Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Побудова системи моніторингу та візуалізації показників життєдіяльності людини

Текстова частина магістерської роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник магістерської роботи
к.н., доцент М. В. Почебут

_____
(підпис)

“____” _____ 2021 р.

Виконав студент С. О. Гребенович
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики, к.ф.-м.н.
_____С. С. Гороховський
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

Студенту 2 р.н. магістерської програми Інженерія програмного забезпечення Гребеновичу Сергію Олексійовичу

Розробити систему моніторингу та візуалізації показників життєдіяльності людини

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1 Опис проблеми та постановка задача
2 Огляд існуючих систем моніторингу та візуалізації
2 Визначення архітектурно-значущих вимог
3 Розробка архітектурних рішень, що задовольняють вимоги
4 Реалізація прототипів систем на основі архітектурних рішень
5 Порівняння та вибір кращої архітектури
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2020 р.

Керівник
М. В. Почебут, кандидат наук, доцент _____
(підпис)

Завдання отримав
С. О. Гребенович _____
(підпис)

Тема: Побудова системи моніторингу та візуалізації показників життєдіяльності людини

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	01.11.2020	
2.	Огляд технічної літератури за темою роботи	15.11.2020	
3.	Виконання аналізу сучасних рішень	29.11.2020	
3.	Розробка архітектури системи моніторингу та візуалізації	27.12.2020	
4.	Реалізація програмного застосунку на базі архітектури	03.02.2021	
5.	Налаштування програмного застосунку для обраного прикладу даних та оцінка ефективності роботи	27.03.2021	
6.	Написання пояснювальної записки	24.04.2021	
7.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
8.	Аналіз отриманих результатів з керівником, написання доповіді	30.04.2021	
9.	Попередній захист магістерської роботи	14.05.2021	
10.	Корегування роботи за результатами попереднього захисту	15.05.2021	
11.	Остаточне оформлення пояснювальної записки та слайдів	25.05.2021	
12.	Захист магістерської роботи (проекту)	17.06.2021	

Студент _____

Керівник _____

“____”_____

ЗМІСТ

Анотація

У даній роботі було проведено огляд поширених систем та застосунків, що використовуються для реєстрації, візуалізації та аналізу показників життєдіяльності людини. Запропоновано уніфікований підхід для отримання інформації з різних джерел та накопичення у централізованому хмарному сховищі, з метою забезпечення її подальшої презентації в зручному для кінцевого користувача вигляді та ефективного аналізу, а саме визначення міри кореляції між різними параметрами.

Надалі, в рамках моделі було визначено набір архітектурно-значущих вимог та атрибутів якості. Розроблено альтернативні варіанти архітектури, що їх задовольняють.

На базі хмарних ресурсів Microsoft Azure (таких як Azure Logic Apps, Azure Functions, Azure Storage, Azure Cosmos DB та ін.) та бібліотек мови Python (Dash, pandas) було реалізовано прототипи програмних продуктів для різних варіантів архітектури. Проведено порівняння ефективності реалізацій та визначено більш оптимальну архітектуру для поставленої задачі.

Ключові слова: моніторинг та візуалізація параметрів життєдіяльності, міри кореляції, Microsoft Azure, Azure Logic Apps, Azure Functions, Azure Storage, Azure Cosmos DB, Python, Dash, pandas.

ВСТУП

Актуальність. Інформаційні технології проникають у найрізноманітніші аспекти людського життя. І чи не найбільш важливою та актуальною на сьогодні є сфера догляду за здоров'ям. З кожним днем нам стають доступними все більше даних про себе. Ми повноцінно використовуємо інтернет речей, інтегруючи фітнес-трекери, що вимірюють нашу активність, або прилади, що слідкують за медичними показниками, отримуємо результати аналізів на електронну пошту або ж власноруч реєструємо дані в своїх пристроях. І це далеко не повний список.

Вже існує чимало систем, що допомагають зберігати такого роду інформацію та відслідковувати її динаміку. Технологічні гіганти, такі як Apple, Samsung, а особливо Google вибудовують цілі екосистеми для роботи з даними про здоров'я, забезпечені інтерфейсами для інтеграції. Проте, список параметрів, що відслідковуються, зазвичай є фіксованим, можливості для аналізу обмежуються візуальним порівнянням, а інтеграції різних систем між собою можуть вимагати додаткової розробки або мати обмежений доступ.

Додаткова складність виникає, якщо ми хочемо взяти до уваги зовнішні дані, наприклад про середовище, що нас оточує, такі як погода чи забрудненість повітря. Зазвичай тут у пригоді стають універсальні аналітичні системи, але для повноцінного використання вони, здебільшого, вимагатимуть ліцензії або ж можуть бути складними у використанні для кінцевого користувача.

Зважаючи на це, актуальною є проблема побудови системи, яка б не була прив'язана до конкретного типу інформації, виробника або постачальника програмного забезпечення, натомість дозволяла б легко інтегруватися з різноманітними джерелами та забезпечувати можливість ефективного аналізу, характерного для догляду за здоров'ям, наприклад, часову кореляцію між значеннями різних параметрів.

Мета дослідження. Створення сучасної архітектури для системи централізованого моніторингу та візуалізації показників життєдіяльності людини, здатної забезпечити аналіз великих об’ємів даних у реальному часі.

Завдання дослідження. Провести огляд поширених систем та застосунків, що опрацьовують показники життєдіяльності людини. Визначити потенційні недоліки існуючих підходів та запропонувати способи їх вдосконалення за допомогою нового рішення.

Визначити архітектурно-значущі вимоги та атрибути якості для майбутньої системи. Розробити варіанти архітектури, що їх задовольняють.

Обрати інструменти для впровадження та визначити шаблони для типових сценаріїв інтеграції. Створити прототипи програмного продукту, оцінити їх ефективність та обрати найбільш прийнятний варіант архітектури.

Об’єкт дослідження. Моніторинг та візуалізація показників життєдіяльності людини, виявлення кореляцій між такими показниками.

Предмет дослідження. Підходи до інтеграції різних джерел інформації, способи організації та збереження даних. Спеціалізовані рішення для статистичного аналізу, інструменти для візуалізації результатів.

Джерела дослідження. Наукові публікації, програмна документація, вихідний код бібліотек, електронні ресурси спеціалізованих спільнот, відео-версії лекцій та навчальних інструкцій.

Наукова новизна одержаних результатів дослідження полягає у вирішенні проблеми порівняльного аналізу для даних різного походження за допомогою нового рішення на основі хмарних сервісів та сучасних інструментів обробки та візуалізації даних. Порівняння впливу архітектурних рішень на ефективність роботи такої системи.

Практичне значення одержаних результатів. Використаний уніфікований підхід до збереження даних та їх подальшого зчитування, в

поєднанні з використанням стандартних хмарних сервісів та методів роботи з ними, надає можливості для відносно легкого підключення нових джерел інформації до системи з одного боку, а також впровадження нових метрик для аналізу даних – з іншого. Це може допомогти суттєво зменшити час на впровадження нових сценаріїв для аналізу, порівняно з розробкою їх з нуля, не використовуючи при цьому універсальних спеціалізованих інструментів, вартість яких може бути високою незалежно від ступеня використання.

РОЗДІЛ 1. Системний контроль за параметрами життєдіяльності

1.1. Системи для контролю параметрів життєдіяльності

До систем контролю параметрів життєдіяльності можемо віднести такі системи, що відповідають за отримання, обробку та презентацію даних, які так чи інакше відображають стан здоров'я та фізичні характеристики людини або ж чинників, що можуть на них впливати [1]. Прикладами таких даних є рівень артеріального тиску, концентрація глюкози у крові, температура тіла, пульс, вага, рівень активності, тощо. [2] Виділимо основні види систем, що використовуються для їх накопичення та моніторингу:

– Безпосереднє програмне забезпечення медичних та спортивних приладів (наприклад, так званих, фітнес-трекерів);

– Системи для особистого контролю за здоров'ям (зазвичай, мобільні та веб застосунки), що можуть додатково інтегруватися з сенсорами, приладами та іншими джерелами даних;

– Спеціалізовані клінічні системи для відстеження стану пацієнтів. Вони можуть варіюватися від відносно простих до надзвичайно складних програмних комплексів, структура та наповнення яких сильно залежить від безпосереднього напряму діяльності та переліку послуг.

Не зважаючи на різноманіття застосувань, перелічені системи об'єднує єдиний підхід до роботи з інформацією, що в спрощеному вигляді зображено на Рис. 1.1.

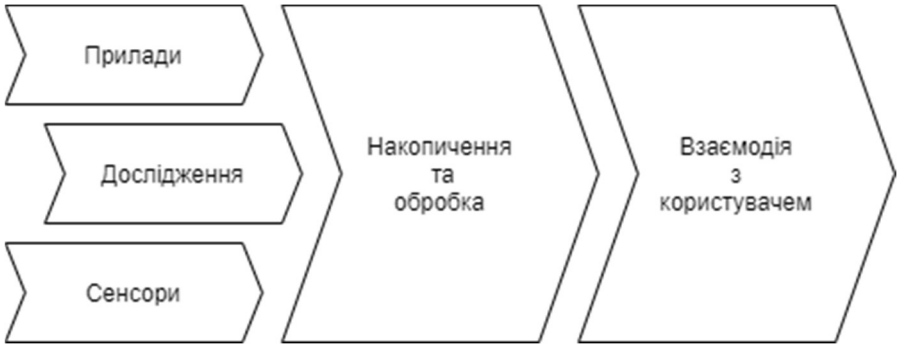

Рис. 1.1. Узагальнена схема роботи системи контролю параметрів життєдіяльності людини.

Відповідно, ключовими функціями систем для контролю за параметрами життєдіяльності людини є:

– **Збір та первинна обробка інформації.** Це відбувається за допомогою приладів або сенсорів, що здатні проводити вимірювання в реальному часі (або ж на вимогу) та передавати дані наступним елементам системи. За використання мережевих технологій для передачі даних це стає гарною ілюстрацією концепції Інтернету речей (IoT), що набуває широкого використання у найрізноманітніших галузях і, як наслідок, вже має чимало стандартних інструментів для застосування цієї ідеї на практиці. Альтернативою залишається ручний ввід даних, так як не всі прилади або способи вимірювання дозволяють інтегрувати їх результати з сучасними системами або ж така інтеграція не є доцільною.

– **Накопичення та забезпечення доступу до даних.** Першим місцем збереження і накопичення даних, зазвичай, є вбудована пам’ять приладу, чого може бути достатньо для певних сценаріїв (наприклад, коли результатом роботи системи є генерація звіту). В загальному ж випадку, дані накопичуються у централізованому сховищі, щоб забезпечити можливості для концентрації даних з різних джерел, аналізу та розподіленого доступу до них. Також стає можливим використання вбудованих або ж сторонніх спеціалізованих сервісів.

– **Візуалізація, порівняння та визначення аномалій.** У найпростішому випадку кінцевим етапом роботи системи є відображення результатів вимірювання користувачу. У цьому випадку окремий прилад теж можна розглядати як таку систему. Проте, більший інтерес представляють складніші системи, що дозволяють проаналізувати результати або, навіть, повідомити про наявність проблеми лікарю чи іншій уповноваженій особі.

1.2. Огляд існуючих рішень

В міру обмеженості доступу до спеціалізованих медичних систем та обладнання, що зазвичай мають підвищені вимоги до конфіденційності або є комерційною таємницею, дане дослідження сфокусується на аналізі загальнодоступних та немедичних систем для особистого використання. Не зважаючи на це, за необхідності, отримані висновки можуть бути розширені також і на них.

Розглянемо два найбільш поширених класи систем, що застосовуються для контролю за індивідуальними параметрами життєдіяльності людини, а саме:

– Рішення, що ґрунтуються на концепції Інтернету речей та використовують відповідні засоби для його впровадження

– Комерційні мобільні застосунки для особистого контролю за параметрами життєдіяльності

Проаналізуємо кожен з них більш детально та визначимо можливості для покращення ідей, закладених у їх основі.

1.2.1. Рішення на основі концепції Інтернету речей

Поняття «Інтернет речей» або IoT (Internet of Things) зазвичай застосовують до сценаріїв, коли об’єкти, сенсори та інші повсякденні предмети, що в загальному розумінні не можуть вважатися комп’ютерами, за допомогою під’єднання до інформаційних мереж та обчислювальних ресурсів, можуть генерувати та отримувати дані, а також обмінюватися ними без суттєвого вручання ззовні [3].

Цей підхід широко використовується в медицині, так як навіть прості сучасні прилади, як, наприклад, ваги, термометри чи вимірювачі кров’яного тиску, можуть оснащуватися інтерфейсами для зчитування або поширення результатів вимірювань. Не кажучи вже про більш складні спеціалізовані прилади.

Спосіб подальшої передачі інформації відбувається в залежності від інтерфейсів, наявних у конкретних моделей приладів. При застосуванні фізичних портів у якості первинного накопичувача та модуля обробки даних може виступати мікроконтролер (такий як Arduino [2] чи Raspberry Pi [4]), за наявності безпровідного підключення – смартфон [5], при прямому ж підключенні до інтернету дані можуть передаватися або зчитуватися IoT-модулем напряму [6].

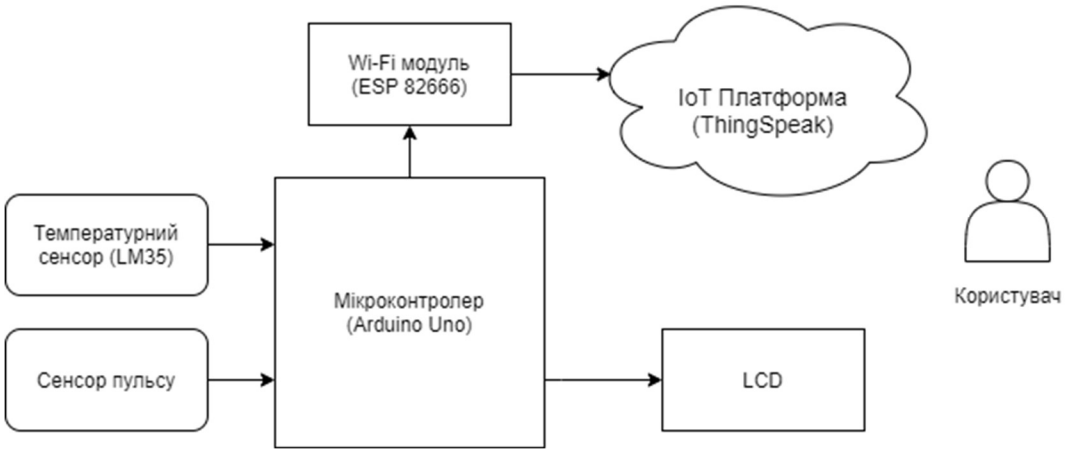

Рис. 1.2. Приклад рішення для вимірювання температури і пульсу з використанням сенсорів та мікроконтролера [2]

Саме IoT-модуль або централізована IoT-платформа виступає основним сховищем даних та джерелом для їх подальшого аналізу та обробки. Сьогодні на ринку хмарних платформ наявні рішення від багатьох провайдерів: Azure IoT Hub, AWS IoT, ThingSpeak, IOT Gecko та інші. Ці IoT-платформи забезпечують накопичення даних або ж їх передачу іншим системам чи сервісам у вигляді повідомлень, забезпечують доступ з допомогою розвинутих прикладних програмних інтерфейсів (API). Стандартним способом візуалізації результатів, накопичених безпосередньо на платформі є використання BI-систем (Business Intelligence systems) або ж створення веб-застосунків на основі шаблонів, наданих розробниками платформи.

Не зважаючи на очевидні переваги використання IoT систем для роботи з сенсорами та приладами, інструменти, що застосовуються можуть бути

недостатньо зручними для кінцевого користувача, що не займається науковою діяльністю або розробкою програмного забезпечення:

– Використання зовнішніх систем для візуалізації та аналізу може бути складним або ж потребувати окремої ліцензії;

– Для простих випадків або сценаріїв з ручним накопиченням інформації використання складних та відносно дорогих сервісів може бути надлишковим.

1.2.2. Комерційні мобільні застосунки для стеження за здоров'ям

На фоні загального тренду до використання портативних пристроїв для стеження за здоров'ям, за останні роки практично кожен великий виробник смартфонів або програмного забезпечення (таких як Apple, Samsung, Google та ін.) почав активно розвивати власні медичні або спортивні застосунки та відповідні сервіси. Більше того, вони вибудовують цілі екосистеми навколо своїх рішень, забезпечуючи централізоване сховище або рушій для інших застосунків.

В основі ідеї лежить можливість відслідковувати рівень активності (пройдені кроки, частота серцебиття, тощо) за допомогою загальнодоступних особистих пристроїв, так званих смарт-годинників та фітнес-трекерів, що синхронізуються зі смартфоном. Доповнивши це можливістю занотовувати дані про дієту або інші показники напряму у смартфоні можна отримати достатньо даних, щоб зробити висновок щодо фізичної форми людини.

Прикладами таких систем є Apple Health, Google Fit, Samsung Health, а яскравими прикладами застосунків, що можуть виступати елементами їх екосистеми є Health2Sync, OneDrop та інші. Приклад користувацького інтерфейсу та функцій таких систем наведено на Рис. 1.3 та Рис.1.4.

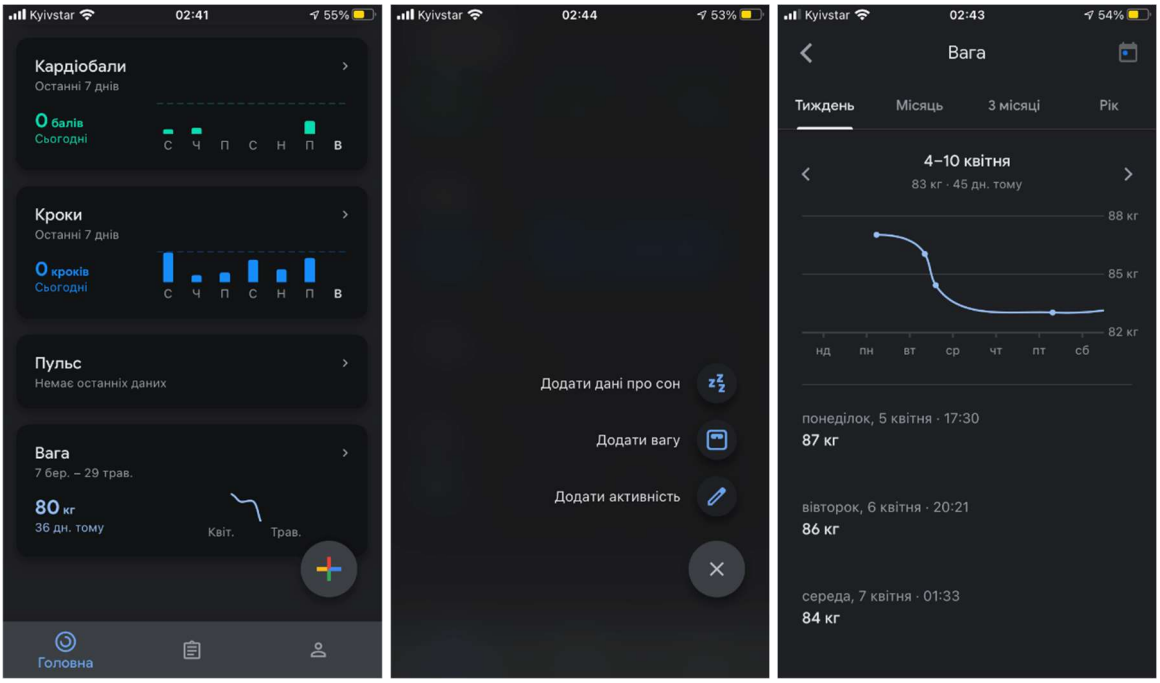

Рис. 1.3. Користувацький інтерфейс та базові функції Google Fit

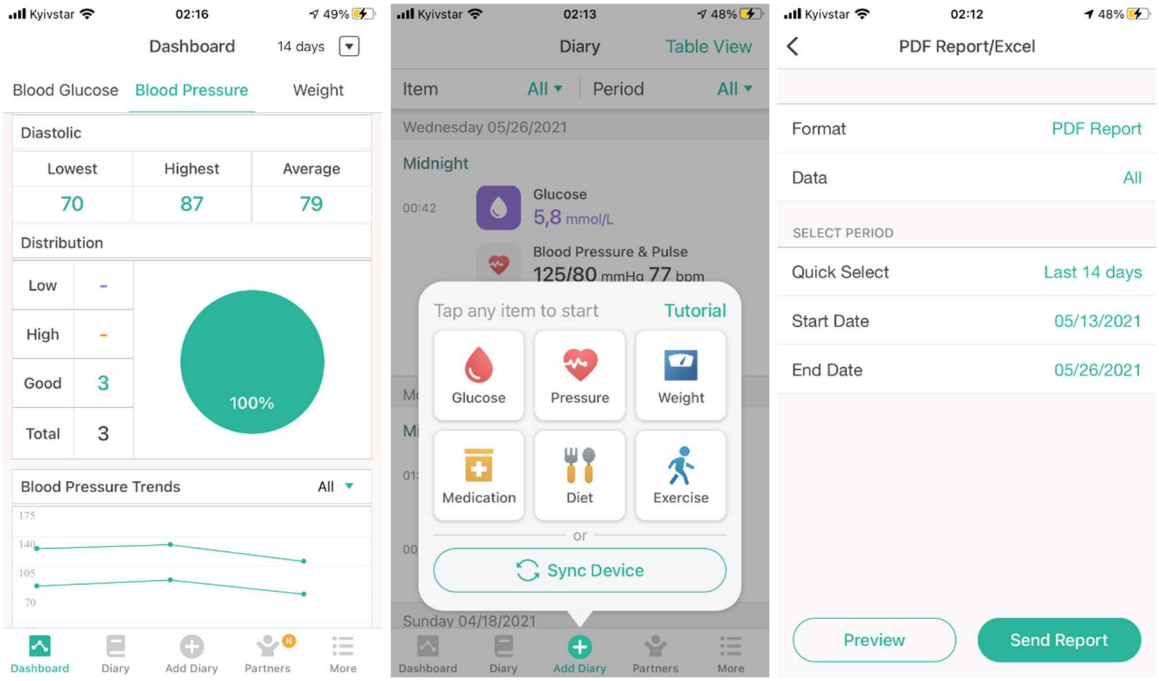

Рис. 1.4. Користувацький інтерфейс та базові функції Health2Sync

Типовими вбудованими функціями для аналізу даних є представлення зміни параметру у часі у вигляді графіку та розрахунок вбудованих параметрів (наприклад, «кардіобали» у Google Fit). Можливості для аналізу даних в сторонніх системах залежить від видів інтеграції, що забезпечується конкретною системою:

– Apple дозволяє доступ до даних HealthKit лише через інші iOS-застосунки, побудовані використовуючи відповідні засоби розробки – SDK (Software Development Kit) [7];

– Samsung Health теж забезпечує доступ з допомогою власного SDK для Android-застосунків чи приладів, а також надає доступ через RESTful API через протокол http. Проте, доступ можливий лише для авторизованих розробників, зареєстрованих як партнери Samsung [8];

– Google Fit – має як Android SDK так і відкритий RESTful API для побудови веб-застосунків, доступний при автентифікації через Google API Service. Доступ же до чутливої інформації вимагає згоди Google та перевірки сирцевого коду системи на предмет безпеки [9];

– Health2Sync – окрім того, що інтегрується з базовими застосунками смартфону, також дозволяє формувати звіти у форматі pdf або Excel та пересилати їх електронною поштою.

Після тестового використання декількох застосунків ми помітили, що список параметрів, що відслідковуються, зазвичай є фіксованим, що дещо обмежує можливості для використання. Для додавання нових параметрів є необхідною побудова окремого мобільного або (у деяких випадках) веб застосунку. Розробка такої системи та її подальший розвиток може бути складною або затратною задачею, особливо при необхідності підтримки різних операційних систем. Більше того, додатковою складністю є необхідність утворення партнерських відносин або сертифікації елементів системи.

1.3. Постановка задачі для нового рішення

Базуючись на результатах огляду, ми прийшли до висновку, що практичне застосування може мати система, що здатна не лише відслідковувати та відображати певні показники життєдіяльності людини, а й забезпечити її більш широкий та глибокий аналіз протягом тривалого періоду часу у зручному для

кінцевого користувача вигляді – зі спрощеним інтерфейсом та розрахунком наперед заданих метрик:

– Не обмежуватися одним способом отримання даних, а натомість розробити підхід, що дозволить максимально розширити спектр інформації, що накопичуватиметься та буде доступною до аналізу;

– Порівняти значення параметрів життєдіяльності людини з зовнішніми даними про середовище, що її оточує;

– Забезпечити при цьому уніфікованих підхід до візуалізації різноманітних даних без застосування спеціалізованих, складних у використанні систем;

– Надати можливості для статистичного аналізу отриманих даних.

В рамках перевірки життєздатності такої ідеї ми пропонуємо розробити систему, що буде:

1. Збирати інформацію з наступних джерел:
   a. Дані з мобільного застосунку, що отримує дані з фітнес-трекера або ж інших застосунків;
   b. Показники, зареєстровані особисто;
   c. Результати аналізів, отримані на електронну пошту;
   d. Параметри навколишнього середовища (погода).
2. Відображати дані для порівняння у єдиному форматі
3. Розраховувати кореляцію між обраними параметрами

РОЗДІЛ 2. Розробка архітектури системи

Для створення дизайну системи використаємо підхід, що керується якостями майбутньої системи або Attribute-Driver Design [10]. Спочатку визначимо набір архітектурно-значущих вимог, обмежень та інших чинників що суттєво впливають на архітектуру системи. На їх основі визначимо атрибути якості, яким повинна відповідати система. І, врешті, виберемо підходи та архітектурні рішення, що задовольняють вимоги.

2.1. Визначення архітектурно-значущих вимог

Сформулюємо вимоги та чинники, що повинні враховуватися при розробці архітектури системи:

– Простота підключення нових джерел інформації та розрахунку нових метрик для аналізу;

– Використання сучасних інструментів, що здатні забезпечити підтримку та розвиток системи у майбутньому;

– Універсальність системи з точки зору аналізу, забезпечення можливостей для порівняння даних з різних джерел між собою;

– Ефективність отримання даних та розрахунку кореляції на великих неструктурованих даних.

Визначимо також параметри якості, яким повинна відповідати розроблена архітектура [11]:

– Розширюваність (Extensibility) – простота, з якою система може розширювати власні можливості без впливу на інші підсистеми та компоненти;

– Безпека (Security) – конфіденційність даних, забезпечення авторизованого доступу та збереження цілісності даних. Цей атрибут набуває особливої ваги, якщо врахувати, що ми можемо працювати з медичними даними, що за багатьма критеріями є дуже чутливою інформацією;

– Продуктивність (Performance) – у нашому випадку швидкість відповіді системи на запит (час видобування, розрахунку кореляції та відображення даних).

2.2. Архітектурні рішення для задоволення вимог

Хмарне середовище. В першу чергу ми пропонуємо побудувати систему за допомогою сервісів одного з трьох основних хмарних провайдерів – Microsoft Azure, Amazon Web Services чи Google Cloud Platform [12]. Таке рішення забезпечить використання сучасних інструментів, так як цей ринок є висококонкурентним і наявні послуги постійно оновлюються та покращуються. З аналогійних міркувань це задовольняє вимогам безпеки, так як більшість сервісів містять засоби безпеки за замовчуванням (authentication, authorization, tokens) та стандартні послуги, що дозволяють посилити безпеку рішень (наприклад, Azure Vault). Багато аспектів безпеки делегуються постачальнику хмарних рішень, а використання послуг саме найбільших провайдерів, що мають велику кількість активних користувачів, також забезпечить своєчасну реакцію на атаки та виявлені вразливості.

Стандартизовані інтерфейси. Задля спрощення підключення нових джерел інформації пропонується використання максимально стандартизованих засобів інтеграції, таких як Azure Logic Apps [13] чи AWS Step Functions. При чому наш вибір впав саме на Microsoft Azure та Logic Apps, так як вони містять багато готових конекторів для широко вживаних продуктів, таких як, поштові клієнти, офісні програми, тощо [14]. Це дозволить скоротити час на розробку для простих інтеграцій і трансформації даних та все одно залишає необмежені можливості для обробки більш складних випадків, викликаючи будь-який потрібний код через Azure Functions [15]. Приклад такого рішення наведено на Рис. 2.1.

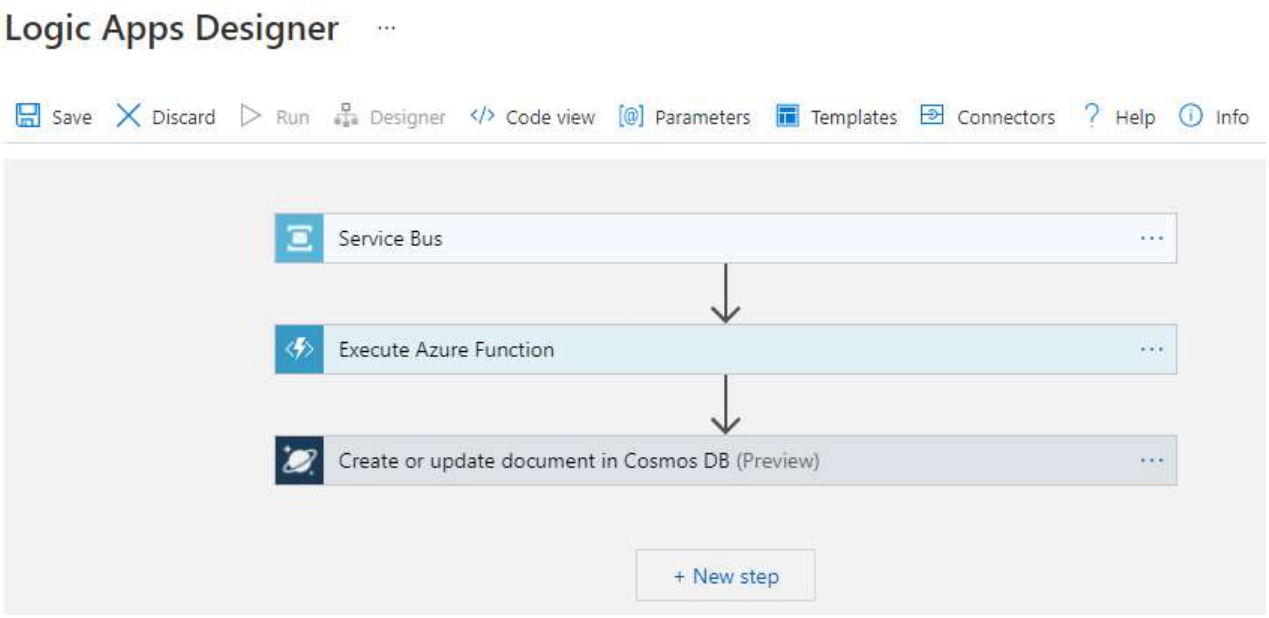

Рис. 2.1. Приклад побудови інтеграції за допомогою Azure Logic Apps

Нереляційне сховище даних. Документна нереляційна база даних [16] дозволить зберігати дані з найрізноманітніших джерел у оригінальному вигляді. Це спростить інтеграцію таких джерел з одного боку, та дозволить впровадити уніфікований підхід до отримання даних – з іншого. Виходячи з попереднього вибору Microsoft Azure як постачальника хмарних сервісів, стандартним рішенням такого типу є Azure Cosmos DB SQL API.

Власний користувацький інтерфейс. З метою забезпечити максимальну зручність у користуванні системою для нефахових користувачів, необхідно впровадити рішення, що буде максимально адаптоване під вимоги саме нашої системи. Для цього доцільним є побудова власного користувацького інтерфейсу на основі існуючих фреймворків для візуалізації даних.

2.3. Загальна архітектура системи

Враховуючи прийняті архітектурні рішення відобразимо компоненти системи та взаємодію між ними (Рис. 2.2.).

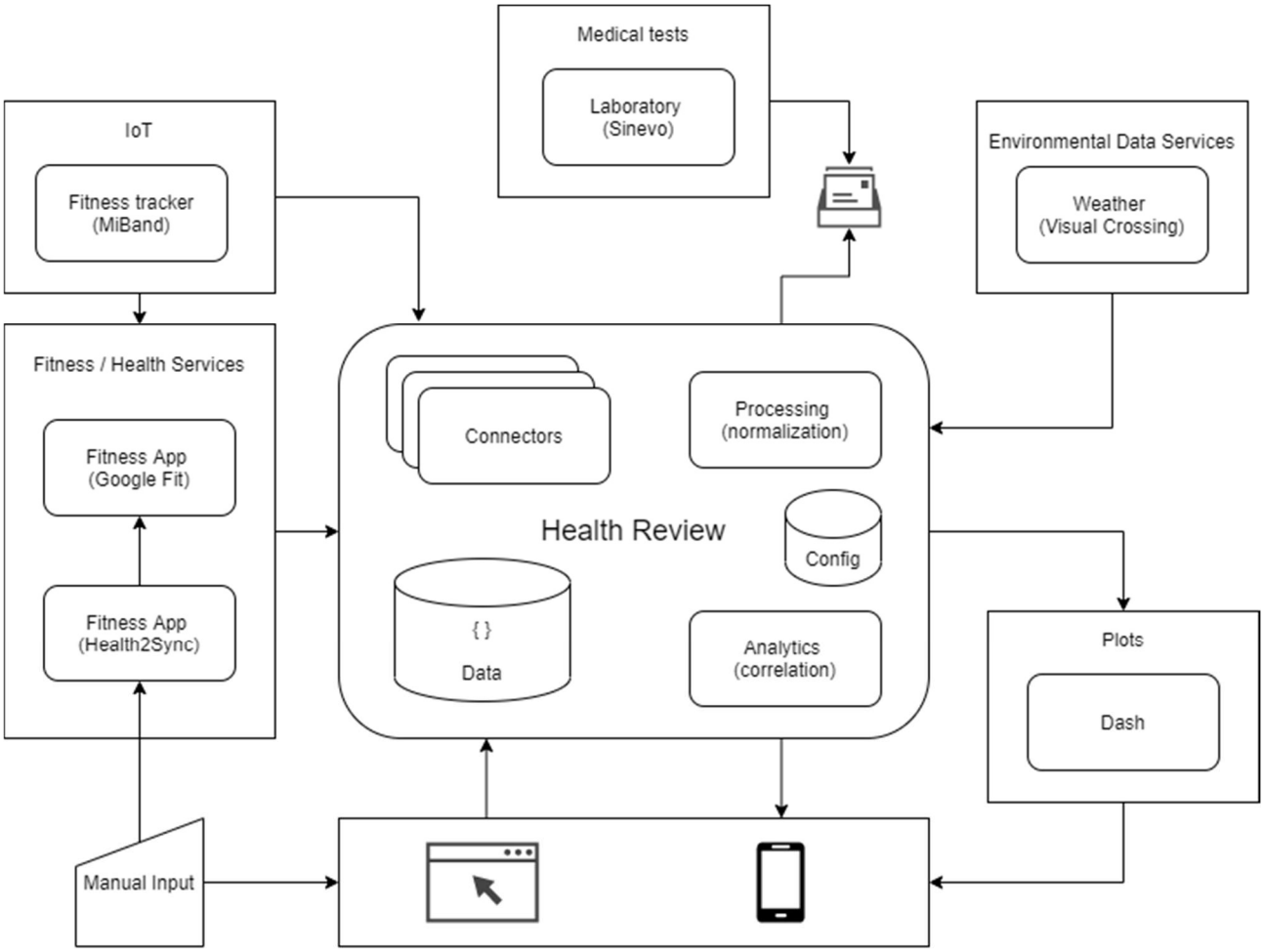

Рис. 2.2. Загальна структура системи і її взаємодія з зовнішніми компонентами

2.4. Альтернативні варіанти архітектури

В процесі розробки архітектури нами було визначено два варіанти архітектури в контексті презентації даних, кожен з яких краще відповідає певним атрибутам якості.

Перший варіант припускає використання єдиної нереляційної бази даних як для накопичення якомога повнішої інформації, так і для зчитування даних у структурованій формі для цілей презентації і обрахунку кореляції. Такий підхід, зображений на Рис. 2.3., краще відповідає вимогам розширюваності (простіша структура системи та відсутність додаткової трансформації даних) та безпеки (менша кількість потоків передачі даних між різними компонентами системи).

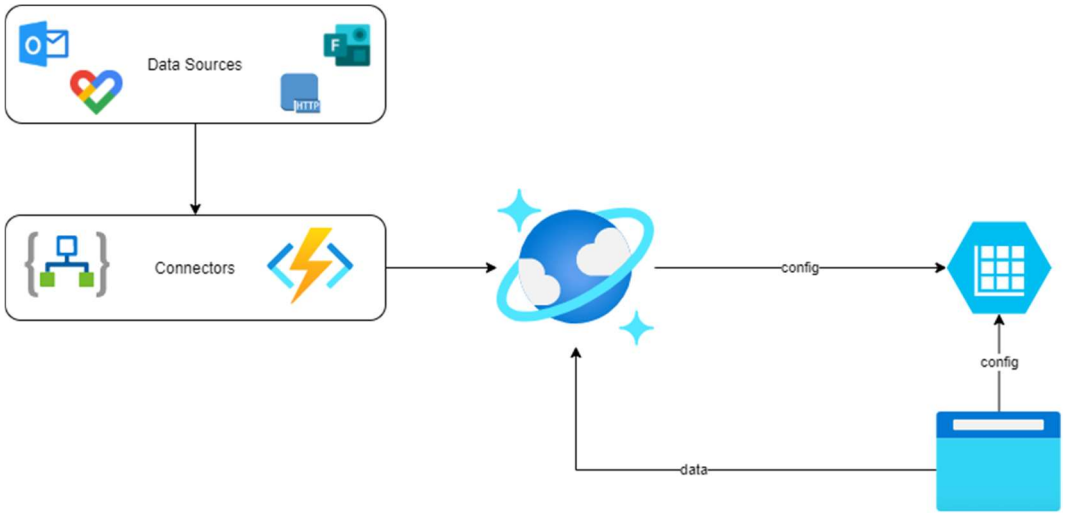

Рис. 2.4. Варіант архітектури 1. Єдина база даних

В другому ж варіанті, зображеному на Рис. 2.4., ми пропонуємо розділити сховище на дві частини. Вхідні дані так само завантажувати в нереляційну базу даних, щоб залишити можливість виокремити нові елементи зі збережених документів у майбутньому. Виключно ж дані для презентації попередньо трансформувати та зберегти у більш структуровану базу даних (реляційну чи колонкову) для більш ефективного зчитування [16].

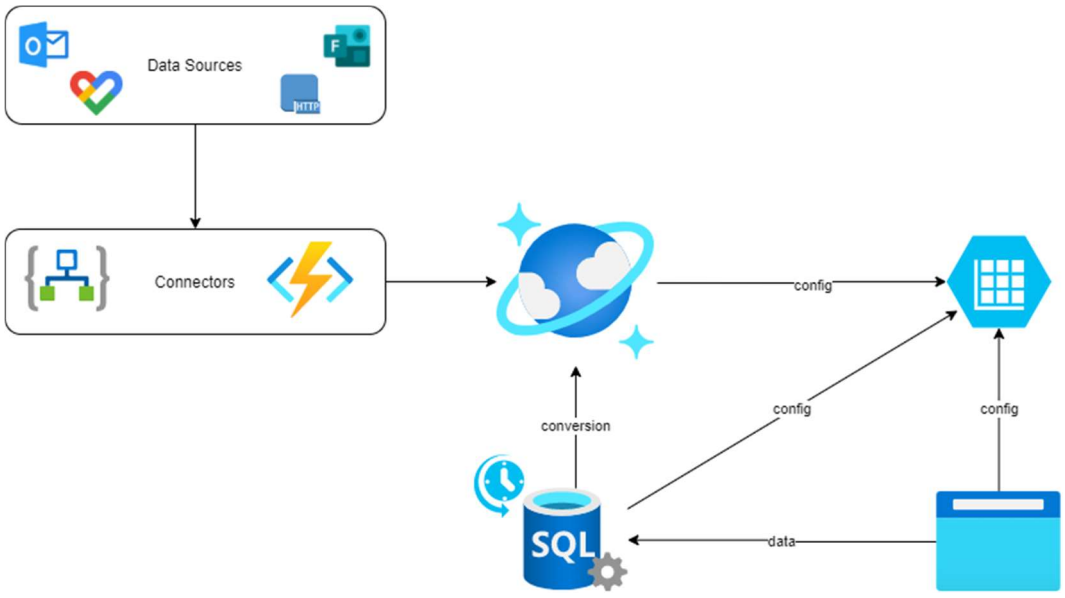

Рис. 2.4. Варіант архітектури 2. Окремі бази даних

Для того, щоб визначити кращий варіант архітектури було прийнято рішення побудувати прототипи системи для обох варіантів архітектури та виміряти метрики продуктивності кожного з них. В подальшому оцінити виграш,

що дає більш складний, але оптимізований варіант архітектури та визначити чи є його впровадження доцільним, порівняно з додатковими витратами на його створення та підтримку.

РОЗДІЛ 3. Реалізація програмного застосунку

3.1. Загальний принцип роботи системи

Згідно з прийнятими архітектурними рішеннями, основна частина системи буде вибудовуватися на основі хмарних сервісів Microsoft Azure. Відповідно, в рамках його підписки, створимо нову групу для ресурсів, що будуть використовуватися. Для ідентифікації елементів будемо використовувати потенційну назву для нашої системи – «Health Review».

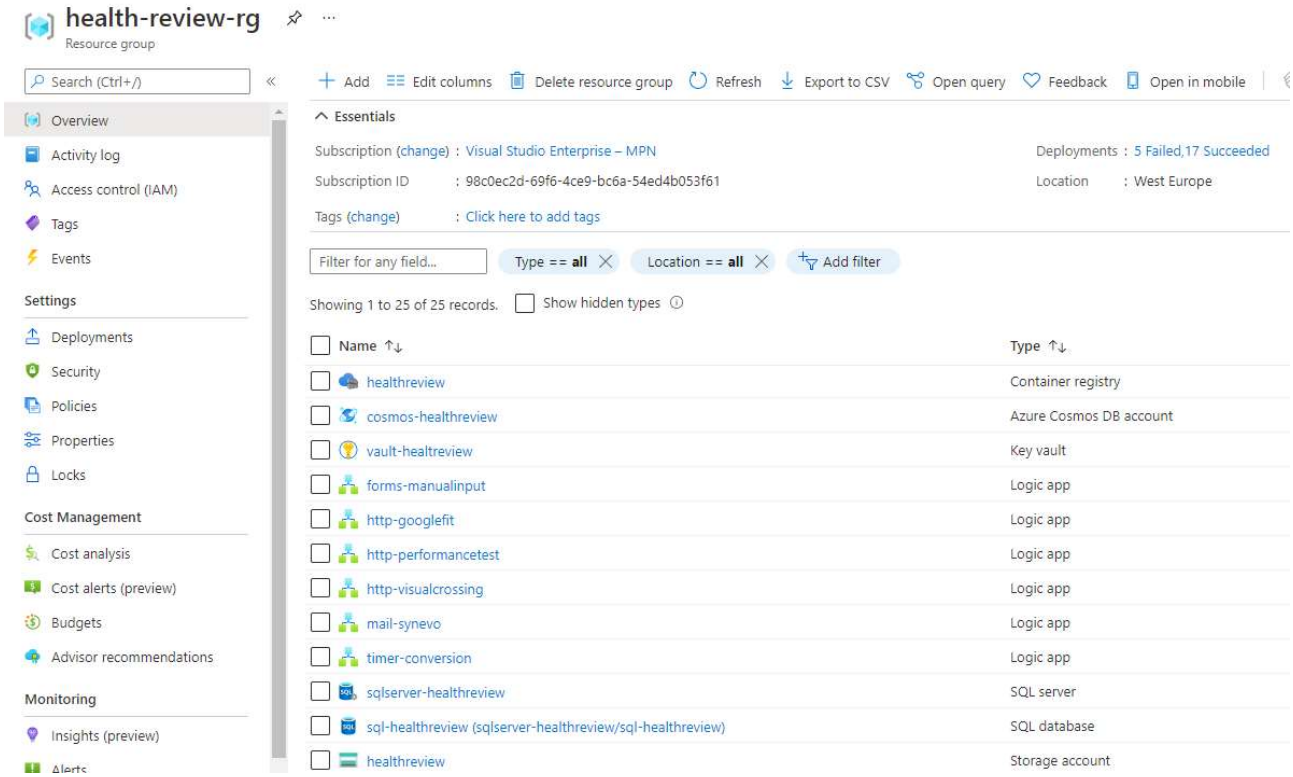

Рис. 3.1. Група ресурсів Microsoft Azure для проекту «Health Review».

В загальному робота системи буде працювати наступним чином:

1. Інформація з окремих джерел інформації буде надходити у систему за допомогою Azure Logic Apps. Де це можливо, будуть використані стандартні елементи, а при необхідності додаткової обробки інформації будуть застосовуватися Azure Functions (наприклад, для парсингу даних з pdf).
2. На виході кожної інтеграції ми повинні отримати документ у форматі json, що зберігатиметься у нереляційній базі даних. Кожен документ буде зберігати максимум інформації для можливості виокремити додаткову інформацію у

майбутньому. Для альтернативного варіанту архітектури необхідним буде ще одне сховище (у першому наближенні це буде звична реляційна база даних Azure SQL) та окремий процес, що буде здійснювати вибірку необхідних даних з документів та збереження в підготовленому форматі.

Окремим елементом у сховищі даних буде таблиця конфігурації, що у стандартизованому форматі буде зберігати інформацію, необхідну, щоб ідентифікувати в документі елемент, що відповідає конкретному параметру.

3. На останньому етапі через користувацьких інтерфейс будуть обиратися параметри, що цікавлять користувача. Для них буде відбуватися зчитування даних зі сховища, розрахунок кореляції та відображення результатів на екрані.

Для написання коду елементів системи, що не можуть бути реалізовані стандартними засобами було обрано мову Python. Це обумовлено її широким використанням, хорошою документацією та великою спільнотою спеціалістів, що займаються розвитком її екосистеми та активно публікують інформацію щодо нововведень або вирішення типових проблем. Але найважливішу роль відіграла наявність великої кількості загальнодоступних бібліотек, в тому числі таких, що можуть бути використані для вирішення задач в рамках побудови системи.

Для зберігання та зчитування конфігурації використаємо Azure Storage Tables – інструмент для зберігання табличних даних (фактично пар ключ-значення), що має низьку вартість і є зручним у використанні – окрім доступу через портал, для конфігурування можна використовувати Azure Storage Explorer або, навіть, Microsoft Office Excel. Зчитувати, записувати та оновлювати дані в таблицях під час виконання програми можна з допомогою бібліотеки `azure.storage.table`, створивши об’єкт `table_service`, та використовуючи його методи `_query_entities`, `insert_entity`, `update_entity` відповідно.

Так як середовищем для виконання окремих елементів буде операційна система Linux, то чутливу інформацію, таку як шляхи для підключення та ключі або поточні токени, для підвищення безпеки будемо зберігати виключно в змінних середовища і звертатися до них через метод `environ` бібліотеки `os`.

Логічно можемо розділити систему на три частини відповідно до їх функціонального призначення. Розглянемо побудову кожної з них окремо.

3.2. Підсистема інтеграції з джерелами даних

В основі кожної інтеграції буде знаходитися Azure Logic Apps, що може, залежно від ситуації, формувати запити до джерела на регулярній основі або ж очікувати запиту та приймати дані. Джерела для побудови прототипу системи були підібрані таким чином, щоб розглянути найбільш типові сценарії та створити шаблони для полегшення її подальшого розширення.

3.2.1. Ручний ввід даних

Почнемо з найпростішого сценарію – коли користувач хоче власноруч реєструвати дані у системі для того, щоб потім відслідковувати динаміку та порівнювати з іншими параметрами. Для прикладу використаємо Forms – зручний інструмент Microsoft Office для створення веб-форм.

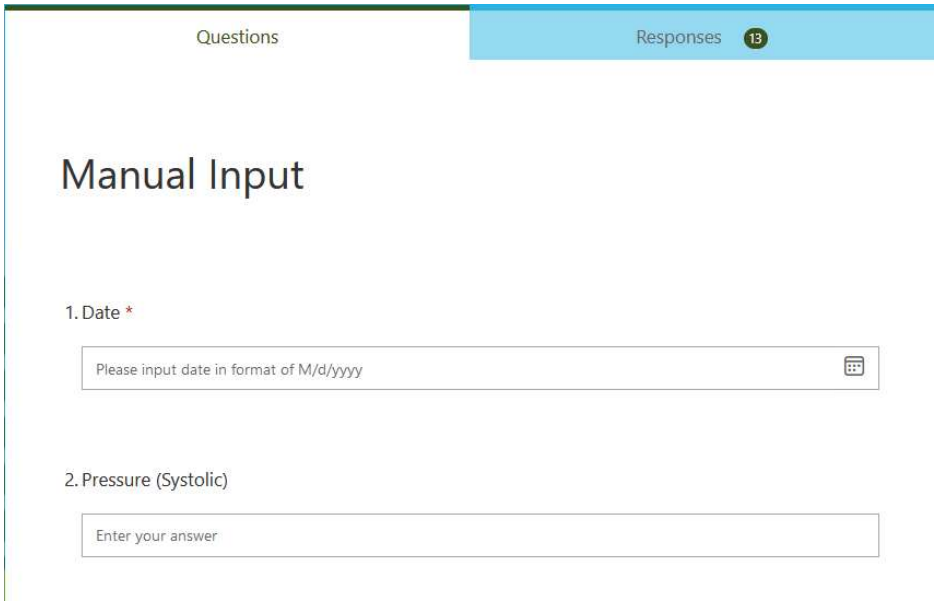

Рис. 3.2. Приклад веб-форми для ручного вводу даних.

Для роботи з такими формами є стандартні елементи у дизайнері для Logic Apps. Побудуємо його з через користувацький інтерфейс порталу Azure.

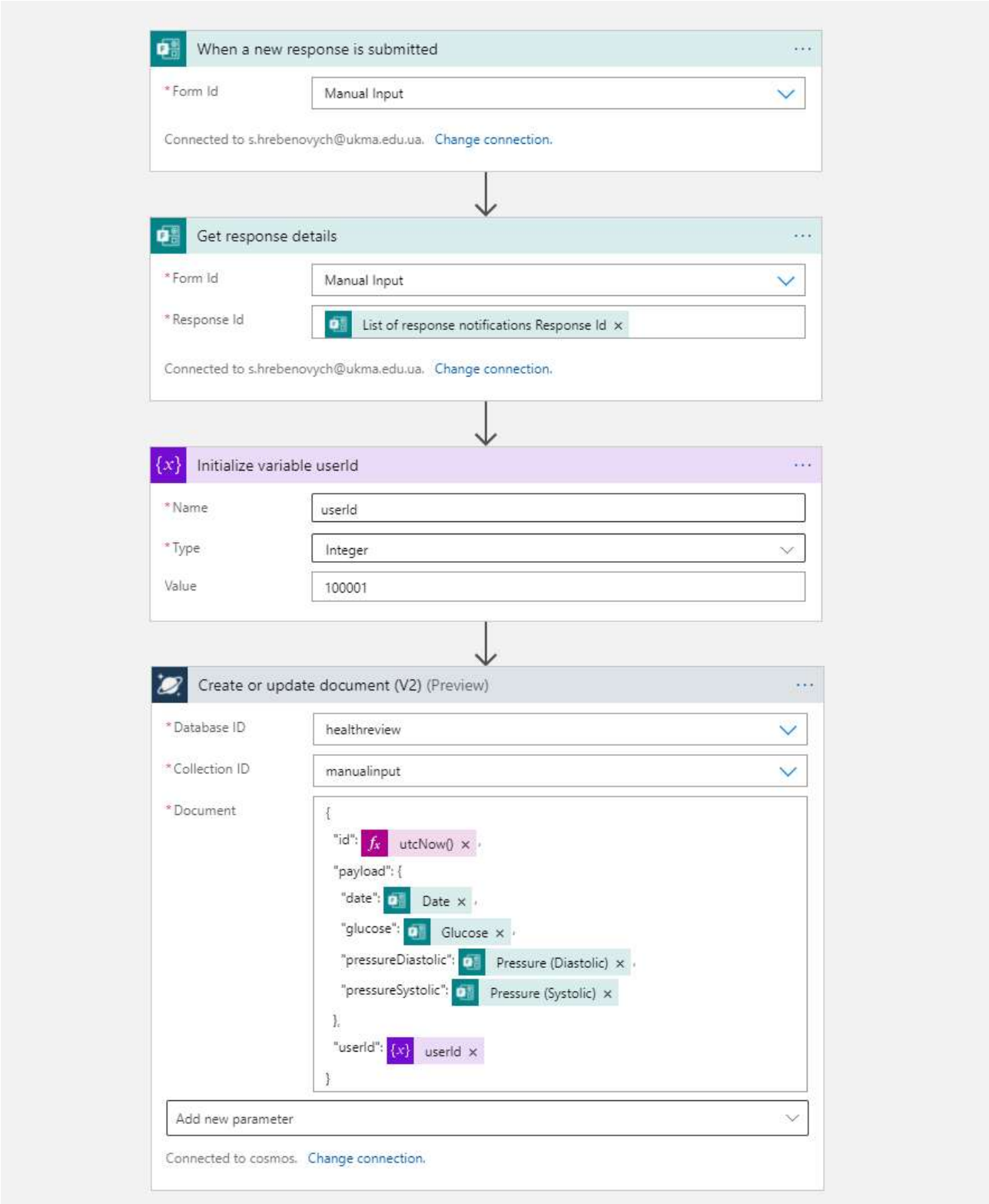

Рис. 3.3. Приклад побудови Logic App через користувацький інтерфейсу порталу Azure.

Альтернативним є використання шаблонів Azure Resource Manager (ARM), що можуть використовуватися для автоматизації створення та масштабування цих елементів системи.

В результаті роботи такого Logic App ми отримаємо наступний документ у Cosmos DB, який зможемо використовувати на наступних кроках:


```
{
    "id": "2021-05-28T00:38:06.1546811Z",
    "payload": {
        "date": "2021-06-01",
        "glucose": "5.5",
        "pressureDiastolic": "80",
        "pressureSystolic": "120"
    },
    "userId": 100001,
    "_rid": "idpeAMWQ+0EYAAAAAAAAAA==",
    "_self": "dbs/idpeAA==/colls/idpeAMWQ+0E=/docs/idpeAMWQ+0EYAAAAAAAAAA==/",
    "_etag": "\"bb009a8c-0000-0d00-0000-60b03b6f0000\"",
    "_attachments": "attachments/",
    "_ts": 1622162287
}
```


3.2.2. Парсинг результатів аналізів

Наступний сценарій, який ми розглянемо – це отримання результатів аналізів на електронну пошту. З допомогою Logic App ми можемо відслідковувати нові повідомлення на електронній пошті та опрацьовувати вкладення.

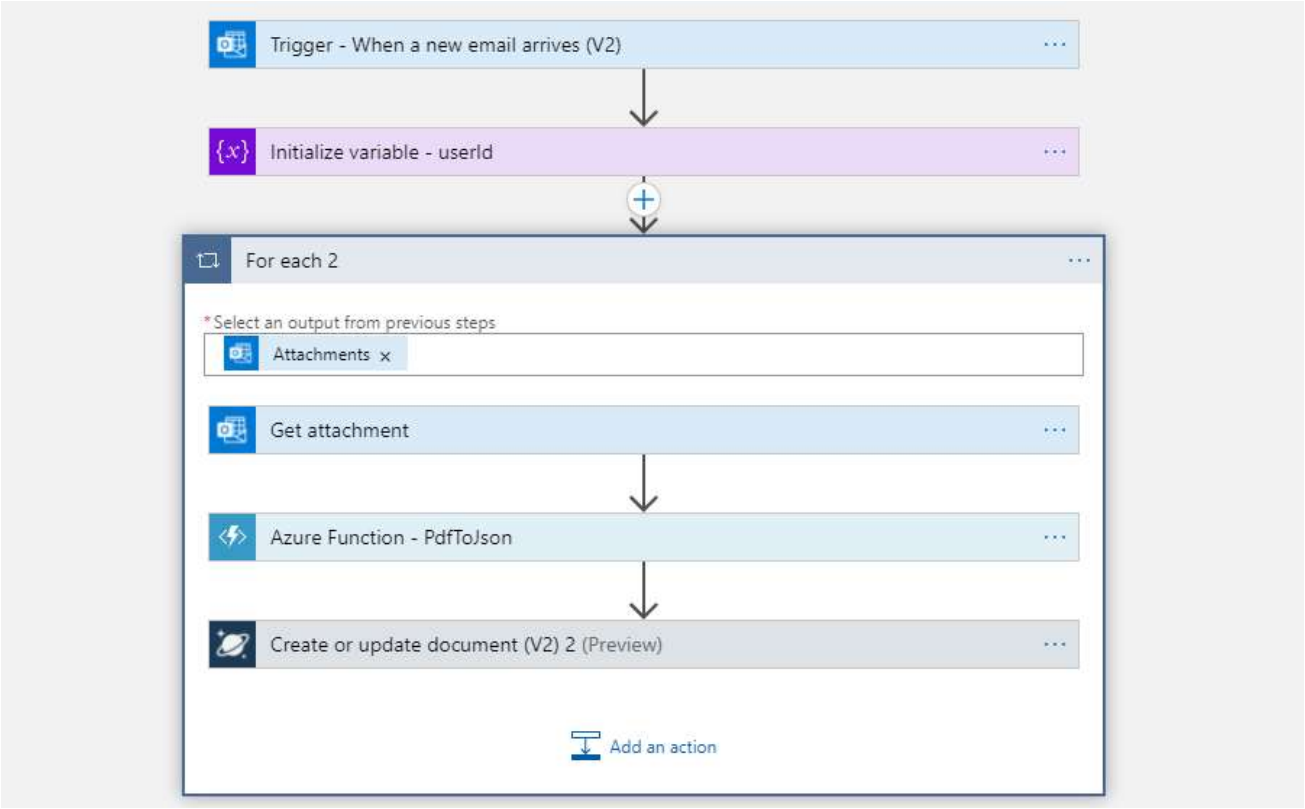

Рис. 3.4. Logic App для обробки даних, що отримуються на поштову скриньку

Оскільки формат вкладення може сильно відрізнятися залежно від постачальника медичних послуг, то в даному випадку ми застосуємо Azure Functions, що буде здійснювати парсинг pdf файлу, та шукати результати з допомогою регулярних виразів по наперед визначених шаблонах (характерних для цього постачальника).

Показник	Результат	Од.	Референтний інтервал
Система гемостазу			
Протромбіновий тест (ПЧ, % за Квіком, МНВ / INR)			
Протромбіновий час (ПЧ, PT, сек.)	22.2	сек	9.9 - 11.8
Протромбін за Квіком, %	29.4	%	70 - 130
Міжнародне нормалізоване відношення (МНВ/INR)	2.2		до 1.0
Біохімія			
Калій (сироватка)	4.51	ммоль/л	3.5 - 5.1

Рис. 3.5. Приклад результатів аналізів з pdf-документу

Для створення та розгортання безсерверної функції, використовується однойменне розширення для середовища розробки від Microsoft Visual Studio Code – Azure Functions. За допомогою елементів розширення ми створили проект (сховище для функцій) та, власне, саму функцію для мови Python під назвою PdfToJson, яка буде викликатися за допомогою HTTP запиту. При цьому автоматично створилися всі необхідні елементи та початковий файл __init__.py, що викликається за замовчування і має використовуватися як відправна точка для написання коду. Змінимо його для виконання потрібних нам інструкцій, враховуючи, що в тілі вхідного запиту ми отримаємо файл у форматі *.pdf:

```
def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    file_content = io.BytesIO(req.get_body())

    output_doc = {
        "clinic": req.headers.get('sender'),
        "document": parse_pdf(file_content)
    }
```

```
    return func.HttpResponse(
        json.dumps(output_doc),
        mimetype="application/json",
    )
```

Для обробки вхідного документу будемо використовувати два open-source фреймворки. PyPDF [17], що краще підходить для роботи з метаданими та pdfplumber [18] – для перетворення даних у текстовий формат. Вибір на користь останнього було зроблено, тому що ця бібліотека добре працює з кодуванням української мови та розбиває табличні дані з документах по рядках, що є зручним для подальшої ідентифікації корисного навантаження. Код основного методу:

```
def parse_pdf(file_content):
    meta_reader = PyPDF2.PdfFileReader(file_content)
    metadata = meta_reader.documentInfo
    pages_count = meta_reader.getNumPages()

    pages = {}
    document_text = ""
    with pdfplumber.open(file_content) as document_pdf:
        for i in range(pages_count):
            current_page = document_pdf.pages[i]
            page_text = current_page.extract_text()
            pages[i] = page_text
            document_text += page_text

    return {
        "date": extract_date(document_text, metadata),
        "results": extract_results(document_text),
        "metadata": metadata,
        "pages": pages
    }
```

Метод для визначення дати аналізів:

```
def extract_date(document_text, metadata):
    document_date = ""

    date_regex1 = ".*? ([0-3]?[0-9])[\-\./]([01]?[0-9])[\-\./](2?0?[0-
9]{2}) "
    meta_date_key = "/CreationDate"

    if re.match(date_regex1, document_text, flags=re.DOTALL):
        search_result = re.search(date_regex1, document_text, flags=re.DO-
TALL)
```

```
        document_date = f"{search_result.group(3)}-{search_result.group(2)}-
{search_result.group(1)}"
    elif meta_date_key in metadata.keys():
        metadata_date = metadata[meta_date_key].replace("'","")
        document_date = format(datetime.strp-
time(metadata_date, '%y%m%d%H%M%S%f%z'))
    else:
        document_date = format(datetime.datetime.now())

    return document_date
```

Методи для визначення корисного навантаження:

```
def extract_results(document_text):
    test_results = {}
    match_result = {}
    open_match = False

    num_regex = "(^[0-9.,]+?) (.*)"

    for line_text in document_text.split('\n'):
        if re.match(num_regex, line_text):
            open_match = True
            match_result = {
                "number": float(re.search(num_regex, line_text).group(1)),
                "comment": re.search(num_regex, line_text).group(2).strip()
            }
        elif open_match:
            test_results[line_text] = match_result
            open_match = False
            match_result = {}

    return test_results

def extract_lines(document_text):
    lines = []
    for line_text in document_text.split('\n'):
        lines.append(line_text)

    return lines
```

В результаті виконання цієї функції для прикладу, наведеному на Рис. 3.5, отримаємо json-документ наступного вигляду (представлений частково):

```
{
    "id": "2021-04-18T00:01:22.4191173Z",
    "payload": {
        "clinic": "promo@vitagramma.com",
        "document": {
            "date": "2021-03-12",
```

```
"results": {
    "Результати досліджень  ": {
        "number": 3680,
        "comment": "Київ, Палладіна 46/2"
    },
    "Протромбіновий час (ПЧ, РТ, сек.)": {
        "number": 22.2,
        "comment": "сек 9.9 – 11.8"
    },
    "Протромбін за Квіком, %": {
        "number": 29.4,
        "comment": "% 70 - 130"
    },
    "Міжнародне нормалізоване відношення (МНВ/INR)": {
        "number": 2.2,
        "comment": "до 1.0"
    },
    "Калій (сироватка)": {
        "number": 4.51,
        "comment": "ммоль/л 3.5 - 5.1"
    }
},
"metadata": {
    "/Producer": "mPDF 8.0.0",
    …
```


В загальному випадку таким самим чином можна опрацьовувати також будь-які відскановані документи з допомогою фреймворків для роботи з OCR. Також, у майбутньому шаблони пошуку параметрів можна узагальнити або визначати з допомогою конфігурації.

3.2.3. Інформація про погодні умови

Отримання інформації про погоду буде прикладом протилежного напряму інтеграції – дані будуть отримуватися через запит на спеціалізований ресурс, що ініціюватиметься на регулярній основі (щоденно).

Серед так званих Weather API ми зупинилися на Weather Crossing (https://www.visualcrossing.com/), який надає достатню інформацію як про поточні показники навколишнього середовища (температуру, атмосферний тиск, вологість, тощо), так і історичні дані для обраного міста.

З міркувань безпеки ключ для запитів не вказувався явно, натомість було використано сховище Azure Key Vault. Logic Apps дозволяють використовувати чутливі дані, що там зберігаються, за допомогою стандартних інструментів і без необхідності явно вказувати їх в нешифрованому вигляді.

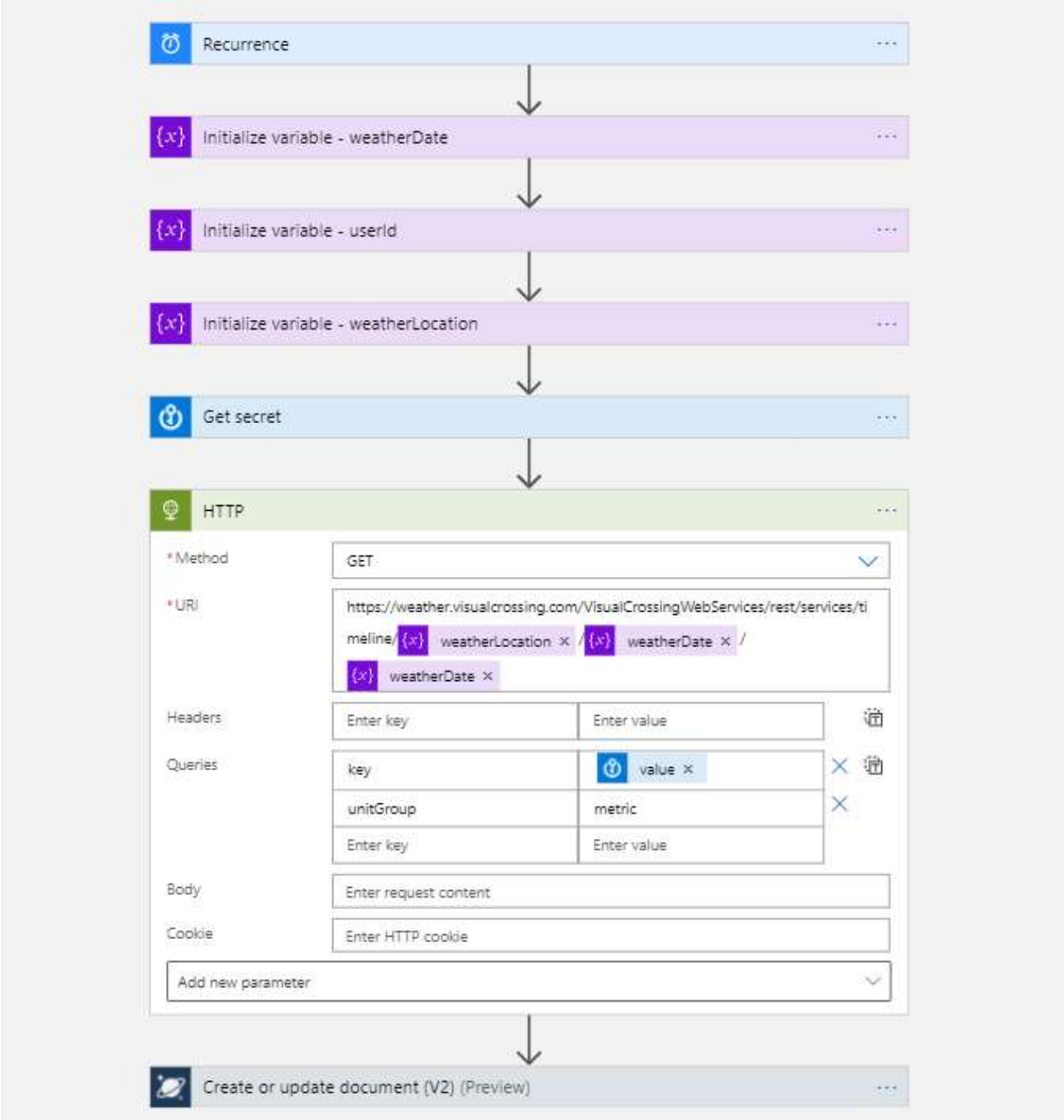

Рис. 3.6. Logic App для відправки запиту на дані до зовнішнього ресурсу

3.2.4. Дані з портативних пристроїв

Виходячи з дослідження, проведеного в попередніх розділах, Google Fit є оптимальним мобільним застосунком (чи точніше – сервісом) для інтеграції з нашою системою. По-перше, як всі подібні застосунки, він може об’єднувати дані, зібрані з різних приладів та інших застосунків, що використовують єдиний

Google-акаунт. Та, по-друге, на відміну від інших, він має доступний RESTful API, що ми можемо використати безпосередньо у Logic Apps.

Для доступу до Fitness API, що відповідає за інтеграцію цієї інформації, необхідно, перш за все, активувати його у Google API Console та отримати дані для подальшої автентифікації. Серед іншого, потрібно задати контексти та налаштувати Google Sign-In, а також авторизувати доступ з нашої системи [9]:

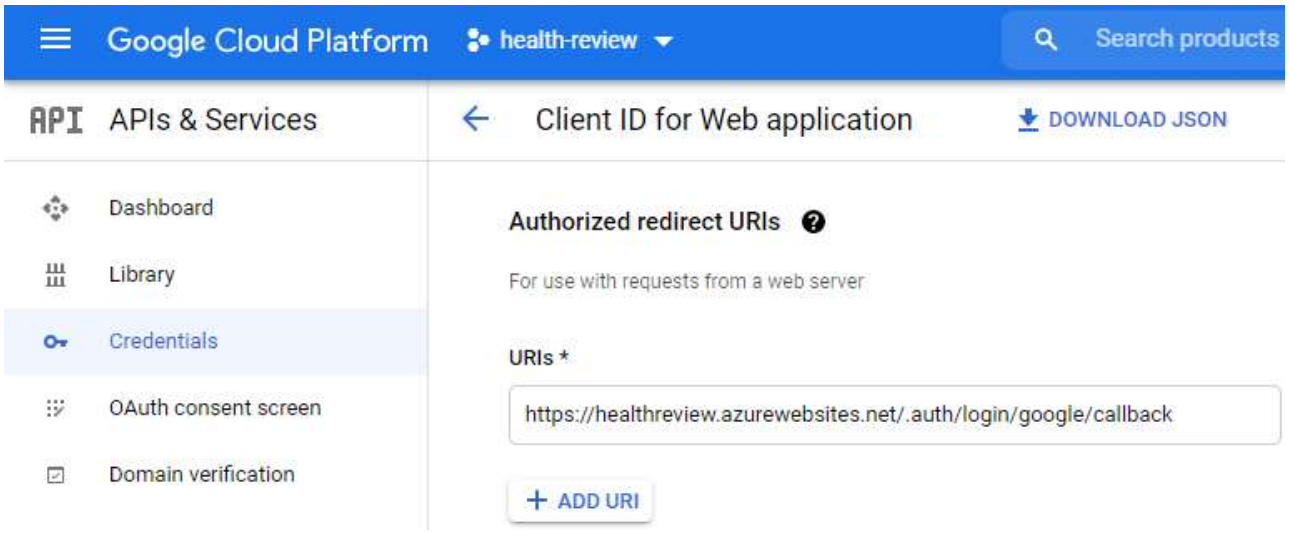

Рис. 3.7. Налаштування авторизації для Google API

Так як Google дозволяє використовувати лише короткотривалі токени для цього API, а вони можуть отримуватися лише за явної згоди користувача через браузер, ми не можемо використати Logic App для фонового оновлення даних без втручання користувача.

З іншого боку, ми можемо використовувати Google Oauth2 для автентифікації при вході в користувацький інтерфейс нашої системи та оновлювати дані на цьому кроці за явної згоди користувача.

В такому випадку Logic App буде виконуватися на вимогу, а токен, необхідний для отримання даних через Fitness API буде передаватися йому при виклику (у наступних версіях системи, з метою поліпшення безпеки планується застосовувати Azure Key Vault для тимчасового зберігання токенів).

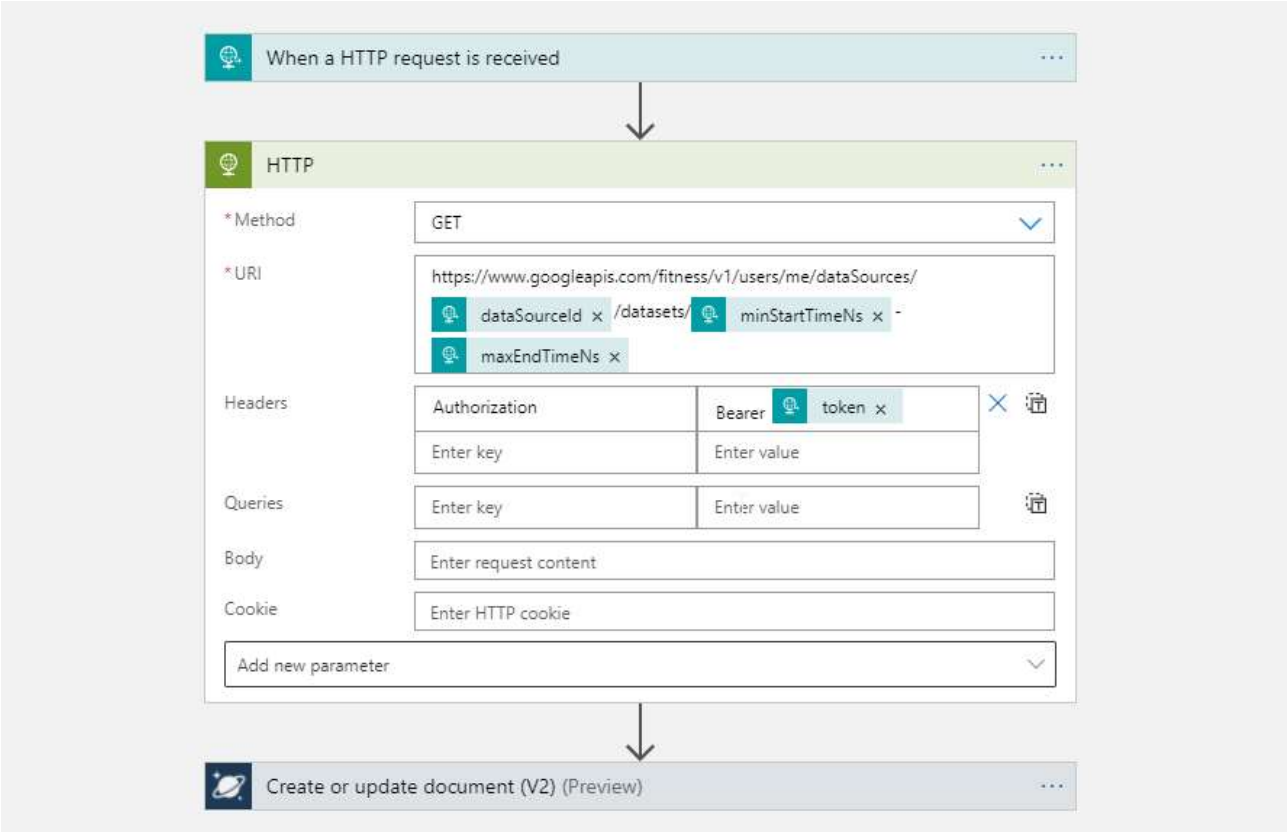

Рис. 3.8. Logic App для взаємодії з Fitness API від Google

Інструкція для виклику буде включена до коду користувацької підсистеми і матиме наступний вигляд:

```
def request_googlefit_update():
    current_epoch = time.time_ns()
    url = os.environ['GOOGLE_FIT_UPDATE_ENDPOINT']
    googlefit_streams = table_service._query_entities('googlefitstream')
    for current_stream in googlefit_streams:
        req_body = {
            "userId": UserId,
            "minStartTimeNs": format(current_stream['maxEndTimeNs']),
            "maxEndTimeNs": format(current_epoch),
            "dataSourceId": current_stream['dataSourceId'],
            "token": os.environ['X-MS-TOKEN-GOOGLE-ACCESS-TOKEN']
        }

        requests.post(url=url, json=req_body)

        updated_stream = {
            "PartitionKey": current_stream['PartitionKey'],
            "RowKey": current_stream['RowKey'],
            "dataSourceId": current_stream['dataSourceId'],
            "maxEndTimeNs": current_epoch
        }
        table_service.update_entity('googlefitstream', updated_stream)
```

Оскільки Fitness API вимагає вказувати конкретні типи даних для кожного запиту, то вони мають бути задані завчасно. Створимо для цього конфігураційну таблицю `googlefitstream`, в якій будемо зберігати типи даних, а також час останнього запиту, щоб також обмежувати запит лише тим інтервалом часу, коли ми ще не отримували даних.

Рис. 3.9. Приклад конфігурації типу даних для інтеграції з Google Fit.

3.3. Сховище даних

Як було описано вище, основним сховищем для даних буде виступати нереляційна база даних Cosmos DB SQL API, де будуть зберігатися вхідні дані в оригінальному вигляді у форматі json-документів, щоб уникнути втрати інформації, що може знадобитися (стати більш актуальною) у майбутньому.

Важливим елементом системи є трансформація сирих вхідних даних у часові ряди для по потрібним параметрам для представлення їх у вигляді графіків і аналізу часової кореляції між різними показниками. Для цього ми будемо використовувати стандартні можливості Cosmos DB SQL API, які дозволяють працювати як з простими, так і доволі складними структурами json-документів [19]. Розглянемо один з найскладніших випадків, а саме отримання показників ваги з Google Fit API. Складність була обумовлена тим, що, на відміну від інших джерел, час зберігається у форматі unixtimestamp, що вимагає трансформації вже на етапі підготовки даних, а сама структура документу не є прямолінійною і складається з кількох рівнів та вкладених масивів.

Приклад документу, створеного на основі інтеграції з Google Fit, що містить інформацію про вагу:

```
{
    "id": "2021-04-18T21:47:32.3057004Z",
    "payload": {
        "minStartTimeNs": "1609452000000000000",
        "maxEndTimeNs": "1618782428000000000",
        "dataSourceId": "raw:com.google.weight:com.google.android.apps.fitness:user_in-
put",
        "point": [
            {
                "startTimeNanos": "1617985644000000000",
                "endTimeNanos": "1617985644000000000",
                "dataTypeName": "com.google.weight",
                "value": [
                    {
                        "fpVal": 83,
                        "mapVal": []
                    }
                ],
                "modifiedTimeMillis": "1618763258296"
            },
            {
                "startTimeNanos": "1618762730638229760",
                "endTimeNanos": "1618762730638229760",
                "dataTypeName": "com.google.weight",
                "value": [
                    {
                        "fpVal": 86,
                        "mapVal": []
                    }
                ],
                "modifiedTimeMillis": "1618762739401"
            }
        ]
    },
    "userId": 100001,
    "_rid": "idpeAMD5yqACAAAAAAAAAA==",
    "_self": "dbs/idpeAA==/colls/idpeAMD5yqA=/docs/idpeAMD5yqACAAAAAAAAAA==/",
    "_etag": "\"720056e3-0000-0d00-0000-607ca8f40000\"",
    "_attachments": "attachments/",
    "_ts": 1618782452
}
```


Запит для отримання інформації у вигляді, готовому для подальшої обробки:

```
SELECT
    TimestampToDateTime(StringToNumber(SUBSTRING(p.endTimeNanos, 0, 13))) AS timestamp,
    p['value'][0].fpVal AS payload
FROM c JOIN p IN c.payload.point
WHERE p.dataTypeName = 'com.google.weight'
```

Результат його виконання (представлений частково):


```
[
    {
        "timestamp": "2021-04-09T16:27:24.0000000Z",
        "payload": 83
```

```
},
{
    "timestamp": "2021-04-18T16:18:50.6380000Z",
    "payload": 86
},
…
```

Для всіх випадків, що розглядалися в даній роботі нам вдалося отримати дані у потрібному форматі без застосування додаткових засобів. Відповідно, у реалізації першого варіанту архітектури ми будемо використовувати прямі SQL-запити до бази даних, отримавши при цьому максимальну гнучкість та простоту. В другому ж, попередньо трансформуємо дані та збережемо їх у реляційній базі даних, виходячи з припущення, що за рахунок простішої та фіксованої структури з вбудованими механізмами індексації, ми забезпечимо кращу швидкодію системи. Справедливість такого припущення ми перевіримо експериментально.

Запропонований уніфікований підхід до зберігання та трансформації даних також дозволяє нам уникнути необхідності програмних змін в частині презентації даних при підключенні нових джерел інформації. Натомість, ми будемо зберігати SQL-запити для отримання часових рядів у конфігураційній таблиці `sourceparameter`.

При зберіганні розіб'ємо SQL-запити на фіксовані структурні елементи таким чином, щоб мати можливість звертатися до них окремо. Поля `timestamp_path` та `payload_path` міститимуть вирази для отримання значень у директиві `SELECT`, в свою чергу `collection_path` та `collection_filter` – директив `FROM` та `WHERE` відповідно. Запит для трансформації у часові ряди буде умовно формуватися наступним чином (відповідні поля конфігураційної таблиці позначено у фігурних дужках):

```
SELECT
     {timestamp_path} AS timestamp,
     {payload_path} AS payload
FROM {collection_path}
WHERE {collection_filter}
```

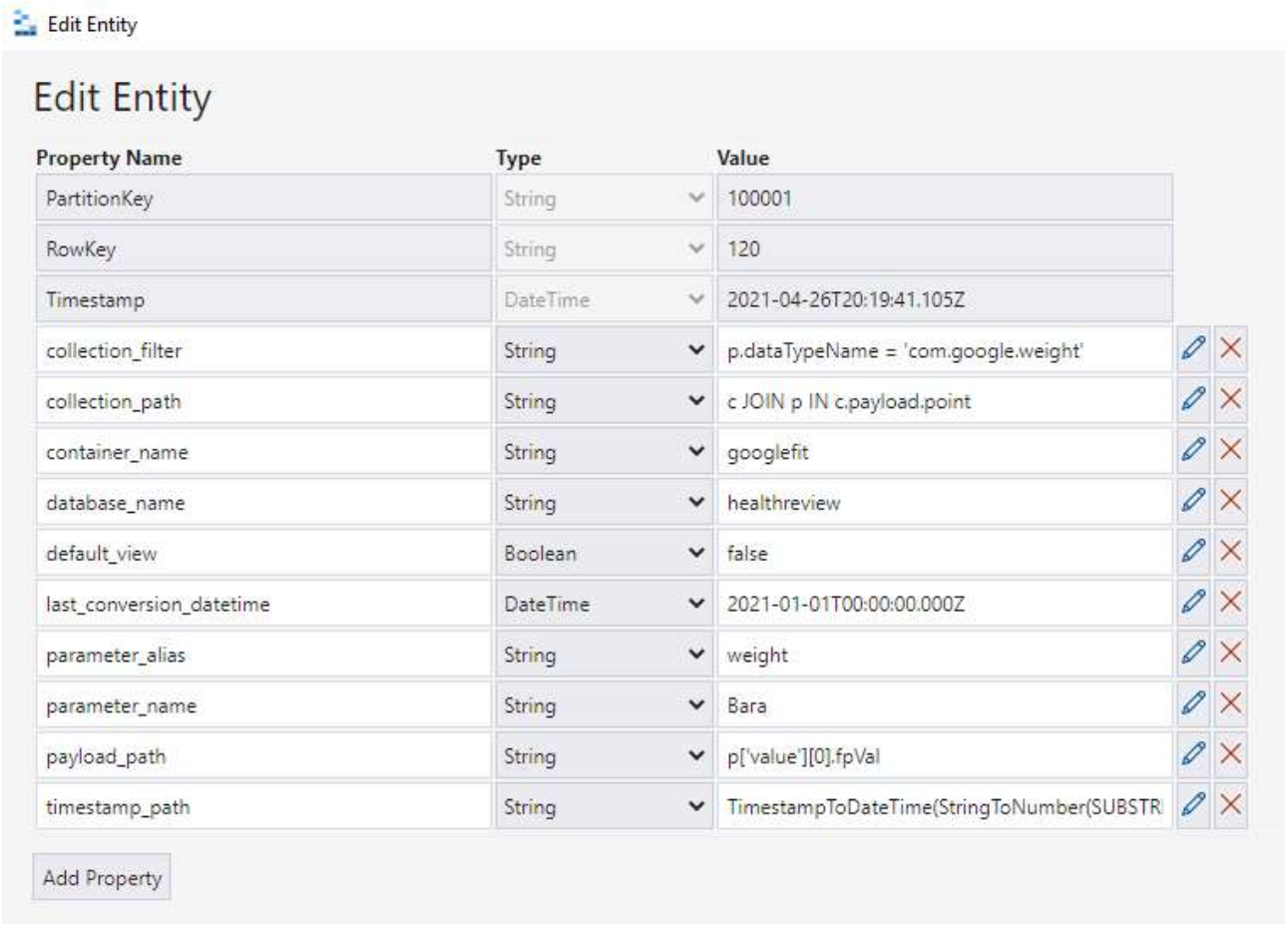

Рис. 3.10. Приклад конфігурації для отримання даних для конкретного параметру.

Це також дозволить нам автоматизувати створення конфігураційних елементів у майбутніх версіях системи. Наприклад, через користувацький інтерфейс, будуть обиратися необхідні елементи в документі – колекцію, час вимірювання, значення параметру, а система буде сама створювати відповідний запис у конфігураційній таблиці.

Ця сама конфігураційна таблиця буде використовуватися для трансформації та збереження даних у реляційну базу даних. Кожен параметр зберігатимемо у окремій таблиці, назва якої буде визначатися полем `parameter_alias`, а колонки будуть співпадати з результатом прямого запиту. Для виконання конвертації створимо нову функцію CostmosToSql в існуючому проекті Azure Functions і будемо викликати її на регулярній основі.

Безпосередній спосіб зчитування даних буде залежати від моделі архітектури, але на виході ми повинні отримати дані у блоках даних – об’єкті

бібліотеки pandas, що широко застосовується для статистичного аналізу, роботи з великими даними та візуалізації [20].

3.4. Підсистема візуалізації та аналізу

В якості аналітичної функціональності, що буде доступна кінцевому користувачу ми обрали розрахунок кореляції між часовими рядами значень показників, що будуть розраховуватися трьома методами [21]:

1. Коефіцієнт кореляції Пірсона – міру лінійної залежності між двома величинами;
2. Коефіцієнт кореляції рангу Спірмена – показує наскільки монотонним є зв'язок між параметрами;
3. Коефіцієнт кореляції рангу Кендала – подібність упорядкованих величин.

Розрахунок проводиться за допомогою бібліотеки pandas наступним чином, в залежності від обраного користувачем методу:

```
    timeseries1 = timeseries1.drop_duplicates('timestamp').set_index('timestamp')
    timeseries2 = timeseries2.drop_duplicates('timestamp').set_in-
dex('timestamp').reindex(timeseries1.index, method='nearest')
    correlation_value = timeseries1.corrwith(timeseries2, method=corr_method)['pay-
load']
```

Для реалізації користувацького інтерфейсу використовувався фреймфорк мови Python для побудови веб-застосунків під назвою Dash [22]. Він побудований на основі іншого широко вживаного фреймворку Flask, але є додатково оснащений інструментами для візуалізації даних.

Для початку роботи потрібно створити об'єкт застосунку і обрати стиль відображення (тему):

```
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.YETI])
```

Також завантажимо конфігураційну таблицю, що містить дані про параметри, що нас цікавлять:

```
table_service = TableService(connection_string=os.environ['CUS-
TOMCONNSTR_STORAGE_ACCOUNT'])
parameters = pd.DataFrame(table_service._query_entities('sourceparame-
ter')).set_index('RowKey')
```

Далі створимо елементи, що будуть відображені на сторінці з допомогою модулю html:

– Два випадаючі списки, що будуть надавати можливість обрати той чи інший параметр з доступних в таблиці конфігурації;

– Радіокнопку для вибору типу кореляції

– Текстові елементи для відображення розрахованих величин

– Кнопка для отримання останніх даних з Google Fit

– Власне, графіки для двох обраних параметрів

```
app.layout = html.Div([
    html.Button('Update Data', id='update-input-but-
ton', n_clicks=0, style={'float': 'right'}),
    html.Br(),
    html.H1(f'Health Review ({arch_model})', style={'textAlign': 'center'}),
    dcc.Dropdown(
        id='data-input-dropdown-1',
        options=[{'label': parameters.loc[i]['parameter_name'], 'val-
ue': i} for i in parameters.index],
        value=parameters[parameters['default_view']==1].index[0]
    ),
    dcc.Dropdown(
        id='data-input-dropdown-2',
        options=[{'label': parameters.loc[i]['parameter_name'], 'val-
ue': i} for i in parameters.index],
        value=parameters[parameters['default_view']==1].index[1]
    ),
    html.Br(),
    dcc.RadioItems(id='corr-input-radio',
        options=[
            {'label': 'Pearson', 'value': 'pearson'},
            {'label': 'Kendall', 'value': 'kendall'},
            {'label': 'Spearman', 'value': 'spearman'}
        ],
        value='pearson'
    ),
    html.Div(id='corr-output-textbox'),
    html.Div(id='time-output-textbox'),
    dcc.Graph(id='data-output-graph')

], style={'width': '500'})
```

Рис. 3.11. Вигляд елементів взаємодії з користувачем.

Виконання логіки системи відбувається реактивно при будь-якій зміні в елементах для взаємодії з користувачем: вибір нового параметру, вибір виду кореляції, натискання кнопки для оновлення даних. Метод, що буде викликатися при таких діях повинен декораторуватися посиланнями на елементи, які ініціюють виконання методу (Input, повинні співпадати з вхідними параметрами), а також елементами, що оновлюються після його завершення (Output, повинні співпадати з вихідними):

```
@app.callback(
    Output(component_id='data-output-graph', component_property='figure'),
    Output(component_id='corr-output-textbox', component_property='children'),
    Output(component_id='time-output-textbox', component_property='children'),
    Input(component_id='update-input-button', component_property='n_clicks'),
    Input(component_id='data-input-dropdown-1', component_property='value'),
    Input(component_id='data-input-dropdown-2', component_property='value'),
    Input(component_id='corr-input-radio', component_property='value')
)
def update_graph(n_clicks, parameter_id1, parameter_id2, corr_method):
     # …
     # код методу
     # …
     return fig, 'Correlation: {}'.format(correlation_value), 'Pro-
cess time: {}'.format(processing_time)
```

Для зчитування даних напряму з CosmosDB буде використовутися наступний підхід, на виході якого ми отримаємо список з потрібними даними:

```
        cosmos_database = cosmos_client.get_database_client(parameter['data-
base_name'])
        cosmos_container = cosmos_database.get_container_client(parameter['con-
tainer_name'])
        cosmos_query = f"""
            SELECT
                c.userId,
                {parameter['timestamp_path']} AS timestamp,
                {parameter['payload_path']} AS payload
            FROM {parameter['collection_path']}
            WHERE {parameter['collection_filter']}
              AND c.id > '{last_conversion_datetime}'
        """

        items = list(cosmos_container.query_items(
            query=cosmos_query,
            enable_cross_partition_query=True
        ))

        df = pd.DataFrame(items).sort_values(by=['timestamp'])
```

Для зчитування ж конвертованих даних з Azure:

```
        sql_con = pyodbc.connect(os.environ['SQLAZURECONNSTR_SQLDB'])
        sql_query = f"""
            SELECT timestamp, payload
            FROM [dbo].[{parameter['parameter_alias']}]
            WHERE userId = {parameter['PartitionKey']}
        """
        df = pd.read_sql(sql_query, sql_con).sort_values(by=['timestamp'])
```

Наступним етапом є побудова графіків на основі отриманих даних. Dash дозволяє розмістити два графіки на одному полотні одночасно для візуального порівняння даних. Для цього необхідно ввімкнути відповідну властивість при створенні об’єкту:

```
    fig = make_subplots(specs=[[{"secondary_y": True}]])
```

Останнім кроком є додавання двох наборів даних:

```
    fig.add_trace(
        go.Scatter(x=timeseries1['timestamp'], y=timeseries1['payload'], name=pa-
rameters.loc[parameter_id1]['parameter_name']),
        secondary_y=False,
```

```
    )

    fig.add_trace(
        go.Scatter(x=timeseries2['timestamp'], y=timeseries2['payload'], name=pa-
rameters.loc[parameter_id2]['parameter_name']),
        secondary_y=True,
    )
```

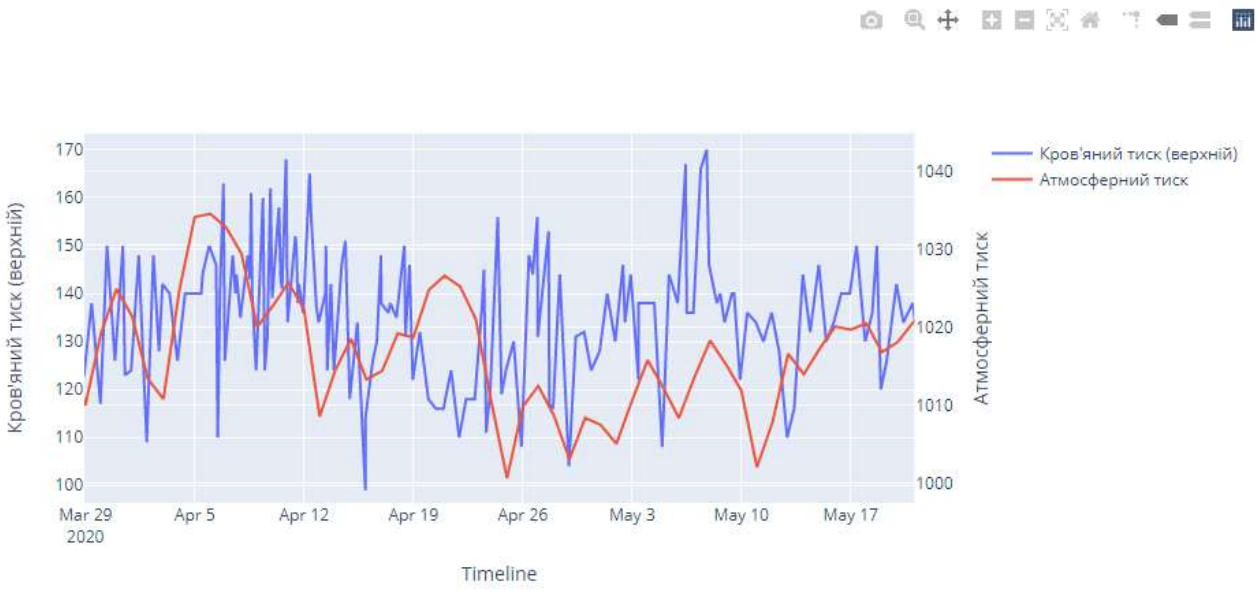

Рис. 3.12. Вигляд елементів візуалізації даних.

3.5. Оцінка ефективності

3.5.1. Вимірювання метрик

Для оцінки ефективності роботи системи у код застосунку була додана функціональність для вимірювання часу виконання наступних етапів: отримання даних, розрахунок кореляції, побудова графіків.

Перед і після виконання кожного етапу замірюється час, а розрахована тривалість, а також додаткова інформація про запит (кількість отриманих рядків, вид кореляції) фіксується у спеціальній таблиці `executionmetrics` у Azure Storage поряд із таблицями конфігурації:

```
    processing_time = time.time() - processing_start
    exec_metrics = {
        "PartitionKey": UserId, "RowKey": format(time.time()),
        "ArchitectureModel": arch_model, "SystemModule": "frontend",
        "DataSet1_Count": timeseries1.shape[0], "Da-
taSet2_Count": timeseries2.shape[0],
```

```
    "Duration_Retrieval": retrieval_time, "Duration_Plotting": plotting_time,
    "Duration_Correlation": correlation_time, "CorrelationMethod": corr_method,
    "Duration_Total_Seconds": processing_time
}
table_service.insert_entity('executionmetrics', exec_metrics)
```

Сам додаток було розгорнуто у Docker-контейнері, який, в свою чергу, розгортається в середовищі Azure як Web App Service. Модель архітектури, що використовується при роботі додатка визначається значенням параметра середовища `ARCHITECTURE_MODEL`, що може набувати значення `COSMOSDB_TABLE_API` або `SQL_SERVER` для першої та другої з розглянутих моделей відповідно.

3.5.2. Результати вимірювань

Задля перевірки ефективності роботи системи на великих наборах даних було задано ряд тестових параметрів і створені скрипти для масової генерації документів з випадковими значеннями цих параметрів та додатковими фоновими даними. Заміри робилися щоразу після додавання даних за один рік (по два вимірювання на кожен день). Результати вимірювань загальної швидкості презентації та аналізу результатів, залежно від архітектурної моделі, наведено на Рис.3.13.

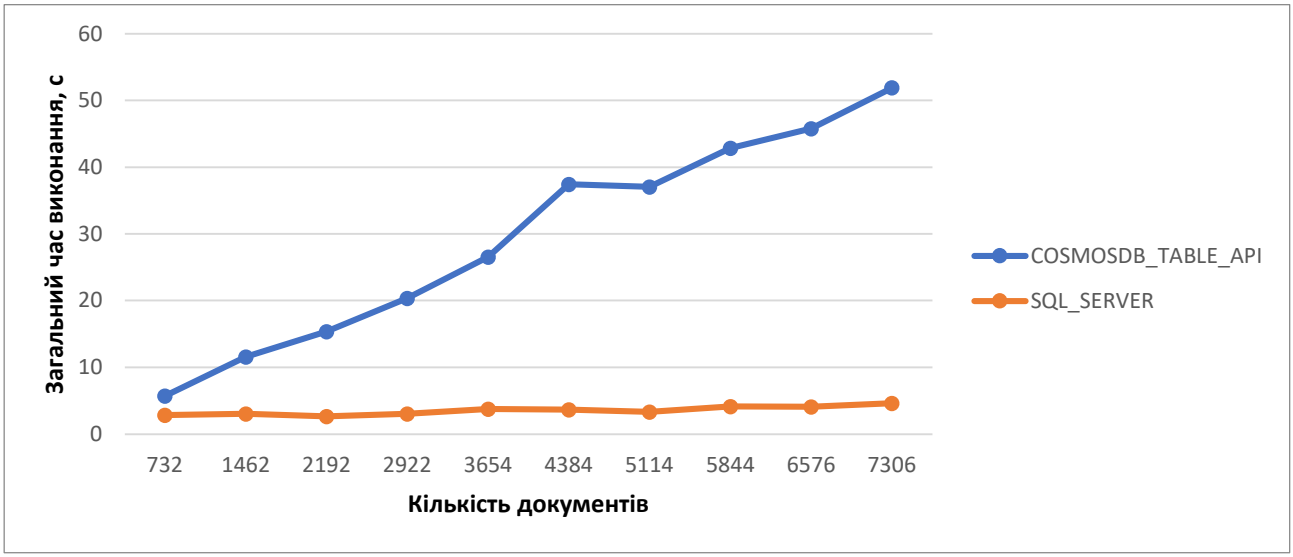

Рис. 3.13. Порівняння швидкості презентації даних для різних варіантів архітектури.

Як бачимо, за попередньої підготовки і збереження даних, швидкість є суттєво вищою. Тоді як швидкість прямого зчитування даних має сильну залежність від кількості накопичених даних.

Якщо розглянути більш детально структуру цієї метрики, то ми побачимо, що основний час витрачається на отримання даних зі сховища, тоді як розрахунок кореляції та побудова графіків є відносно швидкими. Усереднені показники по етапах для прикладу даних за 5 років (3654 документів), наведена у таблиці:

Тривалість, с	**COSMOSDB_TABLE_API**	**SQL_SERVER**
Отримання даних	26,5	3,6
Розрахунок кореляції	0,006	0,005
Побудова графіків	0,04	0,1
Загальна	***26,6***	***3,7***

Висновки по роботі та рекомендації для подальших досліджень

У першому розділі даної роботи було проведено огляд різних видів систем для реєстрації, візуалізації та аналізу показників життєдіяльності людини. Особлива увага приділялася ідеям застосування концепції Інтернету речей (IoT) та використання існуючих API комерційних рішень на базі мобільних застосунків, як таким, що мали найбільшу актуальність та могли бути реалізовані в рамках даного дослідження. В результаті, було запропоновано більш узагальнений підхід до накопичення даних з різних джерел та збагачення їх додатковою інформацією, маючи на меті їх спільне відображення та аналіз взаємозалежності (визначення мір кореляції) у зручному для кінцевого користувача вигляді.

Другий розділ присвячено визначенню та аналізу архітектурно важливих вимог для системи на основі запропонованого підходу. Визначено атрибути якості для архітектури системи, а саме: розширюваність, безпека та продуктивність (швидкість розрахунку кореляції на великих об’ємах даних у реальному часі). Запропоновано два варіанти архітектури системи: варіант більш простої системи, що є легшою для розширення та підтримки, та альтернативний, з попередньою трансформацією та збереженням даних у вигляді, оптимізованому для зчитування.

Останній, третій, розділ описує реалізацію прототипів програмних продуктів для обох варіантів архітектури на базі хмарних ресурсів Microsoft Azure (таких як Azure Logic Apps, Azure Functions, Azure Storage, Azure Cosmos DB та ін.) та бібліотек мови Python (Dash, pandas). Також оцінено метрики швидкодії кожного з них.

Аналіз отриманих результатів показав значну перевагу архітектури з використанням проміжного сховища підготовлених даних. Вже на невеликих наборах даних різниця є суттєвою і вона зростає з накопиченням історичних даних. Також, сильна залежність швидкодії від масиву інформації у першому варіанті архітектури робить її неприйнятною, не зважаючи на погіршення

розширюваності та збільшення периметру для можливих атак у випадку більш складної архітектури.

Запропонована модель оцінки ефективності архітектури такої системи може надалі бути використана для оцінки впливу інших архітектурних рішень на швидкодію систему. Наприклад, цікавим буде перевірити застосування альтернативних фреймворків для створення користувацького інтерфейсу та інших сховищ для трансформованих даних, що також є ефективними для роботи з часовими рядами (Azure Cosmos DB Cassandra API, HBase in HDInsight).

Окрім розробки нових шаблонів для підключення нових джерел інформації та впровадження додаткових можливостей для аналізу даних, подальший розвиток системи може бути направлений на організацію масштабування кількості користувачів, розробку інтерфейсу для налаштування системи та автоматизацію конфігурації правил трансформації даних.

Загалом, можемо зробити висновок, що навики в роботі зі стандартизованими інструментами та хмарними технологіями, а особливо інтеграції різних компонентів між собою є надзвичайно важливими в сучасній інженерії програмного забезпечення. Застосування таких підходів дозволяє доволі швидко та відносно невеликими зусиллями вибудовувати основу для складних та ефективних рішень. А це, в свою чергу, дозволяє скерувати увагу розробників на заточення кінцевих рішень під потреби їх аудиторії та покращення користувацького досвіду.

Список літератури

1. Wan, J. et al. “Wearable IoT enabled real-time health monitoring system”. *J Wireless Com Network.* 298 (2018). 2018. https://doi.org/10.1186/s13638-018-1308-x
2. Patil, S. and Pardeshi, S. “Health Monitoring system using IoT”. International *Research Journal of Engineering and Technology (IRJET).* Vol. 5, No. 4, pp.1678-1682. April 2018. https://www.irjet.net/archives/V5/i4/IRJET-V5I4375.pdf
3. Rose, K. et al. “The Internet of Things (IoT): An Overview”. *The Internet Society.* 15 October 2015. https://www.internetsociety.org/resources/doc/2015/iot-overview/
4. Amru, M. et al. “IoT-based Health Monitoring System with Medicine Remainder using Raspberry Pi”. *IOP Conf. Ser.: Mater. Sci. Eng.* 981 042081. December 2020. http://dx.doi.org/10.1088/1757-899X/981/4/042081
5. Gay, V. and Leijdekkers, P. “A health monitoring system using smart phones and wearable sensors”. *International Journal of ARM.* Vol. 8, No. 2. June 2007. https://www.researchgate.net/publication/228343525_A_health_monitoring_system_using_smart_phones_and_wearable_sensors
6. Azure API for FHIR Documentation. [Electronic resource]. https://docs.microsoft.com/en-us/azure/healthcare-apis/fhir/overview
7. HealthKit. Apple Documentation. [Electronic resource]. https://developer.apple.com/documentation/healthkit
8. Samsung Health. Samsung Developers. [Electronic resource]. https://developer.samsung.com/health/android/overview.html
9. Google Fit. Platform Overview. [Electronic resource]. https://developers.google.com/fit/overview

10. “Attribute-Driven Design: Create Software Architectures Using Architecturally Significant Requirements”. *Software Engineering Institute.* Fact Sheet. February 2018. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513813

11. O'Brien, L. et al. “Quality Attributes and Service-Oriented Architectures”. *Software Engineering Institute, Carnegie Mellon University.* September 2005. https://www.researchgate.net/publication/200086155_Quality_Attributes_and_Service-Oriented_Architectures

12. CloudHealth Tech. “A Cloud Services Comparison of the Top Three IaaS Providers”. Web. 22 December 2020. https://www.cloudhealthtech.com/blog/cloud-services-comparison

13. Azure Logic Apps documentation. [Electronic resource]. https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview

14. Kanjilal, J. "How Azure Logic Apps works and when to choose it". *Web.* 29 April 2019. https://searchapparchitecture.techtarget.com/tip/How-Azure-Logic-Apps-works-and-when-to-choose-it

15. Azure Functions documentation. [Electronic resource]. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview

16. Understand data store models. Azure Application Architecture Guide. [Electronic resource]. https://docs.microsoft.com/en-us/azure/architecture/guide/technology-choices/data-store-overview

17. PyPDF2 Documentation. [Electronic resource]. https://pythonhosted.org/PyPDF2/

18. pdfplumber. Source Code Repository. [Electronic resource]. https://github.com/jsvine/pdfplumber

19. Aspin, A. “SQL For Cosmos DB – Handling Complex JSON Structures”. *Redgate Hub.* Web. 13 May 2019. https://www.red-gate.com/simple-talk/sql/nosql-databases/sql-for-cosmos-db-handling-complex-json-structures/

20. pandas documentation. Development. [Electronic resource]. https://pandas.pydata.org/docs/development/index.html#development

21. Wijaja, C. Yu. “What it takes to be correlated”. *Towards Data Science.* Web. 26 February 2020. https://towardsdatascience.com/what-it-takes-to-be-correlated-ce41ad0d8d7f

22. Dash Python User Guide. [Electronic resource]. https://dash.plotly.com/

Додаток Б. Програмний код підсистеми обробки даних

Azure Function PdfToJson

```
import io
import json
from datetime import datetime
import logging
import azure.functions as func

import PyPDF2
import pdfplumber

import re

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    file_content = io.BytesIO(req.get_body())
    output_doc = {
        "clinic": req.headers.get('sender'),
        "document": parse_pdf(file_content)
    }
    return func.HttpResponse(
        json.dumps(output_doc),
        mimetype="application/json",
    )

def parse_pdf(file_content):
    meta_reader = PyPDF2.PdfFileReader(file_content)
    metadata = meta_reader.documentInfo
    pages_count = meta_reader.getNumPages()

    pages = {}
    document_text = ""
    with pdfplumber.open(file_content) as document_pdf:
        for i in range(pages_count):
            current_page = document_pdf.pages[i]
            page_text = current_page.extract_text()
            pages[i] = page_text
            document_text += page_text

    return {
        "date": extract_date(document_text, metadata),
        "results": extract_results(document_text),
        "metadata": metadata,
        "pages": pages
    }

def extract_date(document_text, metadata):
    document_date = ""
```

```
    date_regex1 = ".*? ([0-3]?[0-9])[\-\./]([01]?[0-9])[\-\./](2?0?[0-9]{2}) "
    meta_date_key = "/CreationDate"

    if re.match(date_regex1, document_text, flags=re.DOTALL):
        search_result = re.search(date_regex1, document_text, flags=re.DOTALL)
        document_date = f"{search_result.group(3)}-{search_result.group(2)}-
{search_result.group(1)}"
    elif meta_date_key in metadata.keys():
        metadata_date = metadata[meta_date_key].replace("'","")
        document_date = format(datetime.strp-
time(metadata_date, '%y%m%d%H%M%S%f%z'))
    else:
        document_date = format(datetime.datetime.now())

    return document_date

def extract_results(document_text):
    test_results = {}
    match_result = {}
    open_match = False

    num_regex = "(^[0-9.,]+?) (.*)"

    for line_text in document_text.split('\n'):
        if re.match(num_regex, line_text):
            open_match = True
            match_result = {
                "number": float(re.search(num_regex, line_text).group(1)),
                "comment": re.search(num_regex, line_text).group(2).strip()
            }
        elif open_match:
            test_results[line_text] = match_result
            open_match = False
            match_result = {}

    return test_results

def extract_lines(document_text):
    lines = []
    for line_text in document_text.split('\n'):
        lines.append(line_text)

    return lines
```

Azure Function CosmosToSql

```
import os
import datetime
import logging
import dateutil.parser

import azure.functions as func

import pyodbc
from azure.storage.table import TableService, Entity
from azure.cosmos import CosmosClient



def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    table_service = TableService(connection_string=os.environ['CUSTOMCONNSTR_STOR-
AGE_ACCOUNT'])
    cosmos_client = CosmosClient.from_connection_string(os.environ['CUS-
TOMCONNSTR_COSMOSDB'])

    sourceparameters = table_service._query_entities('sourceparameter2')
    for parameter in sourceparameters:
        last_conversion_datetime = parameter['last_conversion_datetime']
        parameter['last_conversion_datetime'] = datetime.now().strftime("%Y-%m-
%dT%H:%M:%S.%f")
        table_service.update_entity('sourceparameter2', parameter)

        cosmos_database = cosmos_client.get_database_client(parameter['data-
base_name'])
        cosmos_container = cosmos_database.get_container_client(parameter['con-
tainer_name'])

        cosmos_query = f"""
            SELECT
                c.userId,
                {parameter['timestamp_path']} AS timestamp,
                {parameter['payload_path']} AS payload
            FROM {parameter['collection_path']}
            WHERE {parameter['collection_filter']}
              AND c.id > '{last_conversion_datetime}'
        """

        items = list(cosmos_container.query_items(
            query=cosmos_query,
            enable_cross_partition_query=True
        ))
```

```
        sql_query_insert = f"INSERT INTO [dbo].[{ta-
ble_name}] ([userId],[timestamp],[payload]) """

        sql_query = sql_query_insert
        counter = 0
        for item in items:
            if counter < 999:
                counter += 1
            else:
                sql_cursor.execute(sql_query.rstrip(','))
                sql_query = sql_query_insert
                counter = 1
            sql_query += f"({item['userId']}, '{dateutil.par-
ser.parse(item['timestamp'])}', {item['payload']}),"

        sql_cursor.execute(sql_query.rstrip(','))

        sql_con.commit()


    return func.HttpResponse(f"Hello, {name}. This HTTP triggered function exe-
cuted successfully.")
```

Додаток В. Програмний код підсистеми аналізу та візуалізації

Dash app

```
import os
import time
import requests
import dash
from dash.dependencies import Input, Output
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import pandas as pd
from azure.storage.table import TableService
from azure.cosmos import CosmosClient
import pyodbc

UserId = '100001'

arch_model = os.environ['ARCHITECTURE_MODEL']

table_service = TableService(connection_string=os.environ['CUSTOMCONNSTR_STOR-
AGE_ACCOUNT'])
parameters = pd.DataFrame(table_service._query_entities('sourceparameter')).set_in-
dex('RowKey')

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.YETI])
app.layout = html.Div([
    html.Button('Update Data', id='update-input-but-
ton', n_clicks=0, style={'float': 'right'}),
    html.Br(),
    html.H1(f'Health Review ({arch_model})', style={'textAlign': 'center'}),
    dcc.Dropdown(
        id='data-input-dropdown-1',
        options=[{'label': parameters.loc[i]['parameter_name'], 'val-
ue': i} for i in parameters.index],
        value=parameters[parameters['default_view']==1].index[0]
    ),
    dcc.Dropdown(
        id='data-input-dropdown-2',
        options=[{'label': parameters.loc[i]['parameter_name'], 'val-
ue': i} for i in parameters.index],
        value=parameters[parameters['default_view']==1].index[1]
    ),
    html.Br(),
    dcc.RadioItems(id='corr-input-radio',
        options=[
            {'label': 'Pearson', 'value': 'pearson'},
```

```
            {'label': 'Kendall', 'value': 'kendall'},
            {'label': 'Spearman', 'value': 'spearman'}
        ],
        value='pearson'
    ),
    html.Div(id='corr-output-textbox'),
    html.Div(id='time-output-textbox'),
    dcc.Graph(id='data-output-graph')

], style={'width': '500'})

@app.callback(
    Output(component_id='data-output-graph', component_property='figure'),
    Output(component_id='corr-output-textbox', component_property='children'),
    Output(component_id='time-output-textbox', component_property='children'),
    Input(component_id='update-input-button', component_property='n_clicks'),
    Input(component_id='data-input-dropdown-1', component_property='value'),
    Input(component_id='data-input-dropdown-2', component_property='value'),
    Input(component_id='corr-input-radio', component_property='value')
)
def update_graph(n_clicks, parameter_id1, parameter_id2, corr_method):
    if n_clicks > 0:
        request_googlefit_update()
        n_clicks = 0

    processing_start = time.time()

    # Retrieving data
    retrieval_start = time.time()
    timeseries1 = retrieve_data(parameters.loc[parameter_id1])
    timeseries2 = retrieve_data(parameters.loc[parameter_id2])
    retrieval_time = time.time() - retrieval_start

    # Presenting data
    plotting_start = time.time()
    fig = make_subplots(specs=[[{"secondary_y": True}]])
    fig.add_trace(
        go.Scatter(x=timeseries1['timestamp'], y=timeseries1['payload'], name=pa-
rameters.loc[parameter_id1]['parameter_name']),
        secondary_y=False,
    )
    fig.update_yaxes(
        title_text=parameters.loc[parameter_id1]['parameter_name'],
        secondary_y=False)
    fig.add_trace(
        go.Scatter(x=timeseries2['timestamp'], y=timeseries2['payload'], name=pa-
rameters.loc[parameter_id2]['parameter_name']),
        secondary_y=True,
    )
    fig.update_yaxes(
```

```
        title_text=parameters.loc[parameter_id2]['parameter_name'],
        secondary_y=True)
    fig.update_xaxes(title_text='Timeline')
    plotting_time = time.time() - plotting_start

    # Calculating correlation
    correlation_start = time.time()
    timeseries1['timestamp'] = pd.to_datetime(timeseries1['timestamp'])
    timeseries2['timestamp'] = pd.to_datetime(timeseries2['timestamp'])
    timeseries1 = timeseries1.drop_duplicates('timestamp').set_index('timestamp')
    timeseries2 = timeseries2.drop_duplicates('timestamp').set_in-
dex('timestamp').reindex(timeseries1.index, method='nearest')
    correlation_value = timeseries1.corrwith(timeseries2, method=corr_method)['pay-
load']
    correlation_time = time.time() - correlation_start

    # Capturing metrics
    processing_time = time.time() - processing_start
    exec_metrics = {
        "PartitionKey": UserId, "RowKey": format(time.time()),
        "ArchitectureModel": arch_model, "SystemModule": "frontend",
        "DataSet1_Count": timeseries1.shape[0], "Da-
taSet2_Count": timeseries2.shape[0],
        "Duration_Retrieval": retrieval_time, "Duration_Plotting": plotting_time,
        "Duration_Correlation": correlation_time, "CorrelationMethod": corr_method,
        "Duration_Total_Seconds": processing_time
    }
    table_service.insert_entity('executionmetrics', exec_metrics)

    return fig, 'Correlation: {}'.format(correlation_value), 'Pro-
cess time: {}'.format(processing_time)

def retrieve_data(parameter):
    if arch_model == 'COSMOSDB_TABLE_API':
        cosmos_client = CosmosClient.from_connection_string(os.environ['CUS-
TOMCONNSTR_COSMOSDB'])

        cosmos_database = cosmos_client.get_database_client(parameter['data-
base_name'])
        cosmos_container = cosmos_database.get_container_client(parameter['con-
tainer_name'])

        query = f"""
            SELECT
                {parameter['timestamp_path']} AS timestamp,
                {parameter['payload_path']} AS payload
            FROM {parameter['collection_path']}
            WHERE {parameter['collection_filter']}
        """
```

```
        items = list(cosmos_container.query_items(
            query=query,
            enable_cross_partition_query=True
        ))

        df = pd.DataFrame(items).sort_values(by=['timestamp'])

    elif arch_model == 'SQL_SERVER':
        sql_con = pyodbc.connect(os.environ['SQLAZURECONNSTR_SQLDB'])
        sql_query = f"""
            SELECT timestamp, payload
            FROM [dbo].[{parameter['parameter_alias']}]
            WHERE userId = {parameter['PartitionKey']}
        """
        df = pd.read_sql(sql_query, sql_con).sort_values(by=['timestamp'])

    return df

def request_googlefit_update():
    current_epoch = time.time_ns()
    url = os.environ['GOOGLE_FIT_UPDATE_ENDPOINT']
    googlefit_streams = table_service._query_entities('googlefitstream')
    for current_stream in googlefit_streams:
        req_body = {
            "userId": UserId,
            "minStartTimeNs": format(current_stream['maxEndTimeNs']),
            "maxEndTimeNs": format(current_epoch),
            "dataSourceId": current_stream['dataSourceId'],
            "token": os.environ['X-MS-TOKEN-GOOGLE-ACCESS-TOKEN']
        }

        requests.post(url=url, json=req_body)

        updated_stream = {
            "PartitionKey": current_stream['PartitionKey'],
            "RowKey": current_stream['RowKey'],
            "dataSourceId": current_stream['dataSourceId'],
            "maxEndTimeNs": current_epoch
        }
        table_service.update_entity('googlefitstream', updated_stream)

if __name__ == '__main__':
    app.run_server(host="0.0.0.0", port=80)
```