь отриманих знань про місцезнаходження. Рух і зір, навпаки, можуть використовуватися для фіксації активності та інших динамічних аспектів, але вилучення конкретного контексту обчислювально дорого і проблематично в мобільних та неконтрольованих середовищах через брак ресурсів для обчислень, зберігання даних, пропускна здатність мережі та ємність акумулятора.

У цих випадках зберігання контексту та контекстне міркування часто надто непрактично, щоб розміщувати їх на мобільному пристрої. Деякі алгоритми, що використовуються для контекстних програм, такі як розпізнавання мови або обробка зображень, також вимагають інтенсивних обчислень і дають великий обсяг даних.

Крім того, нерозумно зберігати певні типи даних на персоналізованих роботах. Ці дані занадто великі, і зберігати їх на мобільних роботах неможливо. На відміну від мобільних пристроїв, хмарні обчислення надають великі можливості для зберігання та обробки. Мобільні додатки будуються за веб-стандартами, де обчислення завантажуються на потужні хмарні ресурси. Однак

іноді (наприклад, у випадках підрахунку кроків або виявлення падінь за допомогою акселерометра) обробка мобільних даних на місці пристрою буде більш ефективною та менш сприйнятливою до мережевих обмежень порівняно з розвантаженням даних та обробкою до віддаленої хмари.

Зв'язок, затримка та пропускна здатність.

При споживанні контекстно-залежних хмарних послуг на мобільному пристрої необхідна безперервна передача даних. У мобільному середовищі мережева підключеність варіюється від високої пропускної здатності до низької пропускної здатності відповідно до доступного мережевого підключення. Крім того, інтеграція даних з віддаленою обробкою вимагає переміщення локальних даних у віддалене місце з високою швидкістю завантаження, орієнтованою на зв’язок. Отже, навіть якщо обробка в хмарі є набагато ефективнішою порівняно з мобільним, затримка, викликана завантаженням даних, може бути викликом контекстно-відомим хмарним службам, що працюють у мобільному , особливо якщо ця затримка може змінюватися в пропускній здатності.

Безпека та конфіденційність.

Зберігання та використання контекстних даних для адаптації хмарних послуг, що надаються сторонніми постачальниками послуг, є часто обговорюваною проблемою безпеки в хмарних обчисленнях. Очевидно, що інфраструктура повинна бути захищена від несанкціонованого доступу, але вона також потребує способів дозволити людям самоаналізуватися, щоб вони могли зрозуміти, як використовуються контекстні дані. Крім того, інфраструктура повинна бути спроектована таким чином, щоб проблеми конфіденційності розглянуто законно та адекватно.

Рішення про розподіл для федеральних розгортань.

Найбільшою проблемою для федерального розгортання розподілених контекстних мобільних додатків у хмарі є вирішення, коли і що розвантажувати автономним способом. Проміжне програмне забезпечення, яке розуміє

контекст, має прийняти це рішення щодо розподілу відповідальності між пристроями, програмами та самим проміжним програмним забезпеченням. Він повинен мати можливість вирішити, для яких випадків використання хмара краща, а для яких хмарне розгортання не приносить ніякої доданої вартості.

Управління контекстним життєвим циклом служби.

Платформа як послуга (PaaS) має на меті абстрагувати складність управління життєвим циклом хмарних служб, надаючи цільові інструменти та послуги. Ультрасучасний PaaS пропонує такі загальнодоступні послуги, як автентифікація, авторизація, виставлення рахунків та надання послуг для декількох роботів як програмне забезпечення як послуга (SaaS). Ще однією основною проблемою для контекстно-обізнаних постачальників хмарних послуг є використання переваг хмарних обчислень для управління зобов’язаннями щодо якості обслуговування (QoS) перед клієнтами протягом життєвого циклу послуги. Контекст відіграє життєво важливу роль у життєвому циклі. [13]

Архітектура контекстних мобільних роботів розвивається у величезних масштабах. Вона еволюціонує так, як багато-до-багатьох, коли велика кількість користувачів може використовувати розподілене програмне забезпечення для обробки контексту асинхронно, і це для великої кількості персоналізованих послуг, розміщених сторонніми постачальниками послуг.

Не можна нехтувати роллю контекстної інформації, оскільки життєво важливо змусити всі служби працювати в гармонії та динамічно адаптуватися, щоб досягти інтелектуальної поведінки служб під час виконання масштабованим способом. Факторами, що сприяють цим взаємопов’язаним мобільним послугам, є контекстна розвідка, взаємодія, стандартизовані протоколи зв'язку та платформи для розгортання та ефективної адаптації цих послуг. Будь-яка архітектура програмного забезпечення, розроблена для контекстних мобільних додатків, об'єднаних із хмарними службами, повинна

передбачати зростаючу неоднорідність пристроїв, що захоплюють контекст. Він також повинен забезпечити надання послуг для різного контексту.

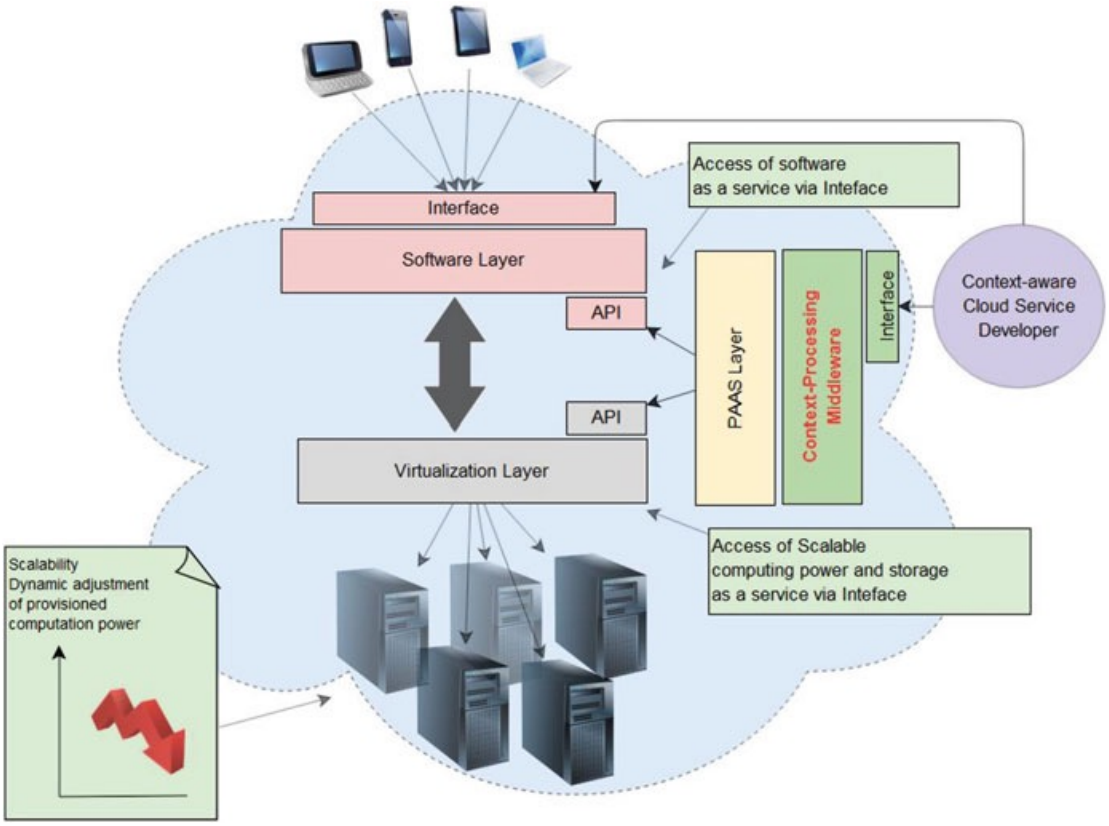


Рисунок 30. Обробка контексту в хмарній архітектурі

Контекстна обізнаність впливає на модель хмарних обчислень та взаємодію розробника хмарних служб, що знає контекст, із рівнями парадигми хмарних обчислень. Проміжне програмне забезпечення для обробки контексту показано як абстрагований рівень в Platform-as-a-Service для розробки хмарних служб, що знають про контекст, та їх динамічної адаптації. Platform-as-a-Service вже забезпечує надання послуг для масштабованості хмарних служб. Проміжне програмне забезпечення для обробки контексту може використовувати інфраструктуру надання послуг, додавши ще кілька служб для управління та адаптації послуг на основі різного контексту. [14]

Це полегшить розробку додатків з урахуванням контексту, обробку контексту та надання послуг під час виконання. Найбільша перевага такого типу

проміжного обладнання полягає в тому, що датчики, послуги та пристрої можна змінювати незалежно, навіть динамічно, коли працюють інші датчики, послуги, пристрої та програми. Інкапсульована обробка контексту збільшує можливість повторного використання та полегшує розробку великомасштабних розумних мобільних роботів.

Хмарні служби загального призначення не повинні бути тісно пов'язані з проміжним програмним забезпеченням для обробки контексту. Це дозволяє еволюціонувати управління контекстом у міру того, як стають доступними нові джерела інформації або зникають існуючі джерела.

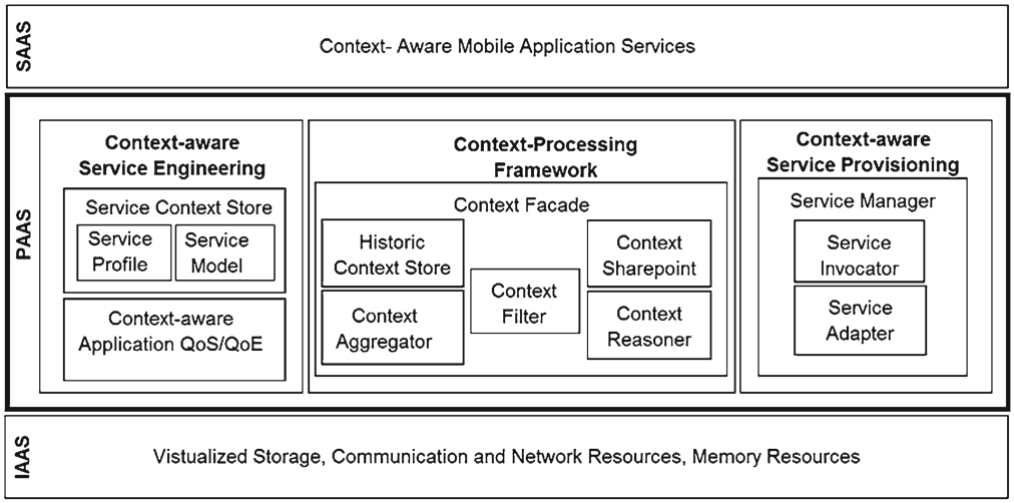


Рисунок 31. Архітектура проміжного програмного забезпечення для контекстної обробки та надання послуг з підтримкою контексту в PaaS

Окремий рівень абстракції менеджера послуг потрібен, щоб відповідати контексту користувача та послуг, на які користувач підписався. Менеджеру послуг потрібні розширені алгоритми адаптації для адаптації хмарних служб відповідно до зміни контексту користувача та його пристрою. Метод аналізу компромісу необхідний для розподілу рішення про те, як динамічно визначати, що і коли розвантажувати в хмару, не впливаючи на QoS та QoE. Користувачі повинні мати можливість додавати власні уподобання та вимоги щодо того, яка контекстна інформація та програми повинні залишатися локальними на

мобільному пристроями. Архітектура повинна бути відкритою та розширюваною для інтеграції нових компонентів управління контекстом, в той же час залишаючись масштабованою та еластичною для управління все більшою кількістю користувачів.

Інформаційна точка зору архітектури обробки контексту.

У контекстно-обізнаних додатках дані контексту широко розподіляються і, можливо, надходять з них і використовуються де завгодно незалежно від часу. Розумні додатки, засновані на контекстних хмарних послугах, відрізняються від традиційних додатків, оскільки життєвий цикл обробки інформації широко розподілений. Важливо, щоб інтерпретація контекстної інформації однаковою для кожної хмарної служби, що бере участь.

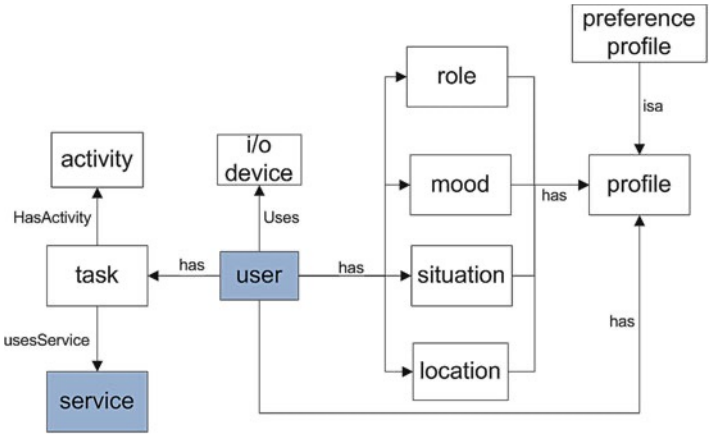


Рисунок 32. Онтологія контексту мобільно пристрою

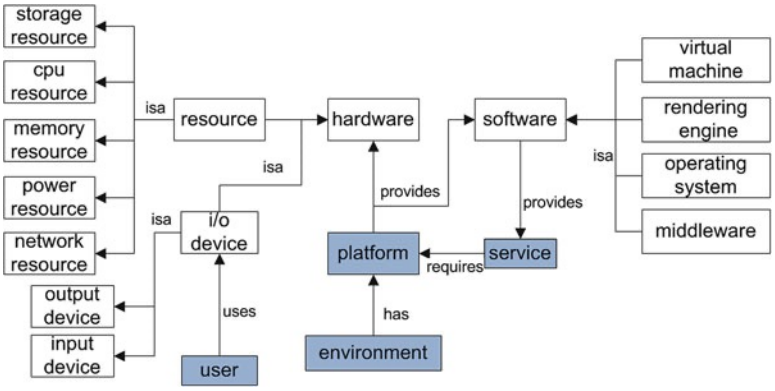


Рисунок 33. Онтологія контексту платформи

Три основні контекстні концепції в онтології:

1. Контекст мобільного робота: будь-яка інформація, що стосується робота хмарної служби, розглядається як контекст робота. Це може бути місце розташування, діяльність, рівень шуму, світло, температура тощо. На рис. 32. показано дизайн онтології для контексту мобільного робота.

2. Контекст платформи: інформація, що стосується мобільного пристрою, що є важливою для виконання послуги, класифікується як контекст пристрою, наприклад рівень заряду акумулятора, доступна пропускна здатність та роздільна здатність дисплея.

3. Контекст послуги: будь-яка інформація, пов’язана з послугою протягом її життєвого циклу, що стосується адаптації послуги, наприклад опис функціональних можливостей, інтерфейси та необхідні ресурси. Цей тип інформації є важливим для менеджера служби для прийняття рішення про розподіл.

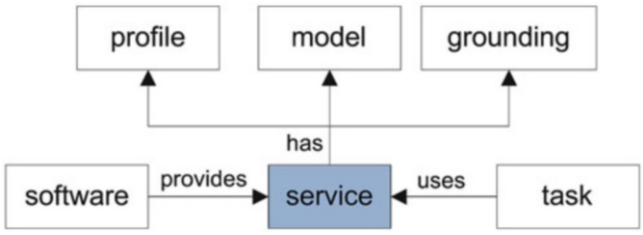


Рисунок 34. Онтологія контексту обслуговування

Розуміння контексту дозволяє автоматично адаптувати послугу. Це допомагає системі адаптувати свою поведінку відповідно до бажаних цілей, пропонуючи сумісні та масштабовані персоналізовані послуги. На рис. 35. представлена функціональна точка зору архітектурного дизайну, представленого на рис. 32 у шарі PaaS. Цей функціональний вигляд архітектури розроблений для задоволення функціональних вимог проміжного програмного забезпечення для обробки контексту. Він визначає ключові функціональні

елементи, їх обов'язки, інтерфейси, які вони виставляють, та взаємодію між ними. У PaaS вже існує механізм надання веб-сервісів і на ходу створення нових сервісів, які не порушують існуючу структуру. [14]

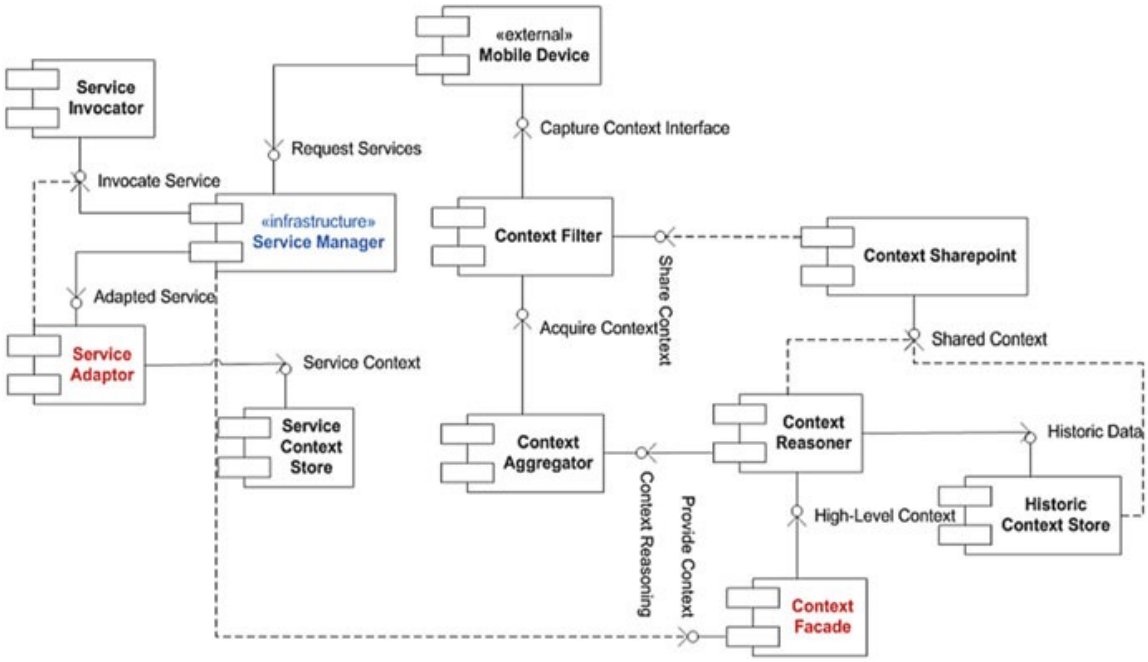


Рисунок 35. Функціональна структура

Мобільний робот запитує послугу до диспетчера служб у хмарі.

Якщо послуга не має контексту, менеджер служби викликає послугу за допомогою координатора служби виклику. Якщо служба знає контекст, менеджер служб зв'язується з компонентом проміжного програмного забезпечення, що обробляє контекст, під назвою Context Facade. Контекстний фасад діє як абстракція до інших компонентів, що обробляють контекст. Ця послуга надсилає запит на отримання контекстної інформації до засобу реалізації контексту. Служба міркувань за контекстом запитує у агрегатора контексту попередньо оброблені та історичні дані контексту. Він визначає запитуваний контекст і передає його на контекстний фасад. [8]

Послуга Context Sharepoint для отримання спільного контексту показана на цій схемі як додаткова послуга. Служба Context Sharepoint зв’язується зі службою Context Filter для забезпечення контексту кількох користувачів. Як

тільки контекст користувача визначається фасадним контекстом, менеджер послуг отримує доступ до контексту і зв'язується із сервісним адаптером, щоб викликати необхідну послугу, сумісну з поточним контекстом. Сервісний адаптер залежить від служби контекстного магазину. Магазин контексту послуг – це місце, де публікуються семантично визначені служби, і кожна служба має дійсний профіль контексту. Сервісний адаптер узгоджує профіль контексту послуги з контекстом користувача та відповідно налаштовує послугу. Менеджер служб викликає службу Сервісного адаптера, коли контекст змінюється.

Впровадження та оцінка WSO2 Stratos – відкрите рішення PaaS, для реалізації контекстно-орієнтованих компонентів як вільно пов'язаних служб та емпіричної оцінки. В процесі оцінки масштабованості проміжного програмного забезпечення для обробки контексту для контекстного мобільного роботу, використовуючи розпізнавання людської діяльності як склад декількох служб обробки контексту. Хоча присутні попередні результати дрібномасштабних експериментів, вони не надають жодної статистичної значущості для широкомасштабних розгортань із великою кількістю роботів. Для цього нам потрібні великі контекстні набори даних та екосистеми на бізнес-рівні для оцінки різних стратегій надання, щоб надати докази щодо оптимальних стратегій, що покращують якість якості споживачів.

Після того, як проміжне програмне забезпечення для обробки контексту буде повністю функціонувати, необхідно перехресно перевіряти ефективність наших стратегій адаптації, запускаючи ті самі інструментальні програми на існуючих інфраструктурах загальнодоступних хмар (наприклад, Amazon або Google App Engine).

Хмарні обчислення є будівельним елементом Інтернету майбутнього, а обізнаність про контекст є невід'ємною частиною Інтернету майбутнього. Це дає спільне бачення розгортання незалежних федеративних служб та додатків, що характеризуються високим ступенем автономного збору даних, передачі

подій, мережевого зв’язку та взаємодії. Додаткова цінність полягає в контекстних мобільних роботах, таких як моніторинг навколишнього середовища та моніторинг охорони здоров’я, де великі обсяги та швидкість передачі даних потребують швидкої обробки, щоб перетворити необроблені дані в контекстуальну інформацію. Мобільні додатки, що адаптуються до контексту, підтримуватимуть взаємодію між гетерогенними контекстними хмарними службами [9].

Контекстні хмарні служби та мобільний зв’язок дадуть початок новій взаємодії з хмарою, в якій інформація та комунікація вбудовуються в навколишнє середовище у вигляді бездротових мереж. В результаті надзвичайного збільшення сучасних мобільних пристроїв з підтримкою Інтернету та поширених бездротових та стільникових мереж передачі даних, велика кількість мобільних користувачів потребує послуг персоналізації відповідно до їхнього контексту. Наступна хвиля в епоху хмарних обчислень буде поза сферою традиційних хмарних сервісів та інфраструктури. Більшість хмарних служб будуть з урахуванням контексту.

У цьому розділі зосередили увагу на необхідності вільно зв'язаних компонентів, що підтримують контекст, які працюють з контекстною інфраструктурою надання послуг, щоб адаптувати послуги до контексту користувача та його мобільного пристрою. Прийняття проміжного програмного забезпечення для обробки контексту на основі PaaS дозволить пришвидшити розробку майбутніх мобільних хмарних рішень, абстрагуючи все управління контекстом у багаторазові служби, а також пришвидшити дослідження в області контекстних мобільних додатків для хмари .

3.2. Модель контролю доступу для віртуальних об’єктів як зв’язок для AWS IoT

Розробка архітектури орієнтована на управління доступом на основі AWS IoT

Інтернет речей (IoT) вимагає вирішення нових проблемами безпеки, які потребують значного перегляду та вдосконалення існуючих рішень безпеки, включаючи системи контролю доступу. Нещодавно розроблена архітектура, орієнтована на управління доступом (ACO – Access Control Oriented) для IoT із підтримкою хмар, яка включає чотири шари: [1]

- рівень об’єкта;
- рівень віртуального об’єкта (VO);
- рівень хмарних служб;
- рівень додатків (ACO визнає необхідність управління зв’язком у кожному рівні та між сусідніми шарами, а також необхідність контролю доступу до даних у хмарних службах та прикладних рівнях).

Архітектура ACO складається з чотирьох шарів: об’єктного, віртуального об’єктного рівня, рівня хмарних служб та рівня додатків, як показано на рис. 36.

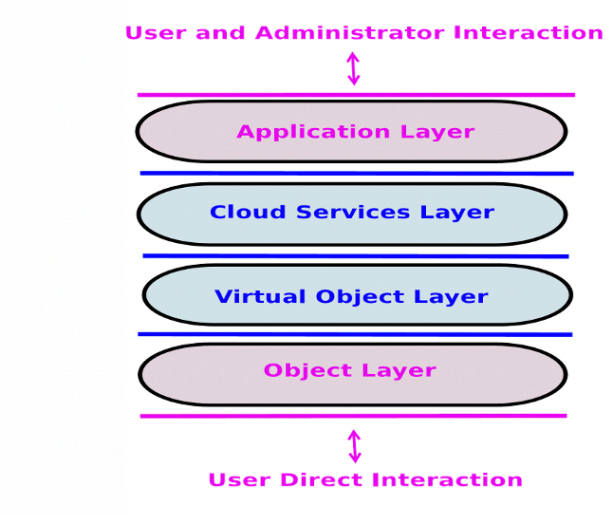


Рисунок 36. Архітектура ACO для IoT із підтримкою хмар

Об’єктний рівень включає неоднорідні фізичні об’єкти, такі як датчики, виконавчі механізми, камери тощо. Користувачі можуть безпосередньо спілкуватися з об’єктами, натискаючи кнопку, змінюючи пристрій, включаючи об’єкт тощо. Також об’єкти можуть спілкуватися безпосередньо один з одним за допомогою комунікаційних технологій або опосередковано через віртуальні об’єкти.

Віртуальний об’єкт представляє постійний поточний стан фізичного об’єкта, коли вони з’єднані. В іншому випадку віртуальний об’єкт може представляти останній отриманий стан, бажаний майбутній стан або обидва.

Віртуальний об’єкт може мати підмножину послуг фізичного об’єкта, усі послуги фізичного об’єкта або підмножину служб фізичного об’єкта. Віртуальні об’єкти можуть рівномірно спілкуватися один з одним незалежно від неоднорідності та локалізації в об’єктному шарі. Асоціація віртуального з фізичним об’єктом може бути один-на-один, багато-до-одного, один-до-багатьох або багато-до-одного.

Рівень хмарних служб допомагає зберігати та обробляти зібрані дані. Ці дані можна використовувати для інтелектуального моніторингу та опрацьовування, а також їх можна візуалізувати більш значущими для користувачів способами.

Крім того, багато хмар IoT також можуть спілкуватися між собою, починаючи від надання лише послуг та інформації на місцевому рівні до співпраці з іншими підключеними IoT, щоб обмінюватися інформацією на широкому рівні та виконувати спільні цілі.

Рівень програми є найвищим шаром запропонованої архітектури АСО IoT і пропонує інтерфейс, за допомогою якого користувачі можуть легко спілкуватися. Адміністратори також можуть взаємодіяти з програмами для створення політик або оновлення / додавання політик на основі отриманої інформації. Більше того, налаштування та управління зв’язком об’єктів та

віртуальних об’єктів організовується адміністраторами за допомогою додатків. Загальні користувачі та адміністратори можуть віддалено спілкуватися з об’єктами IoT та віртуальними об’єктами лише через програми.

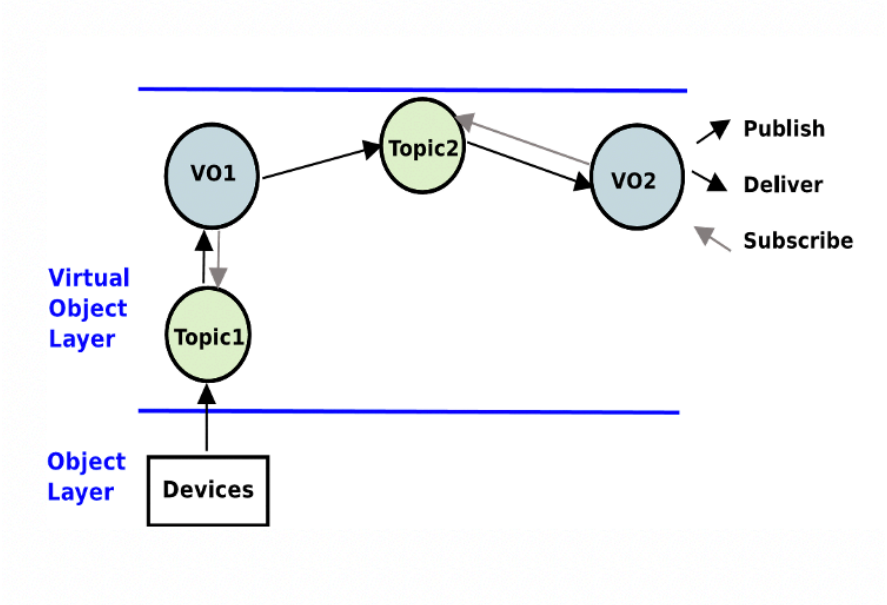


Рисунок 37. Схема публікації/підписки за темами в ACO-IoT-ACMsVO

Моделі контролю доступу для VO (віртуального об’єкту).

Архітектура ACO наголошує на необхідності контролю зв’язку та контролю доступу до даних всередині та між рівнями ACO. Поточні домінуючі моделі контролю доступу, а саме: списки контролю доступу (ACL), списки можливостей та рольовий контроль доступу (RBAC), формально визначені як в оперативній, так і в адміністративній моделях зв’язку VO. Також пропонуються моделі контролю доступу на основі атрибутів (ABAC), оскільки ABAC охоплює переваги попередніх традиційних моделей, а також пропонує нові функції, придатні для динамічних та відкритих середовищ, таких як IoT.[10]

Ця схема підходить для широкомасштабних розподілених взаємодій, таких як IoT. Основним стилем реалізації парадигми публікації/підписки є тематична схема. Тематична схема порівнянна з ідеєю груп, де виробники (видавці) публікують дані для теми, а споживачі (передплатники) стають

членами теми (групи). На рис. 38 продемонстровано загальну ідею публікації/підписки на основі теми, яка використовується в ACO-IoT-ACMsVO.

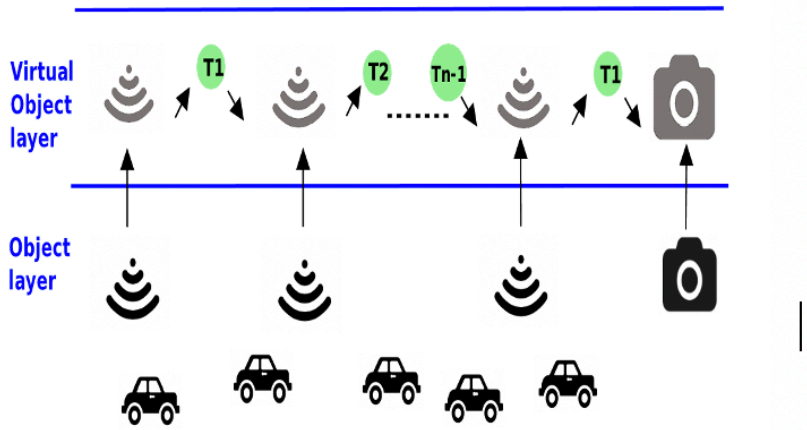


Рисунок 38. Схема керування в ACO архітектурі

На рис. 38 показано спрощене зображення застосованого випадку використання. Один автомобіль розпізнає швидкість, якщо два датчики на певній відстані відчувають швидкість, яка перевищує ліміт, і камера знімає фотографії автомобілів, що перевищують ліміт. Фізичні датчики та камера ідентифікують автомобілі за допомогою приєднаних RFID та передають зібрані дані (наприклад, RFID та Speed) до їх віртуальних об'єктів, де можуть відбуватися більш потужні обчислення та зв'язок. Варіант використання передбачає, що датчики можуть спілкуватися лише в межах віртуального об'єктного рівня, і вони не можуть взаємодіяти безпосередньо один з одним.

Загальна модель контролю доступу для AWS-IoT (AWS-IoTAC).

Бхатт та ін. вивчають AWS IoT як основну комерційну хмарну платформу IoT та AWS-IoTAC включає всі компоненти та відносини AWSAC із модифікованим або додатковим набором компонентів та відносини, пов'язані з послугою AWS IoT. Основним компонентом AWSAC є Облікові записи (A), Користувачі (U), Групи (G), Ролі (R), Послуги (S), Типи об'єктів (OT) та Операції (OP). Додатковими компонентами AWS-IoTAC, які пов'язані з

послугою AWS IoT, є сертифікати (C), об’єкти IoT (IO), операції IoT (IOP), правила (Ru), призначення віртуального дозволу (VPA) та пристрої (D).

Модель AWS-IOT-ACMVO для комунікації AWS IOT SHADOWS.

У цьому розділі, на основі нашого широкого вивчення платформи AWS IoT, її документації та реалізованих випадків використання, пропонується модель контролю доступу для спілкування віртуальних об’єктів під назвою AWS-IoT-ACMVO як абстрактний вигляд можливостей AWS IoT .

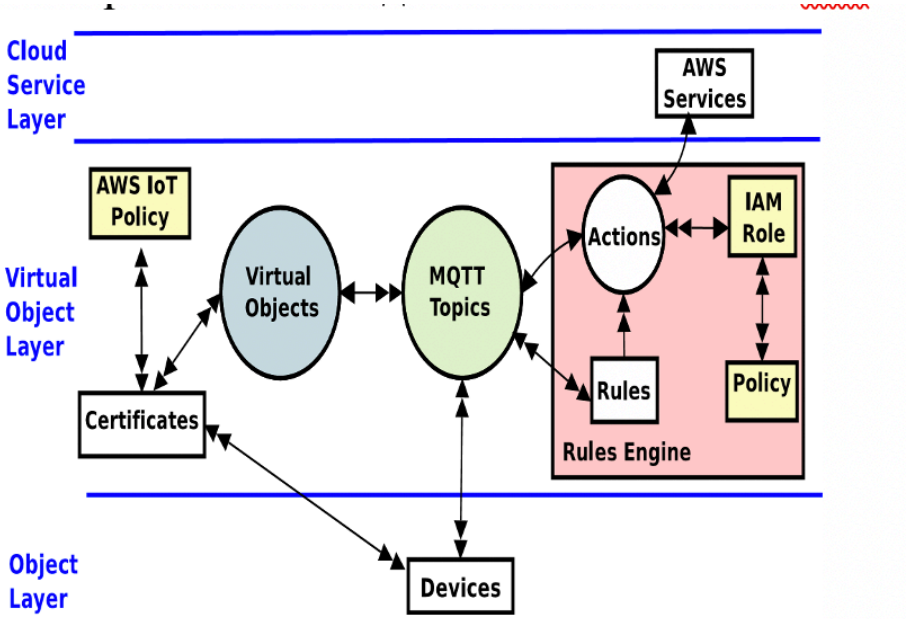


Рисунок 39. Компоненти AWS-IoT-ACMVO

На рис. 39 показані основні компоненти цієї моделі, а саме сертифікати, політики AWS IoT, віртуальні об’єкти , теми телеметрії транспорту в чергу масових повідомлень (MQTT), механізм правил та їх дії. Деталі їх функціональних можливостей обговорюються нижче.

AWS IoT використовує сертифікати X.509 як ідентифікаційні дані для автентифікації пристроїв. Сертифікати можуть бути або сертифікатом, створеним AWS IoT, або сертифікатом, підписаним зовнішнім органом сертифікації, зареєстрованим у AWS IoT. Як правило, один сертифікат може

бути наданий багатьом пристроям, але рекомендується, щоб кожен пристрій мав унікальний сертифікат, щоб забезпечити дрібне управління пристроями. На рис. 39 показано, що кожен сертифікат може бути наданий декільком пристроям, і кожен пристрій може мати кілька сертифікатів (стрілка з подвійним кінцем означає кратність). Однак кожного разу, коли пристрій підключається, він може активувати лише один сертифікат.

Після створення підключення сертифіката до сертифіката потрібно приєднати дві сутності AWS IoT, які дозволяють аутентифікувати та авторизувати пристрої AWS IoT, що бажають взаємодіяти з віртуальними об’єктами , а саме: політика AWS IoT та віртуальна об’єктів. Політика AWS IoT – це документ JSON, який додається до сертифіката з метою авторизації. Він містить одну або кілька заяв політики, кожна з яких визначає ефект, дію, ресурси та необов’язкові умови. Дія – це операція, яка може бути надана або відмовлена ресурсу відповідно до значення ефекту. Дії можуть бути діями політики MQTT або діями віртуальної політики. Дії політики MQTT – це операції, що стосуються підключення, надсилання або отримання даних, які є iot: Connect, iot: Publish, iot: Subscribe та iot: Receive.

З іншого боку, дії віртуально\ політики стосуються дозволів на обробку віртуальних об’єктів , які є iot: DeleteThingShadow, iot: GetThingShadow та iot: UpdateThingShadow. На рис. 39. продемонстровано, що кожна політика AWS IoT може бути приєднана до декількох сертифікатів, і кожен сертифікат може приєднати кілька політик AWS IoT. Як правило, політика AWS IoT додається до сертифіката, щоб дозволити будь-які дії (наприклад, iot: Publish та iot: GetThingShadow) для пристроїв, що містять цей сертифікат (та його приватний ключ).

Віртуальні об’єкти також потрібно приєднати до сертифіката як ресурс, до якого пристрій має повний або частковий доступ. Віртуальному об’єкту може бути надано більше одного сертифіката, а сертифікат може бути приєднаний до

більш ніж одного пристрою. На рис. 39 показано взаємозв’язок "багато-до-багатьох" між сертифікатами та віртуальними об’єктами. Віртуальний об’єкт – це також документ JSON, який зберігає інформацію про поточний стан підключеного пристрою та бажаний майбутній стан підключеного пристрою (останнього або історичного стану). Однією з переваг віртуального пристрою є те, що його інформацію можна використовувати для встановлення або отримання стану свого пристрою, навіть якщо пристрій не підключено. Як правило, пристрій, що має сертифікат із прикріпленими політиками та віртуальними об’єктами, має права на зв’язок та доступ до приєднаних віртуальних об’єктів (один віртуальний об’єкт при кожному з’єднанні) на основі прикріплених політик.

В AWS IoT програми не можуть безпосередньо оновлювати або отримувати дані пристроїв. Віртуальні об’єкти в AWS працюють як проміжна точка зв’язку між програмами та фізичними пристроями. Єдиний спосіб взаємодії додатків або пристроїв з віртуальним об’єктом – це спілкування з його темами MQTT. Іншими словами, теми MQTT віртуального об’єкта дозволяють програмам і пристроям отримувати, оновлювати або видаляти інформацію про стан віртуального об’єкта, публікуючи або підписуючись на теми MQTT. [3]

Назва кожної теми MQTT починаються з thingName – це ім’я віртуального об’єкта, а символ # може бути одна з тем thingName MQTT, які можуть бути використані для взаємодії з thingName. Для кожного віртуального об’єкта є зарезервовані теми MQTT thingName, які можна використовувати для публікації або передплати віртуального об’єкта. Для того, щоб надіслати запит на thingName (віртуальний об’єкт), можна використовувати update, get або delete як thingName MQTT теми цього thingName. Поки update, accept, update, reject, update, delta, update documents, get, accept, receive, reject delete MQTT теми

використовуються самим thingName для опублікування підтвердження про прийняття або відхилення отриманого запиту.

Як правило, служба AWS IoT генерує зарезервовані теми MQTT для кожного створеного віртуального об'єкта. Зарезервовані теми MQTT – це єдиний спосіб спілкування зі створеним віртуальним об'єктом. На рис. 39 показано, що кожен віртуальний об'єкт має певні зарезервовані теми MQTT, і кожна зарезервована тема MQTT пов'язана лише з одним віртуальним об'єктом. Більше того, кожен пристрій може спілкуватися з кількома темами MQTT, якщо він має авторизований сертифікат, і кожна тема MQTT може використовуватися більше ніж одним пристроєм, якщо пристрої авторизовані. [10]

Потужний механізм в AWS IoT полягає в тому, що повідомлення, надіслане темі MQTT, може розпізнаватися і аналізуватися правилом. Правила забезпечують обробку надходження повідомлень до тем MQTT і дозволяють взаємодіяти з різними службами AWS. Правило складається з імені правила, необов'язкового опису, оператора SQL, версії SQL та однієї або декількох дій. Оператор SQL використовується для фільтрації отриманих повідомлень до тем MQTT, а потім механізм правил пересилає його до служб AWS або повторно публікує до інших тем MQTT за допомогою поля дії, зазначеного в правилі. Є фіксовані дії AWS, які можна вибрати, наприклад, вставити повідомлення в таблицю DynamoDB, запустити функцію Lambda та повторно опублікувати повідомлення в темах AWS IoT. Таким чином, правила, прикріплені до тем MQTT, забезпечують способи взаємодії віртуальних об'єктів із службами AWS або повторної публікації отриманих повідомлень в інших темах MQTT (зарезервовані чи без резервування). На рис. 39 показано, що кожне правило може запускатися кількома темами, і кожна тема може запускати більше одного правила. Крім того, коли правило запускається, може бути виконана одна або кілька дій.

Коли правила пересилають опубліковані повідомлення до іншої служби AWS, наприклад AWS Lambda, авторизація доступу до іншої служби та дії іншої служби можуть контролюватися за допомогою ідентифікації AWS та управління доступом (IAM). До кожної ролі IAM додається принаймні одна політика, яка надає дозволи на доступ до ресурсів, зазначених у дії правила, або на керування діями щодо отриманих даних. Наприклад, коли створюється правило Amazon SNS, до цього правила SNS буде приєднана роль IAM, щоб дозволити доступ до ресурсів SNS.

Використання сенсорної швидкості автомобілів в AWS-IOT-ACMVO

Представимо два сценарії використання випадків зондування швидкості автомобілів. Два сценарії матимуть однакову кількість датчиків і камеру на фізичному рівні. Усі пристрої на фізичному рівні будуть надсилати зібрані дані до своїх віртуальних об’єктів. У сценаріях зосереджуємось на спілкуванні між віртуальними об’єктами та тому, як цим спілкуванням можна керувати.

У цьому простому сценарії матимемо два фізичних датчика та одну фізичну камеру, кожна з яких підключена до одного віртуального об’єкта. На рис. 40. показані підключені пристрої, віртуальні об’єкти , сертифікати, політика AWS IoT, правила, дії та їх ролі IAM та служби AWS.

По-перше, створено один віртуальний об’єкт для кожного фізичного об’єкта. В консоі управління AWS IoT приєднано один сертифікат X.509 для кожного віртуального об’єкта. До кожного сертифіката додано політику AWS IoT. Сертифікати копіюються у відповідні фізичні об’єкти, щоб дозволити автентифікацію та авторизацію фізичних об’єктів, коли вони спілкуються з відповідними віртуальними об’єктами. Іншими словами, додана політика AWS IoT дозволяє окремі дії (підключення та публікація) для фізичних об’єктів. Коли сертифікати надаються відповідним фізичним об’єктам, вони супроводжуються приватним ключем сертифіката та кореневим сертифікатом CA AWS.

Імітовано датчики та фізичні об’єкти камери за допомогою AWS SDK для JavaScript (Node.js). Існує прикладена правило для кожного MQTT поновлення теми AWS, яке запускає функцію Lambda. Функції лямбда відповідають за повторне оприлюднення повідомлених даних, що надійшли до віртуального об’єкта (Virtual Sensor або Virtual Camera) з відповідного фізичного об’єкта (Sensor або Camera) до наступного віртуального об’єкта (Virtual Sensor або Virtual Камера), як показано на рис. 40. Крім того, кожна функція лямбда приєднана до ролі IAM, яка дозволяє AWS IoT отримувати доступ до ресурсів та послуг AWS та AWS IoT. Роль IAM також контролює операції з функцією лямбда, наприклад, повторне опублікування даних в іншій темі або отримання поточного стану віртуального об’єкта.

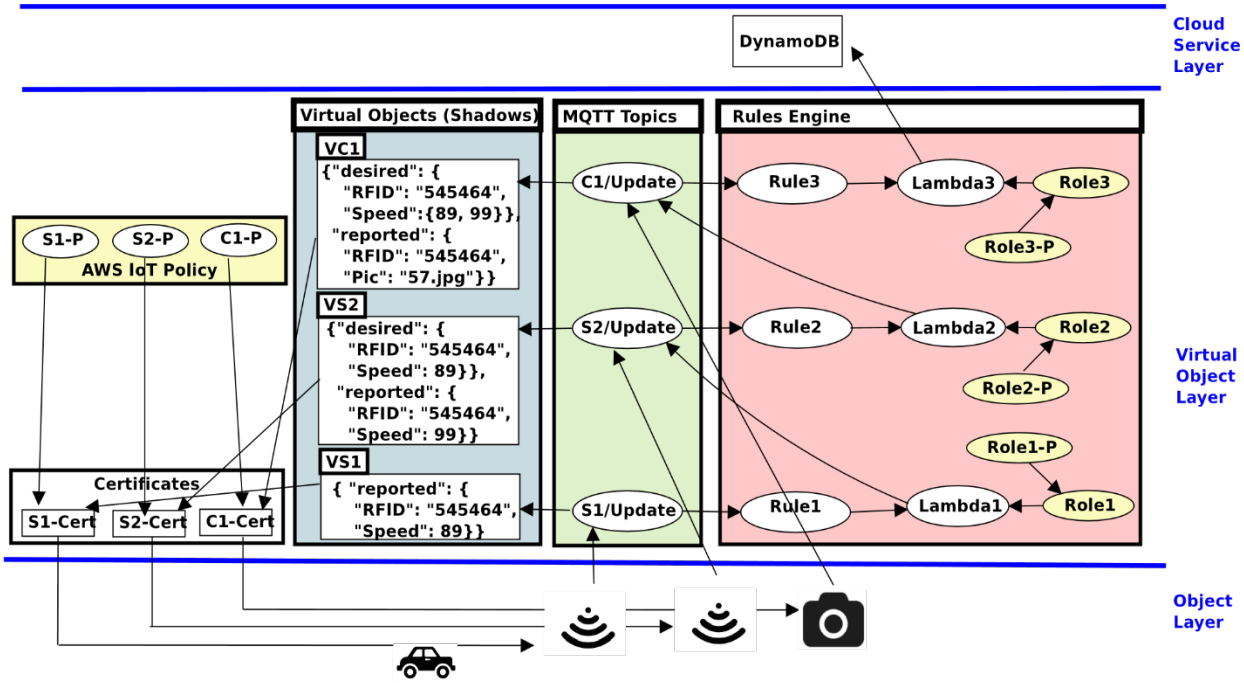


Рисунок 40. Простий приклад використання заміру швидкості одного роботизованого автомобіля

Датчик посилає RFID і швидкість автомобіля як повідомлення для віртуального датчика (VS 1) шляхом публікації теми в Sensor 1 MQTT. Правило 1, яке додається до теми оновлення MQTT Sensor 1, запускатиме функцію Lambda 1 кожного разу, коли дані надходять до теми

оновлення MQTT VS 1. Функція Lambda 1 повторно публікує отримані дані у Virtual Sensor (VS 2) із потрібним тегом. На рис. 40 показано, що повідомлена RFID і швидкість до VS 1 надсилається до VS 2 як бажаний стан за допомогою функції Lambda 1.

Датчик 2 також посилає RFID і швидкість автомобіля як повідомлення для VS 2 шляхом публікації в тему поновлення MQTT. Правило 2 буде запускати функцію Lambda 2 кожного разу, коли дані надходять до теми оновлення MQTT VS 2. Функція Lambda 2 перевіряє, чи надходять дані з тегом, про який повідомляється, вона порівнює збережений бажаний RFID із майбутнім повідомленим RFID від Sensor 2. Якщо обидва RFID збігаються, функція Lambda 2 поєднує дві швидкості та один RFID та публікує його з потрібним тегом на віртуальній камері (VC 1). [11]

На рис. 40. показано, що повідомлений RFID відповідає бажаному RFID у віртуальному датчику 2. Таким чином, VC 1 отримає від функції Lambda 2 дві швидкості, які повідомляються від датчика 1 і датчика 2 для одного і того ж RFID.

Камера також посилає RFIDs і фотографії про пропущені автомобілі, як повідомлення для віртуальної камери шляхом публікації в MQTT. Правило 3 збирається активувати функцію Lambda 3 кожного разу, коли дані надходять у тему оновлення MQTT VC 1. Функція Lambda 3перевіряє, чи надходять дані з тегом, про який повідомляється, він порівнює збережений прийдешній бажаний RFID від датчика 2 з майбутнім повідомленим RFID з камери. Якщо обидва RFID збігаються, функція Lambda 3 поєднує RFID, швидкості та зберігає їх у Amazon DynamoDB. На рис. 40. показано, що повідомлений RFID відповідає бажаному RFID у VC 1. Таким чином, об’єднані RFID, швидкості будуть зберігатися в Amazon DynamoDB.

Політика авторизації. До кожного сертифіката додається політика AWS IoT, щоб санкціонувати певні дії щодо фізичних об’єктів. Наприклад, датчику 1

дозволяється підключатись та публікувати на VS 1 лише для того, щоб надіслати зібрані RFID та швидкість швидкісних автомобілів. Таким чином, AWS IoT S 1– P та VS 1 прикріплені сертифікатом S 1, який копіюється до датчика 1. У політиці зазначено, що підключення та публікація дій дозволяються до зазначених ресурсів, тобто VS 1. Подібним чином AWS IoT S 2– P на рис. 40 та VS 2 будуть приєднані до S 2– Cert, який копіюється до датчика 2, а AWS IoT C 1– P та VC 1 будуть приєднані до C 1– Cert, який копіюється в Камеру 1. AWS IoT визначає змінні політики, які можна використовувати в політиках AWS IoT у блоці ресурсу або стану.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {  "Effect": "Allow",
       "Action": "iot:GetThingShadow",
       "Resource": "arn:aws:iot:us-west-2:760000000000:
        thing/Sensor2"
    },
    {  "Effect": "Allow",
       "Action": "iot:Publish",
       "Resource": "arn:aws:iot:us-west-2:760000000000:
        topic/$aws/things/Camera/shadow/update"
    }
  ]
}
```

Рисунок 41. Політика ролі 1, яка додається до ролі 2

Отже, розглянуто AWS IoT та розроблено модель контролю доступу для спілкування віртуальних об’єктів у AWS IoT (AWS-IoT-ACMVO). Використано AWS-IoT-ACMVO для реалізації двох сценаріїв випадку використання, який застосовується в ACO-IoT-ACMsVO: простий варіант використання, що визначає швидкість одного автомобіля двома датчиками, і варіант використання, що визначає швидкість декількох автомобілів з декількома датчиками. Реалізуючи ці два сценарії за допомогою ACOIoT-ACMsVO,

визначено як налаштувати політики та керувати зв'язком віртуальних об'єктів запропонованої нами моделі. Час на поширення інформації про підозрілі машини та перевищення ліміту автомобілів через усі віртуальні об'єкти вимірюється та обговорюється. Нарешті, під час вивчення та впровадження, запропоновано обговорення питань AWS IoT та пропозиції щодо вдосконалення комунікації VO та контролю їх доступу.

3.3. Побудова хмарної архітектури на основі подієвих моделей (Event Drive Development)

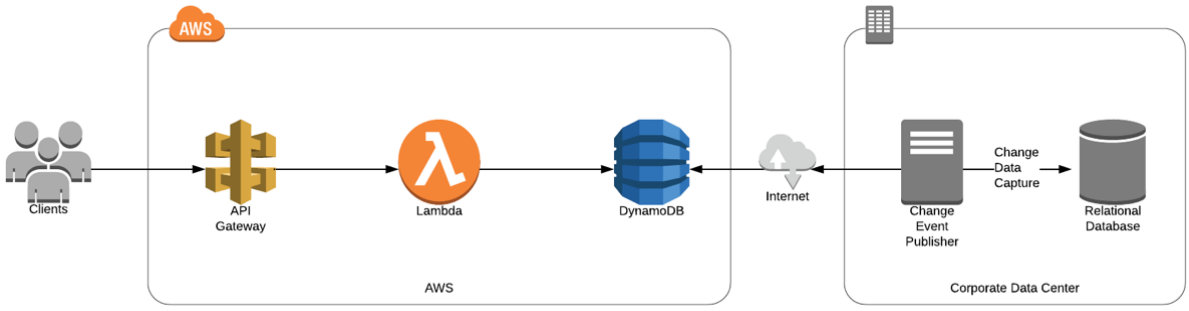


Рисунок 42. Компоненти шаблону синхронізації даних

На рис. 42 показано компоненти шаблону синхронізації даних на основі подій.

- Локальна реляційна база даних: зберігає частини даних, до яких додаток повинен отримати доступ.
- Локальний видавець подій змін: створює подію змін кожного разу, коли хтось створює, оновлює або видаляє відповідний рядок у реляційній базі даних.
- Підключення до Інтернету: використовується для транспортування подій змін від корпоративного центру обробки даних до AWS.
- DynamoDB: база даних NoSQL зберігає дані, що стосуються програми.

- Lambda: запускає код, який отримує доступ до даних, що зберігаються на DynamoDB.
- Шлюз API: точка входу у програму, що дозволяє авторизацію та перевірку вхідних запитів.[3]

Між старим світом, застарілими програмами та локальною інфраструктурою, а також новим світом, додатком, який поверх AWS, існує лише обмежений зв'язок. Описаний підхід має наступні переваги:

- Висока доступність: усі частини програми та інфраструктура розподілені між кількома машинами в декількох зонах доступності, що веде до високо доступної системи. Кожна година простою застарілої програми або корпоративного центру обробки даних не впливатиме на доступність додатка в хмарі.
- Масштабованість: усі частини хмарної інфраструктури можуть масштабуватися автоматично. Ні застаріла програма, ні інфраструктура корпоративного центру обробки даних не додають вузького місця до архітектури.
- Ціни з оплатою за використання: всі хмарні ресурси виставляються за користування.

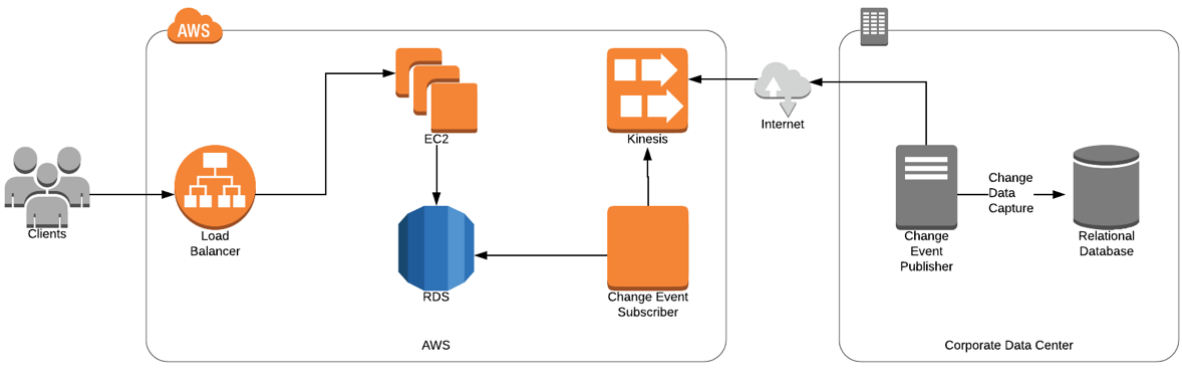


Рисунок 43. Компоненти шаблону синхронізації даних

Компоненти шаблону синхронізації даних.

- Локальна реляційна база даних: зберігає частини даних, до яких додаток повинен отримати доступ.
- Локальний видавець подій змін: створює подію змін кожного разу, коли хтось створює, оновлює або видаляє відповідний рядок у реляційній базі даних.
- Підключення до Інтернету: використовується для транспортування подій змін від корпоративного центру обробки даних до AWS.
- Kinesis: масштабований та керований потік, наданий AWS.
- Змінити абонента події: невелика послуга, яка підписується на потік даних Kinesis і перетворює кожну подію на оператор CREATE, UPDATE або DELETE.
- RDS Aurora: сумісна з MySQL або Postgres реляційна база даних, що зберігає дані програми.
- Екземпляри EC2: додаток працює на парку віртуальних машин.
- Load Balancer: точка входу у вашу інфраструктуру, що розподіляє запити між парком екземплярів EC2. [12]

3.4. Хмарна паралельна реалізація SLAM для мобільних роботів

Одночасна локалізація та картографування (SLAM – simultaneous localization and mapping) для мобільних роботів є обчислювально дорогим завданням. Робот здатний до SLAM потребує потужного бортового комп'ютера, але це може обмежити рухливість робота через вагу та вимоги до керування. Розглядаємо можливість перенести це завдання на пульт дистанційного керування обчислювальної хмару, пропонуючи загальну хмарну архітектуру для обчислень робототехніки в режимі реального часу, а потім впроваджувати SLAM на основі Rao-Blackwellized Particle Filtering алгоритму у багатовузловому кластері в хмарі. У реалізації дорогі обчислення виконуються паралельно, даючи значні покращення в часі обчислень. [12]

Це дозволяє алгоритму збільшити складність і частоту обчислень, підвищення точності отриманої карти, звільняючи при цьому бортовий комп'ютер робота від інших завдань. Метод для здійснення фільтрації частинок в хмарі не є специфічним для SLAM і може застосовуватися до інших обчислювальних завдань.

Паралельні реалізації алгоритмів робототехніки в режимі реального часу переважно працюють на багатоядерних машинах, що використовують потоки як основний механізм розпаралелювання, але це обмежує паралельність за кількістю ядер процесора і обсягом пам'яті в одній машині. Цей ступінь паралелізму часто не є достатньо, щоб забезпечити обчислювально дорогі алгоритми відповіді в реальному часі. Паралельні обчислення в розподіленому навколишньому середовищі може дати економічно вигідний варіант забезпечивши низький рівень латентності, а також масштабування вгору або вниз залежно від вимог до обробки.

Одночасна локалізація та картографування (SLAM) є важливою проблемою, при якій мобільний робот намагається оцінити як карта невідомого середовища, так і його положення в цьому середовищі дають недосконалі датчики з помилкою вимірювання. Ця проблема є обчислювально складною і широко вивчалася в літературі. Розглянемо популярний алгоритм GMapping, який будує карту сітки шляхом поєднання (шумних) вимірювань відстані від лазерного далекоміра та робота-одометра з використанням фільтрації частинок RaoBlackwellized (RBPF). Алгоритм є обчислювально дорогим і дає кращі результати, якщо доступні більше обчислювальних ресурсів.

У реалізації GMapping надсилаються лазерні зразки та показники одометра від робота як потоку подій до внутрішньої хмари, де вони обробляються SLAM і результати повертаються до роботів негайно. Алгоритм працює в повністю розподіленому середовищі, де можна запускати різні деталі різних машин, користуючись перевагами паралелізму для розколу до дорогих обчислень.

Головний внесок полягає в новій структурі для обчислення алгоритмів на основі фільтрації частинок, зокрема RBPF SLAM, в хмарному середовищі для досягнення високої ефективності часу обчислень.

IoTCloud – це розроблений фреймворк з відкритим кодом для підключенняпристроїв IoT до хмарних служб. [4] Як показано на рис. 44, він складається з набору розподілених вузлів, що працюють близько до пристроїв для збору даних, набір брокерів, що видають-підписуються для передачі інформації хмарних служб та розподіленої системи обробки потоків (DSPF) у поєднанні з механізмами пакетної обробки в хмарі для обробки даних та повернення (контролю) інформації на пристрої IoT. Додатки виконують аналіз даних на рівень DSPF, забезпечуючи потокову обробку в режимі реального часу RabbitMQ або Kafka в якості посередника повідомлень, а також хмара OpenStack як платформа. Використано службу координації та виявлення на ZooKeeper для масштабування кількості пристроїв.

Загалом, програма в режимі реального часу, що працює в DSPF, може бути змодельована як спрямований графік з потоками, що визначають ребра та завдання на обробку та вузли. Потік є необмеженою послідовність подій, що протікають за графіком, і кожна подія складається з масиву даних. [15]

Завдання обробки на вузлах вирішують вхідні потоки і виробляють вихідні потоки. DSPF забезпечує необхідний API та інфраструктуру для розробки та виконання програм на кластері вузлів. Щоб підключити пристрій до хмарних служб, користувач розробляє додаток шлюзу, який підключається до потоку даних пристрою. Після розгортання програми в шлюз IoTCloud, хмарні програми виявляють їх додатки та підключатися до них для обробки даних за допомогою служба виявлення.

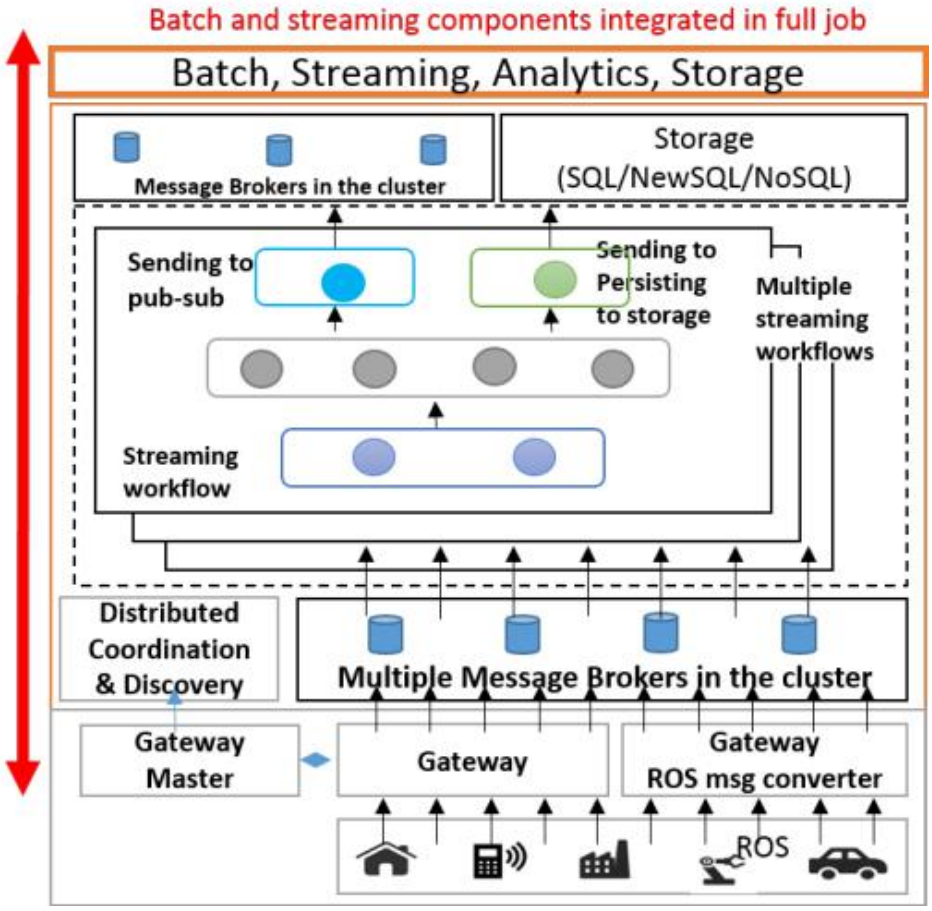


Рисунок 44. Архітектура IoTCloud

Алгоритм RBPF SLAM. Припустимо, у нас є ряд лазерних показань. Після оновлення кожної частинки алгоритм нормалізує ваги всіх частинок на основі загальної суми в квадраті ваги, а потім проводить повторну вибірку, витягуючи частинки із заміною з імовірністю, пропорційною вагам. Зразки часток використовуються з наступним зчитуванням. На кожному зчитуванні алгоритму приймається карту, пов'язана з частинкою найбільшої ваги. Час роботи алгоритму залежить від кількості частинок. Взагалі точність алгоритму покращується, якщо більше частинок використовуються. Виявлено, що RBPF SLAM витрачає майже 98% коштів час обчислення на перевірку співпадінь. Оскільки перевірка співпадінь робиться для кожної частинки незалежно, частинки можуть бути розділені на різні вузли обчислень і обчислюватись паралельно. Однак крок дискретизації вимагає інформації про всі частинки та

його потрібно виконувати послідовно, після збору результатів з паралельних обчислень. Дискретизація також видаляє і дублює деякі частинки, а це означає, що деякі частинки повинні бути перерозподілені по різних вузлах після дискретизації.

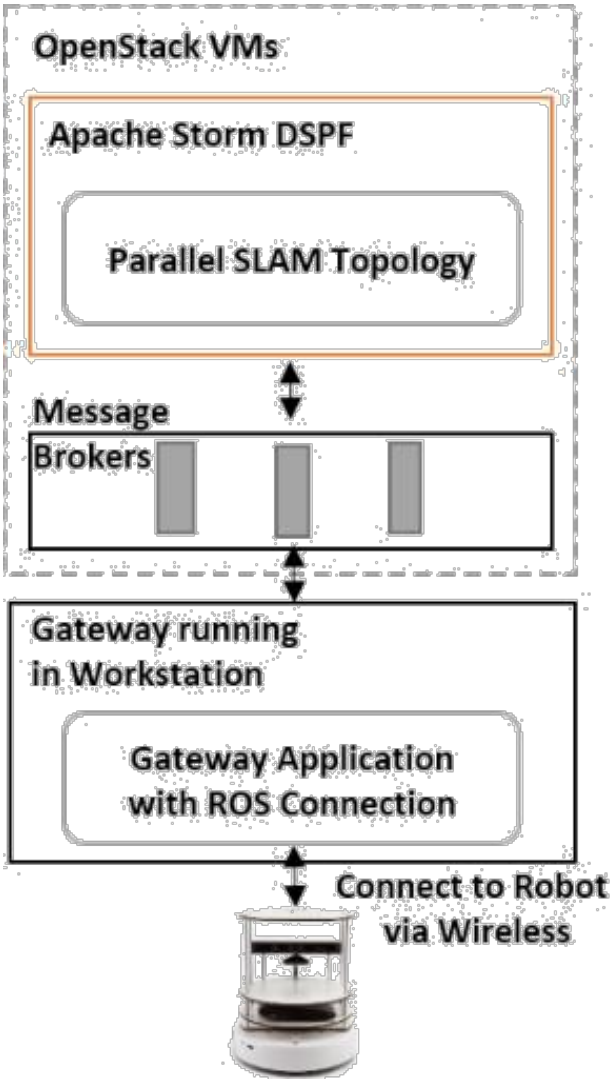


Рисунок 45. Додаток GMapping Robotics

Потік робочого процесу алгоритму показаний на рис. 45, реалізованого як топологію Apache Storm. Топологія визначає графік потоку даних програми з реалізацією завдань на базі Java на вузлах та лініях зв'язку, що визначають краї.

Основні завдання алгоритму поділяються на ScanMatcherBolt і ReSamplingBolt. LaserScanBolt отримує дані від робота і надсилає його решті програми. Після обчислення результати передаються в SendOut, які надішлють його назад роботові. За потреби дані можна зберегти в постійне зберігання.

```
   input  : pose u and laser reading z
   output: bestPose and l
 1 steps ← 0; l ← −∞; bestPose ← u; delta ← InitDelta;
 2 currentL ← likelihood(u, z);
 3 for i ← 1 to nRefinements do
 4 |   delta ← delta/2;
 5 |   repeat
 6 |   |   pose ← bestPose; l ← currentL;
 7 |   |   for d ← 1 to K do
 8 |   |   |   xd ← deterministicsample(pose, delta);
 9 |   |   |   localL ← likelihood(xd, z);
10 |   |   |   steps+ = 1;
11 |   |   |   if currentL < localL then
12 |   |   |   |   currentL ← localL; bestPose ← xd;
13 |   |   |   end
14 |   |   end
15 |   until l < currentL and steps < cutoff;
16 end
```

Рисунок 46. Алгоритм 1: Збіг сканування

Виявлено, що RBPF SLAM витрачає майже 98% своїх коштів на час обчислення та перевірку співпадінь. Оскільки перевірка співпадінь робиться для кожної частинки незалежно, розподілено.

Ключовою ідеєю реалізації є розповсюдження частинки в наборі завдань, що виконуються паралельно. Це специфічний для частинок код інкапсульований у ScanMatcher, тому можна контролювати паралельність алгоритму за допомогою змінм кількості екземплярів ScanMatcher.

Сервіс для дискретизації повинен зачекати, поки не отримає результати результати сервісу ScanMatcher. Після того, як відбудеться дискретизація, алгоритм видаляє деякі існуючі частинки та дублює інші. Всі дані, що

протікають через різні канали зв'язку мають байтовий формат, серіалізований Kryo.

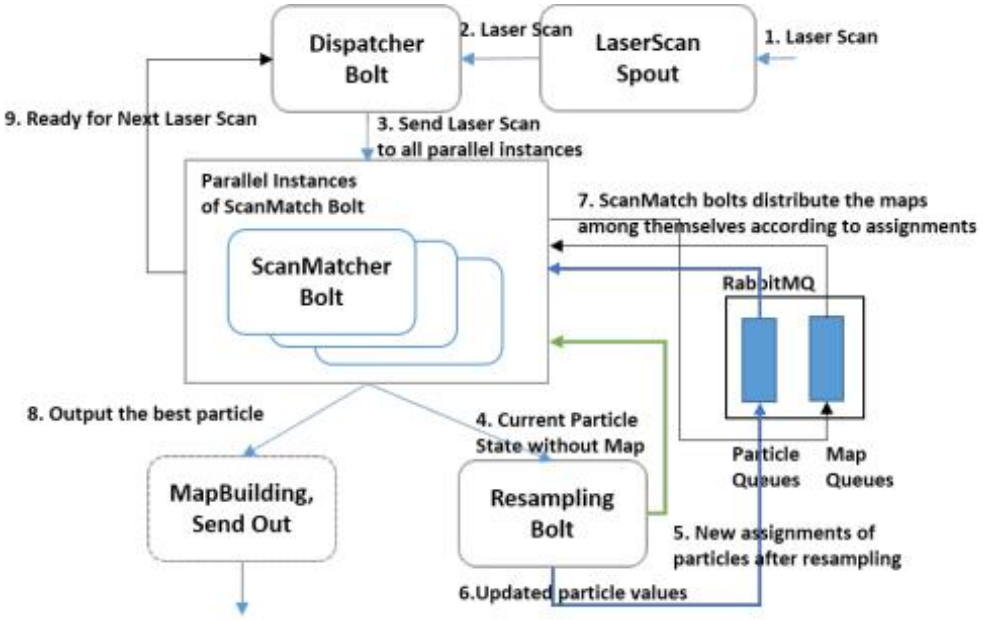


Рисунок 47. Робочий процес потокового алгоритму для паралельного RBPF

Етапи для одного читання здійснюються, як показано в рис. 47:

1. Сервіс LaserScan отримує показання лазера та одометра через посередника повідомлень.
2. Зчитана інформація надсилається диспетчеру, який транслює це до паралельних завдань.
3. Кожне завдання ScanMatcher отримує лазерне зчитування, оновлює призначені частинки та надсилає оновлені значення на сервіс дискретизації.
4. Після повторної вибірки сервіс для повторної вибірки обчислює нове призначення частинок для ScanMatchers, використовуючи алгоритм врахування витрат.
5. Паралельно кроку 4, сервіс для дискретизації посилає значення часток до нових пунктів призначення.

6. Після того, як ScanMatchers отримають нові дані, вони розподіляють карти, пов’язані з перевибраними частинками, за правильними пунктами призначення, використовуючи черги RabbitMQ та надсилають повідомлення безпосередньо завданням.
7. ScanMatcher з найкращими частинками видає свої значення та карту.
8. Сервіси ScanMatcher надсилають повідомлення диспетчеру, вказуючи на свою готовність прийняти наступне читання.

Продемонстровано як розвантажити обробку даних робототехніки до хмари через загальну структуру паралельних обчислень в реальному часі, і результати показують значний приріст в продуктивності. Оскільки алгоритм працює на розподіленій хмарі, вона має доступ до потенційно необмеженої пам'яті та потужності процесора. Це дозволяє системі добре масштабуватись і може будувати карти у великих, складних середовищах , що потребують великої кількості частинок або щільних лазерних показань.

Зменшення складності програмування також є цікавим напрямком. Сучасні механізми обробки розподілених потоків надають API низького рівня, що робить розробку складних програм IoT досить складною. У майбутній роботі слід запропонувати API вищого рівня для обробки складних взаємодій шляхом абстрагування деталі. В даний час розподіл стану між паралельними обчисленнями вимагає третього вузла, такого як зовнішній брокер або потокове завдання, що виступає посередником. API групового спілкування між паралельними завдання та гідним доповнення до DSPF.

3.5. Системи безперервного розгортання

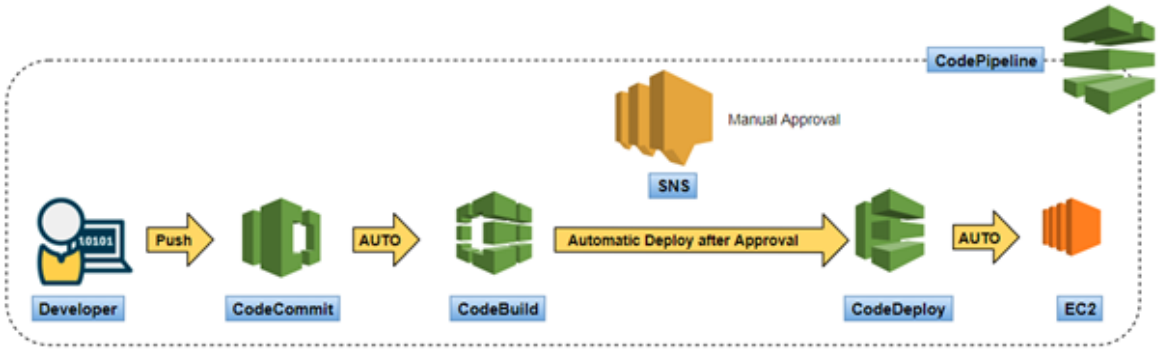


Рисунок 48. Схема безперервного розгортання

На платформі AWS створено сервіси, розміщені на сервері EC2, де використано всі служби AWS. Короткий короткий опис того, що важливо у виборі правильних інструментів для впровадження CI / CD:[4]

Безпека – Першим кроком у будь-якій реалізації має бути безпека. Колективна архітектура програм, апаратне забезпечення, мережа та дані повинні сприяти побудові безпечного середовища.

Архітектура повинна допускати кілька одночасних реалізацій. У той же час, це повинно дозволити ретельне тестування збірки додатків бути першим правильним джерелом (FTR). Для досягнення цих цілей найкраще підходить архітектура мікро сервісу. Однак різні програми можуть мати різні архітектури, що підходять. Такі як архітектура, орієнтована на послуги (SOA), та архітектура Lambda. Зрештою, основна увага повинна бути спрямована на досягнення цілей.

Використано сервіси, наведені нижче, для побудови прикладу конвеєру CI / CD.

- Репозиторій керування джерелом: AWS CodeCommit.
- Збірка: AWS CodeBuild.
- Розгортання: AWS CodeDeploy.
- Повідомлення: AWS SNS.

- Хостинг веб-сервера: EC2 AMI.
- CI / CD Pipeline: AWS Code Pipeline.

Етап 1: Репозиторій керування джерелами – AWS CodeCommit

Фіксація коду AWS без серверів. Дані за замовчуванням зашифровані в стані спокою та мають ввімкнені кінцеві точки SSH або HTTPS для передачі даних. Він пропонує довговічність даних, доступність даних, автоматичне масштабування, а більш важливі дані зберігаються подалі від обчислювальних ресурсів. У той же час це досить недорого.

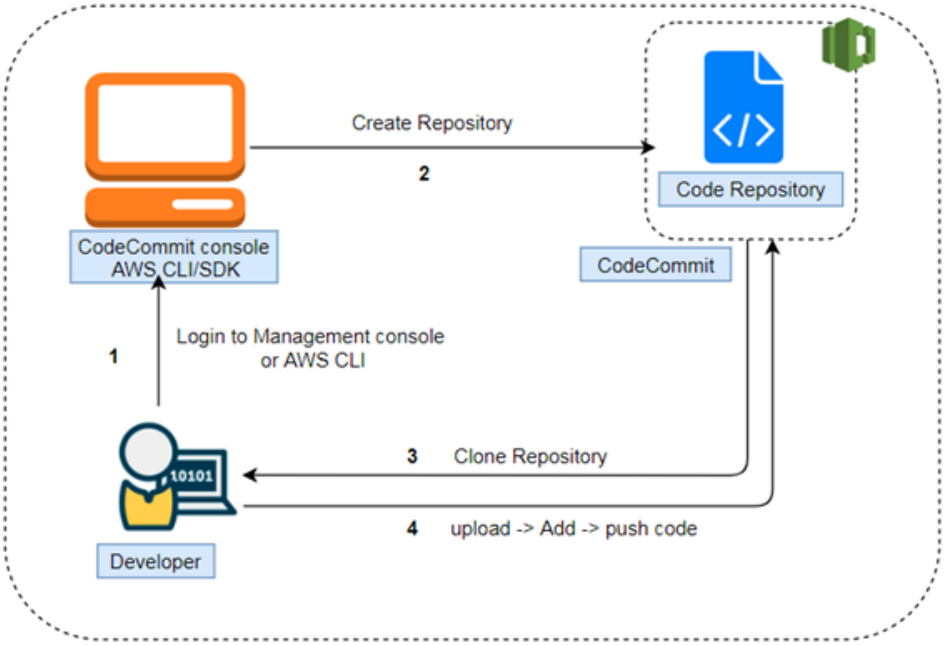


Рисунок 49. CodeCommit Flow

Встановимо репозиторій коду на AWS, передамо код з локальної машини до сховища комітів. Створити сховище – слід увійти на консоль управління AWS і виконати пошук коміту. На домашній сторінці натиснути «Створити сховище». Заповнити деталі нижче та натиснути «Створити».

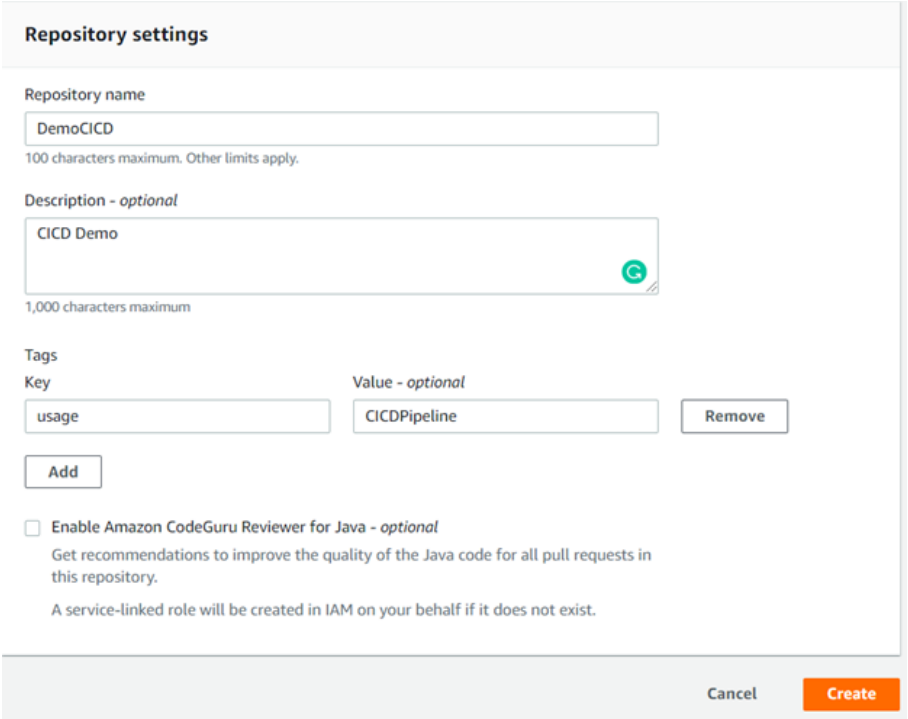


Рисунок 50. Конфігурація репозиторію

Можна використовувати консоль CodeCommit, AWS CLI або клієнт GIT для надсилання коду до CodeCommit. В даному випадку використано GIT bash. Встановимо клієнт GIT, якщо це ще не зроблено, дотримуючись інструкцій на офіційному веб-сайті.

Етап 2. Клонувати сховище – у консолі AWS необхідно обрати «Клонувати URL» у верхньому правому куті сторінки, а потім «Клонувати HTTPS». Адреса для клонування сховища GIT копіюється у буфер обміну. Слід відкрити GIT bash і ввести команду GIT clone з подальшою URL-адресою, скопійованою раніше. Вперше буде запропоновано облікові дані для підключення до AWS через GIT.

Етап 3. Завантаження коду до сховища – необхідно завантажити зразок коду зі сховища Github. Як найкращу практику слід створити гілку та об’єднати код у гілку. Необхідно перейти до поточного каталогу та використати наведену команду нижче, щоб додати нові файли. Потім потрібно перейти до поточного каталогу та використати команду нижче, щоб додати нові файли.

Необхідно здійснити перевірку сховища в AWS, усі файли слід скопіювати. appspec.yml містить налаштування CodeDeploy, buildspec.yml містить налаштування CodeBuild.

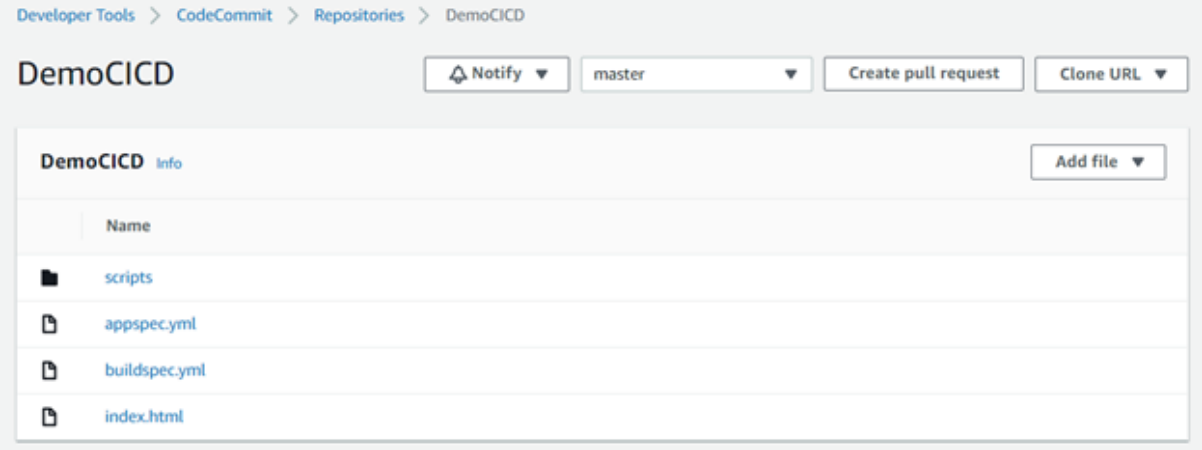


Рисунок 51. Codebuild налаштування

Передумова для побудови є екземпляр EC2 для розміщення програми та визначення агента розгортання коду. Запустимо екземпляр EC2 Linux для розміщення сервісу. Потім розгорнемо агент AWS CodeDeploy, щоб розгорнути артефакт збірки на машині. Перед запуском екземпляра слід створити роль IAM, щоб призначити правильний доступ.

Порядок створення екземпляра ролі.

- Відкрити консоль IAM.
- На інформаційній панелі консолі вибрати ролі.
- У розділі вибір типу довіреної сутності обрати службу AWS.
- Знайти і обрати політику з назвою AmazonEC2RoleforAWSCodeDeploy, а потім вибрати далі – теги.
- Обрати огляд. Ввести ім'я ролі (наприклад, DemoCICDEC2InstanceRole), а потім обрати створення ролі.

Запуск екземпляра EC2:

- Відкрити консоль Amazon EC2.
- На інформаційній панелі консолі запустити екземпляр.

- Обрати сервер Amazon Machine (AMI), знайти Amazon Linux 2 AMI (HVM), тип обсягу твердотільного накопичувача.
- Обрати сервер типу t2.micro (доступний безкоштовний рівень), а потім вибрати налаштування деталей екземпляра.
- Налаштувати деталі екземпляра виконавши такі дії: в автоматичному присвоєнні загальнодоступної IP-адреси вибрати увімкнення, а у ролі IAM – роль IAM, яку створено в попередній процедурі (наприклад, DemoCICDEC2InstanceRole).
- Створити тег із ключем та значенням як DemoCICD. За допомогою цих тегів визначимо екземпляр EC2 для розгортання.
- У розділі налаштування груп безпеки створити групи захисту як «SG-EC2-CICD» з типом порту SSH (для доступу до екземпляра за допомогою терміналу) та HTTP (для доступу до веб сервісів) та з джерелом «My IP». Здійснити запуск групи безпеки.

Служба AWS CodeBuild – це повністю керована служба, яка компілює код, запускає тести та створює програмний пакет, готовий до розгортання. Причиною вибору цієї послуги є:

- Без сервера – не потрібно керувати сервером збірки.
- Масштабованість – він масштабується автоматично і може одночасно обробляти кілька збірок.
- Економічно вигідна – AWS CodeBuild працює за моделлю «плати за використання».
- Безпека – Артефакти збірки шифруються за допомогою спеціальних ключів, керованих AWS KMS. Крім того, розмежований доступ можна надати за допомогою AWS IAM.

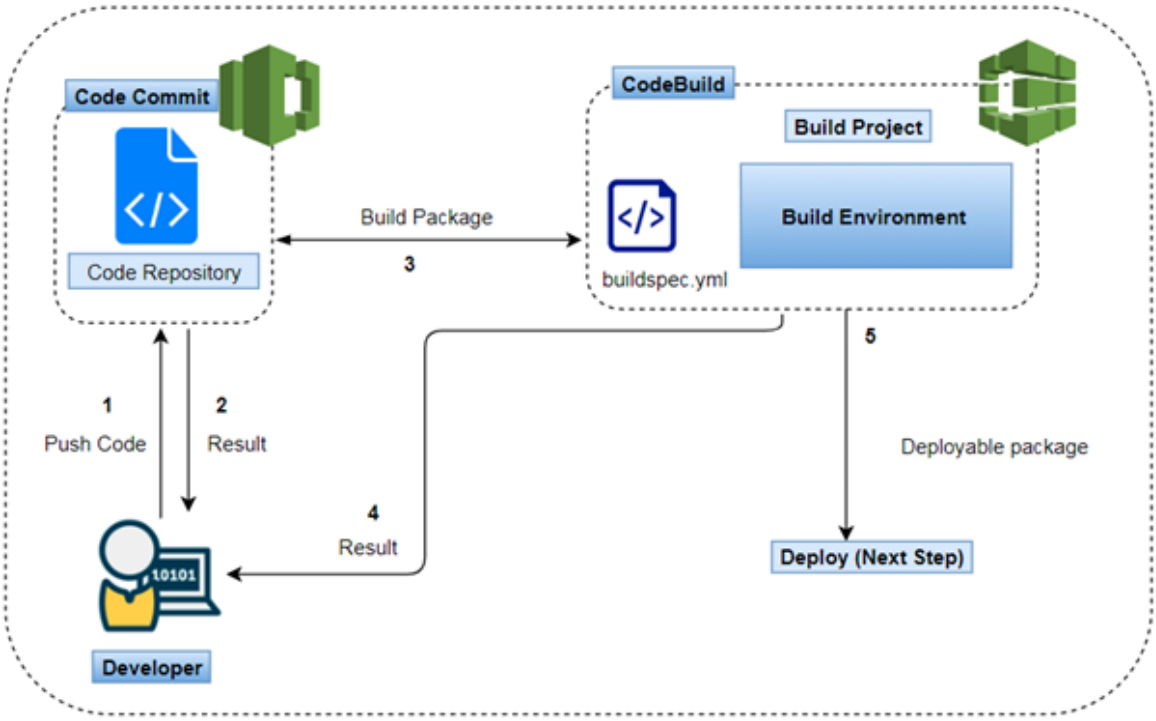


Рисунок 52. Схема збірки проекта

Наступним кроком є створення пакета з коду, розміщеного в CodeCommit. На цьому кроці слід створити проект збірки, який AWS CodeBuild використовуватиме для запуску збірки. Створити проект AWS CodeBuild з назвою проекту BuildCICD. Заповнити опис і створити тег відповідно до рекомендацій за замовчуванням. Наступним кроком є визначення джерела даних, серверу розгортання та визначення файлу Buildspec як конфігураційного. BuildSpec – це сукупність команд та відповідних параметрів, які служба AWS CodeBuild використовує для запуску збірки.

Розгортання за допомогою AWS CodeDeploy. AWS CodeDeploy – це повністю керована послуга, яка використовується для повної автоматизації розгортання програмного забезпечення, що може бути використана для розгортання коду на AWS EC2 або локальних серверах, AWS Fargate та AWS Lambda. Перевагою є повністю керований сервіс: CodeDeploy автоматично масштабується з інфраструктурою, що означає відсутність необхідності

керувати іншою інфраструктурою. AWS CodeDeploy може інтегруватися з групою автоматичного масштабування для автоматичного масштабування ємності EC2.

Централізоване управління – CodeDeploy дозволяє легко запускати та відстежувати стан розгортання. Він надає детальний звіт про розгортання, і можна налаштувати push-сповіщення для оновлень. Ціноутворення – за розгортання на EC2 або AWS Lambda через CodeDeploy не стягується жодна плата. Для решти розгортань це дуже недорого.

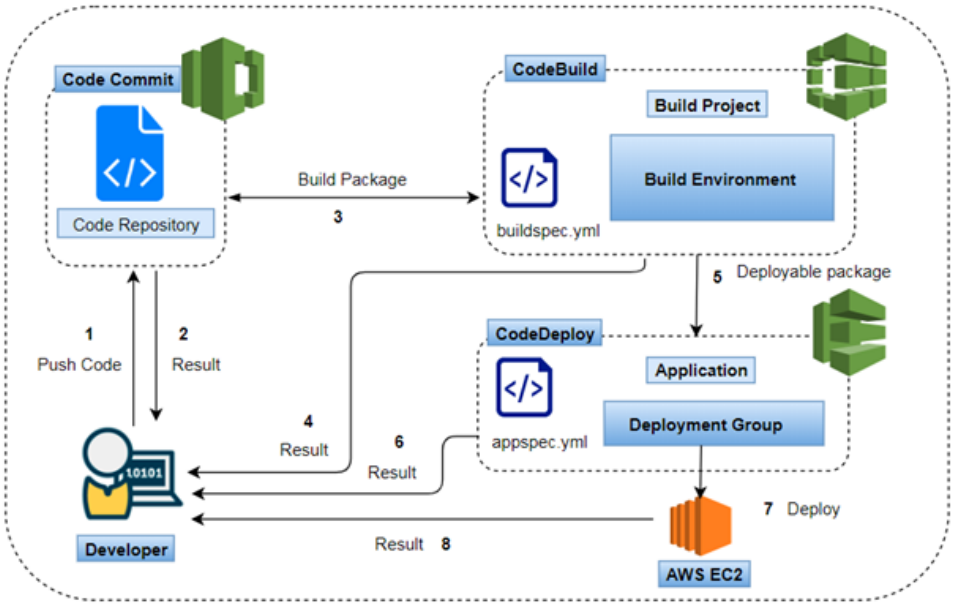


Рисунок 53. Черговість збірки проекта

Роль CodeDeploy. У консолі IAM створити роль обравши політику AWS IAM – AWSCodeDeployRole. Слід створити програму розгортання в AWS CodeDeploy з назвою CICDDemoApplication та обрати групу розгортання як DemoCICDDeploy.

У службовій ролі виберіть роль служби, яку створено раніше (наприклад, CodeDeploy).

В результаті чого здійснено налаштування всіх служб, необхідних для побудови безперевної інтеграції.

Висновок до розділу

Створена архітектура базується на основі використання сервіс–орієнтованого підходу з усвідомленими контекстом, тобто кожен сервіс відображає певну доменну область обробки даних. Архітектура на основі контексту є яскравим представником DDD (Domain Driven Development), що полягає в детальному вивченні доменої області та визначення правильних доменів – такий підхід забезпечує мінімальну кількість зв’язків між сервісів та відсутночті утворення черг повідомлень. Поінформованість про контекст – це механізм адаптації послуги. Адаптація послуги – це також складний процес, який передбачає швидкий аналіз різних контекстних даних та налаштування відповідних послуг. У контекстно-обізнаних додатках дані контексту широко розподіляються і, можливо, надходять з них і використовуються де завгодно та в будь-який час. Розуміння контексту дозволяє автоматично адаптувати послугу. Це допомагає системі адаптувати свою поведінку відповідно до бажаних цілей, пропонуючи сумісні та масштабовані персоналізовані послуги. Прийняття проміжного програмного забезпечення для обробки контексту на основі PaaS дозволить пришвидшити розробку майбутніх мобільних хмарних рішень, абстрагуючи все управління контекстом у багаторазові служби, а також пришвидшити дослідження в області контекстних мобільних додатків для хмари.

Враховуючи особливості хмарної платформи Amazon Web Services розроблено модель доступу до об’єктів як зв’язок AWS IoT (Internet of Things), що складається з шарів: шар обробників даних, шар хмарних сервісів, шар віртуальних об’єктів та фізичних шар мобільних роботів. В AWS IoT програми не можуть безпосередньо оновлювати або отримувати дані пристроїв. Віртуальні об’єкти в AWS працюють як проміжна точка зв’язку між програмами та фізичними пристроями. Єдиний спосіб взаємодії додатків або

пристроїв з віртуальним об’єктом – це спілкування з його темами MQTT. Розроблено модель контролю доступу для спілкування віртуальних об’єктів у AWS IoT (AWS-IoT-ACMVO). Використано AWS-IoT-ACMVO для реалізації двох сценаріїв випадку використання, який застосовується в ACO-IoT-ACMsVO: простий варіант використання, що визначає швидкість одного автомобіля двома датчиками, і варіант використання, що визначає швидкість декількох автомобілів з декількома датчиками. Реалізуючи ці два сценарії за допомогою ACOIoT-ACMsVO, визначено як налаштувати політики та керувати зв’язком віртуальних об’єктів запропонованої нами моделі. Час на поширення інформації про підозрілі машини та перевищення ліміту автомобілів через усі віртуальні об’єкти вимірюється та обговорюється. Нарешті, під час вивчення та впровадження, запропоновано обговорення питань AWS IoT та пропозиції щодо вдосконалення комунікації VO та контролю їх доступу.

Процес обміну спроектовано на основі подієвої моделі (Event Driven Development) що дозволяє реалізувати обмін інформації за допомогою системи розсилки та підписки (pub/sub) на основі хмарних сервісів брокерів повідомлень AWS: AWS SQS, AWS Kinesis.Внаслідок чого в разі затримки на сервісах–обробниках лані не буде втрачено, а тимчасово збережено на сервері брокеру повідомлень.

В спроектованій архітектури дотримано принцип безперервного розгортання (CI/CD) за допомогою таких сервісів як AWS CodeCommit, AWS CodeBuild, AWS CodeDeploy. Система безперервного розгортання керується за допомогою AWS CodePipeline.

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ СЕРВІСІВ

4.1. Обробка даних за допомогою сервісу Atrack Service.

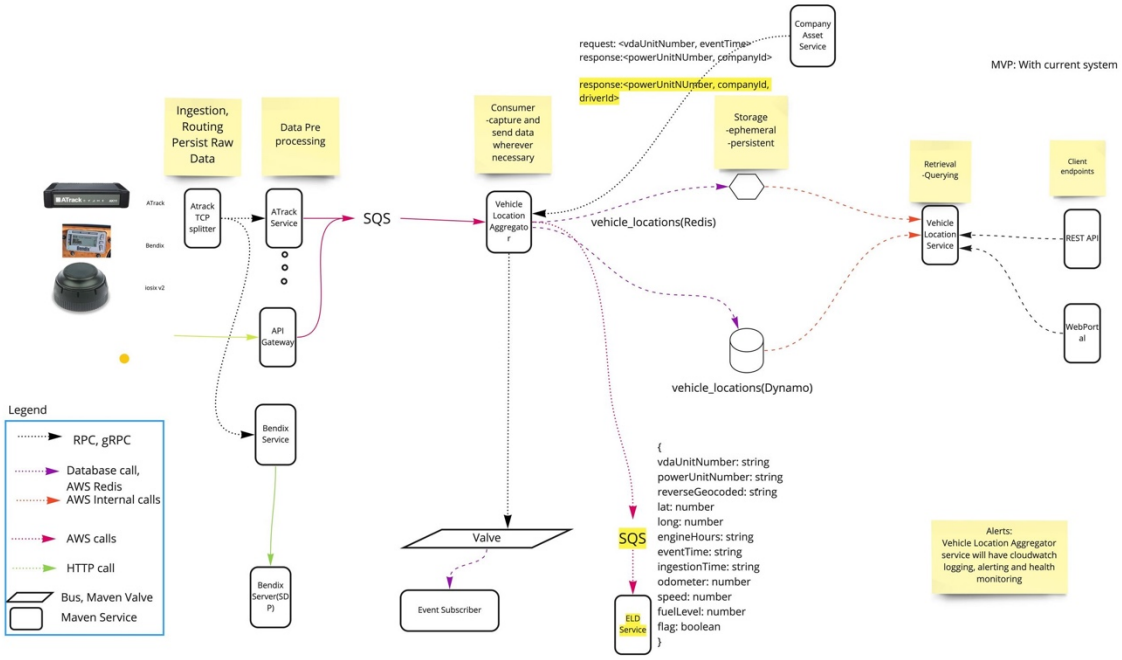


Рисунок 54. Загальна схема сервісів

Схема обробки даних. Дані з вантажівки надсилаються до Atrack Service. Дані зі служби с потрапляють в чергу повідомлень AWS SQS, що агрегуються в Vehicle Location Aggregator з отриманої черги повідомлень. З метою отримання додаткових метаданих на основі поля vdaUnitNumber формується запит <vdaUnitNumber, evenTime> в CompanyAssetService де у відповідь надсилається інформація наступного формату: <companyId, powerUnitNumber, driverId>. В Vehicle Location Aggregator здійснюються додаткові обчислення та формуються, що будуть надіслані за допомогою AWS SQS до сервісу ELD – electronic logging devices.[5]

Формат даних надісланих до ELD наступний:

Таблиця 2. Відповідність полів та їх використання в інших сервісах системи

Назва поля	Джерело даних
vdaUnitNumber	ATrackService, APIGatewayService
powerUnitNumber	Company Asset Service
company	Company Asset Service
vdaEventType	ATrackService, APIGatewayService
eventTime	ATrackService, APIGatewayService
ingestionTime	створено в Vehicle Location aggregator
odometer	ATrackService, APIGatewayService
fuelLevel	ATrackService, APIGatewayService
engineHours	ATrackService, APIGatewayService
latitude	ATrackService, APIGatewayService
longitude	ATrackService, APIGatewayService
driverId	Company Asset Service
Назва поля	Джерело даних
vdaUnitNumber	ATrackService, APIGatewayService
powerUnitNumber	Company Asset Service
company	Company Asset Service
vdaEventType	ATrackService, APIGatewayService
eventTime	ATrackService, APIGatewayService
ingestionTime	створено в Vehicle Location aggregator
odometer	ATrackService, APIGatewayService
fuelLevel	ATrackService, APIGatewayService
engineHours	ATrackService, APIGatewayService
latitude	ATrackService, APIGatewayService
longitude	ATrackService, APIGatewayService
driverId	Company Asset Service

Розглядаючи даний формат даних можна визначити яким чином поля використовуються іншими сервісам в системі, що наведено в табл. 2.

Масштабованість Vehicle Location Aggregator Service. Масштабування сервісу можна здійснювати горизонтально, наразі 10 екземплярів цього сервісу, що працюють на тестовому середовищі та 20 екземплярів на реальному середовищі. Кількість екземплярів вибираються на основі спостереження за обсягом подій, що надходять.

Company Assets Service теж може бути масштабований горизонтально, зараз є один екземпляр для тестового та реального середовища, але можна запустити більше екземплярів, якщо це потрібно.

AWS SQS: Оскільки це сервіс AWS, масштабованість принципово необмежена, а єдиним вузьким місцем можуть бути черги повідомлень. Одна черга повідомлень Amazon SQS може містити необмежену кількість повідомлень, проте існує квота на 120 000 повідомлень за хвилину стандартної черги та 20 000 для черги FIFO.

Таблиця 3. Схема залежностей між сервісами

Залежність сервісу	Сервіс	Залежність від сервісу
ATrack TCP splitter		Atrack, Bendix
Atrack	Atrack TCP Splitter	Company Asset service Asset Tracking service Vehicle Events service Vehicle Location Aggregator
APIGateway		Asset Tracking service Vehicle Events service Vehicle Location Aggregator
Vehicle Events	Company Asset Service DirectoryService Geocoding Service	Company Asset Service
Vehicle Location Aggregator	Company Assets Service	Vehicle Location Service
Vehicle Location Service	Company Assets Service Vehicle Location Aggregator(to populate the Redis cache)	REST API service
REST API service	Vehicle Location Aggregator (reads vehicle_locations MySQL table over gRPC)	Customer exposed endpoint

Базову залежність між представленими сервісами подано в табл. 3. За допомогою чого можна отримати базу інформацію про потік даних. Комунікація сервісів здійснюється на основі pub/sub принципу на основі таких хмарних сервісів як AWS SQS, AWS Kinesis, EC2 RabbitMQ. Це дозволяє здійснювати гнучку передачу даних та забезпечити гарантовану доставку повідомлень навіть у випадку затримок у обробках в конкретному сервісі за рахунок черг, що можуть зберігатися тимчасово на сервері брокеру повідомлень.

4.2. Схема сервісу обробки даних Bendix

Пакет Bendix від пристрою Atrack повинен містити деяку кількість даних, щоб визначити, який пристрій Atrack надіслав пакет. Ідентифікація здійснюється за допомогою ідентифікатора VDA або VIN. Дані ідентифікаційних даних пристрою можна використовувати для запиту відповідних даних у CompanyAssetsService. Далі ідентифікаційні дані пристрою та компанії можуть бути об'єднані в пару для механізму кешування.

Робочий процес для цього сервісу розпочинається, коли пакет отримає пристрій Atrack. Звідти він буде сортувати пакет за двома категоріями – пакети Atrack та пакети Bendix. Пакети Atrack надсилаються службі Atrack для обробки. За допомогою пакетів Bendix служба спочатку отримує ідентифікацію пристрою, а потім перевіряє, чи існує ця ідентифікація в кеші. Якщо кеш має ідентичність, повертається ідентифікатор пов'язаної компанії. Якщо кеш-пам’яті немає, тоді здійснюється запит до CompanyAssetsService на отримання ідентифікатора компанії, пов’язаного з ідентифікацією пристрою. Потім ця асоціація додається до кешу. Звідси ідентифікатор компанії додається до пакета Bendix, а новостворений пакет надсилається на сервер Bendix. Потім ця послуга надішле підтвердження назад на пристрій Atrack і повторить цикл.

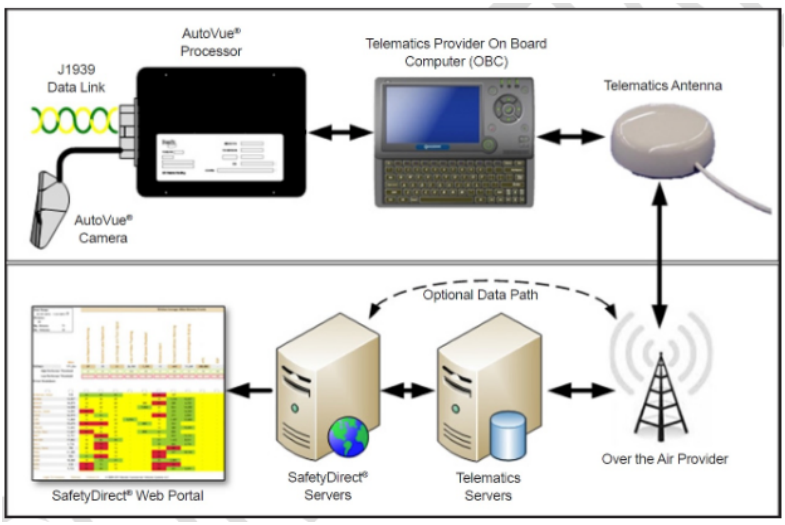


Рисунок 55. Схема роботи серверу Bendix

Розділювач TCP із підтримкою Bendix. Пристрій Atrack надсилає повідомлення службі сортування Atrack TCP. Atrack TCP фільтрує пакети Bendix і направляє їх на сервер Bendix. У відповідь отримуються дані, що потім поєднуються з результатом виклику додаткових даних в CompanyAssetsService та надсилаються на сервіс Attrack.

Переваги:

- Швидке впровадження.
- Враховуючи відсутність всіх необхідних даних для Bendix інтеграції з сервером досить просто змінювати обробку.

Недоліки:

- Не вирішує єдину точку відмови.
- Представляє групу даних в кінцевій точці потоку.
- Ризикований підхід.
- Поганий архітектурний дизайн.

Споживач сортувального сервісу Atrack TCP зчитує пінг з пристрою та надсилає повідомлення новому шлюзу API та потокам AWS Kinesis. Atrack TCP фільтрує пакети Bendix і направляє їх до служби Bendix. Служба Bendix забезпечує додавання ідентифікатора компанії до пакету та маршруту до серверів Bendix. Здійснюється виклик даних до CompanyAssetsService. Після чого дані з серверу Bendix та сервісу CompanyAssetsService відправляються на Vehicle Location Aggregator.

Переваги:

- Кращий архітектурних дизайн.
- Відсутня єдина вузька точка обробки даних.
- Враховуючи можливості AWS Kinesis дані можуть бути передані у вигляді потоку не лише на подальшу обробку а й збережені на кожному етапі у AWS S3.

Недоліки:

- Можливі проблеми з хронологією даних.
- Відсутня повна інформацію про інтеграцію зі стороннім сервером Bendix.

4.3. Сервіс трекінгу та обробки вхідних даних – Asset Signal Service

Метою Asset Signal Service є створення централізованої системи для ведення даних часових рядів будь-яких сигналів, а також взаємозв'язків, як ці сигнали можуть взаємодіяти з будь-якими іншими активами. Відстеження активів розбито на 3 окремі служби.

- Asset Tracking Service – служба відстеження сигналів є основною службою, яка охоплює все відстеження активів. Тут містяться всі API, спрямовані на клієнта, та координати між іншими службами активів.
- Asset Signal Service – служба сигналу активів відповідає за ведення списку сигналів та усіх даних, пов'язаних із цими сигналами.
- Asset Relationship Service – служба зв'язку з сигналами відповідає за картографування взаємозв'язків між різними активами.

Служба передачі активів використовує Timescale як свою базу даних часових рядів. Служба сигналів активів відповідає за ведення списку всіх сигналів та значень даних часових рядів, пов'язаних із цими сигналами. База даних Timeseries, яку зараз використовує ця служба – Timescale, рішення, що підтримується PostgreSQL.

Щоб створити новий сигнал, клієнту просто потрібно опублікувати значення сигналу разом з типом, ініціалізація не потрібна. Сигнали складаються з 2 основних компонентів. Ключ сигналу – це визначений користувачем ключ, унікальний для кожного об'єкта. Тип сигналу – це один із декількох заздалегідь визначених типів, перелічених раніше, яким, як очікується, відповідають дані.

Служба передачі сигналів дещо відрізняється від більшості інших сервісів тим, що в ній фактично є 2 процеси, процес Post та Get. Точками входу для двох процесів є postIndex.js та index.js відповідно. Цей процес розділено на два окремі процеси, оскільки схеми доступу між розміщенням та отриманням у цій службі дуже різні. Кінцеву точку Post викликають інші сервіси дуже часто з відносно меншими обсягами даних, тоді як Get кінцеві точки викликаються рідше, але вони потребують набагато більшої обробки та повертають більше даних. Щоб один тип доступу не блокував інший, розділено на два окремі процеси, що також дозволить масштабувати цю службу на основі її використання. Це означає дві окремі точки входу, два процеси pm2 та дві черги RabbitMQ.

Як обговорювалося вище, цей сервіс можна масштабувати на основі шаблону доступу. Можна горизонтально масштабувати службу сигналізації на основі pub / sub. Як правило, однією з проблем при масштабуванні послуги є будь-яке асинхронне блокування, і ця служба нічим не відрізняється. В даний час служба сигналів блокується щоразу, коли створює нові сигнали, однак оскільки ці блокування знаходяться в пам'яті, можна вважати, що ця служба не буде масштабуватися. Однак навколо цього кроку існує логіка, яка виявляє помилки в результаті локального блокування та повторно обробляє запит, дозволяючи масштабувати службу сигналів.

Щоб створити таблицю SQL Timeseries, можна використовувати Timescale для створення гіпертаблиці – таблиці Postgres, яка розширена для кращої підтримки даних часових рядів. Для збереження цих сигналів визначено таблицю сигналів, яка не є гіпертаблицею, а відповідає за збереження всіх записів (signalKey, signalType, activId). Список сигналів наведено в табл. 4.

Таблиця 4. Опис сигналів

Назва сигналу	Тип	Обов’язковий
signalId	INT	Так
assetId	UUID	Так
signalKey	VARCHAR	Так
signalType	VARCHAR	Так
createdOn	TIMESTAMP	Так
units	VARCHAR	Ні

Висновок до розділу

Сервіс агрегування даних від мобільних роботів Attrack взаємодіє з іншими сервісами за допомогою черг повідомлень AWS SQS. Це дозволяє здійснювати гнучку передачу даних та забезпечити гарантовану доставку повідомлень навіть у випадку затримок у обробках в конкретному сервісі за рахунок черг, що можуть зберігатися тимчасово на сервері брокеру повідомлень. Обмін повідомлень здійснюється на основі асинхрнного патерну віддаленого виклику процедур RPC. Враховуючи відсутність необхідності в збереженні стану сервісу, здісйнен масштаубуванно до 20 екземплярів на реальному середовищі. Пропускна здатність сервісу при заданих конфігураціях сягає 50 тисяч повідомлень за секунду, тобто на один сервіс подається навантаження об’ємом 2.5 тисяч повідомлень за секунду.

Сервіс Attrack надсилає дані на обробку до сервісу обробки даних Bendix. Споживач сортувального сервісу Atrack TCP зчитує пінг з пристрою та надсилає повідомлення новому шлюзу API та потокам AWS Kinesis. Atrack TCP фільтрує пакети Bendix і направляє їх до служби Bendix. Служба Bendix забезпечує додавання ідентифікатора компанії до пакету та маршруту до серверів Bendix. Здійснюється виклик даних до CompanyAssetsService. Після чого дані з серверу Bendix та сервісу CompanyAssetsService відправляються на Vehicle Location Aggregator. Створений сервіс має такі переваги як відсутня єдина вузька точка

обробки даних, враховуючи можливості AWS Kinesis дані можуть бути передані у вигляді потоку не лише на подальшу обробку а й збережені на кожному етапі у AWS S3.

Сервіс трекінгу Asset Signal Service є централізованою системою для ведення даних часових рядів будь-яких сигналів, а також взаємозв’язків, як ці сигнали можуть взаємодіяти з будь-якими іншими активами. Представлений сервіс можна масштабувати на основі шаблону доступу. Можна горизонтально масштабувати службу сигналізації на основі pub / sub. Як правило, однією з проблем при масштабуванні послуги є будь-яке асинхронне блокування, і ця служба нічим не відрізняється. В даний час служба сигналів блокується щоразу, коли створює нові сигнали, однак оскільки ці блокування знаходяться в пам'яті, можна вважати, що ця служба не буде масштабуватися

РОЗДІЛ 5. ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ ДАНИХ В ХМАРНІЙ АРХІТЕКТУРІ

5.1. Опис проблем захищеності даних

Згідно з визначенням NIST, хмарні обчислення – це модель забезпечення повсюдного, зручного мережевого доступу за запитом до спільного пулу сконфігурованих обчислювальних ресурсів таких як сервери, мережі передачі даних, серверів, програм та послуг, які можна швидко надати з мінімальними експлуатаційними витратами або взаємодією з постачальником.

Існує три моделі надання хмарних обчислювальних послуг (згадані в попередніх розділах) та чотири моделі розгортання: Public Cloud, Private Cloud ,Common Cloud, Hybrid Cloud. Незалежно від моделі, хмарні обчислення характеризуються низкою загальних переваг: можливістю спільного використання ресурсів та послуг, гнучкістю в управлінні послугами та можливістю динамічного масштабування ресурсів, зручною системою оплати за використані послуги.

Однак є й проблеми. Переважна більшість проблем хмарних обчислень обумовлені самою природою концепції. Водночас безпека займає центральне місце в цьому списку. Оскільки хмарні обчислення включають багато технологій та концепцій (мережі передачі даних, операційні системи, віртуалізація, бази даних, планування ресурсів, управління паралельністю, транзакції, пам’ять, балансування навантаження), більшість проблем безпеки, пов’язаних з ними, стосуються хмарних обчислень. .

Хмарні несправності можуть спричиняти як зловмисники, так і перебої хмарної інфраструктури. Згідно з дослідженнями, у 2016 році проблема безпеки хмарних обчислень торкнулася 13% компаній. Водночас близько третини з них (32%) стикалися з втратою даних. Відсутність єдиного стандарту безпеки для хмарних обчислень створює певні труднощі при вирішенні таких проблем. У деяких випадках безпека в хмарі досягається за допомогою стороннього

контролю або впровадження хмарних провайдерів власних стандартів та моделей безпеки.

Загрози безпеці в хмарі можуть різнитися залежно від моделі надання послуги. Однак існують типи загроз, які є загальними для всіх моделей. Їх можна класифікувати, впливаючи на характеристики, що характеризують дані та послуги хмарних обчислень.

У 2016 році Cloud Alliance Security (CSA) визначив 12 основних загроз безпеці хмарних обчислень. До них належать:

- витік даних;
- компрометація облікового запису та обхід автентифікації;
- незахищені інтерфейси та API;
- вразливості використовуваних систем та додатків;
- викрадення вірчих грамот;
- зловмисні інсайдери;
- цілеспрямовані кібератаки;
- втрата даних;
- недостатня обізнаність про хмарні можливості;
- зловживання хмарними сервісами;
- відмова в обслуговуванні;
- загрози спільних технологій.

Статистика загроз та аналіз ринку хмарних обчислень показують, що актуальність розробки методів та розробки засобів захисту хмарного середовища (Cloud Environment) дуже висока. Однак сьогодні не існує єдиного механізму та єдиної моделі захисту хмари від існуючих загроз безпеці. Одним із підходів до забезпечення безпеки хмарного середовища можна вважати використання розподілених систем моніторингу та виявлення вразливостей на основі принципів імунної системи людини.

Такі рішення характеризуються розподілом та самоорганізацією, що пристосовує їх до роботи в хмарному середовищі. Однак проблемною стороною використання таких систем для виявлення аномалій у хмарному середовищі та підвищення його безпеки є оцінка їх ефективності, оскільки багаторівнева структура середовища ускладнює використання стандартних методів оцінки. Таким чином, виникає суперечність між необхідністю комплексного підходу до забезпечення безпеки хмарного середовища на основі розподілених систем та відсутністю методологічної бази для оцінки ефективності цих систем. Ця суперечність є наслідком проблемних питань теорії та практики, пов'язаних із безпекою хмарних обчислень.

5.2. Організація доступу авторизованих користувачів до ресурсів

Глобальний розвиток хмарних сервісів призводить до того, що кожен користувач мережі Інтернет оточений в незалежності від платформи доступу, величезною кількістю служб, що дозволяють створювати і поширювати медіа–контент або отримувати доступ до електронних послуг. Очевидно, що перед розробниками сервісів з’являється завдання забезпечення безпеки. Ситуація ускладнюється тим, що робота користувача не повинна утруднятися внутрішніми механізмами безпеки і переміщення між сервісами має відбуватися максимально швидко і безпечно для послуг наданих користувачеві.

Щоб вирішити задачу, пов'язану зі спрощенням авторизації користувача при роботі з великою кількістю застосунків і онлайн сервісів розроблений протокол OAuth. При використанні OAuth-авторизації до основних переваг прийнято відносити відсутність передачі логіна і пароля в додаток, з яким працює користувач. Таким чином, програма може виконати тільки ту, що явно дозволив користувач. Так само, відпадає необхідність вирішення питання забезпечення захищеного зберігання пароля і логіна застосунком.

Актуальна версія стандарту OAuth 2.0, опублікована в 2012 році в документі дозволяє додаткам отримувати доступ від свого імені або обмежений доступ до HTTP службі від імені власника ресурсу, організувавши процес узгодження взаємодії між власником ресурсу і HTTP-службою. Результатом авторизації є Access Token – ключ, пред'явлення якого є пропуском до захищених ресурсів. Стандарт не визначає формат ключа, який отримує додаток, тому ключ сам по собі не може бути використаний для автентифікації користувача.

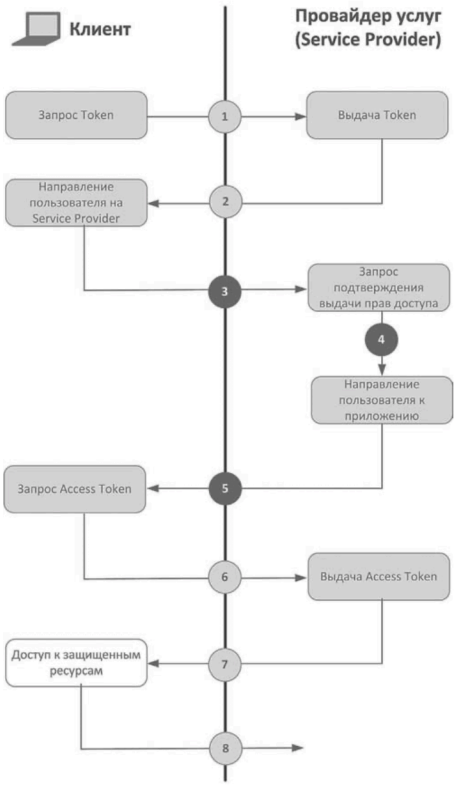


Рисунок 56. Схема OAuth2

Наведений нижче (рис. 57) алгоритм демонструє ключові особливості логіки роботи протоколу, що дозволяють вирішити задачу авторизації, тобто надання права на використання ресурсу. Щоб визначити наявність прав, необхідний токен (запис або значення, що є унікальною ідентифікацію). Відмітимо, що один і той же токен може бути повторно використаний для різних користувачів, в той же час у одного і того ж користувача в процесі авторизації

можуть помінятися токени при наявності специфічних тимчасових вимог для їх поновлення. Щоб отримати права на роботу з ресурсом необхідно пред'явити відповідний токен.

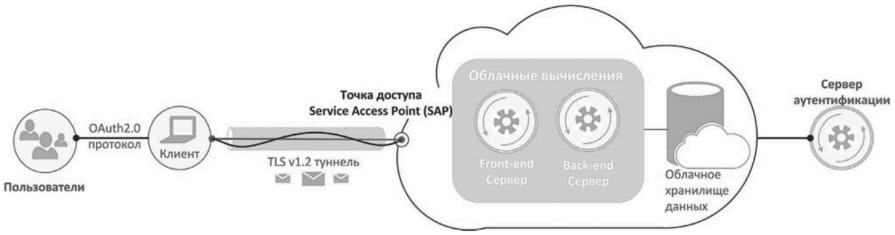


Рисунок 57. Алгоритм роботи OAuth2.0

Висновок до розділу

В результаті розгляду концепції хмарних обчислень підтверджено проблемні аспекти технології, актуальність дослідження даної предметної області, розглянуто можливі підходи до забезпечення безпеки хмарного середовища за допомогою розподілених систем виявлення аномалій, виявлено відсутність методологічної бази для оцінки ефективності цих систем та сформульовано протиріччя між проблемами. практика і теорія.

Щоб вирішити задачу, пов'язану зі спрощенням авторизації користувача при роботі з великою кількістю застосунків і онлайн сервісів розроблений протокол OAuth. Зниження ризику неавторизованого доступу до ресурсів, і, як наслідок, забезпечення доступності інформації, можна домогтися за рахунок впровадження механізму автентифікації. В Amazon Web Services основним механізмом розподілу прав є IAM та RBAC, що дозволяються створювати індивідуальні ролі доступу для кожного сервісу, де роль може мати доступ лише до обраних ресурсів. Єдиною складністю такого підходу є комплексність побудови максимально гнучкої системи розмежування прав.

РОЗДІЛ 6. МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЄКТУ

6.1. Опис ідеї проєкту

Таблиця 5. Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користучача
Проектування та розробка хмарної системи, що дозволить ефективно здійснювати керування та узгодження мобільних роботів з метою оптимізації витрат та забезпечення мінімальної затримки при узгоджені дій мобільних роботів.	Забезпечення керування роботизованими логістичними системами та прикладі безпілотних вантажних вристроїв.	Зменшення витрат на програмне забезпечення, так як модель базується на оплату за об'єм використання не враховуючи простоїв. Ефективна роботизація промислових секторів за рахунок збільшення кількості паралельних завдань.
	Створення роботизованого аграрного виробництва, що за допомогою синхронізації даних в реальному часі забезпечить максимальну узгодженість дій мобільних роботів-помічників.	

Таблиця 6. Опис ідеї стартап-проєкту

№	Техніко-економічні характеритики ідеї	Продукція конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проєкт	DAvinCi	RoboEarth			
1	Спосіб передачі даних	MQTT, HTTP, AMQP	MRID	HTTP		+	
2	Уніфікованість інтерфейсів	AWS IOT	Закрита розробка	Закрита розробка			+
3	Готовність до швидкого масштабування	Гнучкі хмарні сервіси	В межах наданого плану	В межах наданого плану			+
4	Витрати за користування	За користування	За наданий план	За наданий план			+

6.2. Технологічний аудит ідеї проєкту

Таблиця 7. Технологічна здійсненність ідеї проєкту

№	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Забезпечити узгодження даних в реальному часі групи мобільних роботів в хмарному середовищі	Хмарні рішення, що мають значний запас сервісів направлених на обробку даних та взаємодії зі сторонніми інтерфейсами.	Хмарні провайдери AWS, Google Cloud, Azure	Можливість отримати безкоштовний доступ до більшості необхідних сервісів та плата за користування.
2	Забезпечити узгодження даних в реальному часі групи мобільних роботів в класичному середовищі	Класичний серверна технологія з великою кількості наявних сервісів за замовчуваннями та масштабуванням в будь-який час.	Присутні лише звичайні сервери, де плата експоненціально залежить від кількості фізичних серверів, а не часу користування. Складно керувати сервісами.	Технологія не вирішує визначену ідею.
3	Забезпечити узгодження даних в реальному часі групи мобільних роботів в розподіленому середовищі	Використання моделі обробки даних, не обчислювальними вузлам може виступати будь-який пристрій за рахунок надання у користування певної обчислювальної потужності.	Технологія знаходиться лише в стадії теоретичних гіпотез.	Відсутні практичні реалізації
Обрана технологія реалізації ідеї проєкту:1				

Висновок: технологічна реалізація продукту – можлива, вибрана технологія №1.

6.3. Аналіз ринкових можливостей запуску стартап-проєкту

Таблиця 8. Попередня характеристика потенційного ринку

№	Показники стану ринку	Характеристика
1	Кількість головних гравців, од	2–3 (приватні провайдери)
2	Загальний обсяг продаж, грн./ум.од	1 000 000 ум.од. за місяць
3	Динаміка ринку	Стагнує
4	Наявність обмежень для входу	Необхідність високо кваліфікованого персоналу
5	Специфічні вимоги до стандартизації та сертифікації	ISO
6	Середня норма рентабельності в галузі або по ринку, %	75%

Висновок: враховуючи стагнуючу динаміку ринку та середню норму рентабельності отримано висновок, що ідея стартап-проекту є привабливою.

Таблиця 9. Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці цільових груп клієнтів	Вимоги споживачів до товару
1	Необхідність в програмному забезпеченні що дозволяє керувати роботизованими галузями господарство	Власники малих та середніх господарств, що не мають змоги самостійно розробляти серверне рішення.	Кількість мобільних роботів, наявність стабільного інтернет з’єднання.	Просто масштабованість системи, мінімально затримка в прийняті рішень щодо керування, уніфікований інтерфейс доступу, гнучка модель оплати.
2	Забезпечення масштабованості при обробці даних в реальному часі	Сфера обслуговування, що внаслідок збільшення кількості пристроїв не може забезпечити вимоги швидкодії	Направленість сфери обслуговування, об’єм оброблюваної інформації.	

Таблиця 10. Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Відсутність хмарних провайдерів	Архітектура системи базується на хмарних провайдерах, тому їх стабільне функціонування є фундаментом	Зміна хмарного провайдера, створення механізмів бекапів.
2	Відтік кадрів	Розробка та обслуговування системи потребує великої кількості кваліфікованої праці.	Забезпечення сприятливих робочих умов, створення центрів навчання для нових співробітників, фінансування внутрішніх навчальних програм.

Таблиця 11. Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання попиту зі сторони малих та середніх господарств	Внаслідок збільшення попиту виникає необхідність в надалі більш гнучких рішень	Переговори з новими партнерами, інвестиції з метою охоплення більш широкого списку галузей господарств.
2	Зростання рівня підготовки нових працівників	Внаслідок покращення технічної освіти та наявності мережі центрів перекваліфікації кількість кваліфікованих співробітників зростає	Можливість розробки нового функціоналу та обслуговування більшої кількості клієнтів за рахунок зростання ресурсів
3	Політика держав направлена на інформаційне суспільство.	Держави зацікавлені в автоматизації процесів в державі.	Спроба отримати гранти та інвестиції від різних країн світу та надання необхідного об’єму функціонала.

Таблиця 12. Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1	Чиста монополія	Відсутність замінників для наданих послуг.	Надання широкого спектру унікальних послуг згідно попиту користувачів.
2	Світова конкуренція	Продукт доступний в будь-яких куточках світ без прив'язки до місцезнаходження.	Орієнтація на найбільш зацікавлені держави, нові перспективи для виходу на ринок.
3	Галузева ознака: внутрішньогалузева	Відбувається в мажах галузі інформаційних технологій.	Надання найбільш якісних та своєчасних послуг.
4	Конкуренція за видами товарів: товарно-родова	Може бути конкуренція в межах товар різного роду, проте пов'язані з мобільними роботам.	Впровадження функціональності яка відсутня у товарів-замінників; Спрощення інтерфейсів; Надання підтримки.
5	Характер конкурентних переваг: цінова та не цінова	Орієнтація на новий індивідуальний функціонал.	Формування функціоналу залежно від індивідуальних вимог клієнтів.
6	За інтенсивністю: марочна	Користувачі звертають увагу на стабільність бренду	Забезпечення публічності бренду.

Таблиця 13. Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	DAvinCi, RoboEarth	Доступ до ресурсів, доступ до каналів розподілу	Концентрація постачальників	Система інформації та контроль якості	Лояльність споживачів
Висновки	Середня	Так, на ринок можна входити. Можлива їх поява враховуючи швидкість розвитку інформаційних технологій.	Розширення мережі певної формує монополізм на ринку, тому необхідно вести додаткову конкурентну боротьбу.	Клієнти хочуть отримувати максимально ефективне рішення в умовах обмеженого бюджету.	Споживачі можуть бути не готові довіряти сторонній системі.

Можна сформувати висновок, що кожна з систем має обмежений спектр виконуваних задач. Щоб успішно вийти на ринок слід знайти підтримку у організаціях, що готові спробувати нову модель системи.

Таблиця 14. Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1	Актуальність технологій	Побудова архітектури на основі хмарних сервісів є сучасним рішенням, що заміняє класичне серверне обслуговування
2	Простота масштабування	Кожен ресурс може бути масштабований залежно від необхідної обчислювальної потужності.
3	Гнучка модель оплат	Оплата здійснюється лише на основі об’єму використаного ресурсу, а не на основі передбаченого плану.
4	Зручний моніторинг	Всі ресурси системи можна відслідковувати в сервісах моніторингу, налаштовуючи гнучку систему сповіщень.
5	Універсальність інтерфейсу	Інтерфейс взаємодії з мобільними роботами дозволяє здійснювати підключення на основі різних протоколів.

Таблиця 15. Порівняльний аналіз сильних та слабких сторін системи

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з запропонованим						
			-3	-2	-1	0	+1	+2	+3
1	Актуальність технологій	18	3	2	1				
2	Простота масштабування	19		2	3	1			
3	Гнучка модель оплат	16		3	2	1			
4	Зручний моніторинг	15	1	2	3				
5	Універсальність інтерфейсу	16	3	2	1				

Таблиця 16. SWOT аналіз стартап-проєкту

Сильні сторони (S): – низька затримка в обробці даних – гнучка система інтеграцій – проста масштабованість – універсальність моделі	Слабкі сторони (W): – відсутній значний об’єм даних про тестування системи на різних практичних реалізаціях – складність міграції існуючих систем, враховуючі інноваційність рішення
Можливості (O): – використання нових технологій та підходів до оптимізації	Загрози (T): – зменшення попиту на роботизовані системи – зменшення кваліфікації працівників

Таблиця 17. Альтернативи ринкового впровадження стартап-проєкту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Надання безкоштовного функціонал для перших клієнтів з метою апробації рішення	Висока, так як відсутні прямі аналоги	2-3 місяці
2	Залучення рекламних проспектів на сайтах в сфері інформаційних технологій	Залучення власних коштів для реклами системи	2-3 місяці
3	Переговори з посадовцями	Можливість використання ресурсу в рамках державних програм інформатизації	6-12 місяців

6.4. Розроблення ринкової стратегії проєкту

Таблиця 18. Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Малі та середні господарства та представники сфери послуг	Висока зацікавленість в готових програмних комплексах.	**Високий.** Ефективне рішення враховуючи динамічність господарств.	**Середня.** Доступні декілька аналогів з обмеженими можливостями.	**Просто. Доступна значна ніша.** Представники групи зацікавлені в отримані такого роду систем
2	Великі корпорації	Низька зацікавленість в сторонніх рішеннях.	**Низький.** Можливість створювати власні розробки враховуючи потужність.	**Середня.** Частина корпорацій має власні закриті рішення.	**Складно. Закрита корпоративна ніша.** Складно мігрувати існуючі системи.
Які цільові групи обрано: 1					

Отримано висновок, що великі корпоративні структури не є цільовою аудиторію продукту, адже мають власні великі системи, які досить складно мігрувати на нові технології та мають достатньо потужностей для забезпечення самостійних розробок. Малі ж та середні господарства є досить динамічними в

планах розвитку, тому потребують масштабованого рішення, що забезпечить роботу роботозованих відділів.

Таблиця 19. Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Забезпечення гнучкості та швидкодії рішення	Акцент на можливостях масштабування та охоплення різних сфер господарства.	Нова система інтеграції та синхронізації даних, що не має прямих аналогів	Стратегія диференціації

Таблиця 20. Визначення базової стратегії конкурентної поведінки

Чи є проєкт «першопрохідцем» на ринку	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, які?	Стратегія конкурентної поведінки
Враховуючи актуальність технологіє є перщопроходцем	Шукати нових споживачів, що потребують нових програмних рішень	Копіювати частково підходи до взаємодії з мобільними роботами та алгоритми узгодженості керування	Стратегія заняття конкурентної ніші

Таблиця 21. Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію власного проєкту
1	Проста масштабованість, ефективне використання ресурсів	Стратегія диференціації	Хмарна провайдери дозволяють оплачувати лише за використані ресурси та збільшувати потужність залежно від необхідної пропускної здатності	Обробка великих масивів даних, мінімальні затримки, майбутнє обчислень

Стартап грунтується на основі стратегії диференціації та стратегії заняття конкурентної ніші.

6.5. Розроблення маркетингової програми стартап-проєкту

Таблиця 22. Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Масштабованість	Горизонтальна та вертикальна масштабованість залежно від поточного об’єму навантажень	Система автоматично збільшує чи зменшує пропускну здатність без необхідності запиту та очікування додаткових ресурсів.
2	Ефективність використання ресурсів	Плата лише за використаний ресурс	Ооплата лише за реально спожитий об’єм обчислень.
3	Швидкість обробки даних	Мінімальна затримка в обробці за рахунок хмарних сервісів.	Велике різноманіття хмарних сервісів дозволяють вирішити спеціалізовані задачі залежно заданої мети.

Продовження таблиці 22

4	Стабільність системи	Збереження вхідних та вихідних даних	Система зберігає всі дані на кожному етапі обробки та логує всі операції, що дозволяє в будь-який момент здійснити повторне обчислення.

Таблиця 23. Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
1. Товар за задумом	Система синхронізації даних в реальному часі групи мобільних роботів		
2. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Масштабованість	М	Тх
	Ефективність використання ресурсів	М	Тх
	Швидкість обробки даних	Нм	Тх
	Стабільність системи	М	Тх
	Якість: ISO/IEC 9126		
	Спосіб взаємодії: інтерфейс MQTT, пристрій для доступу в мережу Інтернет з метою керування системою		
	Марка: IoT_Now Ltd.		
3. Товар із підкріпленням	До продажу: тестова модель системи, документація, короткий опис можливостей та перспектив		
	Після продажу: адаптації залежно від потреб, служба підтримки		
За рахунок чого потенційний товар буде захищено від копіювання: патент на корисну модель, захист інтелектуальної власності,			

Таблиця 24. Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
100 000 – 1 000 000 грн залежно від плану	100 000 – 1 000 000 грн залежно від плану	> 500 тис. грн	0 – 1 000 000 Залежно від використаних ресурсів

Таблиця 25. Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Індивідуальні потреби господарства	Налагодження взаємодії з господарствами, демонстрація тестової моделі.	0. Безпосереднє надання інформаційних послуг господарствам	Власна система збуту

Таблиця 26. Концепція маркетингових комунікацій

№	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Різні моделі бізнесу	Електронна пошта, телефонний зв’язок, соціальні мережі, бізнес зустрічі	Залучення великої клієнтської бази, можливість регуляції та масштабування роботи	Стимул користувачів до користування системою протягом довготривалого періоду часу..	Система дозволяє ефективно масштабувати та забезпечувати керуванням.

Висновки по розділу

Розроблено маркетингову програму, що визначає переваги системи, цінову модель, системи збуту та маркетингові комунікації.

ВИСНОВКИ

У результаті виконання магістерської дисертації спроектовано архітектуру синхронізації даних в реальному часі на основі хмарного середовища Amazon Web Services. Така архітектура може бути використана як для синхронізації даних групи мобільних роботів та і для обробки даних з IoT пристроїв за рахунок уніфікації інтерфейсів та побудови коректної багатошарової архітектури, де внутрішні рівні є незалежними від зовнішніх.

В першому розділі поставлено задачу використання хмарних сервісів в роботизовних системах, встановлено проблеми, що виникають в побудові систем на основі класичної серверної обробки та шляхи вирішення такиї проблем. Здійснено аналіз систем-аналогів таких як DAvinCi, RoboEarth, Rapyuta. Визначено переваги та недоліки таких систем та можливості вирішенняі на основі хмарного провайдеру AWS, що підтримує вузькоспеціалізовані сервіси для обробки даних з IoT пристроїв – AWS IoT.

Далі здійснено огляд хмарних сховищ, сервісів передавання інформації та хмарних серверних рішень. В результаті чого обрано DynamoDB в якості основного сховища збереження дані про пристрої, що дозволяє здійснювати ефективний пошук за допомогою основного та додаткових індексів. AWS Kinesis забезпечує потокове передавання даних без необхідності групування у блоки та дозволяє здійснити запис в проміжні сховища S3, сервіс аналітики AWS CloudWatch та почати обчислення в без серверному рішенні AWS DynamoDB.Крім цього обраний брокер повідомлень має можливість накопичувати повідомлення в разі відмови сервісів-обробників. Без серверні рішення представлені у вигляді функцій, які можуть бути запущені паралельно в кількості сотень тисяч. Такий підхід забезпечує мінімум затримок на кожному етапі та просто масштабується.

Спроектовано архітектуру системи на основі доменних моделей, контролю доступу до віртуальних об’єктів, підходу подієвих моделей. В результаті кожен сервіс виконує визначену функцію в рамках проаналізованої доменної моделі, мінімізуючи кількість операцій обміну даних з іншими сервісами. Обмін даних відбувається асинхронно використовуючи віддалений виклик процедур RPC, що реалізовано на методі обміну повідомлення типу pub/sub. Використання віртуальних об’єктів дозволяє зберігати інформацію щодо останнього працездатного стану та є проміжним шаром між зчитувачем даних сигналів та власне фізичним роботом. Представлена архітектура забезпечує просте масштабування, мінімальні простої між обміну даними за рахунок використання через повідомлень та ефективне використання ресурсів завдяки хмарному рішенню.

В наступному розділі представлені прикоади реалізованих сервісів обробки вхідних сигналів такі як Attrack, Bendix. Визначено основний формат даних та схему взаємодії з іншими сервісами в системі.

Питання безпеки є особливо важливим в контекст використання хмар, тому визначено основні можливі прогалини в безпеці доступу в системі та описано шляхи побудови системи з максимальним рівнем захищеності на основі AWS RBAC та AWS ABAC.

Проведений маркетинговий аналіз стартап-проекту. Чітко визначено ідею проекту, можливості виходи на ринок та конкурентоспроможність, шляхи вирішення загроз та перспективи. Також проаналізовано ринок збуту, концепцію маркетингових комунікацій.

Результати роботи над магістерською дисертацією опубліковані у двох статтях. Додати інформацію про статті.

Перспективою подальшого дослідження може виступати впровадження хмарного сервісу RoboMaker. Це хмарне робототехнічне рішення, яке

забезпечує індивідуальне середовище розробки, повністю керовану інфраструктуру моделювання та вбудовані розширення хмари,

що спрощує створення та тестування програм-роботів та їх інтеграцію з іншими службами AWS

Наукова новизна одержаних результатів магістерської дисертації полягає у наступному:

вперше:

- проаналізовано розроблені архітектури керування роботами на основі хмарних рішень;
- визначено основні вимоги до проектування масштабованої хмарної системи синхронізації даних мобільних роботів;
- спроектовано сучасну архітектуру системи синхронізації даних, використовуючи поєднання підходів доменно орієнтованої розробки, контролю доступу до віртуальних об’єктів та подієво орієнтованох розробки;
- застосовано такі рішення як AWS DynamoDB, AWS Kinesis Data Warehouse, AWS SQS, AWS IOT як складові компоненти системи;
- розроблено механізми логування та генерації інформативних логів на кожному етапі обчислювальної обробки.

удосконалено:

- зменшено навантаження на мобільні роботи за рахунок винесення обробників даних в хмарну інфраструктуру;
- забезпечено горизонтальне та вертикальне масштабування системи на різних етапах обробки;
- покращено систему безпеки за рахунок використання RBAC, ABAC, Iam підходів для гнучкої організації доступів до різних обчислювальних ресурсів.

здобуло подальший розвиток:

– використання хмарних сервісі як підгрунтя до побудови хмарних систем;
– актуальні використання моделей передачі даних на основі publisher/subscriber;
– підходи до уніфікації програмних інтерфейсів.

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Amazon Web Service, Amazon Inc [Електронний ресурс] // Режим доступу: http://aws.amazon.com/
2. T. C. Chieu Scalability and Performance of Web Applications in a Compute Cloud : report / A. Mohindra, and A. A. Karve – IEEE International Conference on eBusiness Engineering, 2011
3. AWS IoT Docs [Електронний ресурс] // Режим доступу: https://aws.amazon.com/ru/iot/
4. AWS Kinesis Data Warehouse Docs [Електронний ресурс] // Режим доступу: https://docs.aws.amazon.com/firehose/latest/dev/what-is-this-service.html
5. AWS CloudWatch Docs [Електронний ресурс] // Режим доступу: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html
6. AWS DynamoDB [Електронний ресурс] // Режим доступу: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
7. Amazon EC2 Instance Types, Amazon Web Services [Електронний ресурс] // Режим доступу: http://aws.amazon.com/ec2/instance-types/
8. Antonic, A. A mobile crowdsensing ecosystem enabled by a cloud-based publish/subscribe middleware: report / Roankovic, K., Marjanovic, M., Pripuic, K., Zarko, I.P. – International Conference on Future Internet of Things and Cloud (FiCloud), 2014 – p. 107-114
9. Happ, D. Meeting IoT Platform Requirements with Open Pub/Sub Solutions: report / Karowski N., Menzel T., Handziski V., Wolisz A. – 1st Int. Conf. on Cloudification of the Internet of Things Paris, France (2015)

10. G. Banavar A case for message oriented middle-ware: report / T. Chandra, R. Strom, and D. Sturman – Proceedings of the 13th International Symposium on Distributed Computing (DISC99), 1999 – p. 1-18
11. F. Buschmann It Pattern-oriented software architecture: a system of patterns: report – John Wiley and Sons, Inc. New York, NY, USA l'1996 ISBN:0-471-95869-7
12. P. Kogge Exascale computing study: Technology challenges in achieving exascale systems: science book / W. Yu – IEEE access, 6:6900–6919, 2018
13. Papakos P. VOLARE: context-aware adaptive cloud service dis– covery for mobile systems: report / Capra L., Rosenblum D.S. – Proceedings of the 9th international work– shop on adaptive and reflective middleware, Bangalore, 2010
14. Wright Evaluating Interconnect and virtualization perfor– mance for high performance computing: report – ACM Performance Evaluation Review, 2012
15. P. Mehrotra Performance evaluation of Amazon EC2 for NASA HPC applications: report – Proceedings of the 3rd work– shop on Scientific Cloud Computing (ScienceCloud '12). ACM, New York, NY, USA – p. 41-50
16. Розроблення стартап-проєкту [Електронний ресурс] : Методичні рекомендації до виконання розділу магістерських дисертацій для студентів інженерних спеціальностей / За заг. ред. О.А. Гавриша. – Київ : НТУУ «КПІ», 2016. – 28 с.

ДОДАТКИ

ДОДАТОК А

НАУКОВІ СТАТТІ

№ 1512/2020-3
15.12.2020

СПРАВКА АВТОРА ПУБЛИКАЦИИ

научной статьи в международном научном журнале
International Academy Journal Web of Scholar
ISSN print - 2518-167X
ISSN online - 2518-1688

Издательство RS Global Sp. z O.O., Варшава, Польша

Научная статья на тему “ ДОСЯГНЕННЯ ЕФЕКТИВНОГО РОЗПОДІЛЕНОГО ПЛАНУВАННЯ ЗА ДОПОМОГОЮ ЧЕРГ ПОВІДОМЛЕНЬ У ХМАРІ ДЛЯ БАГАТОЗАДАЧНИХ ОБЧИСЛЕНЬ ТА ВИСОКОПРОДУКТИВНИХ ОБЧИСЛЕНЬ” отвечает требованиям издательства и принята к публикации в журнал International Academy Journal Web of Scholar.

Авторы:

1. Старовойтенко Олексій Володимирович

Статья будет издана в журнале №8(50), 2020 и включена в наукометрические базы данных: Index Copernicus и другие.

Editorial board of academic journal
Science Review
RS Global Sp. z O.O.
REGON: 367026200,
NIP: 5213776394

Досягнення ефективного розподіленого планування за допомогою черг повідомлень у хмарі для багатозадачних обчислень та високопродуктивних обчислень

Старовойтенко Олексій Володимирович
студент кафедри ТК
Факультет інформатики та обчислюваної техніки
Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"
alexey.starovoytenko@gmail.com
https://orcid.org/0000-0001-7365-3940

Annotation
Due to the growth of data and the number of computational tasks, it is necessary to ensure the required level of system performance. Performance can be achieved by scaling the system horizontally / vertically, but even increasing the amount of computing resources does not solve all the problems. For example, a complex computational problem should be decomposed into smaller subtasks, the computation time of which is much shorter. However, the number of such tasks may be constantly increasing, due to which the processing on the services is delayed or even certain messages will not be processed. In many cases, message processing should be coordinated, for example, message A should be processed only after messages B and C. Given the problems of processing a large number of subtasks, we aim in this work - to design a mechanism for effective distributed scheduling through message queues. As services we will choose cloud services Amazon Webservices such as Amazon EC2, SQS and DynamoDB. Our FlexQueue solution can compete with state-of-the-art systems such as Sparrow and MATRIX. Distributed systems are quite complex and require complex algorithms and control units, so the solution of this problem requires detailed research.

Keywords: FlexQueue, multitasking, cloud message queues, high-performance system, data consistency,

I. ВСТУП

Метою системи планування завдань є ефективне управління розподіленою обчислювальною потужністю робочих станцій, серверів та суперкомп'ютерів з метою максимізації пропускної здатності роботи та використання системи. Прогнозується, що до кінця цього десятиліття ми матимемо систему масштабування з мільйонами вузлів і мільярдами потоків виконання [1].

На жаль, сучасні планувальні системи мають централізовану архітектуру Master/Slaves (наприклад, Slurm [2], Condor [3][4], PBS [5], SGE [6]), де централізований сервер відповідає за надання ресурсів та виконання завдань. Ця архітектура добре працювала в масштабах обчислювальних мереж та грубих гранульованих робочих навантаженнях [7], але вона має погану масштабованість при екстремальних масштабах систем із дрібнозернистими робочими навантаженнями [8][9]. Рішенням цієї проблеми є перехід до децентралізованих архітектур, які уникають використання одного компонента в якості менеджера. Розподілені планувальні системи, як правило, реалізуються в ієрархічній [10] або повністю розподіленій архітектурі [31] для вирішення проблеми масштабованості. Використання нових архітектур може вирішити потенційну єдину точку відмови та покращити загальну продуктивність системи до певного рівня, але можуть виникнути проблеми при розподілі завдань та балансуванні навантаження між вузлами [25].

Ідея використання хмарних сервісів для високопродуктивних обчислень існує вже кілька років, але вона не набула популярності в першу чергу через багато проблем. Маючи великі ресурси, публічні хмари можна використовувати для виконання завдань в екстремальних масштабах розподіленим способом. Наша мета у цьому проекті - забезпечити компактну та легку розподілену структуру виконання завдань, яка працює на Amazon Elastic

Compute Cloud (EC2) [17], використовуючи складні розподілені будівельні блоки, такі як Amazon Simple Queuing Service (SQS) [18] та Amazon, розподілений NoSQL ключ/значення сховища (DynamoDB) [33].

Було багато дослідницьких робіт з використання загальнодоступного хмарного середовища для наукових обчислень та високопродуктивних обчислень (HPC). Більшість цих робіт показують, що хмара не могла виконувати добре працюючі наукові програми [11][12][13][14]. Більшість існуючих дослідницьких робіт використовували підхід використання публічної хмари як подібний ресурс до традиційних кластерів та суперкомп'ютерів. Використання спільних ресурсів та технологій віртуалізації робить загальнодоступні хмари зовсім іншими, ніж традиційні системи HPC. Замість того, щоб запускати одні й ті ж традиційні програми на іншій інфраструктурі, ми пропонуємо використовувати публічні хмарні сервісні програми, які оптимізовані для хмарного середовища. Використання загальнодоступних хмар, таких як Amazon, як ресурсу для виконання завдань може бути складним для кінцевих користувачів, якщо воно надає лише необроблену IaaS[34]. Було б дуже корисно, якби користувачі могли лише увійти в свою систему та надіслати завдання, не турбуючись про управління ресурсами.

Ще однією перевагою хмарних служб є те, що користуючись цими службами, користувачі можуть впродовж короткого періоду впровадити порівняно складні системи з дуже короткою базою коду. Наша мета - показати докази того, що користуючись цими послугами, ми можемо надати систему, яка надає високоякісні послуги, які відповідають сучасним системам, із значно меншою базою коду. *У цій роботі ми розробляємо та впроваджуємо масштабовану структуру виконання завдань на хмарі Amazon, використовуючи різні хмарні сервіси AWS, і спрямовуємо її на підтримку обчислень із багатьма завданнями та високопродуктивних робочих навантажень.*

Найважливішим компонентом нашої системи є служба простої черги Amazon (SQS), яка діє як служба доставки вмісту для виконання завдань, дозволяючи клієнтам ефективно, асинхронно та масштабовано спілкуватися з сервісами. Amazon DynamoDB - це ще один хмарний сервіс, який використовується, щоб переконатися, що завдання виконуються рівно один раз (це потрібно, оскільки Amazon SQS не гарантує семантику доставки точно один раз). Ми також використовуємо Amazon Elastic Compute Cloud (EC2) для управління віртуальними ресурсами. Завдяки можливості SQS одночасно доставляти надзвичайно велику кількість повідомлень великій кількості користувачів, система планування може забезпечити високу пропускну здатність навіть у більших масштабах.

Сучасна аналітика даних направлена до інтерактивних коротших завдань із більшою пропускною здатністю та меншою затримкою [35][10]. Більше програм використовується до запуску більшої кількості завдань з метою покращення пропускної здатності та продуктивності програм. Хорошим прикладом для цього типу програм є багатозадачні обчислення (MTC) [15] [16] [39] [40]. Додатки MTC часто вимагають короткого часу для вирішення і можуть вимагати значних комунікацій або даних [41].

Розподілені системи управління роботою мають проблему низького використання через їх погану стратегію балансування навантаження. *Ми пропонуємо FlexQueue як систему управління робочими місцями, яка забезпечує гарне балансування навантаження та велике використання системи у великих масштабах.* Замість використання таких методів, як випадкова вибірка, FlexQueue використовує розподілені черги, щоб коректно доставити завдання сервісам, не вимагаючи при цьому системи вибору між вузлами. Розподілена черга служить великим пулом завдань, які є високодоступними. Сервіс вирішує коли отримати нове повідомлення. Такий підхід забезпечує простоту та ефективність дизайну. Більше того, застосовуючи такий підхід, компоненти системи нещільно зв'язані між собою. Тому *система буде високо масштабованою, надійною та простою в оновленні.* Хоча мотивацією цієї роботи є підтримка завдань MTC, вона також забезпечує підтримку розподіленого планування HPC. Це дозволяє FlexQueue бути ще більш гнучким, одночасно виконуючи різні типи робочих навантажень.

Основним внеском цієї роботи є:

1. Спроектувати та впровадити просту та легку структуру виконання завдань за допомогою сервісів Amazon Cloud (EC2, SQS та DynamoDB), що підтримує як навантаження MTC, так і HPC

3. Оцінка продуктивності до масштабу 1024 екземплярів порівняно з Sparrow та MATRIX: FlexQueue здатний перевершити інші дві системи після масштабування 64 екземплярів з точки зору пропускної здатності та ефективності.

Решта розділів цієї статті є такими. В Розділ II продемонстрвоано деталі проектування та реалізації FlexQueue. Розділ III оцінює ефективність FlexQueue у різних аспектах, використовуючи різні показники. Розділ IV вивчає відповідну роботу в галузі систем виконання завдань. Нарешті, розділ V обговорює обмеження поточної роботи та висвітлює майбутні напрямки цієї роботи.

II. РОЗРОБКА ТА ВПРОВАДЖЕННЯ FLEXQUEUE

Метою цієї роботи є впровадження системи планування/управління сервісами, яка відповідає *чотирьом основним цілям:*

• *Масштаб: пропонуємо збільшити пропускну спроможність із більшими масштабами через розподілені послуги*

• *Баланс навантаження: пропонуємо балансування навантаження у великих масштабах при неоднорідних робочих навантаженнях*

• *Слабко зв'язана: вирішальне значення для того, щоб зробити систему стійкою до несправностей і простою в обслуговуванні*

Для досягнення масштабованості FlexQueue використовує SQS, який є розподіленим та дуже масштабованим. Як основний елемент FlexQueue, SQS може завантажувати та завантажувати велику кількість повідомлень одночасно. Незалежність сервісів та клієнтів забезпечує зручність роботи системи в більших масштабах. Для забезпечення інших функціональних можливостей, таких як моніторинг або послідовність виконання завдань, FlexQueue також використовує такі хмарні сервіси, як DynamoDB, які є повністю розподіленими та масштабованими.

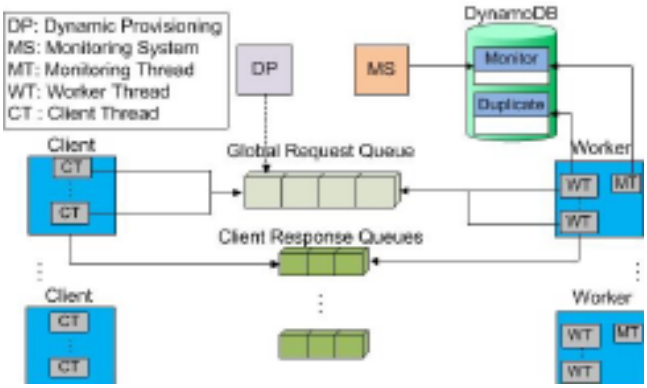


Рисунок 1. Огляд архітектури FlexQueue

Через використання хмарних служб витрати на обробку FlexQueue дуже низькі. Багато програмних викликів у FlexQueue - це виклики до хмарних служб. Маючи абсолютно незалежні сервіси і клієнтів, FlexQueue не потрібно зберігати будь-яку інформацію про свої вузли, таку як IP-адреса або будь-який інший стан своїх вузлів.

Компоненти FlexQueue можуть працювати незалежно від компонента SQS посередині, щоб відокремити різні частини фреймворку один від одного. Це робить наш дизайн компактним, надійним і легко розширюваним.

Система планування може працювати в міжплатформенній системі з можливістю обслуговування в неоднорідному середовищі, яке має системи з різними типами вузлів з

різними платформами та конфігураціями. Використання розподілених черг також допомагає зменшити залежність між клієнтами та сервісами. Клієнти та сервіси можуть самостійно змінювати свою швидкість pub/sub без будь-яких змін у системі.

Усі вищезазначені переваги покладаються на розподілену чергу, яка може забезпечити хорошу продуктивність у будь-якому масштабі. Amazon SQS - це високошвидкісна хмарна служба, яка може надати всі функції, необхідні для реалізації масштабованої системи планування завдань. Використовуючи цей сервіс, ми можемо досягти мети - мати систему, яка ідеально вписується в загальнодоступне хмарне середовище та оптимально працює на своїх ресурсах.

Система полегшує користувачам розподілену роботу над хмарними ресурсами, просто використовуючи клієнтський інтерфейс, без необхідності знати деталі базових ресурсів, а також налаштовувати кластер.

A. Архітектура

У цьому розділі пояснюється системний дизайн FlexQueue. Ми використали дизайн на основі компонентів для цього проекту з двох причин. (1) Дизайн на основі компонентів краще підходить для хмарного середовища. Це також допомагає розробити проект у вільному поєднанні. (2) Поліпшити впровадження в майбутньому буде простіше.

У наступних розділах пояснюється архітектура системи як для навантажень MTC, так і для HPC. FlexQueue має можливість запускати робочі навантаження за допомогою суміші обох типів завдань. Перший розділ показує архітектуру системи на випадок виконання виключно завдань MTC. Другий розділ описує процес у разі запуску завдань HPC.

1) Управління завданнями MTC

На Рис. 1 показані різні компоненти FlexQueue, які беруть участь лише у запуску завдань MTC. Завдання MTC визначається як завдання, яке вимагає обчислювальних ресурсів, які може задовольнити один сервіс (наприклад, там, де сервіс керує ядром або вузлом). Клієнтський вузол працює як інтерфейс для користувачів для подання своїх завдань. SQS має обмеження в 256 КБ для розміру повідомлень, якого достатньо для розміру завдання FlexQueue. Для надсилання завдань через SQS нам потрібно використовувати ефективний протокол серіалізації з низькими накладними витратами на обробку. З цієї причини ми використовуємо буфер протоколу Google. Завдання зберігає системний журнал під час процесу, передаючи різні компоненти. Таким чином, ми можемо повністю зрозуміти різні компоненти, використовуючи докладні журнали.

Основними компонентами FlexQueue для запуску завдань MTC є Клієнт, Сервіс, Глобальна черга запитів та Черги відповідей клієнта. Система також має динамічний провайзер для управління ресурсами. Він також використовує DynamoDB для забезпечення моніторингу. Для кожного сервіса запущений потік моніторингу, який періодично повідомляє про використання кожного сервіса до сховища значень ключа DynamoDB.

Клієнтський компонент не залежить від інших частин системи. Він може почати виконувати та подавати завдання без необхідності реєструватися в системі. Наявності адреси глобальної черги достатньо, щоб компонент Клієнта приєднався до системи. Клієнтська програма багатопоточна. Тож він може подавати кілька завдань паралельно. Перед відправкою будь-яких завдань Клієнт створює для себе чергу відповідей. Усі подані завдання містять адресу черги відповідей Клієнта. Клієнт також має можливість використовувати групування завдань, щоб зменшити накладні витрати на зв’язок.

З метою підвищення продуктивності та ефективності системи ми вирішили встановити два режими. Якщо в системі виконуються завдання MTC, усі сервіси працюють як звичайні сервіси. Але у випадку запуску навантажень HPC або навантажень із комбінацією завдань HPC та MTC, крім звичайних сервісів, сервіси також можуть стати або менеджерами сервісами, які керують сервісам HPC, або субсервісами, які виконують завдання HPC.

Подібно до компонента Клієнт, компонент Сервіс самостійно працює в системі. Що стосується підтримки MTC, робоча функціональність є відносно простою і прямою. Маючи загальну чергу запитів, Сервіси можуть приєднатися до системи та вийти з неї в будь-який час під час виконання. Глобальна черга запитів діє як великий пул завдань. Клієнти можуть подавати свої завдання до цієї черги, а сервіс можуть витягувати із неї завдання. Використовуючи такий підхід, масштабованість системи залежить лише від масштабованості Глобальної черги, і це не призведе до додаткового навантаження на сервісів у більших масштабах. Код сервіса також є багатопотоковим і може паралельно отримувати кілька завдань. Кожен потік може об'єднати до 10 об'єднаних завдань. Знову ж таки, ця функція зроблена для зменшення великих накладних витрат на зв'язок. Після отримання завдання робочий потік перевіряє дублювання завдання, а потім перевіряє тип завдання. У разі запуску завдань MTC це одразу ж це зробить. Потім він поміщає результати в завдання і використовуючи заздалегідь задану адресу всередині завдання, відправляє завдання назад в чергу відповіді Клієнта. Як тільки черга відповідей отримує завдання, відповідний потік клієнта витягує результати. Процес закінчується, коли Клієнт отримує всі результати своїх завдань.

2) Управління завданнями HPC

На Рис. 2 показані додаткові компоненти для запуску завдань HPC. Як зазначалося вище, у разі запуску поєднання завдань HPC та MTC кожен сервіс може виконувати різні ролі. У разі отримання завдання MTC сервіс приступає до виконання завдання самостійно. DynamoDB використовується для підтримки статусу системи, щоб сервіси могли приймати рішення про життєздатність виконання завдання HPC. По суті, у DynamoDB ми зберігаємо поточну кількість діючих менеджерів та підпорядкованих сервісів, які зайняті виконанням завдань HPC, що дає іншим сервісам уявлення про те, скільки наявних ресурсів існує.

Якщо сервіс отримує роботу HPC. DynamoDB перевіряється, щоб переконатися, що в системі достатньо доступних вузлів, що імітують для виконання завдання HPC. Якщо це задоволено, сервіс (який тепер називається менеджером робочих) розміщує n повідомлень у другому SQS (Черга завдань HPC), n - кількість робочих місць, необхідних робочому менеджеру для виконання завдання. Якщо недостатньо доступних ресурсів, вузол не може виконувати функції менеджера сервісів; натомість цей вузол перевірить Чергу завдань HPC та виступить у ролі допоміжного сервіса. Якщо в черзі HPC є повідомлення, субсервіс повідомить менеджера, використовуючи IP-адресу робочого менеджера. Менеджер сервіса та субсервіс використовують RMI для спілкування. Після виконання менеджер сервісів надсилає результат до черги відповідей, яку повинен забрати клієнт.

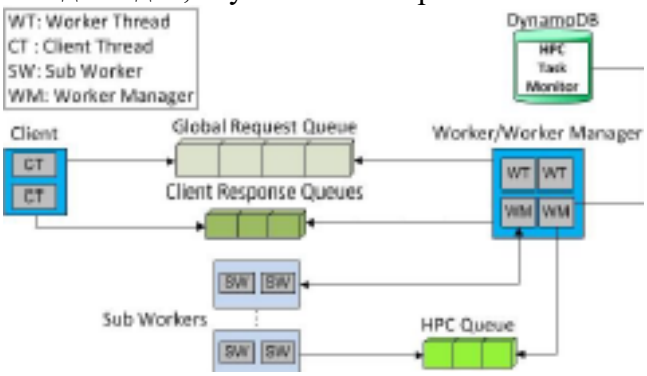


Рисунок 2. Огляд архітектури FlexQueue-HPC

Б. Питання узгодженості виконання завдання

Основним обмеженням SQS є те, що черга гарантує доставку повідомлень принаймні один раз. Це означає, що можуть існувати дублікати повідомлень, що передаються

сервісам. Існування дублікатів повідомлень походить від того, що ці повідомлення копіюються на кілька серверів з метою забезпечення високої доступності та збільшення можливості паралельного доступу. Нам потрібно запропонувати методику, яка запобігає запуску дублікатів завдань, які доставляє SQS. У багатьох типах навантажень виконання завдання більше одного разу неприйнятно. Для того, щоб бути сумісним для таких типів програм, FlexQueue повинен гарантувати рівномірне виконання завдань.

Для того, щоб мати можливість перевірити дублювання, ми використовуємо DynamoDB. Отримавши завдання, робочий потік перевіряє, чи виконується завдання вперше. Робочий потік робить умовний запис у таблицю DynamoDB, додаючи унікальний ідентифікатор завдання, який є комбінацією ідентифікатора завдання та ідентифікатора клієнта. Операція завершується успішно, якщо ідентифікатор не був записаний раніше. В іншому випадку служба видає виняток для сервіса, і сервіс скидає дублікат завдання, не запускаючи його. Ця операція є атомарною операцією.

Як ми вже згадували вище, рівно один раз доставка необхідна для багатьох типів програм, таких як наукові програми. Але є деякі програми, які мають менші вимоги до послідовності і все ще можуть функціонувати без цієї вимоги. Наша програма має можливість вимкнути цю функцію для цих програм, щоб зменшити затримку та збільшити загальну продуктивність. Ми вивчимо накладні витрати на цю функцію щодо загальної продуктивності системи в розділі оцінки.

C. Динамічне забезпечення

Однією з головних цілей у публічному хмарному середовищі є економічна ефективність. Доступна вартість ресурсів є однією з головних особливостей публічної хмари для залучення користувачів. Для такої системи, що підтримує хмару, дуже важливо підтримувати витрати на найнижчому рівні. Для досягнення економічної ефективності ми впровадили динамічну систему забезпечення. Динамічний постачальник відповідає за призначення та запуск нових сервісів до системи, щоб не відставати від вхідного навантаження.

Компонент динамічного постачальника відповідає за запуск нових екземплярів сервісів у разі нестачі ресурсів. Додаток періодично перевіряє об’єм глобальної черги запитів та порівнює довжину черги з попереднім розміром. Якщо швидкість збільшення перевищує дозволену межу, він запускає новий сервіс . Як тільки запущений, сервіс автоматично приєднується до системи. І інтервал перевірки, і поріг розміру налаштовуються користувачем.

Для того, щоб запропонувати рішення для динамічного зменшення масштабу системи, щоб утримати низькі витрат ми додали програму, щоб сервіси могли припинити екземпляр якщо виконуються дві умови. Це трапляється лише в тому випадку, якщо сервіс на деякий час переходить у режим очікування, а також якщо екземпляр наближається до поновлення. Екземпляри в Amazon EC2 стягуються щогодини, і вони будуть поновлюватися щогодини, коли користувач не вимикає їх. Цей механізм допомагає нашій системі автоматично зменшувати масштаби без необхідності отримувати будь-який запит від компонента. Використовуючи ці механізми, система здатна динамічно масштабувати вгору і вниз.

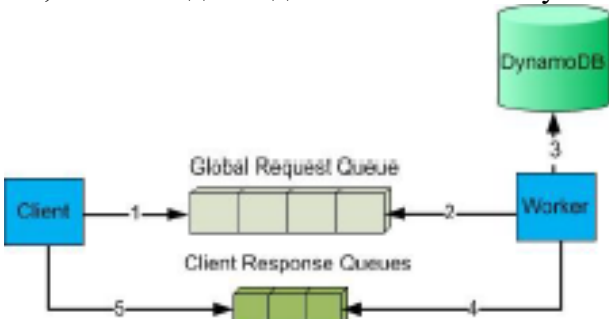


Рисунок 3. Вартість зв’язку

D. Витрати на спілкування

Затримка мережі між екземплярами в загальнодоступній хмарі порівняно висока у порівнянні з системами HPC [36][37]. Для досягнення розумної пропускної здатності та затримки нам потрібно мінімізувати накладні витрати на зв'язок між різними компонентами системи. На Рис. 3 показано кількість зв'язків, необхідних для завершення повного циклу запуску завдання. Для виконання завдання існує 5 кроків. FlexQueue також забезпечує групування завдань під час кроків. Клієнт може надсилати кілька завдань разом. Максимальний розмір пакета повідомлень у SQS становить 256 КБ або 10 повідомлень.

E. Безпека та надійність

Що стосується системної безпеки FlexQueue, ми покладаємось на безпеку SQS. SQS забезпечує надзвичайно безпечну систему, що використовує механізм автентифікації. Лише авторизовані користувачі можуть отримати доступ до вмісту Черг. Щоб зберегти низьку затримку, ми не додаємо жодного шифрування до повідомлень. SQS забезпечує надійність, надлишково зберігаючи повідомлення на декількох серверах і в декількох центрах обробки даних [18].

F. Деталі реалізації

Ми впровадили всі компоненти FlexQueue в Node.js. Наша реалізація багатопотокова як в компонентах клієнта, так і в Сервісах. Багато функцій в обох цих системах, таких як моніторинг, узгодженість, кількість потоків та розмір розбиття завдань, можна налаштувати як аргумент введення програми.

III. ОЦІНКА ЕФЕКТИВНОСТІ

Ми оцінюємо продуктивність FlexQueue і порівнюємо її з двома іншими розподіленими системами управління роботою, а саме Sparrow та MATRIX. Спочатку ми обговоримо їх особливості високого рівня та основні відмінності. Потім ми порівнюємо їх ефективність з точки зору пропускної здатності та ефективності. Ми також оцінюємо затримку FlexQueue.

A. Порівняння FlexQueue з іншими системами планування

Нам було достатньо порівняти нашу систему з Sparrow та MATRIX, оскільки ці дві системи представляють найкращі у своєму роді розподілені системи управління завданнями з відкритим кодом.

Sparrow був розроблений для досягнення мети управління мілісекундними сервісами у широкомасштабній розподіленій системі. Він використовує децентралізований підхід рандомізованої вибірки для планування завдань на робочих вузлах. У системі є кілька планувальників, кожен із яких має список сервісів і розподіляє робочі місця між сервісами, приймаючи рішення на основі довжини черги робочих місць.

У MATRIX на кожному комп'ютерному вузлі працює планувальник, виконавець та сервер ZHT. Виконавець може бути окремим ланцюжком у планувальнику. Всі планувальники повністю пов'язані з кожним, знаючи всі інші. Клієнт - це інструмент стендової розмітки, який видає запит на генерацію набору завдань і передає завдання будь-якому планувальнику. Виконавець продовжує виконувати завдання планувальника. Кожного разу, коли планувальник не має виконувати більше завдань, він ініціює адаптивний алгоритм, щоб забрати завдання у планувальників-сусідів-кандидатів. ZHT - це DKVS, який використовується для збереження метаданих задачі розподіленим, масштабованим та відмовостійким способом.

Однією з головних відмінностей між Sparrow та FlexQueue або MATRIX є те, що Sparrow розподіляє завдання, підштовхуючи їх до сервісів, тоді як FlexQueue та MATRIX використовують підтягуючий підхід. Крім того, в FlexQueue система надсилає клієнтам результати виконання завдань. Але як у Sparrow, так і в MATRIX, система не надсилає будь-

який тип повідомлень назад клієнтам. Це може дозволити Sparrow та MATRIX працювати швидше, оскільки це дозволяє уникнути ще одного кроку спілкування, але також ускладнює для клієнтів з'ясування, чи успішно виконувались їх завдання.

B. Тестовий стенд

Ми розгортаємо та запускаємо всі три системи на Amazon EC2. Ми використовували екземпляри t3.large на Amazon EC2. Ми провели всі наші експерименти в центрі обробки даних Amazon (us.eastl). Ми масштабували експерименти до 1024 вузлів. Для того, щоб зробити експерименти ефективними, клієнтський та робочий вузли запускаються на кожному вузлі. Усі екземпляри мали операційні системи Linux. Наш фреймворк повинен працювати на будь-якій ОС, що має Node12+, включаючи Ubuntu.

C. Пропускна здатність

1) *Завдання MTC*

Для того, щоб виміряти пропускну здатність нашої системи, ми запускаємо задачі sleep 0. Ми також порівняли пропускну здатність FlexQueue із Sparrow та MATRIX. У кожному екземплярі запущено 2 потоки клієнта та 4 робочі потоки. Кожен екземпляр подає 16000 завдань. Рис. 4 порівнює пропускну здатність FlexQueue із Sparrow та MATRIX у різних масштабах. Кожен екземпляр подає 16000 завдань із загальним обсягом до 16.38 мільйонів завдань у найбільшому масштабі.

Пропускна здатність MATRIX значно вища, ніж FlexQueue та Sparrow у 1 екземплярі. Причина полягає в тому, що MATRIX може виконувати багато детальних завдань локально, будь-яке планування або накладні витрати на мережу. Але на FlexQueue завдання повинні проходити через мережу, навіть якщо в системі запущений один вузол. Розрив між пропускною здатністю систем зменшується, оскільки накладні витрати на мережу додаються до двох інших систем. Планувальники MATRIX синхронізуються між собою, використовуючи загальний метод синхронізації. Маючи занадто багато відкритих TCP-з'єднань між сервісами та планувальниками у масштабі 256 екземплярів, MATRIX стає нестабільним. Продуктивність мережі в хмарі EC2 значно нижча, ніж у системах HPC, де MATRIX успішно запущений у масштабах 1024 вузли.

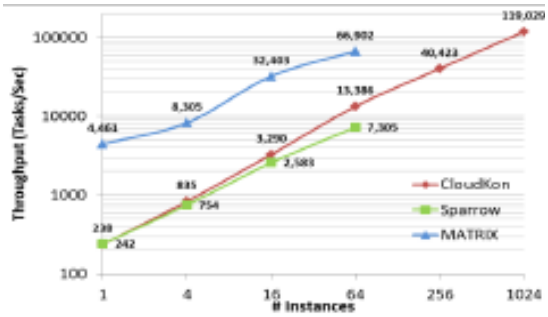


Рисунок 4. Пропускна здатність FlexQueue, Sparrow та MATRIX (завдання MTC)

Sparrow є найповільнішою серед трьох систем з точки зору пропускної здатності. Вона демонструє стабільну пропускну здатність з майже лінійним прискоренням до 64 екземплярів. Оскільки кількість екземплярів збільшується більше ніж на 64, список екземплярів для вибору для кожного планувальника на Sparrow збільшується. Тому багато сервісів залишаються без роботи, а пропускна спроможність не збільшиться, як очікувалося. Ми не змогли запустити Sparrow у масштабі 128 або 256 примірників, оскільки у планувальників було відкрито занадто багато сокетів, що призвело до збоїв системи.

FlexQueue досягає хорошого прискорення в 500 разів, починаючи з 238 завдань на секунду на 1 екземплярі до 119 тис. Завдань на секунду на 1024 екземплярах. На відміну від інших двох систем, процес планування на FlexQueue не виконується екземплярами. Оскільки

управління завданнями здійснює SQS, продуктивність системи в основному залежить від цієї послуги. Ми прогнозуємо, що пропускна здатність буде продовжувати масштабуватися, поки не досягне обмежень продуктивності SQS (яких ми не змогли досягти до 1024 екземплярів). Через бюджетні обмеження ми не змогли розширити наш масштаб понад 1024 екземпляри, хоча ми плануємо подати заявку на додаткові кредити Amazon AWS та перенести нашу оцінку на шкали екземплярів 10K, найбільшу допустиму кількість екземплярів на користувача без попереднього резервування.

2) *Завдання HPC*

Ми демонструємо пропускну здатність FlexQueue під робочим навантаженням завдань HPC. Запуск завдань HPC додає системі більше накладних витрат, оскільки буде виконано більше кроків для їх виконання. Замість того, щоб виконувати завдання відразу, менеджеру сервіса потрібно пройти кілька кроків і почекати, щоб отримати достатньо ресурсів для запуску роботи. Використання DynamoDB сповільнює проботу системи та знижує ефективність робот. Але це не впливає на масштабованість. Використання FlexQueue може значно покращити час роботи робочих навантажень HPC, паралельно виконуючи завдання, яке зазвичай виконується послідовно. Ми вибрали сервіси з 4, 8 та 16 завданнями. На кожному екземплярі запущено 4 сервісні потоки. Кількість виконаних завдань за кожною шкалою для різних сервісів однакова.

Рис. 5 порівнює пропускну здатність системи у випадку запуску завдань HPC з різною кількістю підзавдань на завдання. Результати показують, що пропускна здатність імітаційних завдань із більшою кількістю завдань на одне завдання менша. На робочих місцях із більшою кількістю завдань потрібно чекати, поки більше сервісів почнуть процес вимирання. Це додає більше затримок і уповільнює роботу системи. Ми бачимо, що FlexQueue здатний досягти високої пропускної здатності в 205 сервісах в секунду. Результати також показують хорошу масштабованість, оскільки ми додаємо більше примірників.

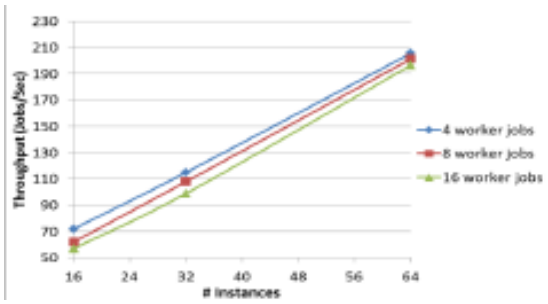


Рисунок 5. Пропускна здатність FlexQueue (завдання HPC)

D. Латентність

Для точного вимірювання затримки система повинна записати запит і відповісти на позначки часу кожного завдання. Проблема Sparrow та MATRIX полягає в тому, що в процесі виконання робочого часу сервіси не надсилають сповіщення клієнтам. Тому неможливо виміряти затримку кожного завдання, порівнюючи мітки часу з різних вузлів. У цьому розділі ми виміряли латентність FlexQueue та проаналізували латентність різних етапів процесу штампа.

На Рис. 6 показана затримка FlexQueue для режиму сну 0 мс, масштабування від 1 до 1024 екземплярів. Кожен екземпляр запускає 1 клієнтський потік і 2 робочі потоки і надсилає 16000 завдань на екземпляр.

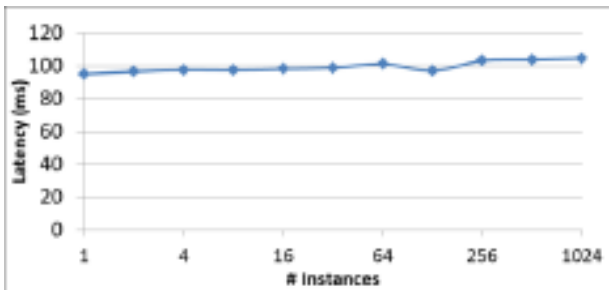


Рисунок 6. Затримка завдань FlexQueue sleep 0 ms

Час затримки системи на 1 вузлі є відносно високим, показуючи накладні витрати 95 мс, додані системою. Але це буде прийнятно у більших масштабах. Той факт, що затримка не збільшується більше 10 мс при збільшенні кількості екземплярів з 1 екземпляра до 1024, свідчить про стабільність FlexQueue. SQS як пул завдань - це надзвичайно масштабований відправник, резервне копіювання якого складається з декількох серверів, що забезпечує відправника дуже масштабованим. Таким чином, масштабування системи шляхом додавання потоків та збільшення кількості завдань не впливає на продуктивність SQS. Клієнтський та сервісний вузли завжди обробляють однакову кількість завдань у різних масштабах. Тому масштабування не впливає на екземпляри. FlexQueue включає кілька компонентів, і його продуктивність та затримка залежать від різних компонентів. Результат затримки на Рис. 6 не показує нам жодних подробиць щодо продуктивності системи. Для того, щоб проаналізувати ефективність різних компонентів, ми вимірюємо час, який кожне завдання витрачає на різні компоненти системи, реєструючи час під час процесу виконання.

Рис. 7, 8 та 9 відповідно показують сукупний розподіл етапу завдання доставки, етапу результату доставки та етапу виконання завдань на FlexQueue. Кожен етап спілкування має три етапи: надсилання, чергу та отримання. Час затримки викликів API SQS, включаючи send-task та receive-task для обох, досить високий у порівнянні з часом виконання завдань на FlexQueue. Причиною цього є дорога вартість дзвінків API веб-служби, яка використовує формат XML для зв'язку. Обробка сервісом займає 16 мс більше 50% випадків. Сюди входить DynamoDB на який відводиться 8 мс у понад 50% випадків. Це свідчить, що гіпотетично затримка FlexQueue може значно покращитись, якщо ми використовуємо низьку розподілену чергу повідомлень, яка може гарантувати рівну доставку завдань. Ми розглянемо це докладніше в наступному розділі роботи.

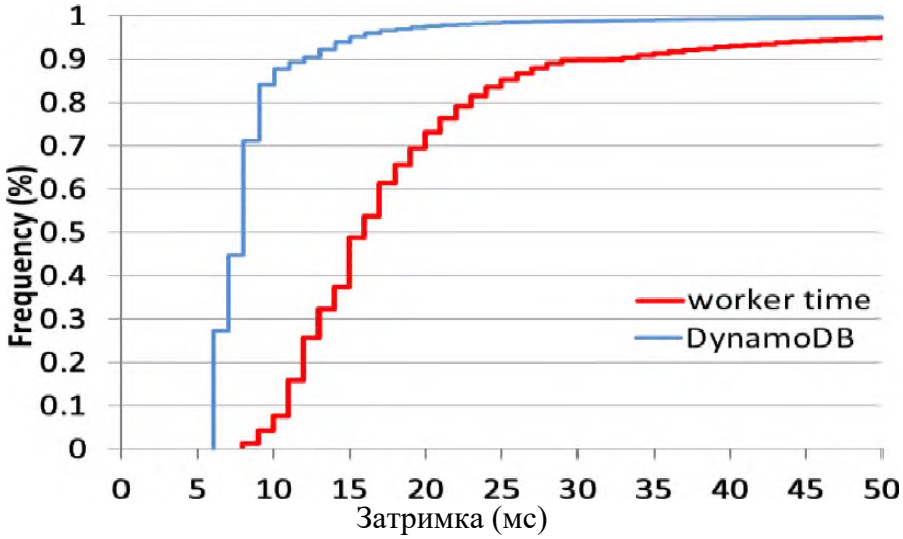


Рисунок 7. Кумулятивний розподіл затримки на етапі виконання завдання

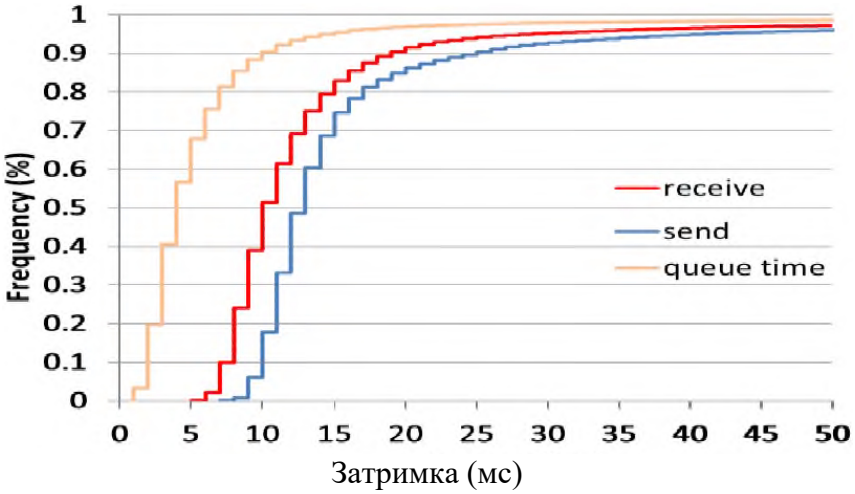


Рисунок 8. Кумулятивний розподіл затримки на етапі подання завдання

Іншим помітним моментом є різниця між часом виконання завдання та результатом доставки як у черзі, так і при отриманні назад, навіть якщо вони мають однакові виклики API. Час, витрачений завданнями на чергу відповідей, перевищує час, витрачений на чергу запитів. Причина полягає в тому, що у кожному екземплярі є два потоки сервісів і лише один потік клієнта. Тому частота виконання завдань вища, коли завдання тягнуть робочі нитки.

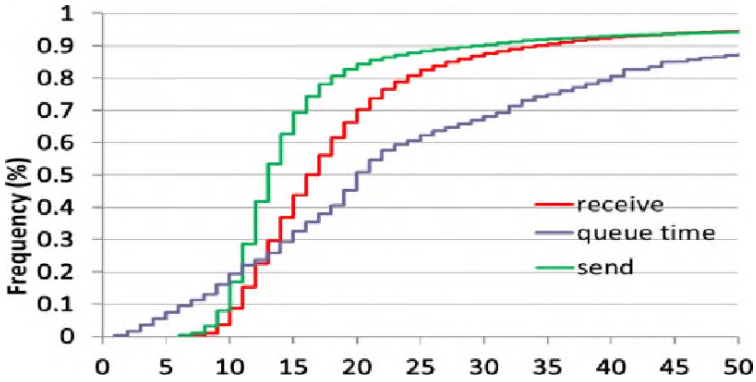


Рисунок 9. Кумулятивний розподіл затримки на етапі доставки результату

E. Ефективність FlexQueue

Для системи дуже важливо ефективно керувати системами. Досягнення високої ефективності в розподілених системах планування сервісів не є тривіальним. Важко справедливо розподілити навантаження на всіх сервсіах і зайняти всі вузли під час виконання у більших масштабах.

Для того, щоб показати ефективність системи, ми розробили два набори експериментів. Ми перевіряємо ефективність системи на випадок однорідних та неоднорідних завдань. Однорідні завдання мають певну тривалість. Тому їх легше розподілити, оскільки планувальник передбачає, що для їх запуску потрібен однаковий час. Це може дати нам хороший відгук про ефективність системи у випадку запуску різних типів завдань з різною деталізацією. Ми також можемо оцінити здатність системи виконувати дуже довгі завдання. Проблема першого експерименту полягає в тому, що для виконання всіх завдань потрібна однакова кількість часу. Це може суттєво вплинути на ефективність системи, якщо планувальник не бере до уваги об'єм завдань. Випадкове робоче навантаження може показати, як планувальник працюватиме у випадку запуску реальних додатків.

1) Однорідні навантаження

У цьому розділі ми порівнюємо ефективність FlexQueue із Sparrow та MATRIX щодо допоміжних завдань. На Рис. 10 показана ефективність завдань на системи 1, 16 та 128 мс. Ефективність FlexQueue при виконанні завдань на 1 мс нижча, ніж у інших двох системах. Як ми вже згадували раніше, затримка FlexQueue велика для дуже коротких завдань через значні накладні витрати мережі, додані в циклі виконання. Матриця має кращу ефективність на менших масштабах, але, як показує тенденція, ефективність надзвичайно падає, доки система не вийде з ладу через занадто велику кількість компіляцій TCP на масштабах 128 екземплярів або більше. Для завдань сну 16 мс ефективність FlexQueue становить близько 40%, що є низьким (у порівнянні з іншими системами). Ефективність MATRIX починається з більш ніж 93% на одному екземплярі, але знову ж вона падає до нижчої ефективності, ніж FlexQueue у більшій кількості екземплярів. Ми можемо помітити, що ефективність FlexQueue дуже стабільна в порівнянні з двома іншими системами в різних масштабах. Це показує, що FlexQueue досягає кращої масштабованості. У завданнях 128 мс для сну ефективність FlexQueue сягає 88%. Знову ж таки, результати показують, що ефективність MATRIX падає у більших масштабах.

Sparrow демонструє дуже хорошу та стабільну ефективність, виконуючи однорідні завдання до 64 екземплярів. Ефективність падає після цієї шкали для коротших завдань. Маючи занадто багато сервісів для розподілу завдань, планувальник не може мати ідеального балансу навантаження, а деякі сервіси залишаються без обробки. Тому система буде недостатньо використана, а ефективність знизиться. Система аварійно завершує роботу на масштабах 128 або більше через те, що в планувальниках надто багато сокетів

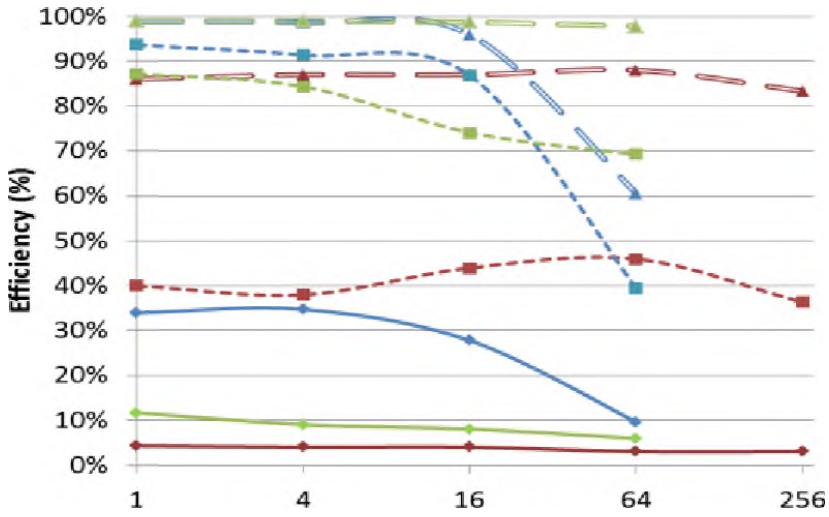


Рисунок 10. Ефективність FlexQueue, Sparrow та MATRIX, що виконують однорідні робочі навантаження різної тривалості завдань (завдання 1, 16, 128 мс)

2) Неоднорідні навантаження

Для того, щоб виміряти ефективність, ми дослідили найбільший доступний слід реальних робочих навантажень MTC [38] та відфільтрували журнали, щоб виділити лише допоміжні завдання, які об'єднали близько 2,07 млн завдань з діапазоном часу роботи від 1 мілісекунди до 1 секунди. Завдання були подані випадковим чином. Середня тривалість завдання різних екземплярів відрізняється одна від одної.

Кожен екземпляр виконує в середньому 2K завдань. Порівняння ефективності на Рис. 9 показує подібні тенденції щодо FlexQueue та MATRIX. В обох системах сервіс тягне завдання лише тоді, коли у нього є доступні ресурси для запуску завдання. Тому той факт, що тривалість виконання завдань різний, не впливає на ефективність системи. З іншого боку, у Sparrow планувальник розподіляє завдання, надсилаючи їх сервісам, у яких менше черги завдань для виконання в черзі. Той факт, що завдання мають різний час виконання, вплине

на ефективність системи. Деякі з сервісів можуть мати багато довгих завдань, а багато інших сервісів можуть виконувати короткі завдання.

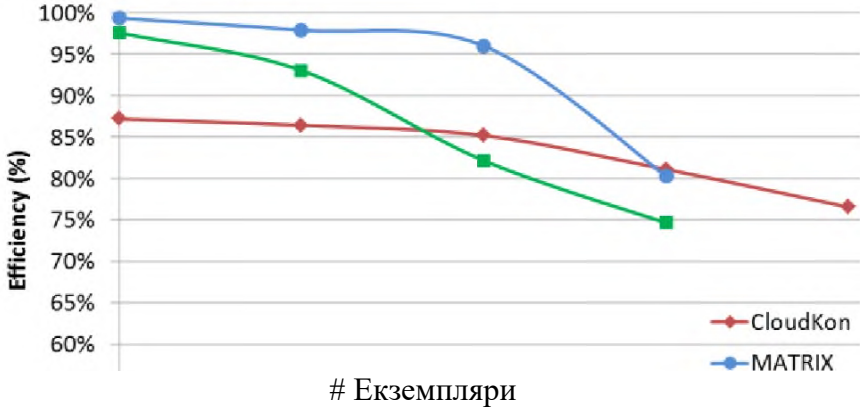


Рисунок 11. Ефективність систем, що використовують неоднорідні навантаження.

Будучи недостатньо використаним, ефективність Sparrow має найбільше падіння з 1 примірника до 64 примірників. Система не працювала в 128 випадках і більше. Подібним чином ефективність MATRIX почалася з високої ефективності, але почала суттєво падати через занадто багато відкритих сокетів на TCP-з'єднаннях. Ефективність FlexQueue не така висока, як у двох інших системах, але вона є більш стабільною, оскільки зменшується лише на 6% від 1 до 64 примірників порівняно з MATRIX на 19% і Sparrow, що падає на 23%. Знову ж таки, FlexQueue була єдиною функціональною системою на 256 екземплярах із 77% ефективністю.

F. Накладні витрати на послідовність

У цьому розділі ми оцінюємо вплив узгодженості виконання завдань на FlexQueue. На Рис. 10 показана система запуску для 16мс сну з контролером дублювання вмикається і вимикається. Витрати на інші завдання зі сну були подібні до цього експерименту. Отже, ми включили лише один із експериментів у цю роботу.

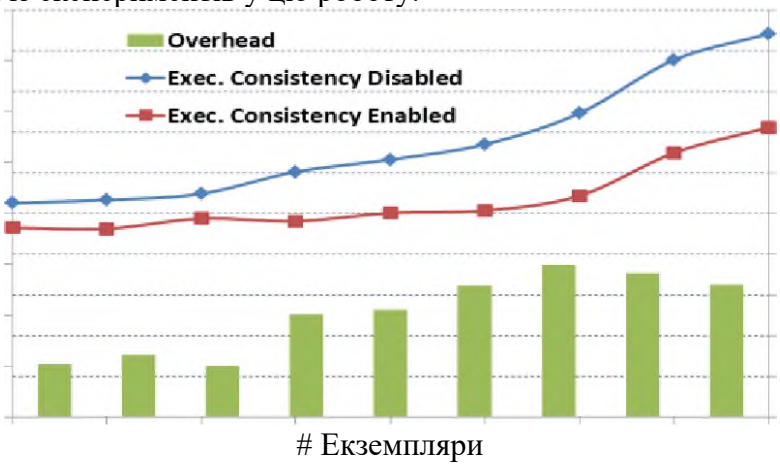


Рисунок 12. Виконайте накладні витрати на послідовність виконання завдань на FlexQueue

Накладні показники консистенції збільшуються зі збільшенням масштабу. Невідповідність у різних масштабах є результатом змінної кількості повторюваних повідомлень при кожному запуску. Це призводить до більш випадкової продуктивності системи в різних експериментах, загалом накладні витрати за шкалою менше 10 становлять менше 15%. Ці накладні витрати служать здебільшого для успішних операцій запису на DynamoDB. Ймовірність отримати дублікати завдань зростає у більших масштабах. Тому винятків буде більше. Це призводить до вищих накладних витрат. Накладні витрати у більших масштабах сягають до 35%. Однак ставка накладних витрат стабільна і не

перевищує цю ставку. Використання розподіленої черги повідомлень, яка гарантує доставку точно один раз, може значно покращити продуктивність.

IV. ПОВ'ЯЗАНА РОБОТА

Планувальники завдань можуть бути централізованими, де єдиний диспетчер керує поданням роботи та оновленням стану виконання; або ієрархічний, де кілька диспетчерів організовані в топології на основі дерева; або розподілений, де кожен обчислювальний вузол підтримує власну структуру виконання завдань.

Condor [3] був реалізований для використання невикористаних циклів процесора на робочих станціях для тривалих пакетних робіт. Slurm [2] - це менеджер ресурсів, призначений для кластерів Linux усіх розмірів. Він надає користувачам ексклюзивний та/або невиключний доступ до ресурсів протягом певного періоду часу, щоб вони могли виконувати роботу, і забезпечує основу для початку, виконання та моніторинг роботи над набором виділених вузлів. Портативна пакетна система (PBS) [5] була спочатку розроблена для задоволення потреб HPC. Вона може керувати пакетними та взаємоактивними робочими місцями, а також додавати можливість сигналізації, повторного запуску та зміни завдань. LSF Batch [19] - це компонент розподілу навантаження та черги пакетів у наборі інструментів управління навантаженням.

V. ВИСНОВОК ТА МАЙБУТНЯ РОБОТА

Широкомасштабні розподілені системи вимагають ефективної системи планування завдань для досягнення високої пропускної здатності та використання системи. Важливо, щоб система планування забезпечувала високу пропускну здатність та низьку затримку на більших масштабах та додавала мінімальну накладну витрату до робочого процесу. FlexQueue - це розподілена програма виконання завдань, що підтримує хмару, яка працює в хмарі Amazon AWS. Це унікальна система з точки зору виконання робочих навантажень HPC та MTC у загальнодоступному хмарному середовищі. Використання служби SQS дає FlexQueue перевагу масштабованості. Оцінка FlexQueue доводить, що вона є дуже масштабованою та забезпечує стабільну продуктивність у різних масштабах. Ми протестували нашу систему до 1024 екземплярів. FlexQueue зміг перевершити інші системи, такі як Sparrow та MATRIX, у масштабах 128 екземплярів або більше з точки зору пропускної здатності. FlexQueue досягає ефективності до 87%, виконуючи однорідні та неоднорідні дрібнозернисті завдання на секунду. У порівнянні з іншими системами, такими як Sparrow, він забезпечує меншу ефективність на менших масштабах. Але у більших масштабах він досягає значно вищої ефективності.

Існує багато напрямків для подальшої роботи. Один із напрямків - зробити систему повністю незалежною та протестувати її на різних державних та приватних хмарах. Ми збираємось впровадити послугу, подібну до SQS, з високою пропускною здатністю на більших масштабах доступу. За допомогою інших систем, таких як розподілена хеш-таблиця ZHT [32], ми зможемо реалізувати таку послугу. Іншим майбутнім напрямком цієї роботи є впровадження більш тісно пов’язаної версії FlexQueue і тестування її на суперкомп’ютерах та середовищах HPC під час запуску завдань HPC розподіленим способом та порівняння безпосередньо з Slurm та Slurm++ в тому ж середовищі.

Список літератури

[1] P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.

[2] M. A. Jette et. al, “Slurm: Simple linux utility for resource management”. In *Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Prarallel Procesing* (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60.

[3] D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: *The Condor Experience” Concurrency and Computation: Practice and Experience* 17 (2-4), pp. 323-356, 2005.

[4] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” *Cluster Computing*, 2002.

[5] B. Bode et. al. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” *Usenix, 4th Annual Linux Showcase & Conference*, 2000.

[6] W. Gentzsch, et. al. “Sun Grid Engine: Towards Creating a Compute Power Grid,” *1st International Symposium on Cluster Computing and the Grid* (CCGRID’01), 2001.

[7] C. Dumitrescu, I. Raicu, I. Foster. “Experiences in Running Workloads over Grid3”, *The 4th International Conference on Grid and Cooperative Computing* (GCC 2005), 2005

[8] I. Raicu, et. al. “Toward Loosely Coupled Programming on Petascale Systems,” *IEEE/ACM Super Computing Conference* (SC’08), 2008.

[9] I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” *IEEE/ACM SC 2007*.

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer,S. Shivakumar, M. Tolton, and T. Vassilakis. “Dremel: Interactive Analysis of Web-Scale Datasets. Proc.” *VLDB Endow*., 2010

[11] L. Ramakrishnan, et. al. “Evaluating Interconnect and virtualization performance for high performance computing”, *ACM Performance Evaluation Review*, 40(2), 2012.

[12] P. Mehrotra, et. al. “Performance evaluation of Amazon EC2 for NASA HPC applications”. In *Proceedings of the 3rd workshop on Scientific Cloud Computing* (ScienceCloud '12). ACM, NY, USA, pp. 41-50, 2012.

[13] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. “Case study for running HPC applications in public clouds,” In *Proc. of ACM Symposium on High Performance Distributed Computing*, 2010.

[14] G. Wang and T. S. Eugene. “The Impact of Virtualization on Network Performance of Amazon EC2 Data Center”. *In IEEE INFOCOM*, 2010.

[15] I. Raicu, Y. Zhao, I. Foster, “Many-Task Computing for Grids and Supercomputers,” *1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS) 2008.

[16] I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Dept., University of Chicago, Doctorate Dissertation, March 2009

[17] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[18] Amazon SQS, [online] 2013, http://aws.amazon.com/sqs/

[19] LSF: http://platform.com/Products/TheLSFSuite/Batch, 2012.

[20] L. V. Kal´e et. al. “Comparing the performance of two dynamic load distribution methods,” *In Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, August 1988.

[21] W. W. Shu and L. V. Kal´e, “A dynamic load balancing strategy for the Chare Kernel system,” *In Proceedings of Supercomputing* ’89, pages 389–398, November 1989.

[22] A. Sinha and L.V. Kal´e, “A load balancing strategy for prioritized execution of tasks,” In International Parallel Processing Symposium, pages 230–237, April 1993.

[23] M.H. Willebeek-LeMair, A.P. Reeves, “Strategies for dynamic load balancing on highly parallel computers,” *In IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993

[24] G. Zhang, et. al, “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers,” *In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW 10, pages 436-444, Washington, DC, USA, 2010.

[25] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. “Sparrow: distributed, low latency scheduling”. In *Proceedings of the TwentyFourth ACM Symposium on Operating Systems Principles* (SOSP '13). ACM, New York, NY, USA, 69-84.

[26] M. Schwarzkopf, A Konwinski, M. Abd-el-malek, and J. Wilkes, Omega: Flexible, scalable schedulers for large compute clusters. *In Proc. EuroSys* (2013).

[27] Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” *In Proc. Conf. on Prog. Language Design and Implementation* (PLDI), pages 212–223. ACM SIGPLAN, 1998.

[28] R. D. Blumofe, et. al. “Scheduling multithreaded computations by work stealing,” *In Proc. 35th FOCS*, pages 356–368, Nov. 1994.

[29] V. Kumar, et. al. “Scalable load balancing techniques for parallel computers,” *J. Parallel Distrib. Comput.,* 22(1):60–79, 1994.

[30] J. Dinan et. al. “Scalable work stealing,” *In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[31] A. Rajendran, Ioan Raicu. "MATRIX: Many-Task Computing Execution Fabric for Extreme Scales", Department of Computer Science, Illinois Institute of Technology, MS Thesis, 2013

[32] T. Li, et al., “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” *in IEEE International Parallel & Distributed Processing Symposium* (IPDPS ’13), 2013.

[33] Amazon DynamoDB (beta), Amazon Web Services, [online] 2013, http://aws.amazon.com/dynamodb

[34] P. Mell and T. Grance. “NIST definition of cloud computing.” National Institute of Standards and Technology. October 7, 2009.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” *in Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*, Boston, MA, June 2010.

[36] P. Mehrotra, et al. 2012. “Performance evaluation of Amazon EC2 for NASA HPC applications” In (ScienceCloud '12). ACM, New York, NY, pp. 41-50.

[37] I. Sadooghi, et al. “Understanding the cost of cloud computing”. Illinois Institute of Technology, Technical report. 2013

[38] I. Raicu, et al. “The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems,” *ACM HPDC* 2009

[39] I. Raicu, et al. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[40] Y. Zhao, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, Nova Publisher 2008.

[41] I. Raicu, et al. "Towards Data Intensive Many-Task Computing", book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009

[42] Y. Zhao, et al. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE CyberC 2011

[43] M. Wilde, et al. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", SciDAC 2009

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

ФАХОВИЙ НАУКОВИЙ ЖУРНАЛ
«КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ: ОСВІТА,НАУКА,ВИРОБНИЦТВО»

м.Луцьк., вул.Львівська 75, 43018 тел. (0332) 74-61-15 e-mail: cit@lntu.edu.ua, сайт журналу: cit-journal.com.ua

2021р.
Випуск №42

ДОВІДКА

Редакція журналу «Комп'ютерно-інтегровані технології: освіта, наука, виробництво» прийняла до друку статтю **Старовойтенко О. В**. «**Дотримання вимог платформи IoT за допомогою хмарних рішень»,** яка буде опублікована у випуску № 42, 2021р.

Відповідальний секретар журналу
мол.наук.співробітник Свиридюк К.А.

15.12.2020р

Старовойтенко Олексій

Національний технічний університет України

"Київський політехнічний інститут імені Ігоря Сікорського"

Дотримання вимог платформи IoT за допомогою хмарних рішень

Ключові слова: AMQP, pub/sub, ZeroMQ, MQTT, XMPP

Аннотація

Інтернет речей (IoT) надає широкий спектр програм, що забезпечують підвищену обізнаність та контроль за фізичним середовищем. Поточні системи, як правило, локально сприймають і опрацьовують фізичні явища, а потім переносяться на хмарну інфраструктуру pub / sub (публікації / підписки) для розподілу даних датчиків та контролю серед кінцевих користувачів та зовнішніх служб. Незважаючи на популярність рішень pub / sub досі незрозуміло, які функції проміжне програмне забезпечення повинно мати, щоб успішно відповідати конкретним вимогам домену IoT. Питання як велика кількість підключених пристроїв, які лише епізодично надсилають невеликі повідомлення датчиків, впливають на пропускну здатність. У цій роботі ми розглядаємо дані обмеження, аналізуючи основні вимоги платформ IoT та оцінюючи, які з цих функцій підтримуються відомими open source (відкритими рішеннями) pub / sub. Далі ми проведимо оцінку продуктивності в загальнодоступній хмарі, використовуючи чотири популярні реалізації pub / sub: RabbitMQ (AMQP), Mosquitto (MQTT), Ejabberd (XMPP) та ZeroMQ. Ми дослідимо максимальну стійку пропускну здатність та затримку в справжніх умовах навантаження, використовуючи дані від реальних датчиків. Хоча основні функції подібні, аналізовані системи pub / sub відрізняються своїми можливостями фільтрації, семантичними гарантіями та кодуванням. Наша оцінка показує, що ці відмінності можуть мати помітний вплив на пропускну здатність та затримку хмарних платформ IoT.

Майбутня мережа фізичних об'єктів, Інтернет речей (IoT) забезпечує зондування навколишнього середовища в режимі реального часу і надає можливість здійснювати автономну взаємодію як реакцію на зміни у фізичному світі. Парадигма призвела до появи різних «розумних» додатків, наприклад розумне місто, розумне підприємство та розумний дім, які, як очікується, збільшать кількість підключених пристроїв до мільярдів [9].

Завдяки своїй величезній обробній потужності, швидкій мережі та надійному сховищу, хмарні обчислення можуть забезпечити інфраструктуру для забезпечення, управління великою кількістю сенсорних пристроїв. Модель гнучкого розподілу ресурсів на вимогу та витрат на оплату за час роботи може забезпечити еластичне узгодження зростаючих вимог до комунікацій, обчислень та зберігання, пов'язаних із програмами IoT [13].

На відміну від класичних бездротових сенсорних мереж (WSN), які пристосовані і використовуються одним додатком, додаткова цінність IoT полягає в загальному використанні апаратного забезпечення датчиків неоднорідними програмами. Пристрої з обмеженими даними зчитують датчики лише один раз, і вони будуть розподілені між кількома зацікавленими програмами та службами. Масштабований рівень обміну повідомленнями на основі хмари може бути використаний для вирішення складного аспекту узгодження потоків даних сенсора та зацікавлених програм або служб та відповідного розподілу даних.

Структура обміну повідомленнями «pub / sub» дає змогу вибірково розповсюджувати повідомлення і стала добре усталеною моделлю розповсюдження даних, наприклад для даних фондового ринку або погоди. Незважаючи на популярність pub / sub рішень у відповідних застосунках, все ще відсутні дані про особливості, як pub / sub система повинна відповідати конкретним вимогам IoT, ступінь цих функцій, що надаються існуючими рішеннями. У цій роботі ми робимо три основні внески у вирішення цих проблем:

1. Ми формуємо набір вимог до систем pub / sub для хмарних IoT та аналізуємо, які доступні відкриті рішення відповідають цим вимогам;
2. Ми визначаємо три класи трафіку IoT на основі реалістичних випадків використання;
3. Ми проводимо хмарну оцінку продуктивності в загальних реалістичних умовах, використовуючи справжні файли трасування.

1. Pub / sub в контексті IoT

Сьогодні декілька відомих академічних та комерційних платформ IoT мають спільну хмарну архітектуру, подібну до тієї, що зображена на рис. 1 [13]. Пристрої підключені до шлюзів, які передають дані датчиків на хмарний рівень за допомогою посередника повідомлень. Додаткові послуги, напр. зберігання, аналіз або агрегування даних, а програми, що стоять перед користувачами, підключаються до посередника, щоб отримати доступ до даних. Ми бачимо тенденцію використовувати посередника повідомлень, використовуючи шаблон pub / sub для розповсюдження даних серед кількох зацікавлених програм [22]. Система pub / sub - це проміжне програмне забезпечення, орієнтоване на повідомлення (MoM) [10], що забезпечує розподілений, асинхронний обмін даними між джерелом повідомлень (видавець) та споживачами повідомлень (передплатник). Проміжне програмне забезпечення pub / sub пропонує три основні типи роз’єднання [12], що є придатним для широкомасштабних розгортань IoT:

1. Джерела повідомлень та споживачі роз'єднуються в часі, тобто їх не потрібно підключати одночасно;
2. Повідомлення адресовані не явно конкретному споживачеві, а символічній адресі (каналу, темі);
3. Повідомлення є асинхронними, не блокуючими.

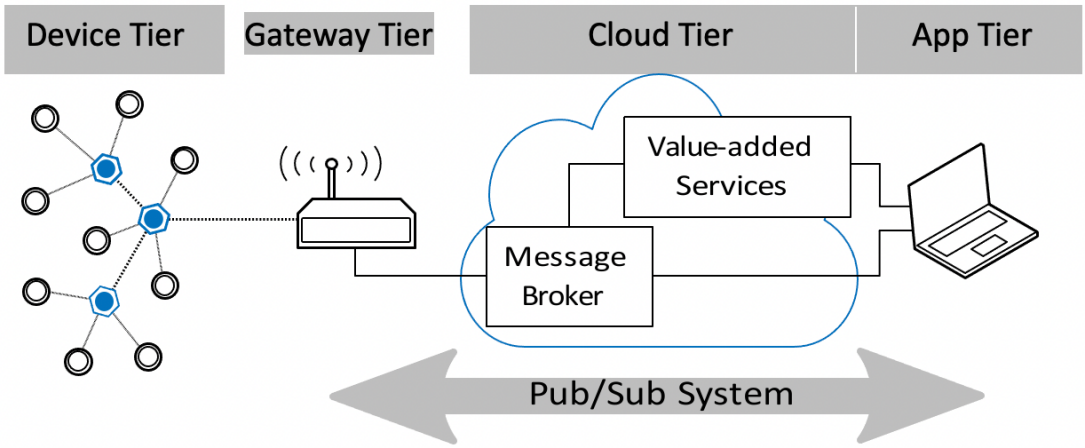


Рис. 1 Агрегування повідомлень в системі pub/sub

Основним елементом систем pub / sub є відповідність між видавцями та передплатниками, яка може базуватися на різних типах фільтрації, переважно за темою чи вмістом. Фільтрування зазвичай здійснюється декількома спеціалізованими посередниками повідомлень. У схемі, заснованій на темах, символічні адреси каналів - це теми, зазвичай у формі рядків, тобто виробники публікують, а споживачі підписуються на теми. Повідомлення доставляються лише відповідним абонентам. Теми можуть бути організовані ієрархічно, тобто тема може бути підтемою іншої теми. Тоді підписки на батьківську тему зазвичай також відповідають усім підтемам. Тематична фільтрація - це статична схема, що пропонує лише обмежену виразність. На противагу цьому, у схемі, що базується на вмісті, абоненти не мають статичного збігу на основі тем, а на основі вмісту окремих повідомлень, наприклад якщо значення досягає певного порогу, визначеного передплатником.

Подібним підходом до великомасштабних даних датчиків є обробка потоків. На відміну від повідомлень на основі pub / sub, програми обробки потоків виступають як складні безперервні запити щодо вхідних потоків даних, що генерують потоки результатів. Особливість потоків обробки потоків полягає в перетворенні вхідного потоку, тоді як pub / sub фокусується на розподілі даних. Незважаючи на те, що ми розглядаємо потокову обробку як перспективний підхід до обробки та аналізу великих потоків даних, ми стверджуємо, що обробка потоку сама по собі не дасть змоги загально використовувати апаратне забезпечення датчиків у кількох додатках, що є однією з ключових властивостей бачення IoT.

Незважаючи на те, що pub / sub є добре встановленим та вивченим шаблоном обміну повідомленнями, використання pub / sub в хмарних налаштуваннях IoT все ще не досліджено в деталях. Існує безліч існуючих рішень без надійних даних про те, яке рішення найкраще відповідає конкретним вимогам IoT. У наступному розділі ми починаємо вирішувати ці проблеми, визначаючи довідкові сценарії для хмарних розгортань IoT.

2.1 Довідкові сценарії

Перш ніж виводити набір вимог до загальних програм IoT, спочатку ми вводимо підмножину можливих сценаріїв, визначивши в якості прикладів три еталонні сценарії в контексті розумних міст:

2.1.1 Соціальна служба погоди

Першим варіантом використання, який ми передбачаємо, є соціальна метеорологічна служба, де власники сенсорних пристроїв підключаються і діляться своїми наявними датчиками з громадськістю. Прикладом є система опалення, вентиляції та кондиціонування повітря (ОВК) в офісній будівлі, яка періодично вимірює температуру та вологість. Система може або публічно надавати власні дані, або використовувати показання з інших подібних систем, або контролювати прогноз погоди, щоб покращити час нагрівання чи охолодження, заощаджуючи енергію та покращуючи комфорт.

2.1.2 Розумний спільний доступ до автомобіля

Другий варіант використання - це система спільного використання автомобілів. Поява підключених автомобілів призвело до появи спільного використання. Спільне використання автомобілів зменшує кількість необхідних місць для паркування та, в свою чергу, зменшує кількість трафіку. У майбутньому фізичні особи можуть почати здавати свої машини в оренду подібним чином: приватні машини можуть періодично запропоновані в оренду , надаючи додаткову інформацію, таку як положення, рівень палива чи стан.

2.1.3 Моніторинг дорожнього руху

Щоб зменшити обсяг руху, іншим варіантом є відстеження дорожнього руху. Камери фотографують умови дорожнього руху та відправляють їх у хмару. Хмарний сервіс аналізує їх та оцінює стан дорожнього руху. Зацікавлені водії або навігаційні підрозділи можуть використовувати цю інформацію, щоб спланувати свій маршрут.

2.2 Вимоги до pub / sub в хмарному IoT

У цьому розділі викладено конкретні вимоги щодо Інтернету речей, які впливають на вибір відповідного проміжного програмного забезпечення pub / sub. Ми починаємо з функціональних вимог, які походять від загальних програм IoT і проілюстровані наведеними вище довідковими сценаріями:

1. Шаблон обміну повідомленнями: Усі випадки використання вимагають моніторингу показань датчиків.
2. Фільтрування: Зацікавлені сторони зазвичай хочуть отримати лише підмножину всієї інформації, наприклад датчики погоди в тому самому районі. Можливості фільтрації проміжного програмного забезпечення визначають виразність підписок, які може видавати клієнтська програма.

3. Семантика QoS: хоча втрата повідомлень даних датчика може бути допустимою в деяких налаштуваннях, інші системи можуть вимагати гарантованої доставки повідомлень, наприклад розумний автомобіль може видавати тривожні повідомлення у випадку крадіжки.
4. Топологія: У контексті хмарно-орієнтованого IoT кожен проміжний пакет pub / sub повинен підтримувати централізовану топологію, де брокер пересилає повідомлення на основі запитуваних фільтрів.
5. Формат повідомлення: Через неоднорідність апаратного забезпечення датчиків, як проілюстровано у вибраних випадках використання, складно передбачити, які саме будуть подані дані датчиків формату.

Ці вимоги будуть використані в наступному розділі для виведення таксономії систем pub / sub. На додаток до вищезазначених вимог, існують дві важливі нефункціональні вимоги до хмарної системи pub / sub:

1. Через велику кількість очікуваних пристроїв система повинна бути масштабованою;
2. Оскільки дані можуть бути терміновими, обмін повідомленнями повинен мати низьку затримку. Ми досліджуємо ці вимоги окремо під час оцінки результатів роботи в Розділі 3.

2.3 Таксономія pub / sub

Спираючись на загальні вимоги до платформ IoT, у цьому розділі ми представляємо підмножину протоколів pub / sub та надамо класифікацію відповідно до визначених вимог. Ми обмежуємось протоколами з відкрито доступна специфікація протоколу з реалізаціями з відкритим кодом, які широко використовуються. Ми обираємо протоколи AMQP, MQTT, XMPP та ZeroMQ для подальшої оцінки. Далі ми коротко описуємо ці протоколи:

AMQP: Розширений протокол черги повідомлень (AMQP) [2] розроблений як відкрита заміна власних протоколів обміну повідомленнями у галузі фінансових послуг. AMQP використовується в популярних реалізаціях, таких як RabbitMQ [22], Apache ActiveMQ [4] та Apache Apollo [5]. Версія AMQP 0-9-1 - це відкрита та безкоштовна специфікація як протоколу дротового рівня, так і моделі брокера. AMQP 1.0 нещодавно став Організацією з удосконалення стандартних стандартних інформаційних стандартів (OASIS), але включає в основному новий провідний протокол і лише абстрактні вимоги до посередника. Більшість функціональних можливостей визначаються брокером та його поведінкою, яка не є частиною AMQP 1.0. Тому ми зосереджуємося на версії 0-91, яка визначає найбільш широко застосовувану модель брокера.

MQTT: Телеметричний транспорт черги повідомлень (MQTT) [18] - це чистий протокол pub / sub для обмежених пристроїв та мереж з низькою пропускною здатністю, високою затримкою та ненадійністю, розроблених IBM та стандартизованих OASIS. Він має різні реалізації з відкритим кодом, такі як Apache ActiveMQ [4] та клієнтська бібліотека Eclipse paho [11] Існує спроба перенести MQTT на датчики вузлів у зменшеному варіанті MQTT -S [16]. MQTT та MQTT-S відкрито публікуються з безкоштовними ліцензіями.

XMPP: Розширюваний протокол обміну повідомленнями та присутності (XMPP) [18] бере свій початок у протоколі обміну миттєвими повідомленнями Jabber, мотивованому різноманітністю власних протоколів чату. XMPP базується на потоковій

передачі XML. Основний протокол стандартизований у декількох RFC IETF та розширений протоколами розширення XMPP (XEP), включаючи обмін повідомленнями pub / sub [16]. XMPP - це протокол, який стоїть за сервером обміну миттєвими повідомленнями Ejabberd [16] та сервером Openfire [17].

ZeroMQ: ZeroMQ [15] - це бібліотека обміну повідомленнями, що пропонує API сокета з більш досконалими шаблонами обміну повідомленнями, ніж сокети Берклі. Підтримувані шаблони включають запит / відповідь та pub / sub. Протоколом дротового рівня є Ze roMQ Message Transport Protocol (ZMTP), що доступний за загальною публічною ліцензією GNU. ZeroMQ - це бібліотека обміну повідомленнями, що дозволяє додавати нові функції до основного протоколу. Для порівняння бібліотеки з іншими системами ми розглядаємо лише функції, включені в офіційну документацію.

В решті цього розділу ми наводимо класифікацію pub / sub протоколів щодо вимог платформи IoT на основі хмарних технологій, яка наведена в табл 1.

		AMQP	MQTT	XMPP	ZeroMQ
Messaging Pattern	Pub/Sub	X	X	X[1]	X
	Point-to-point	X		X	X
Filtering	Topic-based	X	X	X1	X
	Content-based				
QoS Semantics	At-most-once	X	X	X	X
	At-least-once	X	X		
	Exactly-once	X	X		
	Last value caching	X[2]	X	X	X
Topology	Decentralized				X
	Centralized	X	X	X	X
	Hybrid			X1	
Message Format	Payload agnostic	X	X	X	X
	Binary Encoding	X	X		X

Таблиця 1 Класифікація відкритих pub / sub протоколів проміжного програмного забезпечення

Ми використовуємо шаблони pub / sub та запит / відповідь для моніторингу сенсорного пристрою. Хоча AMQP і ZeroMQ підтримують обидва шаблони, MQTT - це чистий протокол pub / sub. Запит / відповідь все ще можливий, але він повинен використовувати спеціальні теми, щодо яких пристроїв чи служб підписуються на запити. XMPP спочатку є протоколом чату для прямого обміну повідомленнями, який був розширений XEP-0060 для підтримки публікацій / додаткових повідомлень [16].

Розглянуті рішення надають лише тематичні pub / sub з ієрархічними темами. AMQP та MQTT підтримують узагальнюючі символи на кожному рівні теми, тоді як ZeroMQ підтримує лише відповідність префіксів.

Ми зосереджуємося на хмарних платформах IoT із централізованою топологією. Це означає, що спеціалізовані розподілені брокери в центральній точці здійснюють узгодження та доставку повідомлень. Усі розглянуті протоколи підтримують таку топологію. В умовах очікуваної кількості пристроїв буде вигідно зменшити навантаження на центральних брокерів. XMPP використовує розподілену мережу серверів XMPP як посередників. Також можливо встановити позасмугові однорангові з'єднання за допомогою XEP-0065, що відповідає гібридній топології. ZeroMQ можна використовувати без центрального брокера як P2P. Крім того, його сокети pub / sub можуть бути використані для формування ієрархії брокерів, які вибірково пересилають публікації та підписки, дозволяючи горизонтальне масштабування.

Усі протоколи, як правило, є агностичними щодо формату повідомлення. AMQP, MQTT та ZeroMQ використовують двійкове кодування, дозволяючи будь-який формат, тоді як XMPP використовує XML, який також може транспортувати інші формати, але зазвичай ціною меншої ефективності пропускної здатності та збільшення накладних витрат на аналіз. AMQP додатково пропонує систему типів, яка відображає передані типи даних у загальні типи даних багатьох мов та платформ.

3 Хмарна оцінка продуктивності

Хоча аналізовані системи pub / sub пропонують однакові основні функції, вони відрізняються семантикою QoS, кодуванням та виразністю фільтрів, що може мати незначний вплив на продуктивність. Щоб проаналізувати нефункціональні вимоги до масштабованості та низької затримки, ми оцінюємо системи в реальних умовах. Одним із основних аспектів цих реалістичних умов є розгортання в хмарі. Іншим аспектом є реалістичне відтворення реальних джерел трафіку за допомогою великої кількості пристроїв.

3.1 Показники ефективності

Ми вибираємо показники продуктивності відповідно до нефункціональних вимог до масштабованості та низької затримки. Оскільки хмарні платформи IoT, можливо, в майбутньому повинні будуть підтримувати мільярди пристроїв [13], важливо, щоб система могла масштабуватися горизонтально. Зазвичай це робиться за допомогою кластера брокерів. Однією з можливостей буде шардинг на основі тем. Однак для роботи всієї системи найважливіше, щоб один брокер міг обробляти якомога більше датчиків і, отже, повідомлень датчиків, тобто він пропонує найбільшу пропускну здатність повідомлень на даній фізичній або віртуальній машині. Тому ми вивчаємо пропускну здатність одного брокера, яку ми визначаємо як кількість повідомлень в секунду, яку може обробити брокер.

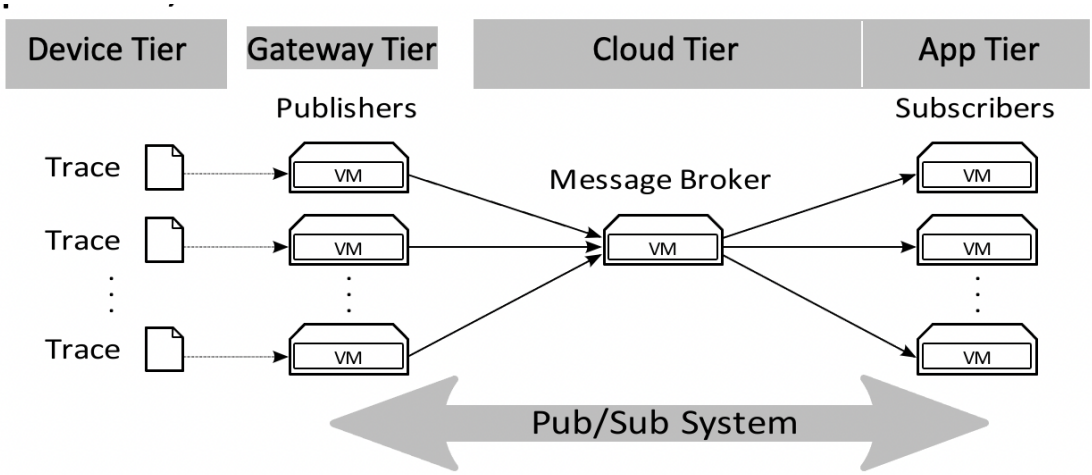


Рис. 2 Налаштування експерименту: реальні дані датчиків збираються та відтворюються в екземплярах хмари з файлів трасування; брокер розподіляє ці дані між процесами підписки, які також моделюються за допомогою хмарних віртуальних машин.

Protocol	Broker	Version	Release date	Language
AMQP	RabbitMQ	3.4.4	2015-02-11	Erlang
MQTT	mosquitto	1.4	2015-02-18	C
XMPP	ejabberd	15.02	2015-02-17	Erlang
ZeroMQ	XPUB/XSUB	-	-	C

Табл. 2 Огляд вивчених систем pub / sub

Іншою важливою вимогою є низька затримка між вимірюванням значень датчика та доставкою до зацікавлених застосунків. У цьому дослідженні ми не зосереджуємося на загальній затримці, а лише на тій частині затримки, яку запроваджує система pub / sub. Таким чином, ми вивчаємо кінцеву затримку між шлюзами та зацікавлені програми, тобто між видавцями та передплатниками.

3.2 Загальні налаштування вимірювань

Далі ми представляємо налаштування вимірювань, які ми використовуємо для оцінки вибраних систем за допомогою представлених показників. Наші вимірювання спрямовані на максимально реалістичну оцінку роботи брокера обміну повідомленнями на основі хмар. Огляд нашої системи наведено на рис. 2. Оскільки брокер, як очікується, буде працювати на хмарній віртуальній машині, ми розгортаємо брокер на екземплярі в Amazon Elastic Compute Cloud (Amazon EC2) [1], який є одним із провідних хмарних провайдерів.

Оскільки ми не можемо розгорнути апаратне забезпечення сенсорів, що взаємодіє з брокером повідомлень, ми дотримуємося подібного підходу, як Rege та ін. [17] та відтворювати джерела трафіку на екземплярах хмар. Рівень пристрою змодельований файлами трасування, зібраними із сенсорних пристроїв реального світу. Детальніше - у наступному розділі. Файли трасування використовуються для копіювання поведінки

синхронізації та розмірів повідомлень трафіку, що спостерігаються на шлюзах IoT. Шлюзи, які виконують роль видавців у системі pub / sub, а також застосунки, що підписуються, копіюються на хмарних віртуальних машинах.

Щоб уникнути відтворення одних і тих самих файлів трасування на різних шлюзах і потенційно зміщення результатів, розподіл розмірів повідомлень та часу між повідомленнями обчислюється для кожних 15-хвилинних інтервалів і відтворюється. Це зберігає інформацію про різні розміри та швидкість повідомлень протягом дня, які зазвичай співвідносяться у розумному місті, але уникає нереальних упереджень, спричинених штучним високим співвідношенням.

Ми оцінюємо протоколи, використовуючи популярні реалізації представників. Наш вибір брокерів наведено в табл. 2. Брокер ZeroMQ використовує сокети XPUB і XSUB, як зазначено в документації ZeroMQ, і написаний на мові програмування C. Брокери Erlang використовують високопродуктивний Erlang (HiPE). Оскільки ми не враховуємо безпеку, шифрування вимкнено. Брокерські та публікаційні та передплатні процеси розгортаються на екземплярах EC2, що працюють у поточному 64-бітній версії довгострокової підтримки Ubuntu (14.04) (ami-234ecc54). Апаратне забезпечення узагальнено в табл. 3. Брокери працюють на одному екземплярі m4.large з двома ядрами, як правило, Intel Xeon E5-2676 v3 з 32 ГБ оперативної пам'яті. Виробники та споживачі повідомлень розміщені на екземплярах m3.medium з одним ядром та 3,75 ГБ оперативної пам'яті. Ми використовуємо 10 екземплярів шлюзу, в яких розміщується до 60 віртуальних шлюзів публікації та до 60 процесів підписки кожен. Під час експериментів контролюється навантаження центрального процесора на кожен екземпляр, тому вузькі місця на стороні клієнта не будуть розцінюватися як погана робота брокера.

Для генерації трафіку та тиражування процесів підписки ми використовуємо клієнт, написаний мовою програмування C. Застосунок публікує та підписується за допомогою єдиного інтерфейсу, який відображається на інтерфейсах окремих систем pub / sub за допомогою спеціальних плагінів. Компонент генерації трафіку та ведення журналу залишається незмінним у кожному експерименті. Щоб уникнути побічних ефектів дискового вводу-виводу, ми буферизуємо всі результати в пам’яті, перш ніж записувати їх на диск. Ми використовуємо непостійні повідомлення з максимальною кількістю семантичних показників із фіксованим коефіцієнтом розсилки 1, тобто кожен видавець має одного передплатника.

У цій роботі ми вдаємося до використання чітко визначеного коефіцієнта розмивання як першого кроку, але працюємо над вдосконаленими моделями абонентів для подальшої роботи. По-друге, наша система спрощує вимірювання пропускної здатності та затримки, оскільки ми можемо розгортати видавців та підписників теми на одній і тій же хмарній віртуальній машині, уникаючи необхідності точної синхронізації між процесами.

Ми оцінюємо кожен протокол, використовуючи три сценарії робочого навантаження, аналогічні нашим еталонним сценаріям, які ми визначаємо нижче.

3.3 Сценарії навантаження

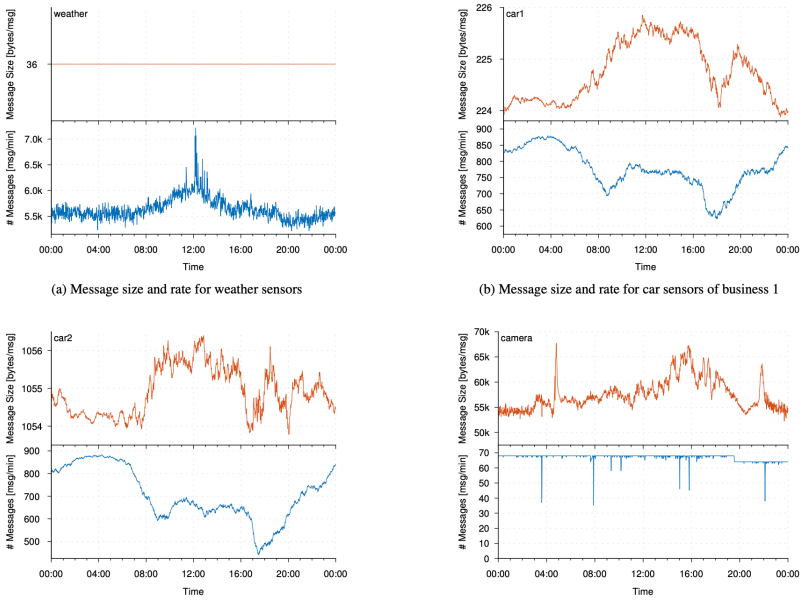
Ми узагальнюємо три еталонні сценарії в Розділі 2.1 для трьох основних класів датчиків, що виробляють чіткі схеми руху:

3.3.1 Прості датчики

Датчики погоди - це приклад того, що ми визначаємо як простий датчик, який відбирає по одній величині за раз, як правило, незалежний від інших датчиків і компактне двійкове представлення.

Тестовий стенд бездротового внутрішнього датчика (TWIST) [14] з 90 вузлами датчиків Telos Rev. B [18], оснащений датчиком вологості та температури Sensirion SHT11, датчиком видимого світла Hamamatsu S1087 (від 320 нм до 730 нм), що вимірює фотосинтетично активне випромінювання (PAR)) і Hamamatsu S1087-01 (від 320 нм до 1100 нм), видимий для ІЧ-датчика світла, що вимірює загальну сонячну радіацію (TSR). Ми збираємо зразки згаданих вище датчиків на кожному вузлі датчика щосекунди, використовуючи програму TinyOS, яка відправляє дані в одному пакеті до послідовного інтерфейсу, який виставляється через порт Universal Serial Bus (USB). Процес збору використовує можливості тестового стенду TWIST: кожен вузол датчика підключений до одного супервузла, який пересилає послідовні пакети на сервер TWIST, де записується файл трасування з усіма значеннями датчика. Створюється файл трасування із загальною кількістю даних за 72 години.

Ми використовуємо ці необроблені дані датчиків для створення сигналів датчиків для публікації. Ми вирішили надіслати повідомлення з оновленими даними датчика, лише якщо перевищені розумні порогові значення, тобто якщо датчик відбирає значення, близьке до значення, відібраного раніше, шлюз не передаватиме дані повторно. Ми вважаємо, що це розумна поведінка в бездротових сенсорних мережах, де енергії недостатньо. Огляд сформованого файлу трасування наведено на рис 3a, де показано один день зразкових даних. Кількість виданих повідомлень за секунду варіюється і досягає максимуму близько обіду. Оскільки повідомлення кодуються двійково, розмір повідомлення постійний на рівні 36 байт.



(a) Message size and rate for weather sensors

(b) Message size and rate for car sensors of business 1

(c) Message size and rate for car sensors of business 2

(d) Message size and rate for cameras

Рис. 3 Часові характеристики сенсорних файлів датчиків протягом одного конкретного дня вимірювання; вказано розмір повідомлення та швидкість повідомлень за хвилину.

3.3.2 Складні датчики

На відміну від простих показників, які пов'язані, але також мають значення самі по собі, датчики автомобіля в основному мають значення лише разом. Беручи до уваги приклад спільного використання автомобілів, лише рівень палива, наприклад, не дуже значущий для потенційних споживачів без положення автомобіля. Ми визначаємо такі датчики як складні датчики, що являє собою комбінацію декількох простих датчиків, де показання датчиків згруповані, щоб утворити цілісні показники, які підключені та передаються разом як одне повідомлення. Ці повідомлення зазвичай кодуються за допомогою JSON або XML.

Дані про транспортний засіб, як складний тип даних датчиків, походять із загальнодоступних веб-сайтів двох підприємств спільного користування автомобілями [6, 11]. Подібно до того, як браузер отримує доступ до даних, щоб скласти карту доступних автомобілів, ми щохвилини отримуємо доступ до загальнодоступних даних цих служб для Берліна і розподіляємо відповідь в індивідуальних звітах про стан для кожного автомобіля. Ми реєструємо розмір цих індивідуальних показань і використовуємо їх для відтворення еквівалентних повідомлень у наших вимірах, не зберігаючи жодних фактичних даних про машини. Файли трасування містять дані за тиждень для обох компаній, що надають послуги спільного використання автомобілів.

Індивідуальні звіти про стан автомобілів використовуються як повідомлення для публікації. Обґрунтування полягає в тому, що кожен автомобіль опублікує його наявність разом із додатковою інформацією про автомобіль, такою як стан чи стан. Один день даних наведено на рис. 3b та 3c для кожного бізнесу, що займається спільним використанням автомобілів. Зміна кількості доступних автомобілів відбувається за тією ж загальною схемою і є низькою близько 9:00 та між 17:00 і 20:00, коли багато пасажирів орендують машини. Хоча компанія, що займається спільним сполученням автомобілів, зазвичай отримує звіти про стан або з фіксованим інтервалом, або за певних подій, кількість приватних автомобілів, що надаються громадянами в оренду, мабуть, має подібну модель. Розмір повідомлення досить стабільний протягом дня, але суттєво відрізняється між двома компаніями.

3.3.3 Мультимедійні датчики

Прикладами камер як датчиків IoT є зображення камер спостереження або відеопотоки, а також спостереження за дорожнім рухом за допомогою камер. Зображення дорожньої камери можна аналізувати в режимі реального часу, щоб запропонувати альтернативні шляхи до розумних навігаційних систем. Ми визначаємо клас мультимедійних датчиків, які виробляють носії, такі як аудіо, нерухомі зображення або відеоматеріали. Ці медіапотоки кодуються як двійкові фрагменти даних.

Як приклад медіа-даних ми використовуємо загальнодоступні камери дорожнього руху міста Берлін. Подібно до автомобілів, ми отримуємо доступ до зображень з камери з фіксованим інтервалом у 30 секунд і реєструємо розмір відповіді. Файли трасування містять дані за тиждень для 34 камер.

Окремі зображення використовуються як повідомлення для публікації. Приклади даних за день показані на рис. 3d. Середній розмір повідомлення збільшується до півдня, можливо, тому, що зображення з більшою кількістю світла матимуть більше деталей, які не будуть стискатися так добре, як темні зображення. Швидкість повідомлень є досить

стабільною, як очікується, приблизно два зображення на хвилину на камеру. Деякі камери, як правило, не оновлюються надійно, тому ми ігноруємо всі точні копії, пояснюючи незначні коливання.

3.4 Вимірювання пропускної здатності

Ми використовуємо класи трафіку, введені раніше, для хмарної вимірювальної кампанії, що визначає типову пропускну здатність та затримку систем pub / sub. Ми починаємо вимірювати пропускну здатність, застосовуючи все більше навантаження на брокера. Це досягається більшою кількістю пар видавців та передплатників. Для сценарію датчика погоди ми змінюємо кількість публікаційних шлюзів від 10 до 600, при цьому кожен шлюз генерує навантаження, еквівалентну 1000 датчикам, що відповідає загальній кількості до 600 000 емульованих датчиків. Для випадку спільного використання автомобілів ми використовуємо від 10 до 600 видавців, які генерують вантаж, еквівалентний до 9,6 мільйона окремих автомобілів. Ми варіюємо кількість камер від 40 до 600, де кожен видавець імітує лише одну камеру. Збільшення кількості пар видавець / передплатник є експоненціальним для точного вимірювання як дуже малих, так і дуже високих значень пропускної здатності. Експерименти складаються з генерування навантаження протягом 5 хвилин та вимірювання кількості повідомлень, які надходять до абонентів за цей час. Навантаження генерується відповідно до випадкової позиції в одному з файлів трасування. Кожна конфігурація параметрів застосовується загалом 8 разів.

На рис.4 показана виміряна пропускна здатність, коли брокер завантажений різними робочими навантаженнями, визначеними раніше. Насиченість пропускної здатності позначає індивідуальну максимальну стійку пропускну здатність протоколів за певного сценарію. Результати показані з 95% довірчим інтервалом.

На рис. 4a зображені результати для погодних датчиків. ZeroMQ має найвищу пропускну здатність - до 522 тис. повідомлень / с. Інші протоколи мають значно меншу пропускну здатність:

MQTT показує максимальну пропускну здатність, меншу за третину від пропускної здатності 134 тис. повідомлень / с. AMQP має максимальну пропускну здатність 30 тисяч повідомлень / с. Максимальний розмір XMPP в 1 тис. повідомлень / с приблизно на два порядки нижче, ніж пропускна здатність MQTT.

Рис. 4b показує результати для випадку використання автомобілів. ZeroMQ знову показує найвищу максимальну пропускну здатність до 83 тис. повідомлень / с. Примітно, що в цьому випадку використання, хоча і досі не досягає пропускної здатності ZeroMQ, розрив між протоколами зменшився: MQTT показує пропускну здатність 69 тис. повідомлень / с. AMQP має максимальну пропускну здатність 26 тис. повідомлень / с. До 2 тис. повідомлень / с пропускна здатність XMPP нижча за інші протоколи, але напрочуд вища, ніж для менших повідомлень.

На рис. 4в наведені результати щодо трафіку камери. У цьому випадку AMQP має найвищу пропускну здатність з максимальною швидкістю 186 тис. повідомлень / с. ZeroMQ також показує максимальну пропускну здатність до 186 тис. повідомлень / с, що трохи нижче AMQP лише після десяткової коми. MQTT показує максимальну пропускну здатність 164 тис. повідомлень / с. XMPP показує пропускну здатність до 92 повідомлень / с, але не може впоратися з навантаженням понад 300 повідомлень / с.

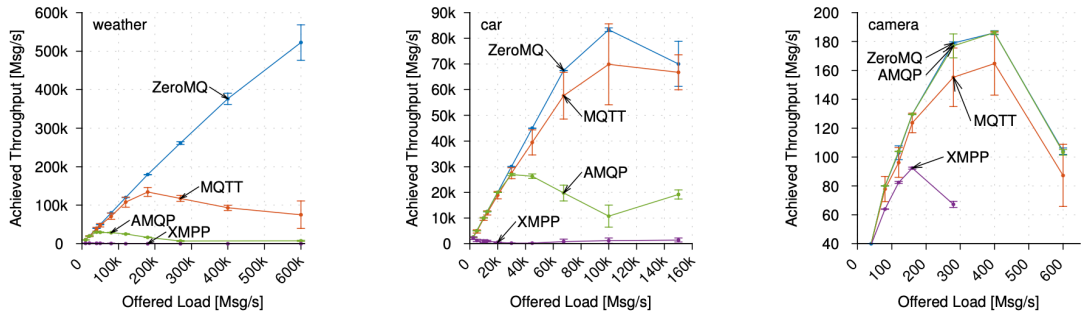
Якщо для інших еталонних сценаріїв ZeroMQ був безперечним лідером, то для сценарію камери протоколи показують набагато більше подібних результатів.

Ми пояснюємо таку поведінку наступним чином: наш брокер ZeroMQ є лише дуже простим і підтримує лише відповідність префіксів. Через це пропускна здатність значно вища, ніж інші протоколи, особливо для менших повідомлень. MQTT та AMQP пропонують більш складну фільтрацію, збільшуючи час, витрачений на кожне рішення про фільтрацію. XMPP має найбільші накладні витрати завдяки використаному кодуванню XML, яке доводиться аналізувати клієнтам і брокеру, що призводить до подальшої обробки накладних витрат. Щодо великих повідомлень, MQTT та AMQP стають більш конкурентоспроможними, оскільки час, витрачений на прийняття рішення про фільтрацію, не залежить від розміру повідомлення, а час копіювання в пам'яті, а також мережева передача повідомлень стають домінуючими факторами часу обробки.

3.5 Вимірювання затримки

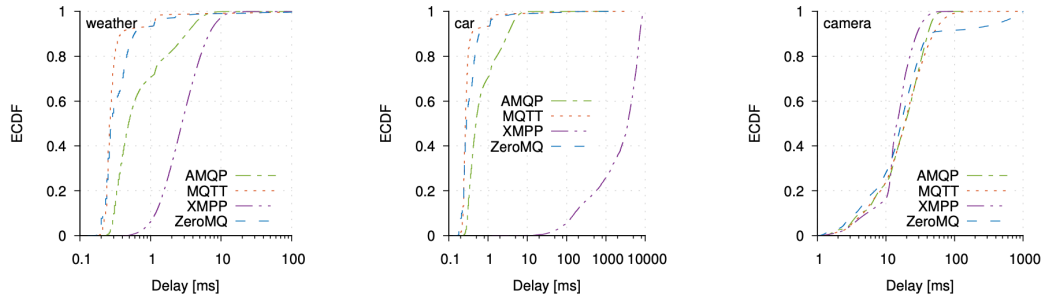
Ми також досліджуємо затримку протоколів. Оскільки затримка у насиченому середовищі була б нереально високою, ми спостерігаємо розподіл латентності протягом одного дня в цьому окремому експерименті. Ми копіюємо 40 мереж датчиків погоди розміром 200 вузлів кожна, що моделює мережу датчиків середнього розміру, таку як тестовий стенд TWIST. Ми використовуємо 100 мереж спільного використання автомобілів, подібних до тих, що записані в Берліні. Нарешті, ми запускаємо одну мережу камер, подібну до тієї, що вимірюється у Берліні з 34 камерами.

Результати представлені на рисунку 5. Окремі графіки показують емпіричну функцію розподілу затримки, введеної системою pub / sub. Графіки ілюструють, який відсоток успішно переданих повідомлень відчуває наскрізну затримку між видавцем та передплатником.



(a) Throughput under weather sensor load. (b) Throughput under car sharing load. (c) Throughput under camera load.

Рис. 4 Пропускна здатність, досягнута протоколами у трьох еталонних сценаріях.



(a) Delay under weather sensor load. (b) Delay under car sharing load. (c) Delay under camera load.

Рис. 5 Емпірична кумулятивна функція розподілу (ECDF) результуючої затримки при застосуванні різних навантажень.

Оскільки виміряна затримка має діапазон, який охоплює кілька порядків величини, ми використовуємо логарифмічну шкалу для кращого порівняння результатів.

Для сценарію датчика погоди результати на рис. 5a показують, що розподіл затримок MQTT та ZeroMQ досить подібний. Більше 90% повідомлень мають затримку менше 1 мс. AMQP показує трохи більшу затримку. Тим не менше, близько 70% виміряних значень затримки нижче 1 мс і 90% нижче 3 мс. XMPP вводить значно більшу затримку, але все-таки менше 3% повідомлень затримується більше ніж на 10 мс.

Отримані розподіли затримок, виміряні для навантаження на спільний рух автомобілів, показані на рис. 5b. Хоча повідомлень значно більше, розподіл затримок ZeroMQ, MQTT та AMQP дуже схожий на попередній сценарій. Використовуючи MQTT та ZeroMQ, більше 90% виміряних значень затримки нижче 1 мс. З AMQP приблизно 70% повідомлень мають затримку менше 1 мс і приблизно 90% менше 3 мс. XMPP показує значно більші затримки до 10 с, що свідчить про ситуацію з перевантаженням.

Розподіл затримки для випадку використання камери наведено на рис. 5c. Загалом затримка всіх протоколів вища, ніж для датчиків з меншими розмірами повідомлень. Крім того, розподіл затримки не відрізняється настільки суттєво, як для менших розмірів повідомлень. Для всіх протоколів більше 90% затримки менше 100 мс. Хоча частіше досягається менша затримка, для ZeroMQ приблизно 10% затримки перевищує 100 мс, іноді досягаючи до 1 с.

Підсумовуючи, поведінка затримки та, отже, вибір систем pub / sub сильно залежить від різних розмірів повідомлень, що використовуються у кожному конкретному випадку використання. У той час як для невеликих повідомлень, MQTT і ZeroMQ виявляються меншими затримками, зі збільшенням розміру повідомлення розподіл затримки зближується. Додаткова затримка, введена за допомогою більш складних методів фільтрації, таких як AMQP, стає неістотною. Крім того, затримка, введена XML-кодуванням інформації заголовка в XMPP, не робить загальний час обробки значно вищим, ніж для інших протоколів у сценаріях, коли надсилаються великі повідомлення.

4 Висновок

У цій роботі ми провели аналіз вимог до pub / sub на хмарних платформах IoT, проаналізували основні особливості існуючих відкритих рішень та представили кількісну оцінку представників кожного протоколу за реального навантаження. Це

демонструє, що, хоча жоден протокол не пропонує всіх функцій, бажаних у налаштуваннях Інтернету речей, між протоколами існують суттєві відмінності: Хоча XMPP є розширюваним відкритим протоколом, XMPP не може досягти продуктивності, що спостерігається з іншими проміжними програмами pub / sub в налаштуваннях IoT. Крім того, на стороні сервера кожен рядок данихповинна бути проаналізований для прийняття рішення про маршрутизацію. Це, мабуть, одна з причин, чому XMPP працює набагато гірше, ніж інші протоколи щодо пропускної здатності та затримки. AMQP - це повнофункціональне проміжне програмне забезпечення, орієнтоване на повідомлення, яке пропонує весь функіонал, необхідний для створення системи обміну повідомленнями з підтримкою IoT. Ефективність з точки зору пропускної здатності повідомлень та середньої затримки відстає від ZeroMQ та MQTT для навантажень з великою кількістю невеликих повідомлень. MQTT спеціально розроблений для транспортування даних, схожих на датчики, і фактично став стандартним у цій галузі. Брокер має достатньо високу пропускну здатність і низьку затримку. Найважливішою частиною, якої не вистачає в MQTT, є можливість безпосереднього контакту з підключеними клієнтами, оскільки MQTT - це суто протокол pub / sub. Наші вимірювання продуктивності свідчать про те, що ZeroMQ може досягти дуже високої пропускної здатності, зберігаючи низьку затримку, переважно незалежно від навантаження. Також привабливим є той факт, що брокер не є обов'язковим і можлива децентралізована система. Однак ZeroMQ не є повнофункціональною реалізацією брокера і менш виразною, ніж інші протоколи, оскільки підтримується лише відповідність префіксів. Отже, для розгортання ZeroMQ в налаштуваннях Інтернету речей можуть знадобитися та можуть бути додані ключові функції.

Тому наш висновок подвійний: ми рекомендуємо ZeroMQ там, де потрібне спеціально розроблене рішення, а зусилля щодо впровадження відсутніх функцій є прийнятними. Для налаштувань, де необхідне легкодоступне рішення, ми рекомендуйте MQTT або AMQP, залежно від очікуваних розмірів повідомлень, оскільки існує кілька програмних рішень з відкритим кодом, які можна використовувати нестандартно.

Список літератури

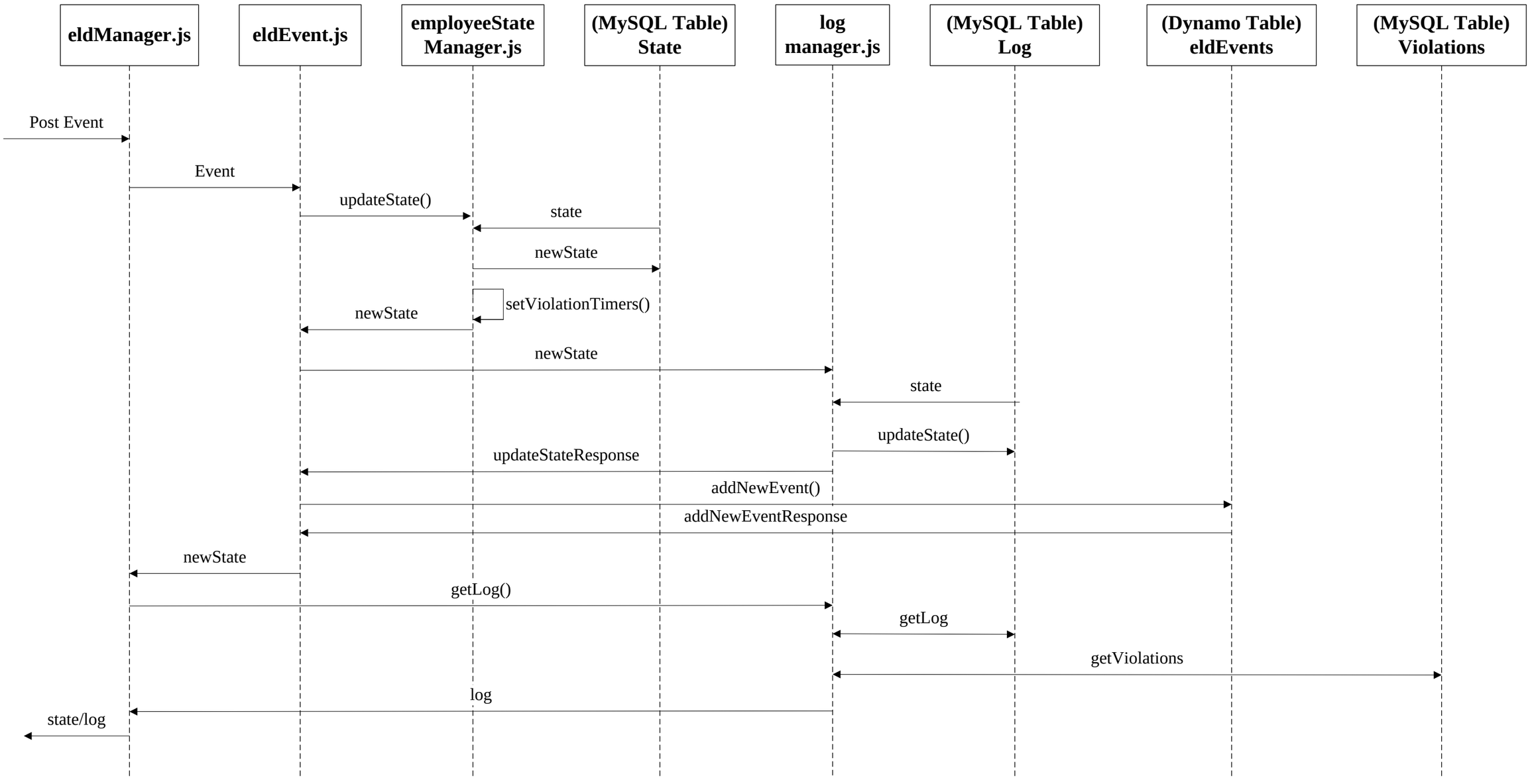
1. Amazon: Elastic Compute Cloud (EC2). URL http://aws. amazon.com/ec2
2. AMQP Working Group: Advanced message queuing protocol (2010). version 0-9-1
3. Apache Software Foundation: ActiveMQ. URL http:// activemq.apache.org/
4. Apache Software Foundation: Apollo. URL http://activemq. apache.org/apollo/
5. Apache Software Foundation: Apollo. URL http://activemq. apache.org/apollo/
6. Car2go: Berlin. https://www.car2go.com/de/berlin/
7. Carzaniga,A.,Rosenblum,D.S.,Wolf,A.L.:Achievingscalability and expressiveness in an internet-scale event notification service. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00), ст. 219–227. ACM, New York, NY, USA (2000). DOI 10.1145/343477.343622
8. Cha,M.,Rodriguez,P.,Moon,S.,Crowcroft,J.:Onnext-generation telco-managed p2p tv architectures. Джерело: Proceedings of the 7th International Conference on Peer-to-peer Systems (IPTPS'08), ст. 5–5. USENIX Association (2008)
9. Chui,M.,Lo¨ffler,M.,Roberts,R.:Theinternetofthings.McKinsey Quarterly 2, 1–9 (2010)
10. Curry, E.: Message-oriented middleware. In: Q.H. Mahmoud (ed.) Middleware for Communications, глава. 1, ст. 1–28. John Wiley & Sons (2005)

11. Eclipse Foundation: Paho. URL https://eclipse.org/paho/
12. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) 35(2), 114–131 (2003)
13. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems 29(7), 1645–1660 (2013)
14. Handziski, V., Ko ̈pke, A., Willig, A., Wolisz, A.: Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In: Proc. of the 2nd Int. Workshop on Multi-hop Ad Hoc Networks: From Theory to Reality (REALMAN '06), ст. 63–70. Florence, Italy (2006)
15. Hintjens,P.:ZeroMQ:MessagingforManyApplications.O'Reilly (2013)
16. Hunkeler, U., Truong, H.L., Stanford-Clark, A.: MQTT-S – A publish/subscribe protocol for Wireless Sensor Networks. Джерело: 3rd Int. Conf. on Communication Systems Software and Middleware
17. Ignite Realtime: Openfire Server. URL http://www.igniterealtime.org/projects/openfire/
18. Locke,D.:MQTelemetryTransport(MQTT)V3.1ProtocolSpeci- fication. IBM developerWorks Technical Library (2010)
19. Menzel, T., Karowski, N., Happ, D., Handziski, V., Wolisz, A.: Social sensor cloud: An architecture meeting cloud-centric iot platform requirements (2014). 9th KuVS NGSDP Expert Talk on Next Generation Service Delivery Platforms
20. Millard, P., Saint-Andre, P., Meijer, R.: XEP-0060: Publish- Subscribe (2010). URL http://www.xmpp.org/extensions/ xep- 0060.html. Version: 1.13
21. Moteiv Co.: Tmote sky datasheet. URL http: //www.crew-project.eu/sites/default/files/tmote- sky- datasheet.pdf
22. Pivotal Software: RabbitMQ. URL https://www.rabbitmq. com/
23. ProcessOne: ejabberd XMPP Server. URL https://www.process- one.net/en/ejabberd/
24. Rege,M.R.,Handziski,V.,Wolisz,A.:CrowdMeter:anemulation platform for performance evaluation of crowd-sensing applications. Джерело: Proc. of the 2013 ACM conf. on Pervasive and ubiquitous com- puting adjunct publication, ст. 1111–1122. Zu ̈rich, Switzerland (2013)
25. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard) (2011). URL http://www.ietf.org/rfc/rfc6120.txt Tran, P., Greenfield, P., Gorton, I.: Behavior and Performance of Message

ДОДАТОК Б

ПОСЛІДОВНОСТЬ СТВОРЕННЯ ЗАПИСІВ ЛОГІВ

Послідовность створення записів логів



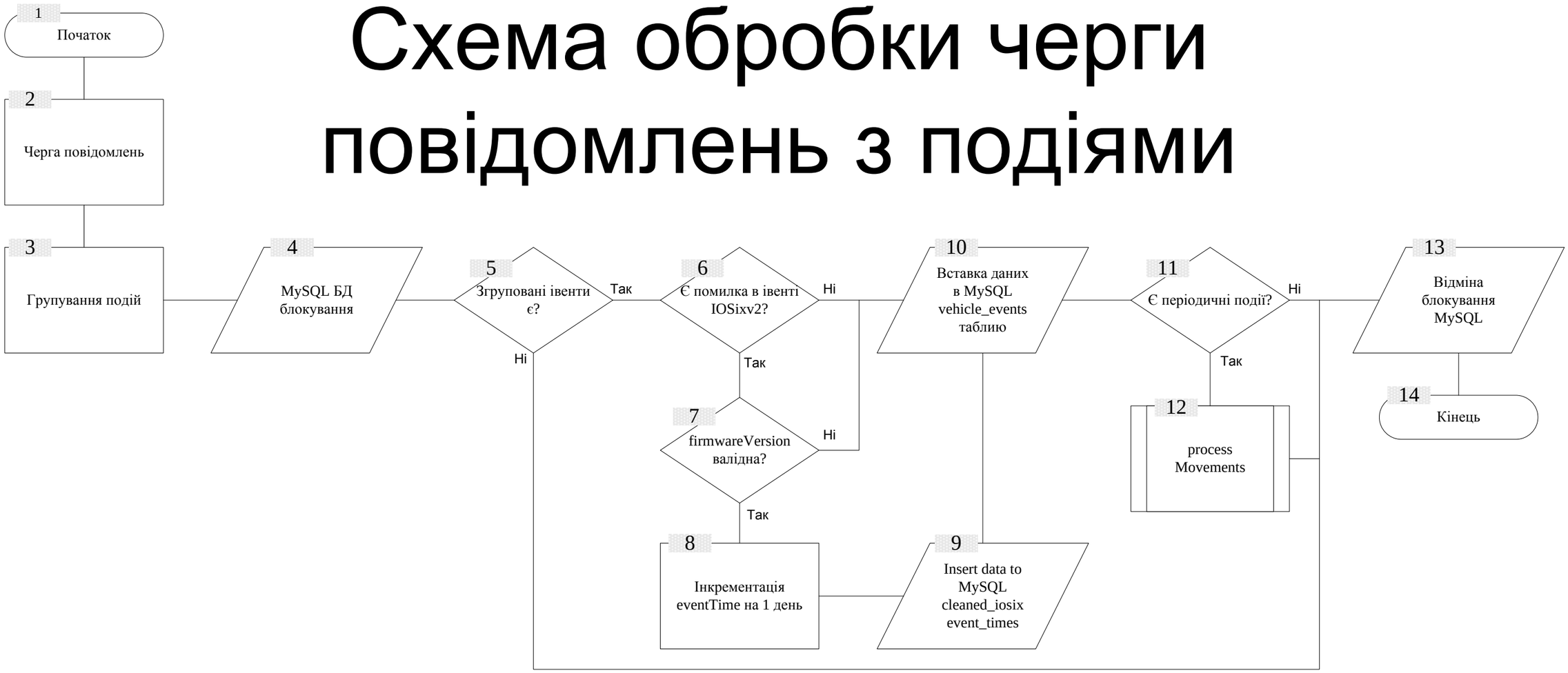
Демонстраційний плакат №1
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК В

СХЕМА ОБРОБКИ ЧЕРГИ ПОВІДОМЛЕНЬ З ПОДІЯМИ

Схема обробки черги повідомлень з подіями



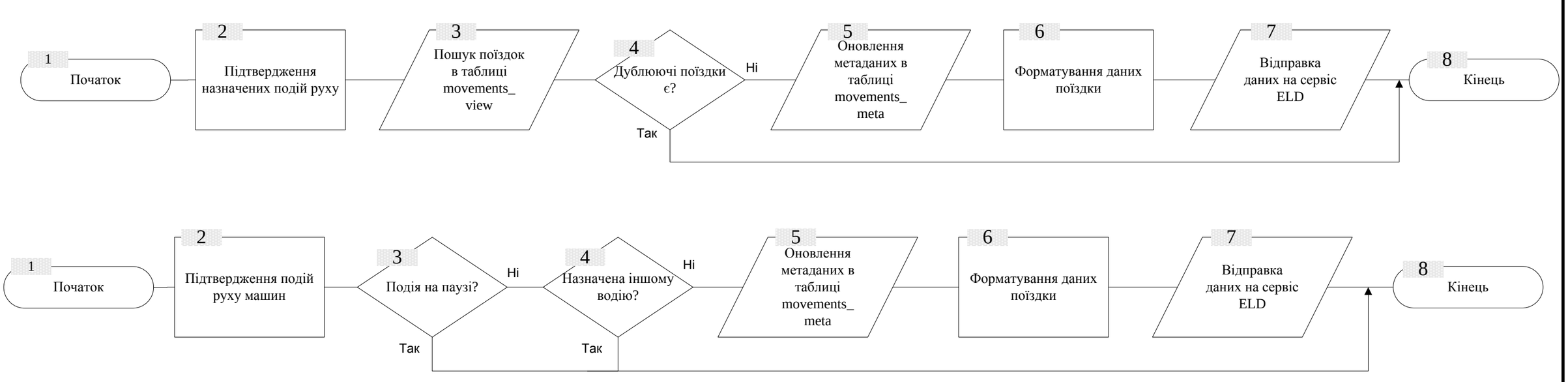
Демонстраційний плакат №2
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК Г

СХЕМА ОБРОБКИ ЧЕРГИ ПОВІДОМЛЕНЬ ПРО РУХ

Схема обробки черги повідомлень про рух



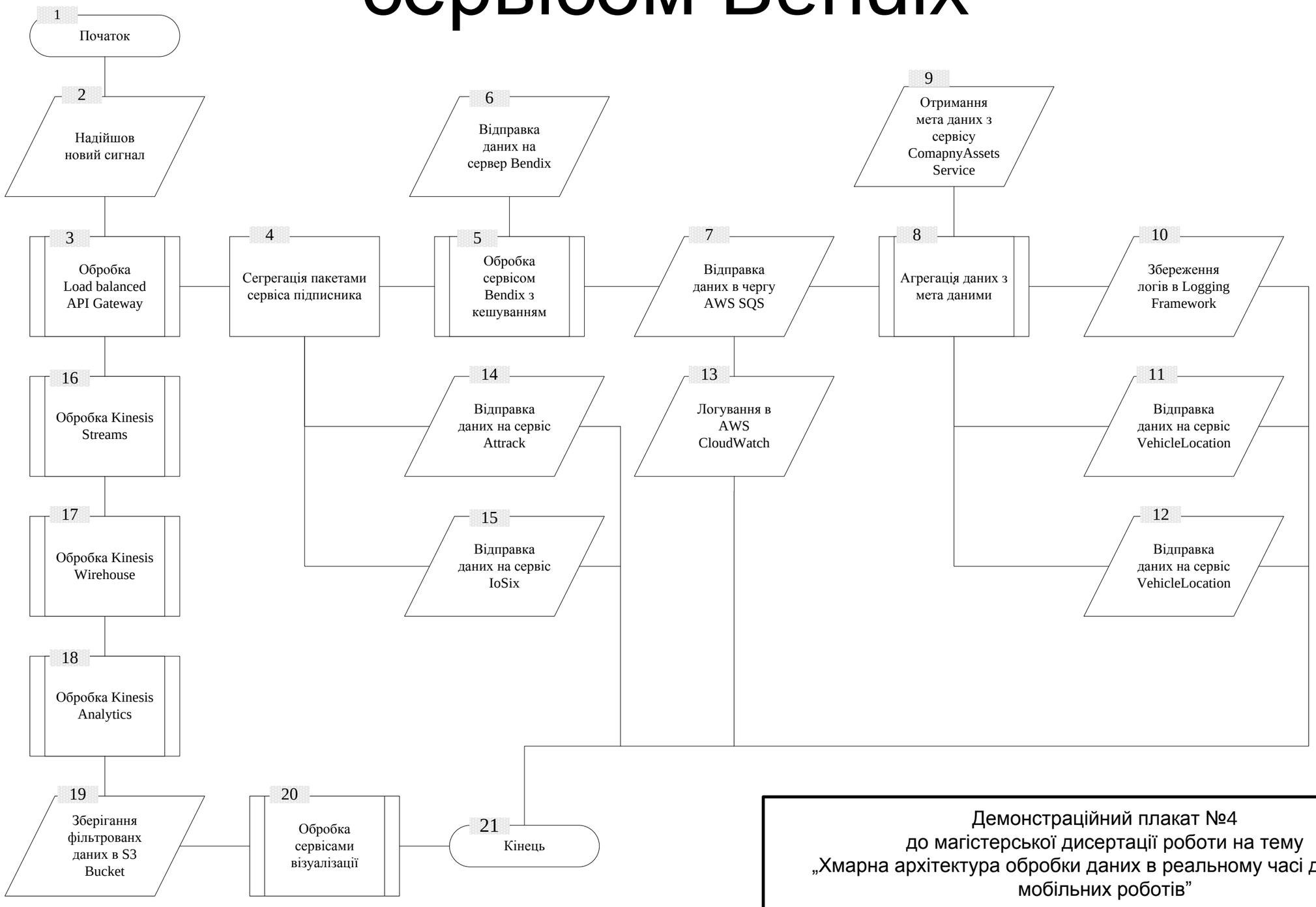
Демонстраційний плакат №3
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК Ґ

СХЕМА ОБРОБКИ СИГНАЛІВ СЕРВІСОМ BENDIX

Схема обробки сигналів сервісом Bendix



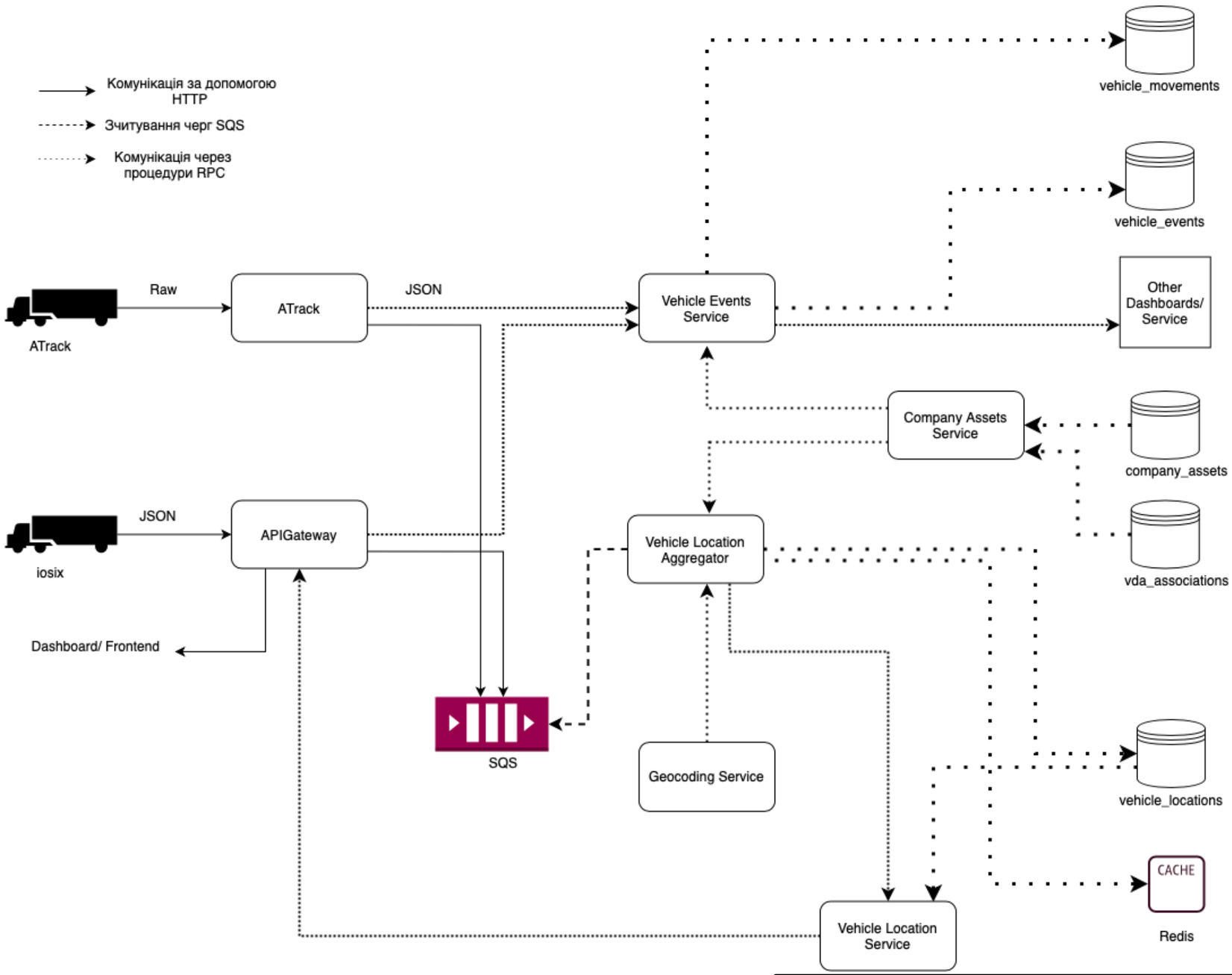
Демонстраційний плакат №4
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК Д

АРХІТЕКТУРА ОБРОБКИ ВХІДНИХ СИГНАЛІВ

Архітектура обробки вхідних сигналів



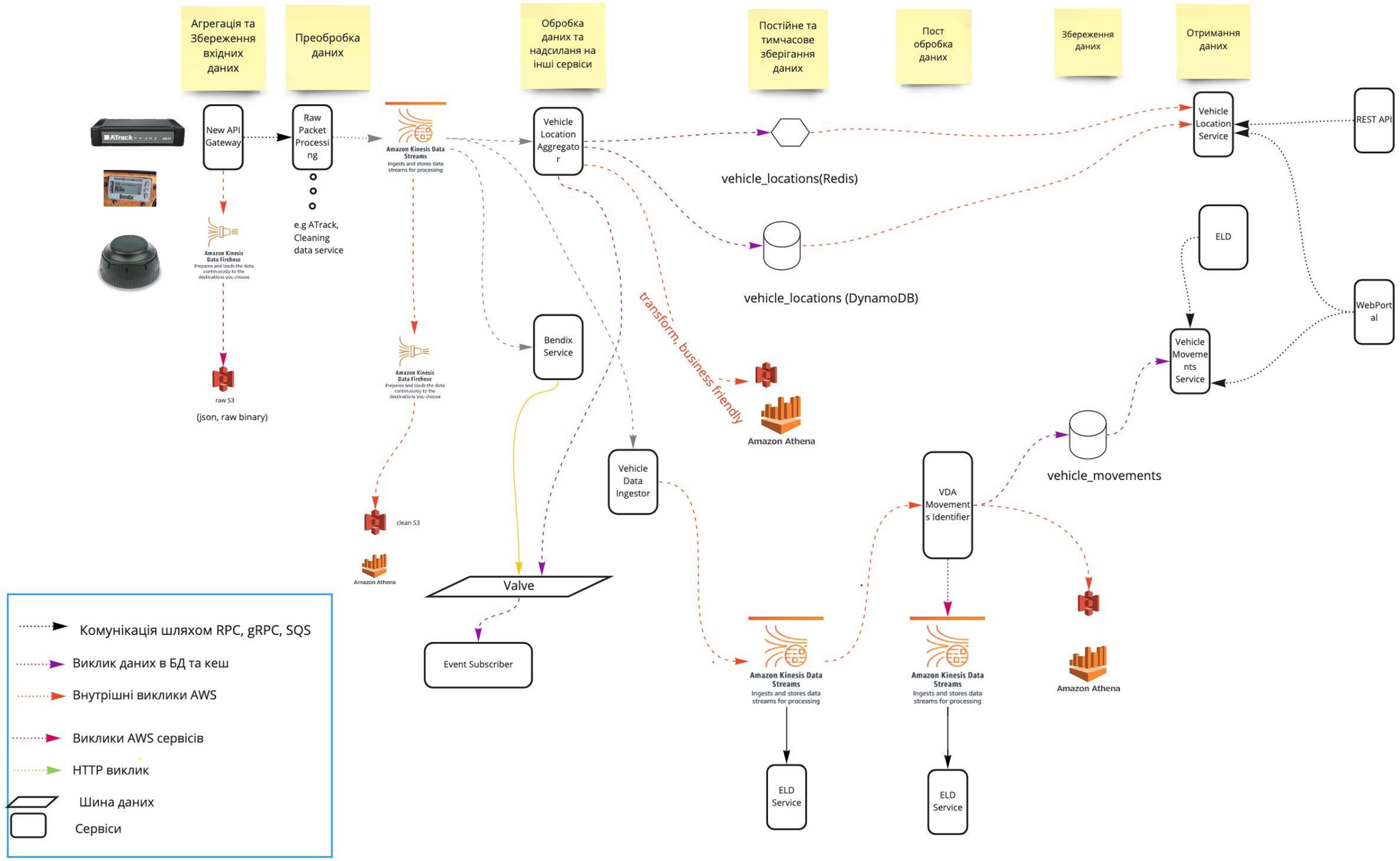
Демонстраційний плакат №5
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК Е

АРХІТЕКТУРА СИСТЕМИ СИНХРОНІЗАЦІЇ ДАНИХ

Архітектура системи синхронізації даних



Демонстраційний плакат №6
до магістерської дисертації роботи на тему
„Хмарна архітектура обробки даних в реальному часі для групи мобільних роботів”

Розробив: студент гр. ІК-91мп Старовойтенко О.В.
Прийняв: к.т.н., доцент Резніков С.А.

ДОДАТОК Є

РЕЗУЛЬТАТ ПЕРЕВІРКИ РОБОТИ НА СПІВПАДІННЯ