Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Центр Післядипломної Освіти
(повна назва)

Кафедра Програмної Інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження можливостей сучасних оркестраторів завдань, для роботи із задачами обробки великих даних

(тема)

Виконав:
Випускник 2 курсу, групи ІПЗздм-19-1
Зініч О.Є.
(прізвище,ініціали)

Спеціальність 121 Інженерія Програмного Забезпечення
(код і повна назва спеціальності)

Тип програми Освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Керівник доц. Чуприна А.С.
(посада,прізвище)

Допускається до захисту
Зав. кафедри ______________ З.В. Дудар
(підпис) (прізвище, ініціали)

2021р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма
(освітньо-професійна або освітньо-наукова)

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри ____________
(підпис)

« 26 » березня 2021 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Зінічу Олександру Євгеновичу
(прізвище, ім’я, по батькові)

1. Тема роботи «Дослідження можливостей сучасних оркестраторів завдань, для роботи із задачами обробки великих даних» затверджена наказом університету від 26.03.2021 № 34Стз.

2. Термін подання студентом роботи до екзаменаційної комісії «08» травня 2021 р.

3. Вихідні дані до роботи *оркестратори завдань, задачі обробки великих даних*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз предметної галузі, постановка задачі, можливості сучасних оркестраторів завдань, задачі обробки великих даних*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів,

ілюстрацій (слайдів) *мета завдання, обґрунтування доцільності розроблення, постановка задачі, методи і алгоритми, структурно-логічна схема взаємодії даних, опис отриманих результатів, інтерфейс програмної системи, демонстраційні матеріали*

6 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доц. Чуприна А.С.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	25.02.2021	*Виконано*
2	Огляд існуючих методів та алгоритмів	05.04.2021	*Виконано*
3	Дослідження існуючих оркестраторів	06.04.2021	*Виконано*
4	Підготовка пояснювальної записки	07.04.2021	*Виконано*
5	Спецчастина	28.04.2021	*Виконано*
6	Підготовка презентації та доповіді	29.04.2021	*Виконано*
7	Попередній захист	01.05.2021	*Виконано*
8	Нормоконтроль, рецензування	02.05.2021	*Виконано*
9	Занесення диплома в електронний архів	07.05.2021	*Виконано*
10	Допуск до захисту у зав. кафедри	08.05.2021	*Виконано*

Дата видачі завдання ____ ________________________ 2021р.

Студент ____

(підпис)

Керівник роботи ________________ доц. Чуприна А.С.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 59 с., 14 рис., 9 табл., 17 джер.

ОРКЕСТРАЦІЯ ЗАДАЧ, ВЕЛИКІ ДАНІ, ОРКЕСТРАТОРИ, ХМАРНІ ОБЧИСЛЕННЯ, Apache Airflow, Apache Luigi, Amazon Step Function, Azure Data Factory.

Об’єктом дослідження є найпопулярніші оркестратори завдань, такі як Apache Airflow, Apache Luigi, Amazon Step Functions та Azure Data Factory.

Метою роботи є дослідження та аналіз існуючих оркестраторів завдань, для роботи із задачами обробки великих даних, їх порівняння та визначення вимог до оркестраторів задач.

Методи розробки базуються на технологіях хмарних обчислень, у разі Amazon Step Functions та Azure Data Factory та мові Python, для використання із Apache Airflow та Luigi.

В результаті роботи досліджено існуючі рішення для оркестрації задач та побудови пайплайнів обробки даних. Було проведено порівняння існуючих оркестраторів, їх можливостей, їх архітектури та істотних принципів їх роботи.

TASK ORCHESTRATION, BIG DATA, ORCHESTRATORS, CLOUD COMPUTING, Apache Airflow, Apache Luigi, Amazon Step Function, Azure Data Factory.

The study focuses on the most popular task orchestrators, such as Apache Airflow, Apache Luigi, Amazon Step Functions and Azure Data Factory.

The purpose of the work is to study and analyze the existing task orchestrators, to work with the tasks of big data processing, their comparison and to determine the requirements for task orchestrators.

The development methods are based on cloud computing technologies, in the case of Amazon Step Functions and Azure Data Factory and Python, for use with Apache Airflow and Luigi.

As a result, the existing solutions for orchestration of tasks and construction of data processing pipelines are investigated. A comparison of existing orchestrators, their capabilities, their architecture and the essential principles of their work was made.

Я, Зініч Олександр Євгенович, студент групи ІПЗздм-19-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження можливостей сучасних оркестраторів завдань, для роботи із задачами обробки великих даних», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП ... 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ... 11
1.1 Задачі обробки великих даних ... 11
1.2 Оркестратори завдань ... 12
1.3 Опис досліджуваних оркестраторів ... 12
1.3.1 Airflow ... 12
1.3.2 Luigi ... 15
1.3.3 AWS Step Functions ... 19
1.3.4 Azure Data Factory ... 22
2 ПОСТАНОВКА ЗАДАЧІ ... 25
2.1 Опис вимог до рішення ... 25
3 ДОСЛІДЖЕННЯ ПАРАМЕТРІВ ДЛЯ ПОРІВНЯННЯ ... 30
3.1 Вартість використання на день ... 30
3.2 Можливість конфігурації для різних середовищ та способів використання ... 31
3.3 Складність розробки ... 31
3.4 Масштабування ... 32
3.5 Складність розробки у команді ... 32
3.6 Підтримка ... 32
3.7 Складність розгортання ... 33
3.8 Підтримка роботи із операціями обробки потокових даних ... 33
4 ДОСЛІДЖЕННЯ ОРКЕСТРАТОРІВ ЗА ХАРАКТЕРИСТИКАМИ ... 34
4.2 Реалізація пайплайну використовуючи Airflow ... 35
4.2.1 Вартість використання на день ... 36
4.2.2 Можливість конфігурації для різних середовищ та способів використання ... 37
4.2.3 Складність розробки ... 37
4.2.4 Масштабування ... 37
4.2.5 Складність розробки у команді ... 38

4.2.6 Підтримка ... 38
4.2.7 Складність розгортання ... 38
4.2.8 Підтримка роботи із операціями обробки потокових даних ... 39
4.3 Реалізація пайплайну використовуючи Luigi ... 39
4.3.1 Вартість використання на день ... 40
4.3.2 Можливість конфігурації для різних середовищ та способів використання ... 40
4.3.3 Складність розробки ... 41
4.3.4 Масштабування ... 41
4.3.5 Складність розробки у команді ... 41
4.3.6 Підтримка ... 42
4.3.7 Складність розгортання ... 42
4.3.8 Підтримка роботи із операціями обробки потокових даних ... 42
4.4 Реалізація пайплайну використовуючи Step Functions ... 43
4.4.1 Вартість використання на день ... 44
4.4.2 Можливість конфігурації для різних середовищ та способів використання ... 45
4.4.3 Складність розробки ... 45
4.4.4 Масштабування ... 46
4.4.5 Складність розробки у команді ... 46
4.4.6 Підтримка ... 46
4.4.7 Складність розгортання ... 47
4.4.8 Підтримка роботи із операціями обробки потокових даних ... 47
4.5 Реалізація пайплайну використовуючи Azure Data Factory ... 47
4.5.1 Вартість використання на день ... 48
4.5.2 Можливість конфігурації для різних середовищ та способів використання ... 49
4.5.3 Складність розробки ... 49
4.5.4 Масштабування ... 49
4.5.5 Складність розробки у команді ... 50
4.5.6 Підтримка ... 50
4.5.7 Складність розгортання ... 50
4.5.8 Підтримка роботи із операціями обробки потокових даних ... 51

5 ПОРІВНЯННЯ ОРКЕСТРАТОРІВ ЗАДАЧ 52
5.1 Вартість використання на день 52
5.2 Можливість конфігурації для різних середовищ та способів використання 53
5.3 Складність розробки 53
5.4 Масштабування 54
5.5 Складність розробки у команді 55
5.6 Підтримка 56
3.6.7 Складність розгортання 56
5.8 Підтримка роботи із операціями обробки потокових даних 57
5.7 Зведені результати порівняння 58
ВИСНОВКИ 59
ПЕРЕЛІК ПОСИЛАНЬ 60
ДОДАТОК А .... **Error! Bookmark not defined.**
ДОДАТОК Б .... **Error! Bookmark not defined.**
ДОДАТОК В .... **Error! Bookmark not defined.**
ДОДАТОК Г .... **Error! Bookmark not defined.**
ДОДАТОК Д .... **Error! Bookmark not defined.**

ВСТУП

У сучасному світі інформації, яка передається в інтернеті стає усе більше. Проходить все більше фінансових транзакцій та здійснюється кількість покупок в онлайн магазинах. За для таргетінгу реклами під кожного окремого користувача збираються усі його дії, а саме: кліки, прокручування екрану, та переходи за посиланнями. Жодна сучасна компанія, яка хоче розвиватись та розширюватись не може обійтись без збору та аналізу даних з яких буде робити висновки.

Для організації роботи з цим незчисленним об'ємом джерел даних, побудови їх обробки та подальшої автоматизації візуалізацій використовуються спеціальні застосунки, які називаються оркестраторами завдань.

Прикладом стандартного процесу, також званого пайплайном, в обробці даних є наступне. До сховища надходить один чи декілька файлів, які необхідно обробити. Як тільки вони надійшли – необхідно перевірити їх цілісність, запустити програму, яка обробить їх та запишіть результат до спеціалізованого сховища, буде це база даних, інше місце у файловій системі, тощо, після чого перевірити що дані записались коректно, кількість співпадає та не було помилок під час читання чи запису. Далі необхідно перенести початкові файли до архіву та надіслати звіт, на який чекають інші системи чи кінцеві користувачі, наприклад у вигляді електронного листа.

На кожному етапі може статись помилка, яка може не залежати від якості коду, як то блокування у базі даних, непрацюючий сервіс відправки електронних листів, тощо. У цьому разі необхідно розуміти який саме етап має помилку та перезапустити виключно його, або навіть пропустити, бо він не є суттєвим на даний час.

Історично для цього використовували самостійно написані скрипти, на мовах shell, python, тощо, які не мали користувацького інтерфейсу, не мали можливості перезапускати окремі частини самостійно, тощо, що було дуже не зручно для звичайного користувача. Подібні застосунки могли бути написані

самостійно, але це вимагало окремої команди розробки та підтримки, тестування цього застосунку, налаштування та інше. Через усі перелічені фактори більшість компаній не могли собі дозволити створення подібних застосунків.

Зараз існує багато варіантів оркестраторів, як з відкритим кодом, які потрібно розгортувати самостійно, так і у вигляді server-less застосунків із закритим кодом. Серед найпопулярніших можна виділити Apache Airflow, застосунок з відкритим кодом, який існує також і в server-less варіанті, Apache Luigi, розробка компанії Spotify, що також має відкритий код та можливість розробки власних модулів, та власні закриті варіанти від основних постачальників хмарних сервісів – Google Workflow, Amazon Step Functions та Azure Data Factory.

Усі вони надають можливість планування запусків чи запуск від подій, запис логів, взаємодію із різними сервісами та можливість створення складних пайплайнів з різними рівнями залежності, помилко-стійкості та автоматичних перезапусків.

Суттєво їх можна розділити на дві групи: застосунки із відкритим кодом, та застосунки із закритим кодом.

До перших відносяться Apache Airflow та Apache Luigi. Плюсом є незалежність від постачальника хмарних послуг, можливість створення власних розширень, можливості перевизначать деякі компоненти та через це робити більш гнучкі та складні пайплайни. Мінусом є потрібність в експертизі при роботі з ними, бо є більше шансів зробити щось не так, та у варіанті не server-less – потрібність у людини з навичками DevOps.

Серед другого типу є сервіси від Amazon, Google та Azure. Перевагами цих сервісів є підтримка від великої компанії, відсутність необхідності DevOps або самостійного налаштування, підтримки та контролю серверів, та більш сильна інтеграція з іншими сервісами постачальника. Недоліками є залежність від постачальника та менша гнучкість при розробці.

Через це метою цієї роботи є дослідження існуючих рішень у галузі оркестрації завдань задля порівняння кожного з них й вибору найкращого.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Задачі обробки великих даних

Великими даними є такі дані, які неможливо, або дуже складно обрахувати використовуючи потужності тільки однієї обчислювальної машини[1].

Задачі їх обробки можна умовно поділити на два типи:

- Batch – обробка великого масиву даних, які вже є у сховищах, та які потрібно обробити за кінцевий задовольняючий час;
- Streaming – обробка безперервного потоку даних, можливо поєднуючи їх із даними у сховищі, при цьому маючи за характеристику поняття latency та throughput.

Де latency – це час з моменту отримання даних, до моменту їх запису до кінцевого сховища, а throughput – кількість оброблених за одиницю часу. Різні рішення у сфері великих даних надають різне співвідношення цих двох параметрів.

Наприклад найбільш популярний Spark має великий показник latency, бо збирає дані у маленькі пакети, але натомість великий throughput, бо вміє дуже швидко їх обробляти, натомість його конкурент Flink має низький показник latency, бо може обробляти кожний шматочок даних окремо, але через це й нижчій показник throughput.

Якщо узагальнювати, то стандартний процес обробки великих даних можна поділити на наступні етапи:

а) отримання даних;

б) обробка даних;

в) вивантаження отриманих результатів;

г) сповіщення інших систем або кінцевих користувачів.

Але кожна з цих дій повинна мати логування, обробку помилок, можливість перезапуску або іншої реакції у разі певних несправностей.

1.2 Оркестратори завдань

Задля надання усіх функції перелічених вище, а також для стандартизації, уніфікації, простоти розробки, підтримки та наочності для кінцевих користувачів використовуються готові рішення, які називаються оркестраторами завдань[2].

У оркестраторів можна виділити декілька загальних функцій:

- планування запусків;
- створення послідовності запуску завдань (паралельно чи послідовно);
- встановлення залежності від результату виконання попередніх завдань;
- користувацький інтерфейс;
- можливість перезапуску конкретного завдання у разі помилки;
- незалежність оркестратора від завдання, яке виконується (помилка у завданні не завдає шкоди оркеструючій програмі);
- єдине місце з логом виконання завдань.

Авжеж, кожен окремий оркестратор може надавати й інші функції. Так, наприклад, Azure Data Factory має вбудовані механізми роботи із майже усіма компонентами хмарного сервісу Azure, але немає можливості динамічного створення порядку завдань, як це вміє Apache Airflow.

1.3 Опис досліджуваних оркестраторів

1.3.1 Airflow

Airflow – це платформа для програмного створення, планування та моніторингу наборів задач. Спочатку він був створений компанією Airbnb у 2015 році, але зараз є платформою із відкритим кодом[3]. Існує декілька комерційних версій Airflow від компаній, які надають його, як сервіс.

Airflow використовується для створення пайнлайнів, як спрямованих ациклічних графів завдань – DAG (див.рис. 1.1). Планувальник Airflow виконує ваші завдання на масиві виконувачів, дотримуючись зазначених залежностей. Утиліти командного рядка дозволяють швидко виконувати складні операції на DAG. Розширений користувальницький інтерфейс полегшує візуалізацію конвеєрів, що працюють у виробництві, відстеження прогресу та усунення проблем за необхідності[3].

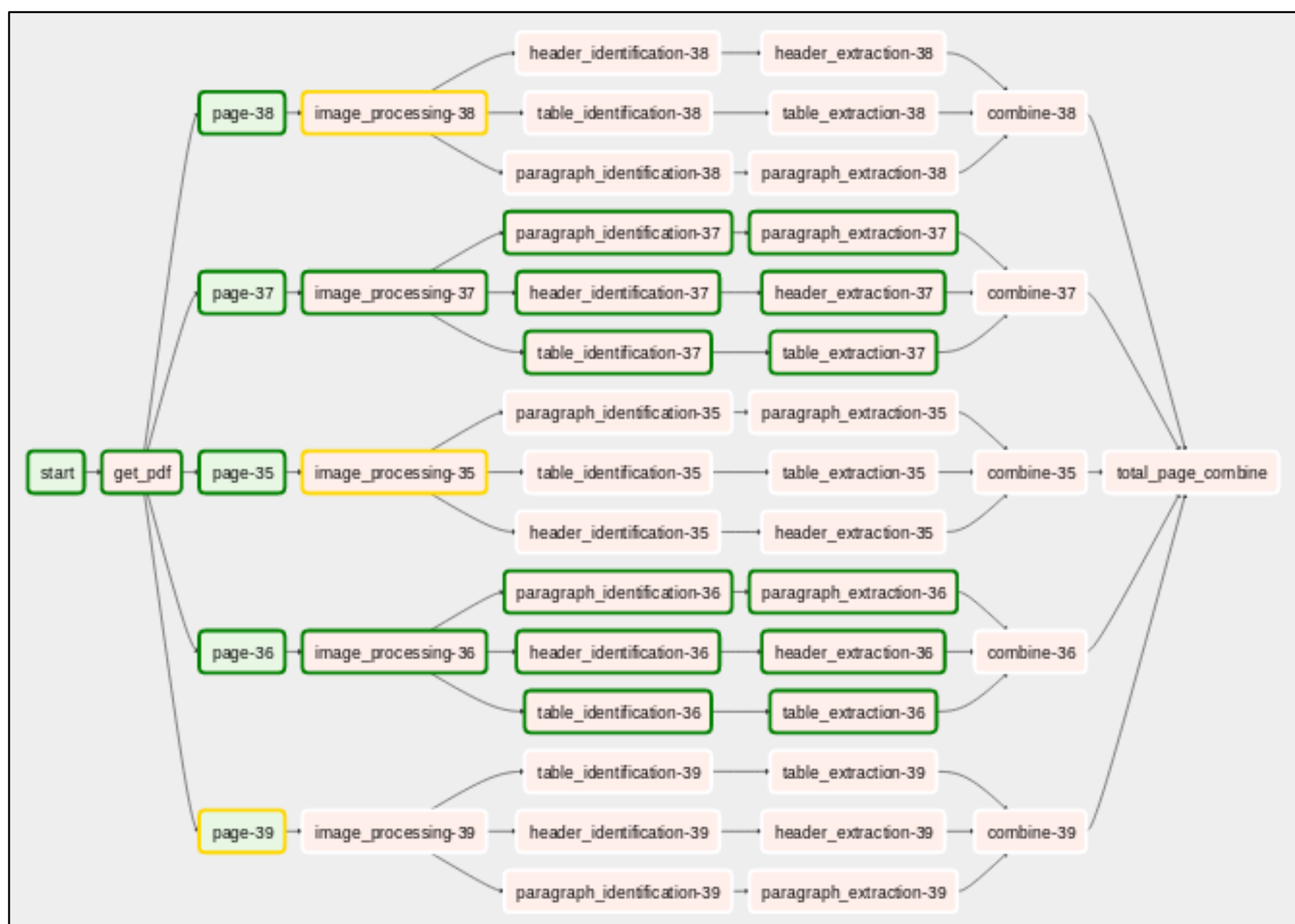

Рисунок 1.1 – приклад DAG у Airlfow[4]

Коли робочі процеси визначаються як код, вони стають більш придатними, перевіреними для підтримки, їх стає легше перевіряти та розробляти спільно.

Основні принципи Airflow це:

– динамічність –пайплайни Airflow мають конфігуруються завдяки коду на мові Python, що дозволяє генерувати динамічні пайплайни. Це дозволяє писати код, який динамічно створює екземпляри пайплайни;

– розширюваність – у Airflow дуже легко визначати власні оператори, виконавці та розширювати бібліотеку так, щоб вона відповідала рівню абстракції, який відповідає вашому оточенню;
– елегантність – пайплайни у Airflow є тонкими та чіткими. Параметризація ваших сценаріїв вбудована в ядро Airflow за допомогою потужного механізму шаблонування Jinja;
– масштабованість – Airflow має модульну архітектуру і використовує чергу повідомлень для організації довільної кількості працівників. Airflow може масштабуватися майже до нескінченності.

Airflow має складну багато сервісну архітектуру (див. рис. 1.2)

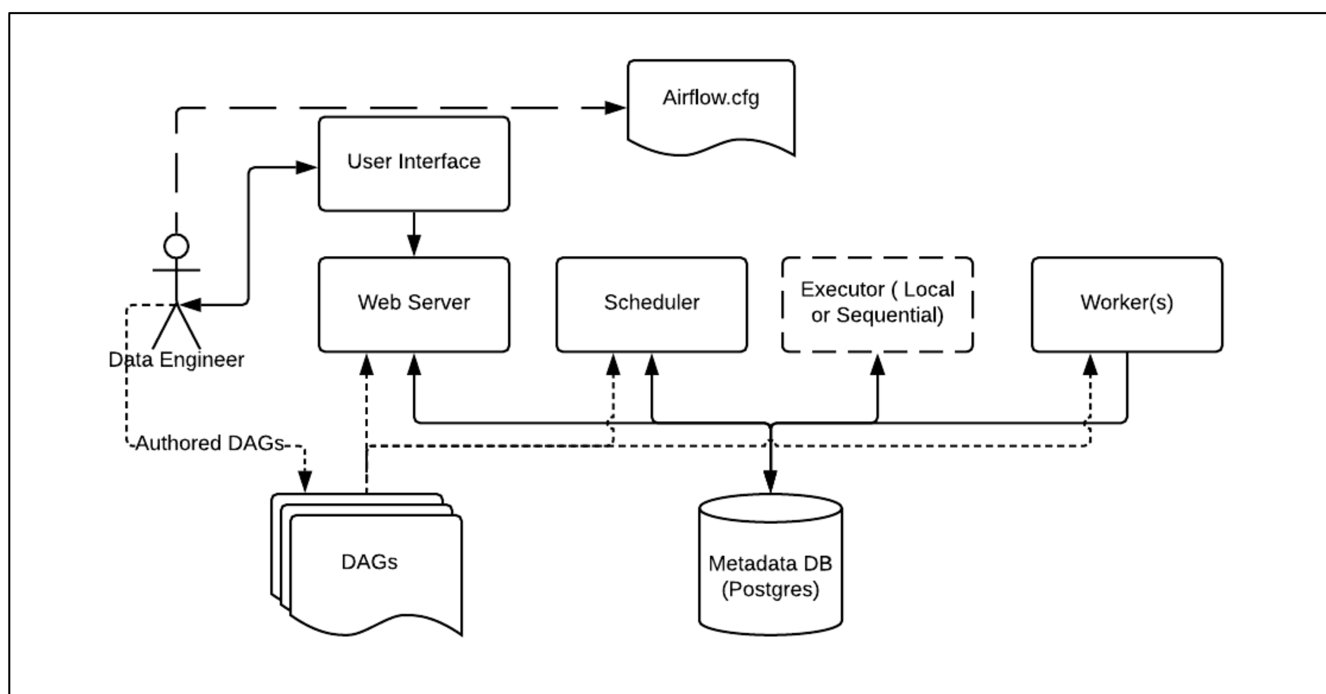

Рисунок 1.2 – архітектура Airflow[5]

Основними складовими Airflow з точки зору архітектури є:
– база даних метаданих – Airflow використовує базу даних SQL для зберігання метаданих про запущені пайплайни. На діаграмі це представлено як Postgres, який надзвичайно популярний у Airflow;
– веб-сервер і Scheduler – Веб-сервер і планувальник Airflow – це окремі процеси, що запускаються (в даному випадку) на локальній машині та взаємодіють із згаданою вище базою даних;

- executor – це зазвичай не окремий процес, а процес запущений у планувальнику;
- worker(и) – це окремі процеси, які також взаємодіють з іншими компонентами архітектури Airflow та сховищем метаданих;
- airflow.cfg – це файл конфігурації Airflow, до якого отримують доступ веб-сервер, планувальник та працівники;
- DAGs – файли що містять код Python, які представляють пайплайни, які будуть запущені Airflow. Розташування цих файлів вказано у файлі конфігурації Airflow, але вони повинні бути доступними веб-серверу, планувальнику та працівникам.

1.3.2 Luigi

Призначення Luigi – вирішити усю координацію, як правило, пов’язану з тривалими пакетними процесами. Ви хочете зв’язати багато завдань, автоматизувати їх, і можете натрапити на збої. Ці завдання можуть бути будь-якими, але це, як правило, тривалі речі, такі як завдання Hadoop, скидання даних у чи з баз даних, запуск алгоритмів машинного навчання або щось інше[6].

Є й інші програмні пакети, які зосереджені на аспектах обробки даних нижчого рівня, наприклад, Hive, Pig чи Cascading. Luigi не є основою для їх заміни. Натомість він допомагає вам поєднати багато завдань, де кожне завдання може бути запитом Hive, завданням Hadoop на Java, завданням Spark у Scala або Python, фрагментом Python, скиданням таблиці з бази даних або будь-чим іншим. Легко створити тривалий конвеєр, який охоплює тисячі завдань і виконується днями або тижнями. Луїджі дбає про більшість з управління робочим процесом, щоб програміст міг зосередитися на самих завданнях та їх залежностях.

Можна створити практично будь-яке завдання, бо Luigi постачається з набором інструментів із декількох типових, найбільш використованих, шаблонів

завдань. Він включає підтримку запуску завдань з картографічного зменшення Python у Hadoop, а також завдань Hive та Pig. Він також постачається з абстракціями файлової системи для HDFS та локальними файлами, які забезпечують атомарність усіх операцій файлової системи. Це важливо, оскільки це означає, що pipeline не буде аварійно працювати в стані, що містить часткові дані.

Сервер Luigi також має веб-інтерфейс (див. рис. 1.3), що дає можливість шукати та фільтрувати серед усіх завдань.

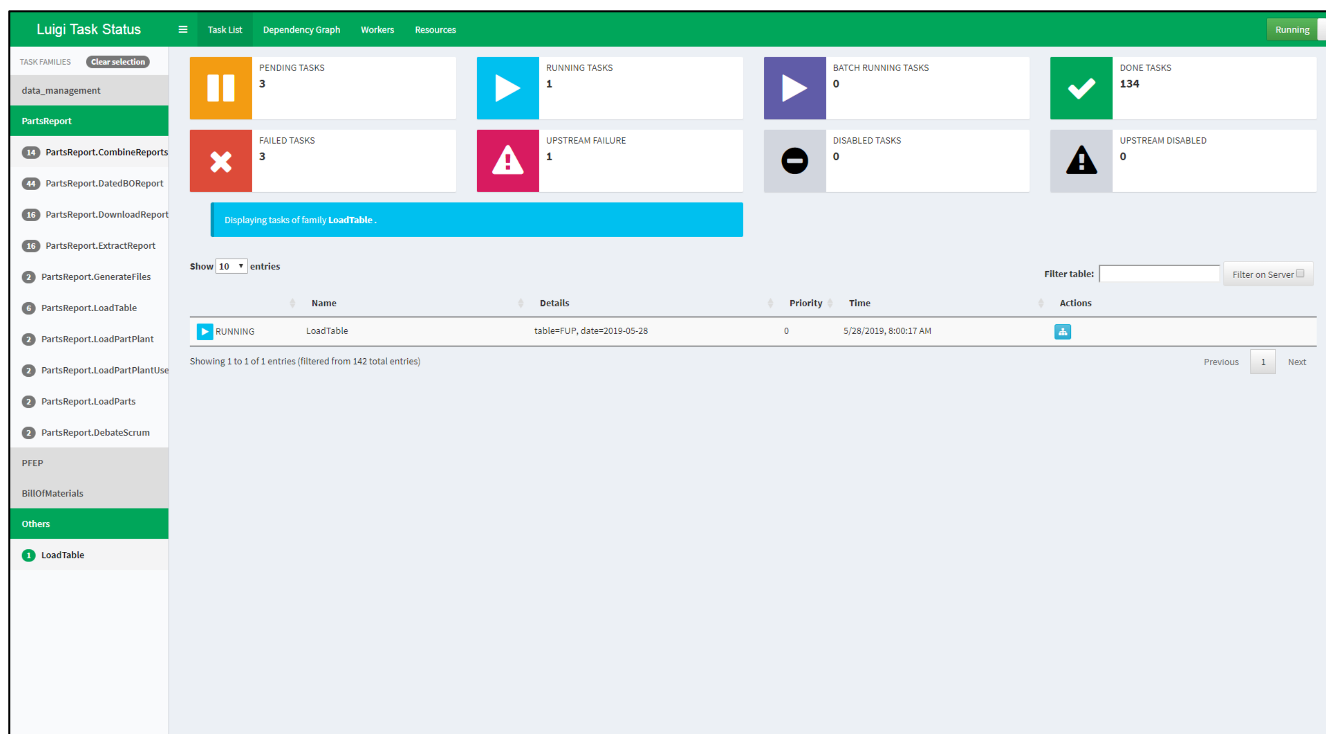

Рисунок 1.3 – веб-інтерфейс Luigi[6]

У прикладі можна побачити використання візуалізатору Луїджі, який дає можливість отримати візуальний огляд графіка залежності робочого процесу. Кожен вузол представляє завдання, яке потрібно виконати. Зелені завдання вже виконані, тоді як жовті завдання ще не виконані. Більшість із цих завдань є завданнями Hadoop, але є також речі, які запускаються локально та створюють файли даних.

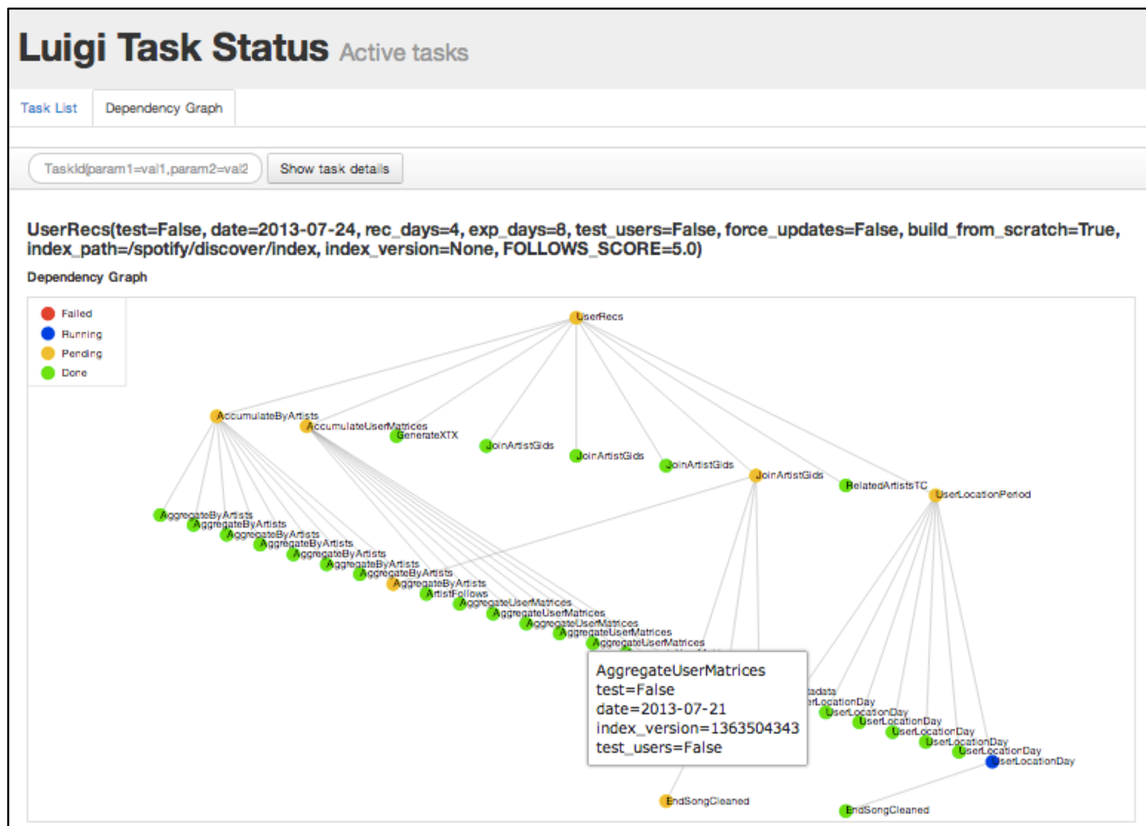

Рисунок 1.4 – граф залежностей завдань Luigi[6]

Концептуально Luigi схожий на GNU Make, де у вас є певні завдання, і ці завдання, в свою чергу, можуть мати залежність від інших завдань. Є також щось подібне до Oozie та Azkaban. Основна відмінність полягає в тому, що Luigi не просто створений спеціально для Hadoop, але його легко розширити за допомогою інших видів завдань (див. рис. 1.4).

Luigi повністю написано на Python. Замість конфігурації XML або подібних зовнішніх файлів даних графік залежностей задається в Python. Це полегшує побудову складних графіків залежностей завдань, де залежності можуть включати алгебру дат або рекурсивні посилання на інші версії того самого завдання. Однак робочий процес може запускати речі, що не належать до Python, наприклад, запускати сценарії Pig або файли scp’ing.

Є два фундаментальні будівельні блоки Луїджі – клас Task та клас Target. Обидва вони є абстрактними класами і очікують впровадження декількох методів. На додаток до цих двох понять, клас Parameter є важливим поняттям, яке визначає, як виконується Task.

Клас Target відповідає файлу на диску, файлу на HDFS або якомусь контрольному пункту, наприклад запису в базі даних. Насправді єдиним методом, який мають реалізовувати Target, є існуючий метод, який повертає True тоді і тільки тоді, коли Target існує.

На практиці реалізація цільових підкласів рідко потрібна. Luigi постачається з набором інструментів з декількох корисних цілей. Зокрема, LocalTarget та HdfsTarget, але є також підтримка інших файлових систем, таких як S3, SSH і так далі. Більшість із цих цілей є файловою системою. Наприклад, LocalTarget та HdfsTarget відображаються у файлі на локальному диску або у файлі HDFS. Крім того, вони також обгортають основні операції, щоб зробити їх атомарними. Luigi постачається з підтримкою Gzip.

Клас Task є трохи більш концептуально цікавим, оскільки саме тут здійснюється обчислення. Task споживає цілі, створені іншим Task. Зазвичай вони також виводять цілі (див. рис. 1.5).

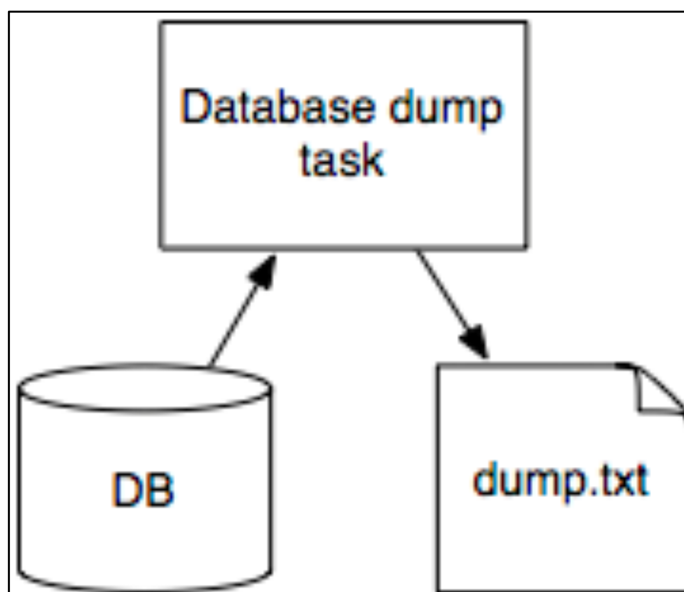

Рисунок 1.5 – Приклад архітектури Task[7]

Є можливість також визначити залежності між Task. Зробити це можна за допомогою методу requires(), у який передати клас Task, від вона якого залежить (див. рис. 1.6).

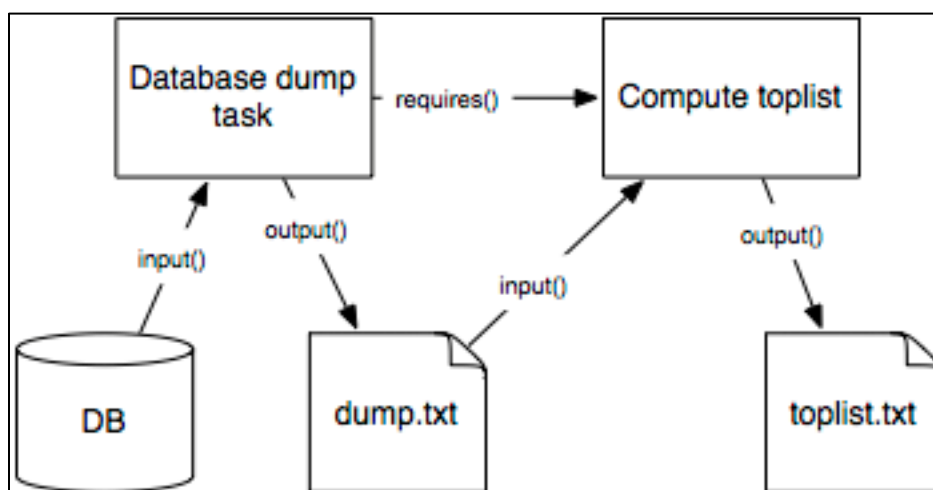

Рисунок 1.6 – приклад залежності однієї Task від іншої[7]

1.3.3 AWS Step Functions

AWS Step Functions – це засіб оркестрації бессерверной функцій, який полегшує створення послідовностей функцій AWS Lambda і різних сервісів AWS для додатків, критично важливих для діяльності компанії. За допомогою візуального інтерфейсу можна створювати і запускати робочі процеси з контрольними точками або управлінням подіями для збереження стану програми. Вихідні дані одного кроку стають вхідними даними наступного кроку. Кожен крок у додатку очікуваним чином виконується в зазначеному порядку згідно заданої бізнес-логікою.

Оркестрації декількох бессерверной додатків, управління повторами і налагодження можуть бути пов'язані з певними труднощами. У міру того як розподілені додатки стають більш складними, управління ними також ускладнюється. Завдяки вбудованим засобам управління Step Functions управляє послідовностями, обробкою помилок, алгоритмом повтору і станом, що дозволяє зняти значну частину операційної навантаження з команди розробників.

Основні переваги Step Function[8]:

- автоматичне масштабування – автоматично масштабує операції та базові обчислення, щоб запустити кроки програми у відповідь на зміну робочого навантаження;
- конфігурація робочого циклу – робочі процеси визначаються як стан-машини, які перетворюють складний код у зрозумілі оператори та схеми;
- вбудовані службові примітиви – Step Function AWS надають готові кроки для вашого робочого циклу, що називаються станами, що реалізують базові примітиви служби. Стани можуть передавати дані в інші стани та мікросервіси, обробляти винятки, додавати час очікування, приймати рішення, паралельно виконувати кілька шляхів тощо;
- інтеграція служб AWS – Step Function дає можливість виклику інших служб AWS. Сюди входять обчислювальні послуги, такі як AWS Lambda, Amazon ECS, Amazon EKS і AWS Fargate, служби баз даних, послуги обміну повідомленнями як то Amazon SNS і Amazon SQS, послуги з обробки даних та аналітики – Amazon Athena, AWS Batch, AWS Glue, Amazon EMR та AWS Glue DataBrew, служби машинного навчання та інше;
- координація розподілених компонентів – Step Function дозволяє координувати будь-яку програму, яка може встановити з'єднання HTTPS, незалежно від того, де вона розміщена;
- повторне використання компонентів – дозволяють координувати існуючі Lambda та дозволяють швидко переоб'єднати їх у нові композиції;
- абстракція робочого процесу – зберігає логіку вашої програми, суворо відокремлену від реалізації вашої програми. Можна додавати, переміщувати, міняти місцями та змінювати порядок кроків без необхідності вносити зміни у бізнес-логіку;
- управління станом – підтримують стан програми під час виконання, включаючи відстеження, на якому етапі її виконання, і зберігання даних, які рухаються між кроками вашого робочого процесу;

- вбудована обробка помилок – автоматично обробляють помилки та винятки за допомогою вбудованих механізмів відловлювання та перезапуску;
- історія кожного виконання – забезпечують діагностику та інформаційні панелі в режимі реального часу, інтегруються з Amazon CloudWatch та AWS CloudTrail, а також реєструють кожне виконання, включаючи загальний стан, невдалі кроки, входи та результати;
- візуальний моніторинг – дозволяє запустити та візуально стежити за виконанням кроків;
- висока доступність – має вбудовану стійкість до несправностей та підтримує сервісну спроможність у декількох зонах доступності в кожному регіоні, щоб захистити додатки від відмов окремих машин або центрів обробки даних;
- безпека – інтегровано із AWS Identity and Access Management (IAM) та рекомендують найменш привілейовану політику IAM для всіх ресурсів, що використовуються у робочому процесі. Підтримує кінцеві точки VPC (VPCE) за допомогою AWS PrivateLink.
- оркестрація великих обсягів – експрес-робочі процеси підтримують швидкість подій більше 100 000 за секунду, що дозволяє створювати великі обсяги короткочасних робочих процесів.

На рисунку 1.7 зображено типову Step Function, яка дістає запис із бази даних, читає його та відправляє повідомлення, після чого повторює доки не прочитає усі.

З цього прикладу можна побачити, що на відміну від Airflow Step Function не є спрямованим графом, що дозволяє робити більш складні послідовності завдань.

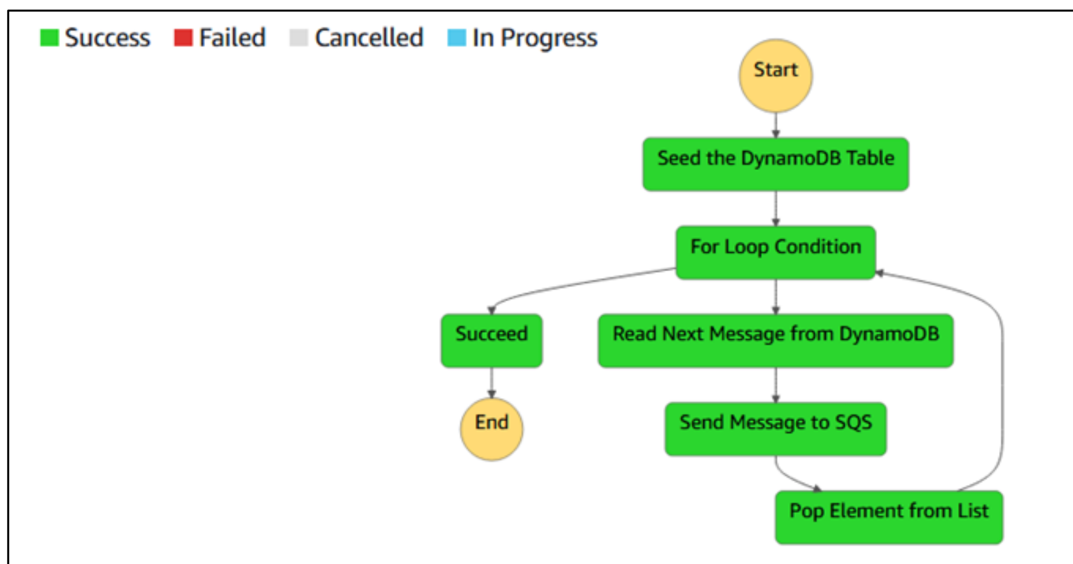

Рисунок 1.7 – приклад Step Function[9]

1.3.4 Azure Data Factory

Фабрика даних Azure – це керована хмарна служба, створена для складних гібридних процесів вилучення, перетворення і завантаження (або вилучення, завантаження і перетворення) і інтеграції даних[10].

Як приклад розглянемо компанію, яка створює хмарні ігри і збирає петабайт інформації у вигляді логів цих ігор. Компанія хоче проаналізувати ці логи, щоб отримати відомості про переваги клієнтів, демографічні параметри і особливості використання. Ці відомості допоможуть зрозуміти, як можна збільшити додаткові і перехресні продажі, розробити нові цікаві функції, стимулювати розвиток компанії і поліпшити якість обслуговування клієнтів.

Щоб проаналізувати ці логи, компанії необхідно використовувати довідкові відомості, наприклад інформацію про клієнтів, грі і маркетингових діях, які зберігаються в локальному сховищі даних. Компанії потрібно об'єднати ці дані з локального сховища даних з додатковими даними журналів, зібраними в хмарному сховище даних.

Щоб отримати аналітичні дані, компанія обробить об'єднані дані за допомогою кластера Spark в хмарі (Azure HDInsight), а потім опублікує перетворені дані в хмарне сховище даних, наприклад Azure Synapse Analytics, з якого можна буде легко отримувати потрібні звіти. Цей робочий процес повинен виконуватися автоматично при приміщенні файлів в контейнер сховища великих бітових об'єктів і повинен щодня відслідковуватися. Крім того, має бути налагоджена щоденне управління ім.

Фабрика даних Azure є ідеальною платформою для таких сценаріїв обробки даних. Це хмарна служба вилучення, перетворення, завантаження і інтеграції даних, яка дозволяє створювати керовані даними робочі процеси для оркестрації і автоматизації масштабних операцій переміщення і перетворення даних. За допомогою фабрики даних Azure можна створювати і включати в розклад керовані даними робочі процеси (пайплайни), які можуть приймати дані з різнорідних сховищ даних, Ви можете створювати складні процеси вилучення, перетворення і завантаження, які перетворюють дані через візуальний інтерфейс потоків даних або служб обчислень , таких як Azure HDInsight Hadoop, Azure Databricks і База даних SQL Azure.

Крім того, ви можете публікувати перетворені дані в сховищах даних (наприклад, Azure Synapse Analytics) для використання додатками бізнес-аналітики. Необроблені дані за допомогою фабрики даних Azure можна організувати в корисні сховища даних і озера даних для прийняття кращих ділових рішень.

Фабрика даних містить серію взаємопов'язаних систем, які надають комплексну платформу для фахівців з інжинірингу даних.

Розглянемо основні складові:

– підключення та збір даних – Azure DF надає можливість підключення до всіх необхідних джерел даних і служб обробки, таким як служби SaaS, бази даних, файлові ресурси із загальним доступом, FTP і веб-служби, і переміщення даних, які потребують подальшій обробці, в централізоване розташування. Також Azure DF надає власні ресурси.

Наприклад, ви можете збирати дані в Azure Data Lake Storage і потім перетворювати ці дані за допомогою служби обчислень Azure Data Lake Analytics. Або ж ви можете зібрати дані в сховищі BLOB-об'єктів Azure і пізніше перетворити їх за допомогою кластера Hadoop під керуванням служби Azure HDInsight;

– перетворення і доповнення даних – Azure DF дозволяє обробляти або перетворювати дані, зібрані в централізованому хмарному сховище, за допомогою потоків зіставлення даних ADF. Потоки даних дозволяють перетворювати дані за допомогою Spark із та без знання самого Spark. Також ADF підтримує зовнішні дії для перетворень в таких службах обчислень, як HDInsight Hadoop, Spark, Data Lake Analytics і Машинне навчання;
– CI / CD і публікація – Azure DF забезпечує повну підтримку CI / CD для конвеєрів даних при використанні Azure DevOps і GitHub. Це дозволяє поступово розробляти і надавати процеси вилучення, перетворення і завантаження перед публікацією готового продукту;
– моніторінг – після створення і розгортання конвеєра інтеграції даних, який витягує корисні дані з оброблених даних, вам знадобиться відстежувати успішне виконання і збої запланованих операцій і конвеєрів. Фабрика даних Azure має вбудовану підтримку моніторингу конвеєрів за допомогою Azure Monitor, API, PowerShell, журналів Azure Monitor і панелей працездатності на порталі Azure.

2 ПОСТАНОВКА ЗАДАЧІ

2.1 Опис вимог до рішення

Метою магістерської атестаційної роботи є аналіз сучасних рішень у галузі оркестрації завдань, таких як Apache Airflow та Apache Luigi, їх архітектури та можливостей. Також слід розглянути хмарні сервіси із оркестрації завдань, такі як Amazon Step Functions та Azure Data Factory, їх можливості, обмеження та способи задання пайплайнів.

Необхідно визначити які є вимоги до оркестраторів задач, тобто яку функціональність вони повинні реалізовувати, як то паралельне виконання завдань, умовні оператори, обробка помилок, логування, тощо.

Після чого потрібно визначити набір характеристик, за якими ми можемо порівняти різні оркестратори завдань та зробили порівняння.

Ці можливості можна розділити на декілька категорій:

а) надавати можливість організовувати задачі у певному порядку та за певними умовами;
б) надавати можливість збору логів;
в) надавати можливість планування запусків;
г) надавати можливість запуску власного коду;
д) передавати різні параметри завданням у залежності від різних обставин;
е) надавати можливість перезапуску частини завдань;
ж) надавати можливість продовжити роботу незалежно від помилок у виконанні задачі.

Далі детальніше про деякі пункти.

Під поняттям «надавати можливість організовувати задачі у певному порядку та за певними умовами» маються на увазі наступні можливості:

– організація виконання задач в не залежності від результату їх виконання;
– організація паралельного виконання задач;
– організація виконання тих чи інших задач у залежності від умов.

Організація виконання задач в не залежності від результату їх виконання – якщо задача завершилася із помилкою – мати можливість продовжити виконання наступної, завершити виконання або виконати іншу задачу.

Для прикладу подивимось на пайплайн зображений на рисунку 2.1

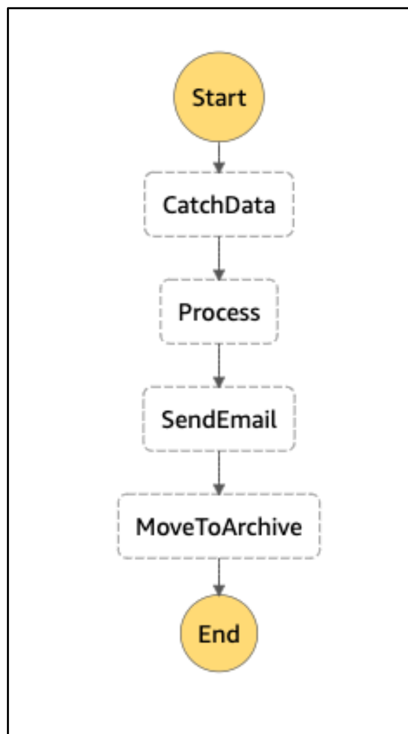

Рисунок 2.1 – Приклад продовження роботи навіть у разі помилки

У цьому прикладі береться файл, з нього вибираються якісь дані, формується новий файл з результатами, який зберігається у сховищі, надсилається електронний лист до користувача та переноситься початковий файл до архіву.

Через те, що операція архівації може зайняти багато часу ми хочемо відправити листа до цього, але якщо сервіс відправки пошти не відповідає, ми все одно маємо архівувати файл. Через це нам необхідно продовжити виконання.

Організація паралельного виконання задач – деякі задачі, які можуть бути навіть непов’язані одна з одною, можуть виконуватись паралельно, але якась наступна задача може очікувати виконання обох. Приклад на рисунку 2.2

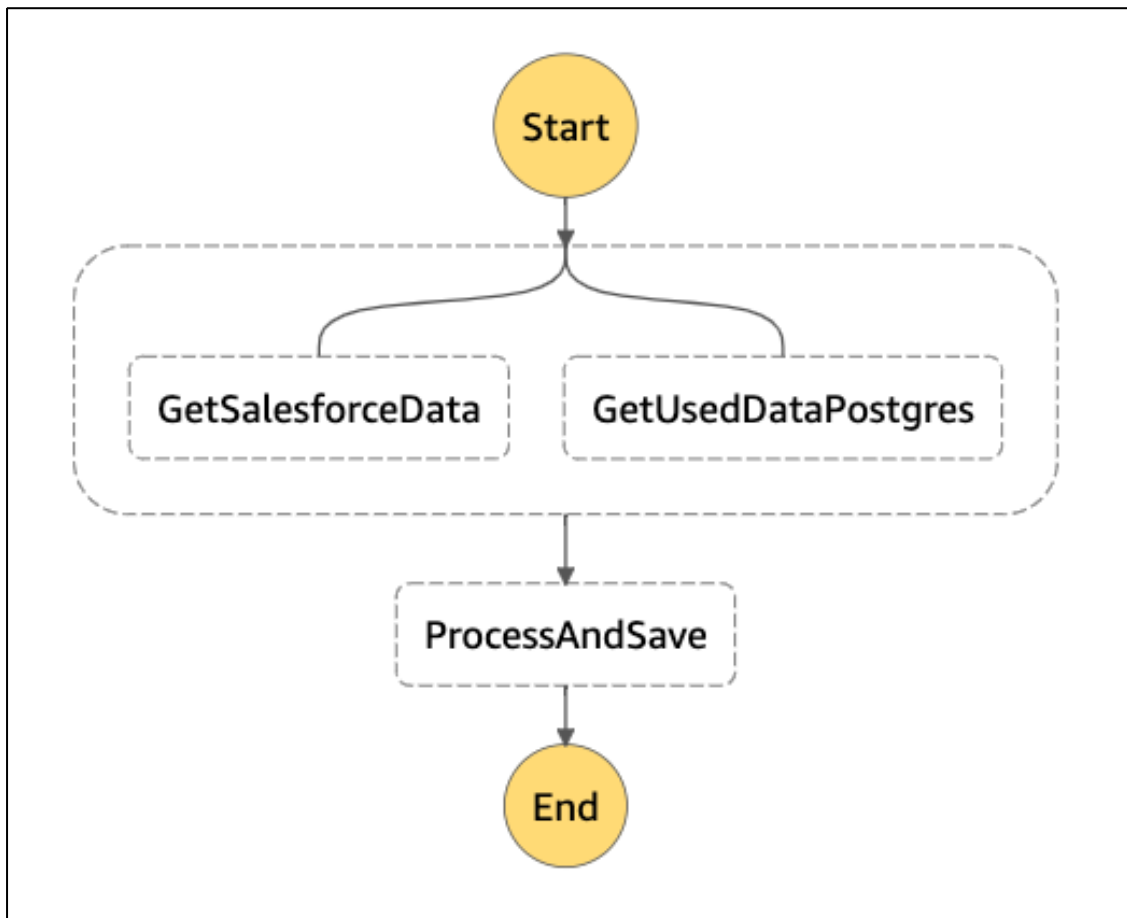

Рисунок 2.2 – Приклад паралельного виконання завдань

На рисунку 2.2 ми можемо побачити, що перше і друге завдання виконуються паралельно.

Перше завдання повинне зібрати дані з сервісу Salesforce та сформувати файл із результатом у сховищі, друге повинне зібрати дані з бази даних, та сформувати файл із результатом у сховищі, а третє, яке йде після їх обох, має сформувати з цих даних якийсь результат та записати його у базу даних.

Організація виконання тих чи інших задач у залежності від умов – деякі задачі повинні виконуватись тільки за якихось обставин. Приклад на рисунку 2.3.

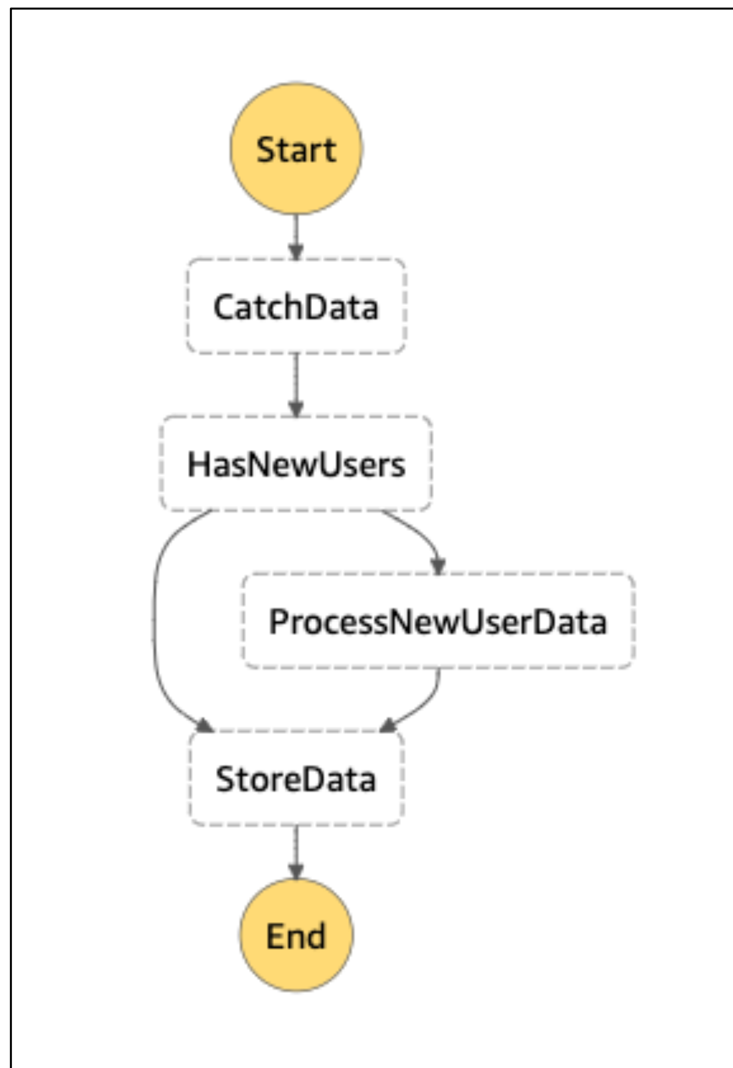

Рисунок 2.3 – Приклад обробки за умовою

На рисунку 2.3 ми можемо побачити, наступний пайплайн.

Надходить файл із даними о діях користувачів. Ми його обробляємо та якщо в нас з’явилися нові користувачі – додаємо їх у базу даних, після чого навіть якщо нових користувачів нема – формуємо звіт та зберігаємо його у сховищі.

Надавати можливість перезапуску частини завдань – звернемось до прикладу із відправкою електронного листа. В ньому ми продовжили роботу без відправки звіту через електрону пошту.

У цьому прикладі ми можливо хотіли би через деякий час самоуруч відправити листа, коли сервіс знов запрацює, не перезапускаючи обробку знову, що може викликати проблеми, о необхідно буде достати файл з архіву, і знов задіяти ресурси кластеру для обробки, яку ми вже зробили.

Надавати можливість продовжити роботу незалежно від помилок у виконанні задачі – цей пункт відрізняється від прикладу із електронним листом тим, що має собою не взаємодію між задачами, а архітектурну особливість.

Оркестратор задач повинен бути відокремлений від ресурсів, які виконують задачі, щоб будь яка помилка, як то OutOfMemory чи StackOverflow, які можуть статися у коді задачі, ніяк не нашкодили виконанню оркестратора, забравши в нього ресурси, чи зламавши програму в цілому.

3 ДОСЛІДЖЕННЯ ПАРАМЕТРІВ ДЛЯ ПОРІВНЯННЯ

У попередньому пункті ми описали вимоги до оркестратора задач, як до програмного продукту, але у випадку, коли ми працюємо із настільки різнорідними сервісами – нам необхідно винайти параметри порівняння, які б враховували кожну із використаних моделей. Для когось проблемою є масштабування, для когось – робота із потоковими даними, і для об’єктивної оцінки це теж необхідно врахувати.

Список параметрів для порівняння:

- вартість використання на день;
- можливість конфігурації для різних середовищ та способів використання;
- складність розробки;
- масштабування;
- складність розробки у команді;
- підтримка;
- складність розгортання;
- підтримка роботи із операціями обробки потокових даних.

Далі, про кожен із пунктів окремо.

3.1 Вартість використання на день

Через те, що AWS Step Functions та Azure Data Factory є сервісами, а Airflow працює у serverless режимі – найкращім індикатором є кількість грошей, які будуть витрачені за день роботи.

3.2 Можливість конфігурації для різних середовищ та способів використання

У кожному проекті є декілька середовищ, у яких повинна виконуватись програма, як то develop, stage, prod, тощо. І для кожного з цих середовищ частина параметрів може відрізнятись. Наприклад адреса бази даних, або шлях до файлу на S3, або список користувачів, яким потрібно відправити електроне повідомлення у разі виникнення помилок. Через це сервіс повинен мати можливість гнучкої конфігурації у залежності від середовища.

Але конфігурація може відрізнятись не тільки за середовищем, але й за обставинами запуску. Наприклад, якщо в нас є алгоритм машинного навчання, який аналізує дані, та визначає який процент може мати помилки, то ми повинні передати цю інформацію подальшим завданням.

3.3 Складність розробки

Цей пункт має у собі як поріг входу для розробки, так і складність реалізації непередбачуваних розробником сервісу речей.

Наприклад, дуже легко розробляти переміщаючи блоки у веб-інтерфейсі, але обробку складних ситуацій у такому разі реалізувати важко. І навпаки, складніше розбиратися із фреймворком на мові програмування Python, але реалізовувати складну поведінку, або інтегрувати якийсь сервіс – набагато простіше.

3.4 Масштабування

Одним із важливих пунктів є можливість масштабування середи для прискорення роботи існуючих пайплайнів, або можливість безболісно додавати нові. Бо який би не був багатофункціональний оркестратор – якщо при використанні у реальному світі він починає мати проблеми із кількістю пайплайнів – його складно назвати гарним оркестратором.

3.5 Складність розробки у команді

Для великих пайплайнів буде виникати необхідність одночасної роботи кількох людей над одним пайплайном, або потрібністю робити зміни одночасно у двох різних гілках системи контролю версій, доки минулі зміни ще перевіряються командою. Тому в залежності від того як задається цей пайплайн буде легше чи складніше вирішувати конфлікти у коді.

3.6 Підтримка

В залежності від популярності сервісу та активності його спільноти, для звичайного розробника буде легше чи складніше вирішувати проблеми, знаходити інформацію у статтях та отримувати відповідь на спеціалізованих ресурсах.

3.7 Складність розгортання

Під складністю розгортання можна розуміти, як складність доставки коду із репозиторію, так і залежність від інших сервісів та можливість автономної роботи.

3.8 Підтримка роботи із операціями обробки потокових даних

Одним із найпопулярніших напрямків у сучасній обробці великих даних є робота із потоковими даними. Тому для оркестраторів задач дуже важливо мати змогу підтримувати у собі операцію роботи із потоковими даними.

4 ДОСЛІДЖЕННЯ ОРКЕСТРАТОРІВ ЗА ХАРАКТЕРИСТИКАМИ

4.1 Опис стандартного пайплайну

Для порівняння можливостей та характеристик сервісів оркестрації завдань у кожному з них було розроблено стандартний пайплайн обробки даних.

Цей пайплайн враховує наступні можливості:

– запуск в залежності від події – у даному випадку від появи файлу у директорії;
– оператор умовних операцій – оператор, який змінює поведінку пайплайну, вибираючи ту чи іншу гілку обчислень, у залежності від якихось подій;
– відправку електронного листа – можливість відправляти електронного листа із змінним змістом, в залежності від результатів роботи попередніх операторів;
– оператори роботи із даними – можливість завантаження та операцій над даними.

Враховуючи основні тенденції у обробці великих даних та стандартні задачі було розроблено наступний пайплайн:

– приходить файл до сховища;
– пайплайн запускається;
– файл переноситься до директорії процесінгу;
– дані з файлу записуються в базу даних;
– якщо є нові користувачі, то дані про них записуються в ще одне сховище;
– файл переноситься в архів;
– відправляється електронного листа про успішне закінчення обробки;
– в разі помилки – відправляться електронного листа про помилку і файл переноситься назад до директорії входу.

Далі розглянемо реалізацію даного пайплайну використовуючи сервіси описані у першому розділі.

4.2 Реалізація пайплайну використовуючи Airflow

Пайплайн із пункту 4.1 було розроблено використовуючи засоби Airflow. Приклад графу обробки наведено на рисунку 4.1.

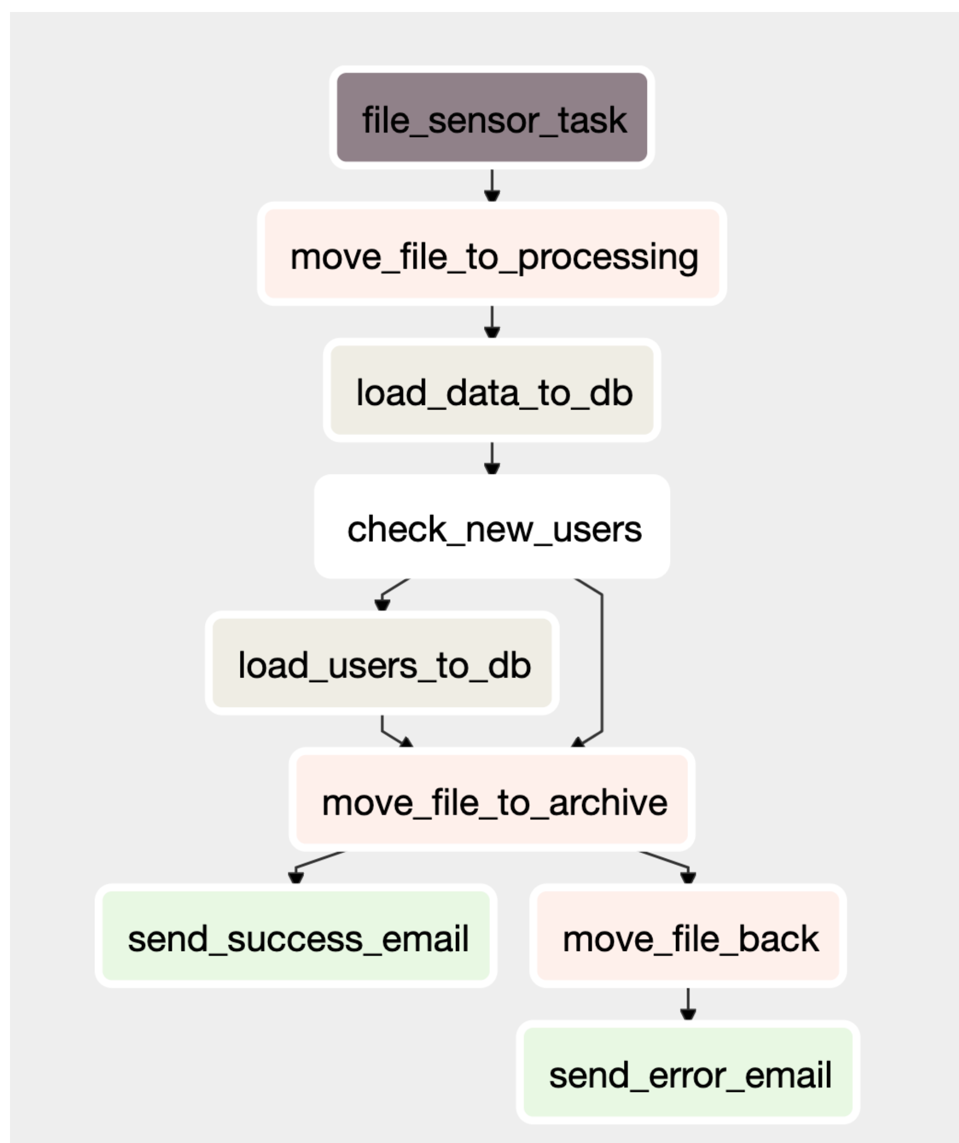

Рисунок 4.1 – граф оркестратору Airflow для реалізації пайплайну із пункту 3.1

Цей пайплайн складається із наступних складових:

- file sensor operator – оператор, який спостерігає за зазначеною директорією та завершується успішно у разі появи файлу;
- file operator – оператор, який переміщує файл між директоріями. Використовується для переміщення файлу до місця обробки, до архіву, та у разі поломки – назад у вхідну директорії;
- database operator – оператор, який завантажує дані із файлу до бази даних, та/або виконує додаткові запити. У даному прикладі завантажує дані із файлу до бази даних, відзначає чи є нові користувачі та завантажує дані про цих користувачів;
- branch operator – оператор, який за умовою визначає який оператор буде виконано наступним. У даному прикладі у разі наявності нових користувачів запускає наступним оператор для завантаження даних про нових користувачів;
- email operator – оператор, який відправляє електронного листа.

Далі ми пройдемо по кожної характеристиці із пункту 3.1.

4.2.1 Вартість використання на день

У нашому випадку ми запускали Airflow у середовищі провайдеру хмарних обчислень Amazon Web Services, завдяки serverless сервісу MWAA (Managed Workflows for Apache Airflow)[11].

Ми використовували small setup, що складається із одного серверу для планувальника (scheduler), одного серверу для виконавця (worker), та половини серверу для веб-інтерфейсу (web-server).

Офіційна ціна – 11.76 доларів на день, але для повноцінної роботи нам необхідно також розгорнути базу даних, сервіс сховища та сервіс відправки електронних листів, що коштуватиме нам ще 0.01 долар на день за сховище, 1.78

долару на день за базу даних та 0.6 долару за 100 000 надісланих електронних листів. У сумі ми маємо 14 доларів на день.

4.2.2 Можливість конфігурації для різних середовищ та способів використання

Завдяки вбудованому механізму роботи із jinja – інструменту роботи із шаблонами, airflow має дуже багаті можливості для конфігурації. Також він має можливість змінювати частину параметрів із користувацького інтерфейсу. Через це ми маємо можливість дуже гнучкої конфігурації Airflow.

4.2.3 Складність розробки

Розробка застосувань Airflow здійснюється завдяки мові Python. Airflow має у собі багатий набір вбудованих операторів, через що велику кількість дій можна здійснити тільки імпортуючи потрібні оператори та додаючи необхідну конфігурацію.

4.2.4 Масштабування

Основною проблемою у масштабуванні Airflow є обмеження бази даних, яка не може, на відміну від кількості виконавців, масштабуватися.

Єдина можливість – це внутрішня оптимізація бази даних, через індекси та партиції.

4.2.5 Складність розробки у команді

Як сказано у попередньому пункті – пайплайни у Airflow будуються використовуючи мову Python, тому усі сучасні можливості систем контролю версій доступні при роботі із Airflow.

4.2.6 Підтримка

Airflow є проектом Apache номер один – набор проектів, які мають найбільшу увагу від користувачів та компаній. Також Airflow має більше 20 000 зірочок у GitHub[12] та 5800 питань у StackOverflow[13], що також є характеристикою популярності цього оркестратору.

4.2.7 Складність розгортання

Через використання Airflow, як сервісу з бібліотеки AWS, розробнику необхідно тільки завантажити файли із кодом пайплайнів та файл requirements.txt за вказаним шляхом на S3.

Для автоматизації цих процесів може використовуватись як вбудовані механізми pipelines із code commit (сервіс AWS для репозиторіїв), або інші CI інструменти.

4.2.8 Підтримка роботи із операціями обробки потокових даних

Airflow більше спрямовано на роботу із пакетними операціями, але через можливість розширювати Airflow своїм кодом – є можливість створення оператору для роботи із потоковими даними.

4.3 Реалізація пайплайну використовуючи Luigi

Пайплайн із пункту 4.1 було розроблено використовуючи засоби Luigi. Приклад графу обробки наведено на рисунку 4.2.

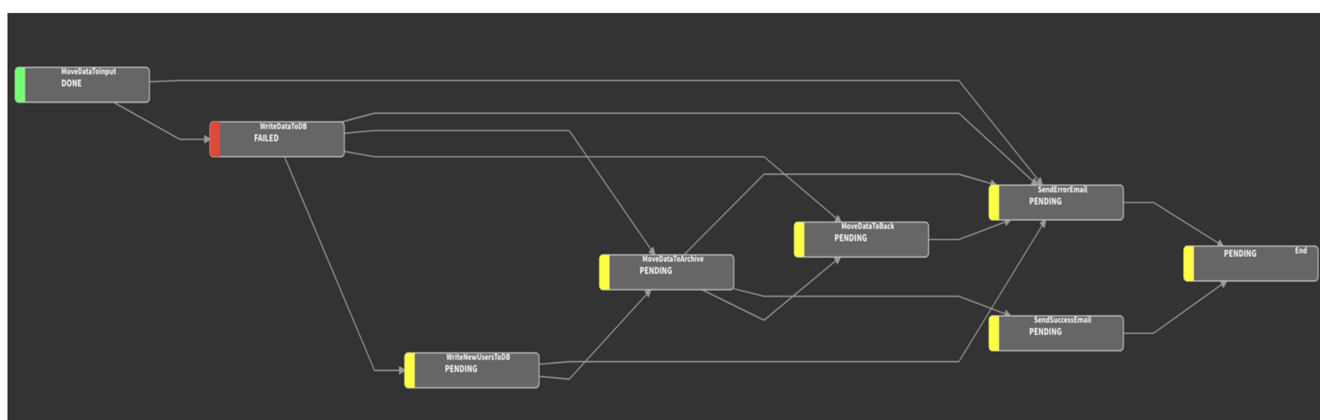

Рисунок 4.2 – граф оркестратору Luigi для реалізації пайплайну із пункту 3.2

Через те, що, у відмінності від Airflow, кожен оператор у Luigi необхідно писати самостійно – важко виділити особливі типи цих операторів. Але потрібно зазначити, що для правильної роботи цього пайплайну необхідно було додати останню, пусту задачу, від якої вже будувався граф залежностей. Ще, Luigi немає вбудованого механізму переходу у залежності від станів, тому для кожного оператору необхідно було самостійно обгортати увесь код у try catch, щоб потім змінити наступну гілку виконання. Для комфортної роботи було виділено декілька класів, але у архітектурі Luigi, навіть при наявності усього коду,

неможливо змінити поведінку тільки параметрами. Завжди потрібно хоча б один раз у коді зробити екземпляр класу. Далі розглянемо усі порівняльні характеристики.

4.3.1 Вартість використання на день

Через не дуже велику популярність, Luigi немає сервісних реалізацій від постачальників хмарних обчислень.

Тому для запуску Luigi ми використали сервіс Amazon EC2, який надає сервери для вільного використання. Офіційна ціна – 4.15 доларів на день.

Також нам необхідно розгорнути базу даних, сервіс сховища та сервіс відправки електронних листів, що коштуватиме нам ще 0.01 долар на день за сховище, 1.78 долару на день за базу даних та 0.6 долару за 100 000 надісланих електронних листів.

У сумі ми маємо 6.54 доларів на день.

4.3.2 Можливість конфігурації для різних середовищ та способів використання

Luigi немає вбудованих механізмів шаблонизації, але через дуже відкриту архітектуру дозволяє інтегрувати будь які методи.

Також він має механізм зміни поведінки в залежності від переданих параметрів, що також дозволяє гнучко налаштовувати поведінку пайплайну.

4.3.3 Складність розробки

Розробка застосувань Luigi здійснюється завдяки мові Python. Luigi має у собі невеликий набір вбудованих операторів.

Але навіть вони потребують реалізації у вигляді наслідних класів, через що навіть для створення простого пайплайну, із використанням тільки вбудованих можливостей, необхідно буде написати велику кількість коду, що призводить до також великого об’єму багів та необхідності тестування.

4.3.4 Масштабування

На відміну від Airflow, Luigi працює тільки на одному сервері, через що усі можливості по масштабуванню можуть бути здійснені лише завдяки збільшенню потужності виконавчого сервера.

Цей факт робить Luigi дуже вразливим до перебоїв.

4.3.5 Складність розробки у команді

Пайплайни у Luigi будуються використовуючи мову Python, тому усі сучасні можливості систем контролю версій доступні при роботі із Luigi.

Але варто зазначити, що через необхідність робити дублікати коду – складність розробки зростає.

4.3.6 Підтримка

Luigi є продуктом Spotify, який передано до Apache, що дає йому сталості, але він не є пріоритетним продуктом для індустрії. Наразі Luigi має 14 000 зірочок у git hub, та тільки 313 запитань на Stack Overflow, що свідчить про не дуже велику популярність цього продукту.

4.3.7 Складність розгортання

Luigi є класичним сервісом написаним на мові Python, тому вам прийдеться зіткнутися з усіма проблемами цього підходу. Від проблем із віртуальними середовищами, до проблем із оновленням і масштабуванням. Але те, що це звичайний пакет Python, дає можливість використовувати велику купу існуючих інструментів CI/CD та навіть розробляти їх самостійно.

4.3.8 Підтримка роботи із операціями обробки потокових даних

Через свою архітектурну спрямованість на роботи із пакетною обробкою даних у Luigi дуже складно зробити обробку потокових даних, але й можливо.

Це дасть велику кількість проблем, і відсутність деякої частини функціоналу, але це можливо.

4.4 Реалізація пайплайну використовуючи Step Functions

Пайплайн із пункту 4.1 було розроблено використовуючи засоби Amazon Step Functions. Приклад графу обробки наведено на рисунку 4.3.

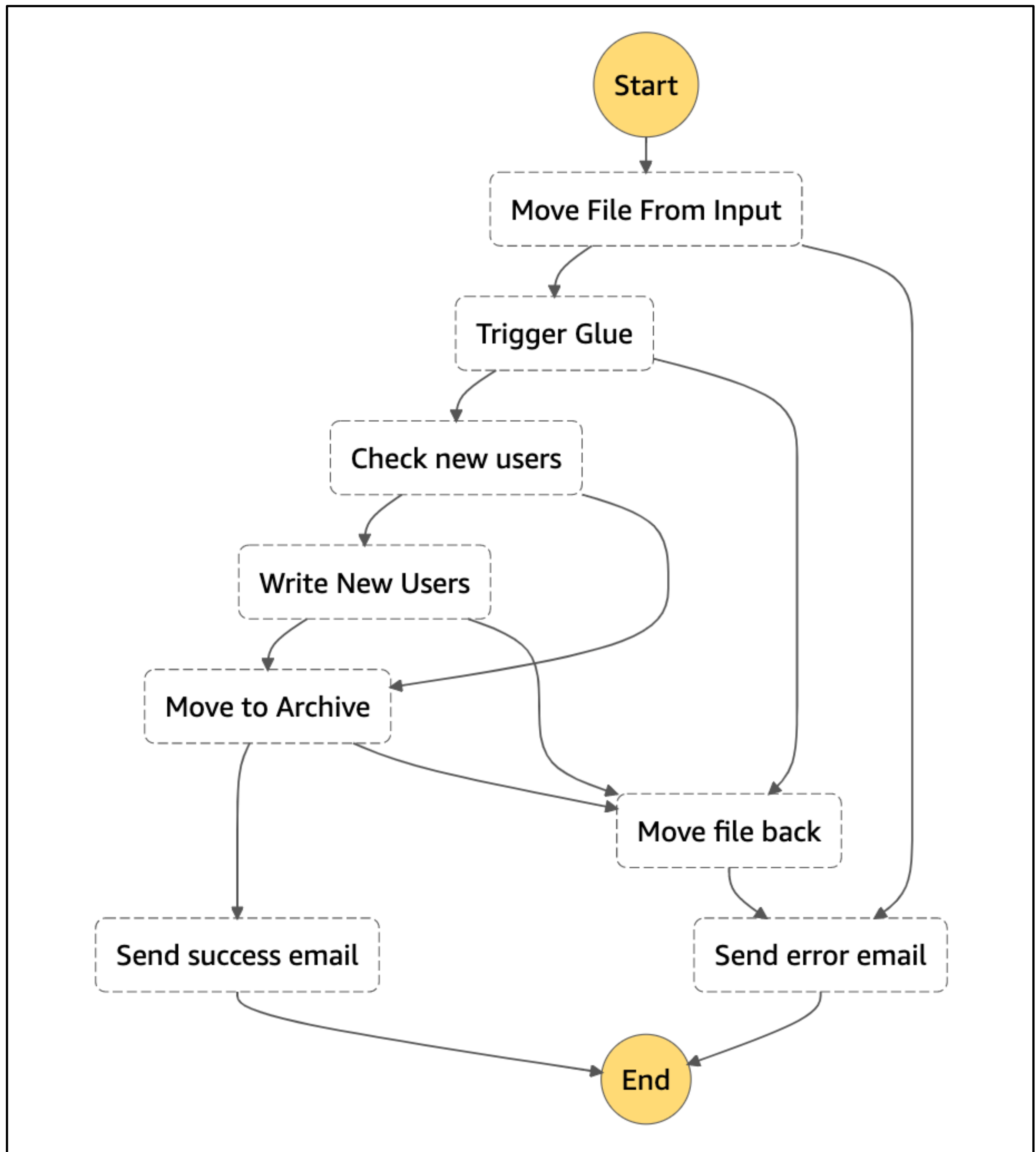

Рисунок 4.3 – граф оркестратору Step Function

Amazon Step Functions є serverless сервісом, який початково був створений для оркестрації лямбда застосунків, але пізніше його функціонал був розширений для інтеграції з більшістю сервісів AWS. Через це архітектурна ідея Amazon Step Functions – це запуск та відстежування станів інших сервісів AWS, та реакція не ці стани. Тобто AWS Step Functions не є самостійним сервісом та тільки передає та корегує результати інших сервісів у вигляді JSON.

Через це усі дії у цьому пайплайні це виклики інших сервісів:

- для запуску пайплайну, як реакції на появу файлу використовуються події S3, які запускають лямбда функцію, яка запускає цей пайплайн;
- для переміщення файлів між директоріями – лямбда функції;
- для обробки файлу – Glue job;
- для відправки електронного листа – виклик події у сервісі SNS;

Єдиним винятком є умовний оператор, який не викликає інших сервісів а має усю функціональність виконану у середовищі Step Function.

4.4.1 Вартість використання на день

Amazon Step Functions є serverless сервісом, і може працювати тільки у середовищі сервісів AWS. Ціна використання складається з багатьох факторів. Самі переходи – близько 25 доларів за 1 мільйон переходів, що важко перерахувати у кількість на день. Але якщо у гіршому варіанті ми будемо отримувати новий файл кожні 10 хв, то на день ми будемо мати 864 переходи, що коштуватиме 0.000864 долара на день. Виклики лямбда функцій будуть коштувати 0.2 долару за мільйон запитів та 0,0000021 долару за секунду виконання. У нашому випадку переміщення файлів відбувається за менше ніж секунду, у нас є 3 переміщення файлу у поганому випадку, тобто ми матимемо 432 переміщення на день у продовж 1296 секунд, що коштуватиме 0.002808 долари на день.

Далі необхідно урахувати інші сервіси. Glue коштує 0.44 долара за одну годину одного серверу, розраховано по секундах, але не менше одної хвилини. У нашій конфігурації це 2 сервери по 1.2 хвилини та 1 хвилина на завантаження. У разі постійної завантаженості кластеру це 144 запусків на день на один оператор завантаження даних у базу. У сумі це виходить 4.65 долара на день.

Додаючи сюди вартість розгортання бази, сховища та сервісу відправки електронних листів ми отримаємо вартість у 7.044 долара на день.

4.4.2 Можливість конфігурації для різних середовищ та способів використання

Архітектура Amazon Step Functions передбачає спілкування між операторами завдяки JSON повідомленням. Тому конфігурація може пересуватись у цих повідомленнях. Також AWS має змогу формувати свої сервіси за допомогою сервісу Cloud Formation, який інтегруючись із фреймворком serverless дозволяє створювати динамічні конфігурації.

4.4.3 Складність розробки

Розробка для Amazon Step Functions можлива у декількох варіантах. Через JSON у веб інтерфейсі, через JSON із загрузкою із репозиторію та через yml через cloud formation. Але через архітектурну ідею того, що усі дії виконуються у інших сервісах – потрібно підтримувати декілька різних стилів коду. Для лямбда функцій, для Glue, тощо. Також це потребує володіння додатковими інструментами, що теж ускладнює розробку.

4.4.4 Масштабування

Amazon Step Functions є serverless застосунком, тому має нескінченні можливості для одночасного запуску та кількості виконань, тому масштабування є майже нескінченним.

4.4.5 Складність розробки у команді

Як було сказано у пункті 3.4.3 для розробки використовується JSON чи YML, що призводить до проблем із перевіркою правильності коду до публікації. Більш того – через те, що розробка повинна бути для різних сервісів, це призводить до складнощів у синхронізації змін.

4.4.6 Підтримка

Amazon Step Functions є закритим сервісом, який підтримується Amazon, завдяки чому усі проблеми вирішуються швидко, а підтримка самого Amazon може допомогти з проблемами.

Наразі на ресурсі StackOverflow зареєстровано біля 734 питань за темою Step Functions. Але така невелика кількість питань може також свідчити про нескладність інструменту.

4.4.7 Складність розгортання

Розгортання є одною із найбільших проблем у контексті Amazon Step Functions, бо потребує використання сторонніх ресурсів та інколи перевірки вже на етапі публікації. Основним інструментом для розгортання є Cloud Formation та фреймворк serverless, але у випадку, коли ці інструменти не підходять для використання – дуже важко знайти інші варіанти.

4.4.8 Підтримка роботи із операціями обробки потокових даних

Amazon Step Functions не підтримує операції із потоковими даними, бо для цього Amazon вважає кращім інструментом Kinesis Streams.

4.5 Реалізація пайплайну використовуючи Azure Data Factory

Пайплайн із пункту 4.1 було розроблено використовуючи засоби Azure Data Factory. Приклад графу обробки наведено на рисунку 4.4.

Azure Data Factory також є serverless сервісом, який із самого початку був створений для оркестрації задач обробки великих даних. Створення пайплайнів виконується завдяки веб-інтерфейсу, який потім трансформується у json, і зберігається у даному вигляді у репозиторії.

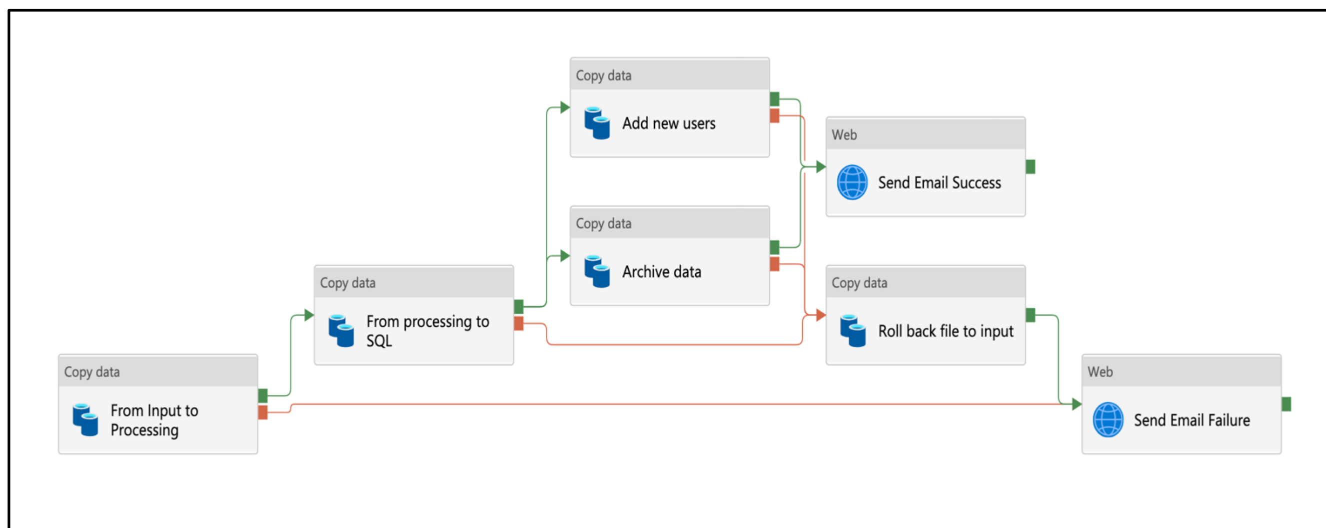

Рисунок 4.4 – граф оркестратору Azure Data Factory

Azure Data Factory складається з вже готових модулів для переміщення та обробки даних, виклику сторонніх сервісів, тощо. Також він має наочні способи задання переходів за станом та інструментарій для виконання циклічних, умовних операцій та операцій зі змінними.

4.5.1 Вартість використання на день

За калькулятором[14], який надає Azure – місячна вартість усього пайплайну із усіма сторонніми сервісами стає 1200 доларів. Що на день – 40 доларів. У цей облік додано репозіторій, запуски лямбда функцій, сховище, база даних та сама Azure Data Factory.

Це робить Azure Data Factory найдорожчім оркестратором серед тих, що є у цій роботі.

4.5.2 Можливість конфігурації для різних середовищ та способів використання

Сам Azure Data Factory дозволяє дуже гнучко конфігурувати пайплайни, але для додавання цих параметрів у різні середовища доводиться мануально у веб інтерфейсі змінювати значення змінних, що не є дуже зручним.

4.5.3 Складність розробки

Розробка простих пайплайнів є чи не найлегшою серед усіх оркетстраторів, бо для цього використовується веб-інтерфейс із дуже зрозумілою параметризацію та швидкими переходами до інших ресурсів, що дозволяє навіть не фахівцям швидко створювати та підтримувати існуючі пайплайни. Але для створення більш складних пайплайнів, із операціями, які не підтримуються у існуючих операторах, доведеться використовувати лямбда функції, які не дуже зручно тестувати і у разі роботи із ними доведеться спиратися на обмеження у часі і ресурсах.

4.5.4 Масштабування

Azure Data Factory є serverless застосунком, тому має нескінченні можливості для одночасного запуску та кількості виконань, тому масштабування є майже нескінченним.

4.5.5 Складність розробки у команді

Для кожного сервісу у архітектурі Azure Data Factory створюється окремий репозіторій, що означає розгалуження коду та складнощі у перевірці. Крім того, кінцевий пайплайн трансформується у JSON, що ускладнює перевірку коду навіть у одному репозиторії.

4.5.6 Підтримка

Azure Data Factory є закритим сервісом, який підтримується Microsoft, завдяки чому усі проблеми вирішуються швидко, а підтримка самого Microsoft може допомогти з проблемами.

Наразі на ресурсі StackOverflow зареєстровано біля 3500 питань за темою Data Factory що свідчить про велику популярність цього інструменту та дає змогу знайти відповіді на велику кількість проблем, які можуть виникнути під час розробки.

4.5.7 Складність розгортання

Azure Data Factory підтримує розгортання для різних середовищ завдяки функціональності Azure DevOps – сервісу для роботи із репозіторями від Azure.

4.5.8 Підтримка роботи із операціями обробки потокових даних

На відміну від інших оркестраторів завдань Azure Data Factory підтримує обробку потокових даних, як один із стандартних операторів, що дає йому велику перевагу над іншими оркестраторами завдань.

5 ПОРІВНЯННЯ ОРКЕСТРАТОРІВ ЗАДАЧ

Для початку необхідно перенести усі характеристики оркестраторів із текстової форми, де цифр або нема, або вони непорівнювані один до одного між оркестраторами, до кількісної.

Після чого необхідно привести їх до єдиного формату у десятибальну шкалу, де 10 – найкращій результат, а 1 – найгірший. Далі, сума усіх характеристик буде давати остаточний результат.

Далі для кожної характеристики буде надано принцип переводу до кількісних значень та таблиця за оркестраторами.

5.1 Вартість використання на день

Для ціни найкращім параметром є те, що чим менша ціна, тим краще. Тому усі суми будуть переведені до зворотних пропорцій, вирівняні по найбільшому та помножені на 10.

Результати обчислень у таблиці 5.1.

Оркестратор	Ціна на день	Зворотна пропорція	Остаточний результат
Airflow	14	0.071	4.76
Luigi	6.54	0.15	10
Step Functions	7.044	0.141	9.46
Data Factory	40	0.025	1.67

Таблиця 5.1 – Приведені результати для вартості на день

Як можна побачити із таблиці – найкращій бал достається Luigi, а найменший достається Azure Data Factory.

5.2 Можливість конфігурації для різних середовищ та способів використання

Усі оркестратори мають можливість конфігурації для різних середовищ та способів використання, але Azure Data Factory має недуже зручний спосіб задання конфігураційних параметрів, а Luigi не має початкової підтримки шаблонизації. Але обидва ці недоліки не є критичними, тому ці оркестратори втрачають тільки по одному балу. Остаточний результат у таблиці 5.2.

Оркестратор	Результат
Airflow	10
Luigi	9
Step Functions	10
Data Factory	9

Таблиця 5.2 – Результати за можливістю конфігурації

5.3 Складність розробки

Усі оркестратори є продуктами, над якими працює велика кількість людей, через що з кожним оновленням користуватися ними стає надалі легше. Розглянемо їх окремо:

- Airflow – дуже зручний інструмент, який може бути використаний для реалізації, як простих, так і важких пайплайнів. Але для нього потрібно знати Python, та розібратися у архітектурних особливостях;
- Luigi – єдиний оркестратор, з яким дуже важко працювати, але він може реалізовувати складні пайплайни;

– Step Functions – самим інструментом Step Functions користуватися не складно, але для реалізації будь-якої функціональності потрібно буде працювати із іншими сервісами;
– Data Factory – найлегший для роботи серед усіх оркестраторів.

Далі приведено таблицю із підрахунками (див. табл. 5.3)

Оркестратор	Результат
Airflow	8
Luigi	3
Step Functions	5
Data Factory	10

Таблиця 5.3 – Результати за складністю розробки

5.4 Масштабування

У масштабуванні велику перевагу мають закриті serverless сервіси, які можуть мати нескінченно багато паралельних запусків та пайплайнів.

Для Airflow єдиним перетином для масштабування є база даних, але самі виконавчі сервери мають великі обсяги масштабування.

Для Luigi єдиним способом масштабування є мікро сервісний підхід[15], де кожен сервер буде незалежним від інших і працювати тільки із своїм набором пайплайнів.

Але це призводить до проблем із розгортанням, складнощам у розробці, та майже неможливості взаємодії між пайплайнами різних серверів.

Результати у таблиці 3.4

Оркестратор	Результат
Airflow	7
Luigi	2
Step Functions	10
Data Factory	10

Таблиця 5.4 – Результати за масштабуванням

5.5 Складність розробки у команді

Усі оркестратори мають змогу бути використаними для розробки у команді, але у випадку оркестраторів які використовують мову Python, а саме Airflow та Luigi, усі набуття сучасних систем керування версіями та середами розробки для вирішення конфліктів.

У той самий час оркестратори, які використовують текстові формати – не мають можливості перевірки помилок, навіть синтаксичних. Важливо також зазначити, що для повноцінного використання оркестратору Step Functions, доведеться використовувати інші сервіси, що іноді призводить до необхідності використання декількох репозиторіїв.

Результати у таблиці 5.5

Оркестратор	Результат
Airflow	10
Luigi	10
Step Functions	4
Data Factory	7

Таблиця 5.5 – Результати за складністю розробки у команді

5.6 Підтримка

Для оцінки якості підтримки ми візьмемо до уваги кількість питань на StackOverflow та зірочки у GitHub, а для закритих сервісів додамо середнє між Airflow та Luigi, бо ми не можемо бути впевнені, що сервіс не буде закрито за невідомих нам обставин.

Результати у таблиці 5.6

Оркестратор	StackOverflow	GitHub	Результат
Airflow	5800	20000	10
Luigi	313	14000	5
Step Functions	734	17000	7
Data Factory	3500	17000	8

Таблиця 5.6 – Результати за підтримкою

3.6.7 Складність розгортання

Найкращі можливості для розгортання надає Azure Data Factory, який підтримує і інтеграцію із репозиторієм, і різні середовища. На другому місці йде Airflow, бо його можна розгортати стандартними інструментами для роботи із додатками на мові Python.

На відміну від Airflow, Luigi складніше розгортати, бо схема запусків в нього йде через Python-daemon.

Amazon Step Functions потребуює використання додаткових сервісів, які не інтегровано із самого початку.

Результати у таблиці 5.7

Оркестратор	Результат
Airflow	8
Luigi	5
Step Functions	6
Data Factory	10

Таблиця 5.7 – Результати за складністю розгортання

5.8 Підтримка роботи із операціями обробки потокових даних

Серед усіх оркестраторів тільки Azure Data Factory підтримує роботи із потоковими даними із початку.

Airflow може бути застосований для цього але із порушенням архітектурних принципів.

Luigi потребує використання деяких хитрощів, та втрачає майже усі надбання, які дає оркестратор.

Step Functions зовсім не може працювати із потоковими даними.

Зведені результати у таблиці 5.8

Оркестратор	Результат
Airflow	7
Luigi	6
Step Functions	0
Data Factory	10

Таблиця 5.8 – Результати за роботою із потоковими даними

5.7 Зведені результати порівняння

У результаті порівняння у попередніх пунктах ми отримали таблицю (див. табл. 5.9), суму балів для кожного оркестратору ми просумуємо та зможемо визначити найліпший за нашими характеристиками.

Де стовбці від 1 до 8 – характеристики із пункту 3.1

Оркестратор	1	2	3	4	5	6	7	8	Результат
Airflow	4.8	10	8	7	10	10	8	7	64,80
Luigi	10	9	3	2	10	5	5	6	50,00
Step Functions	9.5	10	5	10	4	7	6	0	51,50
Data Factory	1.7	9	10	10	7	8	10	10	65,70

Таблиця 5.9 – зведена таблиця результатів

Як ми можемо бачити із таблиці 5.9, найбільшу кількість балів набрали Azure Data Factory та із відставанням у 0.9 балу – Apache Airflow.

Із чого ми можемо зробити висновок, що ці два оркестратора завдань є найдосконалішими серед усіх.

ВИСНОВКИ

У цій роботі було визначено що є великими даними, що є задачами обробки великих даних, на які типи їх можна поділити, та що таке оркестратори завдань.

Було розібрано такі сучасні рішення, як Apache Airflow, його архітектуру та які можливості він має. Розглянули веб-інтерфейс для нагляду за пайплайнами задач та інше. Apache Luigi, та принципи його роботи, організації завдань та можливості, які він надає.

Також розглянули рішення від Amazon – AWS Step Functions, яке трохи відрізняється від попередніх двох тим, що поперед усього це сервіс, який працює за принципом serverless, тобто для нас, як для кінцевого користувача його архітектура невідома, та його не можливо запустити ніде, окрім самого хмарного сервісу Amazon. Та його конкурента від Microsoft – Azure Data Factory, який теж є serverless сервісом, але на відміну від Step Function, який тільки оркеструє задачі, ще має купу можливостей, які розроблені насамперед для роботи із задачами обробки великих даних.

Далі було визначено які є вимоги до оркестраторів задач, тобто яку функціональність вони повинні реалізовувати, як то паралельне виконання завдань, умовні оператори, обробка помилок, логування, тощо.

Після чого було винайдено набір характеристик, за якими ми можемо порівняти різні оркестратори завдань та зробили порівняння. У результаті цього порівняння найбільшу кількість очок набрав Azure Data Factory, та із дуже малим відривом від нього був Apache Airflow, що легко зрозуміти, бо вони обидва є найкращими рішеннями серед оркестраторів свого типу.

За результатами цього дослідження, у разі розробки власного оркестратору, є можливість формалізувати вимоги та мати характеристики для порівняння із існуючими аналогами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Big Data Definition / URL: https://en.wikipedia.org/wiki/Big_data / (дата звернення: 10.04.2021).
2. Workflow Management System definition / URL: https://en.wikipedia.org/wiki/Workflow_management_system / (дата звернення: 10.04.2021).
3. Apache Airflow Documentation / URL: https://docs.microsoft.com/ru-ru/azure/data-factory / (дата звернення: 13.04.2021).
4. Apache Airflow DAG Example / URL: https://stackoverflow.com/questions/57476659/optimizing-airflow-dag / (дата звернення: 13.04.2021)
5. Apache Airflow Architecture UML / URL: https://airflow.apache.org/docs/apache-airflow/stable/concepts.html / (дата звернення: 13.04.2021)
6. Apache Luigi Documentation / URL: https://luigi.readthedocs.io/en/stable / (дата звернення: 16.04.2021).
7. Apache Luigi Architecture / URL: https://luigi.readthedocs.io/en/stable/workflows.html / (дата звернення: 16.04.2021).
8. Amazon Step Functions Documentation / URL: https://aws.amazon.com/step-functions/features / (дата звернення: 17.04.2021).
9. Amazon Step Functions Graph example / URL: https://aws.amazon.com/blogs/aws/new-compute-database-messaging-analytics-and-machine-learning-integration-for-aws-step-functions / (дата звернення: 17.04.2021).
10. Azure Data Factory Documentation / URL: https://docs.microsoft.com/ru-ru/azure/data-factory / (дата звернення 18.04.2021).
11. Managed Workflows with Apache Airflow / URL: https://docs.aws.amazon.com/mwaa/latest/userguide/what-is-mwaa.html / (дата звернення: 19.04.2021).

12. GitHub / URL: https://github.com / (дата звернення: 20.04.2021)

13. Stack Overflow / URL: https://stackoverflow.com / (дата звернення: 20.04.2021).

14. Azure Pricing Calculator / URL: https://azure.microsoft.com/en-us/pricing/calculator/?service=data-factory / (дата звернення: 21.04.2021).

15. Pautasso, Cesare (2017). "Microservices in Practice, Part 1: Reality Check and Service Design". IEEE Software. 34 (1). С. 91–98 (дата звернення: 25.04.2021)

16. Braille character recognition based on neural networks / [K. Smelyakov, A. Chuprina, D. Yeremenko та ін.]. // 2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP). – 2018. – С. 509– 513.

17. Gradational Correction Models Efficiency Analysis of Low-Light Digital Image / [K. Smelyakov, A. Chupryna, M. Hvozdiev та ін.]. // 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream). – 2019. – С. 1–6.