МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження моделей для оптимізації cron завдань

(тема)

Виконав:
Студент 2 курсу, групи ІПЗм-19-1
Афенді К. В.
(прізвище, ініціали)

Спеціальність 121 - Інженерія програмного забезпечення
(код і повна назва спеціальності)

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

Керівник доцент Назаров О.С.
(посада, прізвище)

Допускається до захисту

Зав. кафедри ____________ З.В. Дудар
(підпис) (прізвище, ініціали)

2021 р.

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 спеціальності)

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

Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав.кафедри ____________
(підпис)
« ___ » ____________ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента *Афенді Кирило Вячеславович*
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження моделей для оптимізації cron завдань

затверджена наказом університету від 26.03.2021 № №385 Ст

2. Термін подання роботи до екзаменаційної комісії ______ 2021р.
3. Вихідні дані до роботи модуль для збору cron-задач модуль для оптимізації cron-задач, cron-нотація, системи cron-задач, Apache Airflow, Python, Kubernetes, Linux, Flask
4. Перелік питань, що потрібно опрацювати в роботі вступ, аналіз стану розв’язання проблеми, опис проведених теоретичних досліджень, проектування системи, програмна реалізація, висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, слайдів, ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)

титульна сторінка, аналіз предметної галузі, cron, системи що використовують cron, мета роботи, постановка задачі, цілі оптимізації, модель розкладу, теоретична частина алгоритму розподілу задач, архітектура використаної airflow системи, компоненти розробленої програмної системи, результати парсингу airflow системи, трансформація cron до класичної моделі часу, результати оптимізації розкладу, подальше дослідження, висновки.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1.	Аналіз предметної галузі	27.01.21 –19.02.21	виконано
2.	Огляд методів оптимізації cron задач	20.02.21 –10.03.21	виконано
3.	Аналіз обраних методів на системі cron-розкладів, оптимізація, впровадження	11.03.21 –25.03.21	виконано
4.	Програмна реалізація	26.03.21 –15.04.21	виконано
5.	Підготовка пояснювальної записки	15.04.21 –01.05.21	виконано
6.	Підготовка презентації та доповіді	01.05.21 –01.05.21	виконано
7.	Попередній захист	01.05.21	виконано
8.	Нормоконтроль, рецензування	01.05.21 –04.05.21	виконано
9.	Занесення диплома в електронний архів	04.05.21 –07.05.21	виконано
10.	Допуск до захисту у зав. кафедри	07.05.21	виконано

Дата видачі завдання 27 січня 2021р.

Студент ____________________

(підпис)

Керівник роботи ____________ доцент, Назаров О.С.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до атестаційної роботи: 64 с., 53 рис., 0 табл., 6 додатків, 21 джерел.

АРХІТЕКТУРА, МОДЕЛЬ, CRON, ЕФЕКТИВНІСТЬ, ОПТИМІЗАЦІЯ, APACHE, APACHE AIRFLOW, KUBERNETES, PYTHON, РОЗКЛАД, ЗАВДАННЯ, РЕСУРСИ, ТЕОРІЯ РОЗКЛАДІВ.

Об’єктами дослідження є розклади запуску автоматизованих завдань та ресурсів, що їх використовують, моделі запису та представлення.

Метою роботи є дослідження моделей та методів для оптимізації розкладів автоматизованих cron завдань по ресурсам, та часу виконання.

У результаті роботи була створені моделі та методики оцінювання ефективності розкладів та методи оптимізації їх за ресурсами та часом. Розроблена програмна система для вимірювання ефективності обраних розкладів та методів складання розкладу.

ARCHITECTURE, MODEL, CRON, EFFICIENCY, OPTIMIZATION, APACHE, APACHE AIRFLOW, KUBERNETES, PYTHON, SCHEDULE, TASKS, RESOURCES, JOB SHOP SCHEDULING.

The objects of research are schedules for launching automated tasks and the resources that use them, recording and presentation models.

The purpose of the work is to study models and methods for optimizing schedules of automated cron tasks by resources and execution time.

The models and methods of estimation of efficiency of schedules and methods of their optimization on resources and time were created as a result of work. A software system for measuring the effectiveness of selected schedules and scheduling methods has been developed.

Я, Афенді Кирило Вячеславович, студент гр. ІПЗм-19-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження моделей для оптимізації cron завдань», 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 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 7
1 АНАЛІЗ СТАНУ РОЗВ'ЯЗАННЯ ПРОБЛЕМИ 10
1.1 Системи виконання cron завдань 10
1.2 Масштабування cron систем 11
1.3 Аналіз існуючих cron рішень 13
1.4 Постановка задачі 15
2 ОПИС ПРОВЕДЕНИХ ТЕОРЕТИЧНИХ ДОСЛІДЖЕНЬ 17
2.1 Cron-нотація 17
2.2 Основні поняття та відомості 18
2.3 Теоретична база методів оптимізації розкладів 19
2.4 Цільові функції оптимізування розкладу 19
2.5 Алгоритми та методи оптимізації 20
3 ПРОЄКТУВАННЯ СИСТЕМИ 27
3.1 Основні вимоги до компонентів системи 27
3.2 Опис основних компонентів системи 27
3.3 Поширена схема застосування cron систем 29
3.4 Інфраструктура системи 30
3.5 Взаємодія внутрішніх компонентів airflow 31
3.6 Архітектура компонентів у кластері 37
4 ПРОГРАМНА РЕАЛІЗАЦІЯ 42
4.1 Структура коду проекту 42
4.2 Реалізація cron-завдань у системі airflow 44
4.3 Парсер для airflow системи 48
4.4 Аналізатор та оптимізатор розкладу 50
4.5 Використані програмні забезпечення, бібліотеки та програмні продукти 60
ВИСНОВКИ 61
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 63
ДОДАТОК А. Перелік джерел керівника та науковців кафедри 65
ДОДАТОК Б. Звіт результатів перевірки на унікальність тексту 66
ДОДАТОК В. Слайди презентації 67
ДОДАТОК Г. Лістинг коду 75
ДОДАТОК Д. Стаття 83
ДОДАТОК Е. Результатів перевірки оформлення вимогам ДСТУ 3008:2015 88

ВСТУП

У процесі розвитку програмного забезпечення, програми розширюються до програмних систем [1]. Системи складаються з різних компонентів - від детекторів та програм для запуску тестів до програм відображення інформації. У таких системах багато часу займає розуміння та розробка нових або підтримка та оновлення старих компонентів. Щоб більше займатися першочерговими завданнями, та бути більш творчими при вирішенні завдань, розробники автоматизують ті процеси та задачі, які потребують заданої послідовності інструкцій та виконання у заданий проміжок часу чи події.

У роботі розглядається така автоматизація задач, яка породжує події за якимось планом, наприклад автоматизація процесу тренування моделей машинного навчання за відповідним планом, який визначається швидкістю появи нових даних у базі даних.

Задачі - це послідовність роботи з даними та системою, транспортування, обробка, відображення, збереження, яка виконується у якомусь просторі для обчислення у зазначений час, та має бути опрацьована програмою. На відміну від “Процесу”, який має початок, у “Задачі” є точне або приблизне її завершення. Задачі починають своє виконання тільки коли вони приведені до дії кимось, це може бути ланцюг з механізмів системи який викликав користувач системи, або розробник, або сама система, якщо в неї є механізми планування задач.

Саме цей механізм, приводить до дії та генерує події в системі, для того щоб змінити цю систему, оновити дані в базі, або пересоторувати файли, чи ще щось.

Події - це сутності, які породжуються на зміну системі та середовища, та розглядуються як самою системою, так і розробниками системи. Події бувають різні, нас цікавлять не всі, а конкретні категорії з конкретними властивостями.

Події поділяються на дві категорії:

– зовнішні;

– внутрішні.

Зовнішні події - це такі події, які поступили з інших систем, чи вводяться у систему користувачами чи розробниками цієї системи.

Внутрішні події - це такі події, які зробила сама система, без вводу користувача чи розробника цієї системи.

Наразі існує проблема як рахувати тригери у базі даних. Тут необхідно вводити таке поняття як першопричина події, тому що в багатьох випадках ми працюємо з ланцюгом подій Основним критерієм того що подія в системі є зовнішня є те,що можливо простежити за ланцюгом події до актора (суб'єкта або стороній системи), що породив цю подію.

У цій роботі нас в першу чергу буде цікавити внутрішні події системи, які заплановано генеруються для запуску задач. Необхідно також визначитися із зовнішніми подіями, а саме необхідне моделювання такої системи, опис основних об'єктів та властивостей цих задач. Оптимізація заснована на тому що в нас є система, на якій задачі працюють, тобто задачі потребують певних ресурсів, та планування запуску і розподілу цих ресурсів. Треба зазначити що середовища для запуску задач не завжди гнучкі та мають обмеження з масштабування, тобто залучити чітко необхідні ресурси без їх простоювання це є одна з задач оптимізації.

Оптимізація системи в загалом залежить від наступних цілей:

– зменшити кількість ресурсів;

– розподілити ресурси для заданого рівня навантаження;

– зменшити час виконання задачі.

Звісно, що оптимізація можлива для тих задач, які мають розпорядок запуску, але цей розпорядок можливо здвигати, що в загалому означає, що система потребує запуску якоїсь задачі гарантовано раз в день або в інший проміжок часу, але не потребує конкретно точного часу (наприклад о 9.00 за Київським часом). ожно розрізняти дві категорії задач за планом запуску:

– з чітким часом запуску;

– за не чітким, але гарантованим у якомусь проміжку часі.

Загалом у роботі буде розглянуто всі види таких задач за планом запуску та їх варіацією [2]. На практиці усі ці види задач потрібно реалізовувати, але вони також піддаються оптимізації. Головним принципом є економне використання ресурсів, з таким автоматичним балансуванням задач щоб використання було рівномірне, та не перевантажувати систему. Це дозволить розробникам працювати з більш абстрактними метриками, такими як розподіл ресурсів пам'яті та процесорного часу за часом використання цих ресурсів. За допомогою оптимізації плану виконання за ресурсами та пріоритетами задач можно зробити систему більш надійною та гнучкою. Звісно, що доброчесність кожного розробника полягає у тому, щоб використати найкращим чином дані ресурси та отримати найкращий результат, саме тому ми не будемо дивитись на оптимальність та підхід за яким розробники вирішували та створювали задачі, а будемо казати, що вони розроблені оптимальним чином, та оптимізація на цьому рівні не дає суттєвих переваг.

Основна мета роботи складається з того щоб розробити методи оптимізації внутрішніх задач системи, події запуску яких вона керує самостійно, за планом, пріоритетом, часом, та ресурсами їх виконання. Розробити підходи для аналізу метрик та критеріїв оптимізації для кожної цілі. Розробити модель оптимізації cron завдань, реалізувати цю модель, привести графіки роботи моделі та навести реальні приклади роботи моделі. У результаті планується отримати програмний продукт, модуль якого можно приєднати до різних систем планування внутрішніми задачами системи [3]. Механізм приєднання продукту створено як інтерфейс та його реалізація для поставки та опитування системи щодо її стану, задач які вона виконує та кількість ресурсів на виконання якої вона потребує. Буде програмно розроблено такий інтерфейс, але реалізація цього інтерфейсу для різних систем буде залишена майбутнім розробникам та користувачам програмного продукту для приєднання до конкретних систем. У роботі буде реалізовано для прикладу приєднання цього продукту до системи Apache Airflow [4], тому як вона має стандартний підхід для задання ресурсів та часу виконання задач, як і стандартний cron у системі linux.

1 АНАЛІЗ СТАНУ РОЗВ’ЯЗАННЯ ПРОБЛЕМИ

1.1 Системи виконання cron завдань

На момент початку 2021 року, планування та автоматизація задач у програмних системах набули чітких тенденцій щодо будування їх. Apache Airflow [4] у літку 2015 став такою системою яка поєднувала стандарти системи для автоматизації задач за внутрішнім планом. У системі за базову ідею взяли підхів з unix подібних систем, а саме широко розповсюджений компонент cron [5] (див. рис. 1.1.). Цей компонент став основою для автоматизації задач на рівні операційної системи, наприклад для архівації, резервного копіюванню важливих даних, видалення кешу та тимчасових файлів, та всього на що здатні операційна система та розробник.

```
# /etc/crontab: system-wide crontab
# Unlike any other crontab you don't have to run the `crontab'
# command to install the new version when you edit this file
# and files in /etc/cron.d. These files also have username fields,
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly
#
```

Рисунок 1.1 - Cron в операційній системі Linux

Робота cron проста [5], він просто запускає зазначені інструкції,як з команди терміналу за заданим часом, який встановив програміст. Такі речі як запис логів одразу не налагоджені, роботу задачі у попередні запуски просто так не подивишся, це залишается для розгляду користувачеві.

Налагодити запуск програми за вказаний проміжок часу через cron досить легко, він має зрозумілий синтаксис планування запуску, так би мовити cron-синтаксис, який може бути тонко налагоджений до секунди. Все це заповнюється через термінал або конфігурацію файла у текстовому редактору та редагується так само.

Такий інструмент доволі добре використовується для невеликих систем, тобто в межах однієї машини (серверу), задач, та їх автоматизування. Він став значимим та вплинув на всі подальші такі системи, особливо збереглася та стала стандартним cron-нотація часу запуску задач.

1.2 Масштабування cron систем

Наступним кроком розвитку автоматичних запусків та планування задач стало розширення середовища виконання самих задач, тобто виконання на декількох машинах чи в кластері.

Одним із самих розповсюджених продуктів для цього став Celery - це розподілений планувальник задач та їх виконання на мові програмування Python [6], який з’явився у 2009 році. Він перейняв у cron-а нотацію для розрахунку часу запуску задач, суттєво вдосконалив систему зберігання та моніторингу логів та часу виконання, до появи Apache Airflow [4] він був основним інструментом для роботи з плануванням автоматизованих задач на розподіленому середовищі. Такий підхід доволі значно структурував та полегшував роботу розробникам, але зовсім легким у використанні та налаштуванні на великої кількості машин не був та доставляє значних труднощів і потребував багато часу на налаштування та пошуку проблем (див. рис. 1.2).

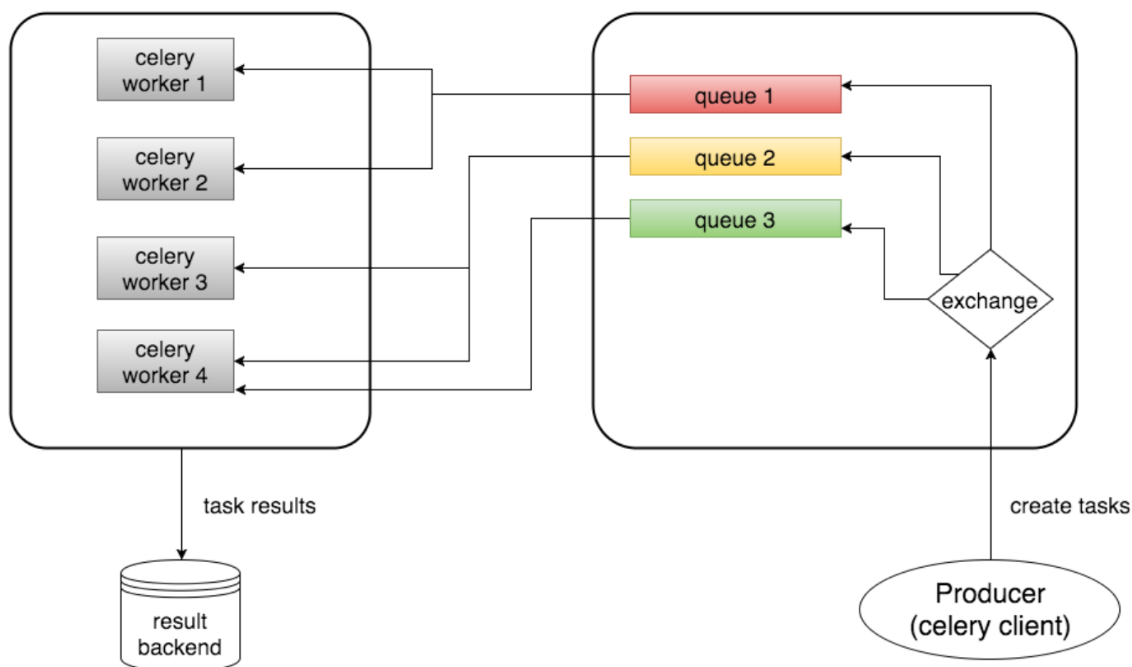


Рисунок 1.2 - Архітектура та компоненти Celery

Звісно є альтернативи Celery, але вони мають такий же підхід і є майже клонами з незначними удосконаленнями. Значною мірою Celery став найпопулярнішим для своїх цілей інструментом та, навіть зараз залишається затребуваним та використовується. Його використання відбувається завдяки злагодженої роботи розробників та розвиненої системі інтеграцій, також він добре адаптувався до реалій та став основою для інших систем та інструментів на його основі, таких, як наприклад Dask (див. рис. 1.3), Apache Airflow та інші інструменти, які мають можливість запускати свої задачі у середовищі Celery.

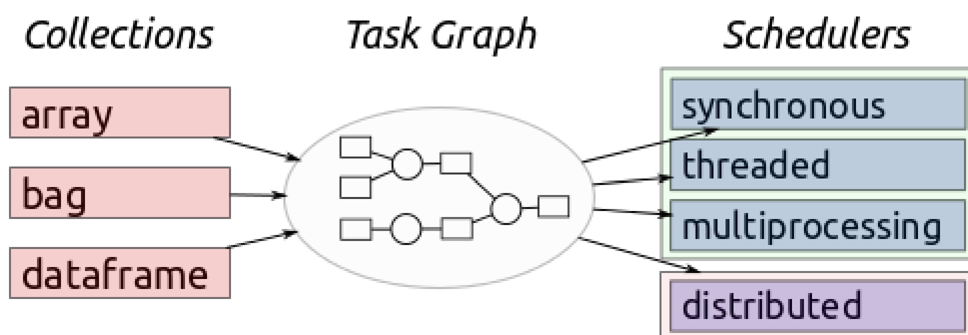


Рисунок 1.3 - Використання системи розкладу в Dask

Для наведення приклада того, як працюють системи планування автоматичних задач буде використано продукт від Apache, Airflow [4] - це досить високорівнева та гнучка система, яка має можливість запуску задач на різних платформах та середовищах, та має cron-нотацію для задання часу запуску задач.

На жаль, на даний момент немає прикладів вирішення, чи методів оптимізації таких планових задач, у вигляді завершеного продукту чи моделі, є окремі задачі, до яких можно звести цю проблему. Але ці підходи не мають конкретики та чіткої спрямованості на систематизацію знань про методи оптимізації. е пов'язано з тим, що проблема має декілька цілей за якими проводиться аналіз з оптимізації запуску задач, а інколи доводиться працювати з грубими інструментами та параметрами аналізу, які не мають запасу гнучкості для ції цілей.

Ця робота направлена на систематизацію та перетворення у чітку модель оптимізації cron планування, визначення проблеми та шляхи її вирішення, введення параметрів оцінки та формул для їх вираховуванням, що дозволяє поліпшити та моніторити процеси розвитку системи та її вдосконалення.

1.3 Аналіз існуючих cron рішень

Існуючі рішення поділяються на різні підходи до розробки та цілей автоматизації. Такі цілі залежать також від ролей в команді, так наприклад для DevOps підхода краще всього використовувати Argo, так як він простіший, не потребує складних маніпуляцій чи створення додаткових програмних додатків.

Такі системи як Apache Airflow [4] досить універсальні та гнучкі, їх називаються DataOps підходом, краще всього вони використовуються для ETL-розробників. ETL - extract, transform, load. Це процес збору, трансформування та завантаження даних в сховище, цей процес чаще за все є доволі великим та потребує складних маніпуляції з сторонніми сервісами, структурами даних, налагодженню інших систем, чаще за все він є частиною компетенцій Data Engineer, але може бути окремою позицією в команді.

Такі системи як Kube Flow дуже схожі на проміжне рішення між Argo та Airflow, вони більш гнучкі чим Argo, але доволі грубі по зрівнянню з Airflow, процеси що будуть виконуватись на Kube Flow будуть доволі надійно налагоджені, адже вони будуть виконуватись на Kubernetes [7] кластері. Така система підійде для MLOps та DataOps. Слід зазначити, що для MLOps вона підходить набагато краще. Дата сайентисти будуть добре почувати себе при правильно налагодженій системі.

Рекомендується використовувати не більше однієї системи, так наприклад доволі складно підтримати документацію та різні підходи до розробки. Популярність сервісів є доволі простий показник, але він відповідає на декілька практичних питань, а саме:

- наскільки швидко можна знайти відповідь по системі;
- кількість користувачів, що виправляють систему;
- кількість розробників які підтримують та покращують систему.

На рисунку 1.4 показано графік популярності вищезгаданих систем.

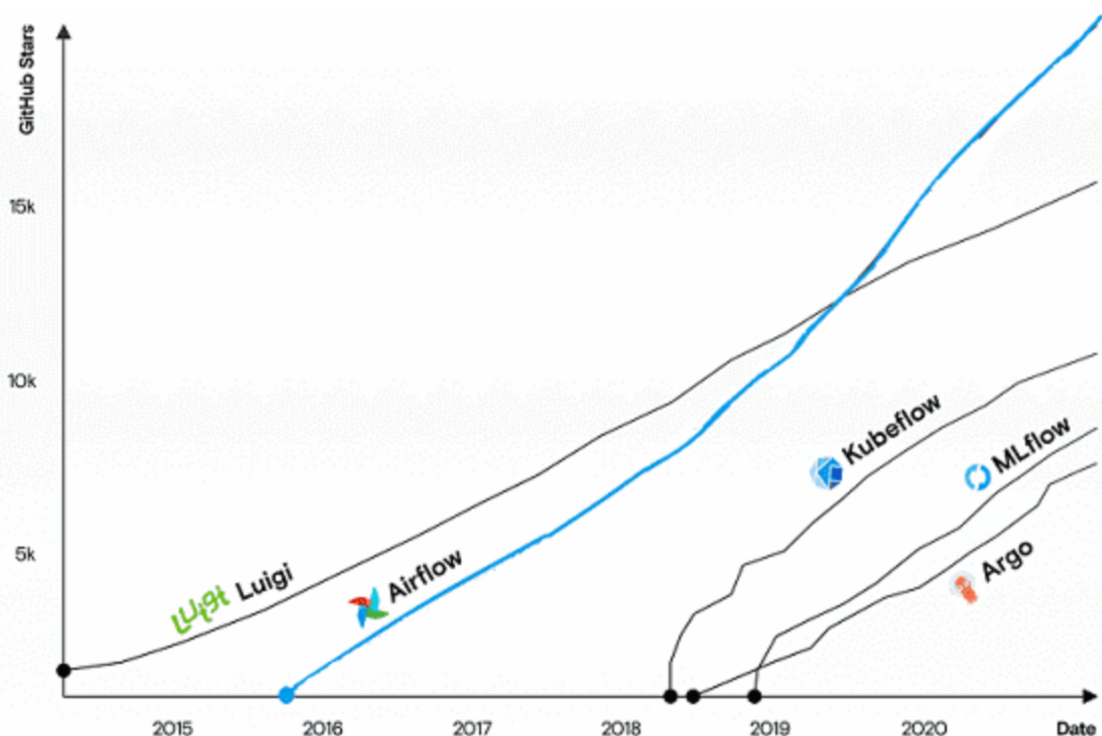


Рисунок 1.4 - Популярність систем автоматизації

Рекомендується використовувати не більше однієї системи, так наприклад доволі складно підтримати документацію та різні підходи до розробки. Наступним кроком порівняння є внесення параметров якості для показників системи, так наприклад систему можно оцінити за:

– простота використання;
– глибина спеціалізованості;
– адаптованість;
– мова програмування.

На рисунку 1.5 приведена таблиця порівняння вищезгаданих показників.

	Maturity	Popularity	Simplicity	Breadth	Language
Apache Airflow	B	A	C	A	Python
Luigi	B	B	A	B	Python
Argo	C	B	B	B	YAML
Kubeflow	C	B	B	C	Python
MLFlow	C	B	A	C	Python

Рисунок 1.5 - Властивості систем автоматизації задач

Як бачимо, Apache Airflow [4] має дуже гнучку систему, але його вивчення та використання недостатньо, Argo дуже близько підійшов до показників Airflow, але має не дуже гнучку систему для адаптації з іншими сервісами, на жаль це важливий показник у подальшому дослідженні та реалізації програмних додатків. Саме було обрано систему Apache Airflow [4].

1.4 Постановка задачі

Постановка задачі складається з дослідження та розробки моделі та системи оптимізації:

– дослідити існуючі підходи, методи та моделі для оптимізування розкладів;

– теоретично і експериментально визначити їх практичність для використання до cron-систем;
– створити додаток для парсингу, аналізу, та оптимізації cron-систем;
– розробити вимоги та прототип архітектури за яким буде проводитись оптимізація;
– розробити цикл операцій для оптимізації cron-систем;
– впровадити нові компоненти до існуючих систем;
– проаналізувати та довести доцільність використання нових компонентів;
– дослідити складність та метрики за системою оптимізації;
– виробити рекомендації для налагодження нових компонентів та їх інтеграції;
– виробити вимоги до системи Apache Airflow та інтеграції з нею;
– проаналізувати складність оптимізації та теоретичну модель з теорії розкладів на ефективність використання;
– розробити парсер для задач с системи Apache Airflow;
– розробити аналізатор для розкладів та оптимізованих розкладів;
– розробти оптимізатор для класичної моделі з теорії розкладів за цільовими функціями;
– реалізувати програмний продукт для автоматизованої оптимізації задач для розробників.

2 ОПИС ПРОВЕДЕНИХ ТЕОРЕТИЧНИХ ДОСЛІДЖЕНЬ

2.1 Cron-нотація

У більшої кількості систем, що використовують cron-нотацію, її використовують в звичному форматі з незначними доповненнями. Основа для введення часу завдання у cron-нотації це строка, що має п'ять колонок, розділені пропусками. Кожна колонка зліва та надалі має значення у часі; хвилини, години, дні, місяці, день неділі, рік. На кожну колонку є допустимий інтервал з чисел та символів. Читати та розуміти запланований час з cron-нотації потрібно виходячи з усього розуміння колонок, адже ігнорування одного з елементів спричиняє некоректне розуміння, коли задача буде виконана та який буде план запусків.

Сама cron-нотація виглядає так; "* * * * * *", або можна записати "* * * * *", ігроруючи останню колонку, тобто рік можно не вказувати, це значить що зміна року не впливає на план запуску, в першому випадку ми явно зазначаємо, що задачу треба виконувати у кожному наступному році, з початку того року в якому ми знаходимось включно. Весь цей запис можно прочитати як - "кожну хвилину, кожного годині, кожного дня, кожного місяця, кожного дня неділі, кожного року", або якщо коротко - "кожну хвилину".

Якщо ми хочемо щоб задача запускалась раз в годину на двадцятій хвилині, то треба записати "20 * * * *" - це значить, що кожний раз, коли настає '20 хвилин, кожного часу' ми запускаємо задачу.

Якщо ми хочемо щоб задача запускалась раз в двадцять хвилин, тобто задача за час виконається тричі, то треба записати "*/20 * * * *".

Якщо ми хочемо задати, щось складне, наприклад - "кожні сім хвилин, по кожні три години, через кожні чотири дні, через кожний місяць, якщо попали на понеділок". Це мабуть найскладніший та майже не використовуємий випадок, але й це можно записати через cron-нотацію "*/7 */3 */4 */2 1". Зазначимо, що п'ятий стовпчик, що відповідає цифрі дня неділі, так наприклад 1 - це понеділок, 5 - п'ятниця, 7 - неділя. Це підтверджує сервіс для генерації опису cron виразів (див. рис. 2.1).

“At every 7th minute past every 3rd hour on every 4th day-of-month if it's on Monday in every 2nd month.”

next at 2021-05-17 00:00:00 random

*/7 */3 */4 */2 1

Рисунок 2.1 - Перевірка коректності заданого часу

Як бачимо, розклад для задачі вказано правильно.

Є також альтернативні форми запису розкладів, так наприклад табличний формат та у вигляді графу, модель що використовується у цій роботі ближче до табличного формату.

2.2 Основні поняття та відомості

Розклад - це структурована дискретна послідовність події у часі, яка записана у вигляді формули чи схемі.

Подія - це точка у часі, яка має початок та кінець.

Задача - це інструкції, програми, алгоритми, які виконуються за якоїсь події, обмежені в часі подією, так як вона має початок та кінець.

Машина - це простір в якому виконуються задачі, цей простір чимось обмеженою, кількістю процессорів та об’ємом пам’яті.

Парсер - це програма, яка сканує систему на предмет закодованих розкладів та приводить їх до практичного та єдиного формату.

Алгоритм оптимізації розкладу - це алгоритм який приводить до цільових метрик розклад.

Вікно простою - це такий проміжок часу у розкладі, коли машина не зайнята такими задачами значний час, під ‘значний час’ - мається на увазі що

співвідношення часу простою до часу остановки до старту машини перевищується у 3-4 рази.

2.3 Теоретична база методів оптимізації розкладів

Сама задача оптимізації cron-задач, може бути зведена до змішаної задачі з теорії розкладів [8]. Теорія розкладів - це такая сфера знань з дискретної математики, яка вивчає аналіз рішення задач розкладу в різних проявах. Наприклад, така наука вивчає, як найкраще оптимізувати розклад вчителів або професорів у школі і університеті, така модель була би доцільною для нас, якщо б вчителі вміли одночасно не заважаючи друг другу для різних груп людей проводити заняття в одній аудиторії чи одному класі [9].

Моделью у теорії розкладів вважається така спрощена математична модель, що відповідає функціональним властивостям системи, в якій відкинуто частини що не беруть участь у аналізу чи оптимізації розкладу [10].

З теорії розкладів нам підійде модель, за якою потрібно оптимізувати події k0, .., kn на пространстві m0, mn. Слід підібрати такі m0, .., mk, які би балансували між достатніми ризиками та витрати за ціною, адже m0, .. mn - мають якусь ціну, c0… ck [11].

Існують також інші моделі оптимізації які будуть розгладжувати частоту задач за часом. Також краще всього обирату ціль оптимізації за одним параметром, або зводити декілька параметров до одного складного, таким чином реалізація та складність моделі буде прийнятною для оптимізації [12].

2.4 Цільові функції оптимізування розкладу

Це такі функції, які на вхід мають параметри складеного розкладу, та використовуються для оцінка цих параметрів у числовому еквіваленті. Такі функції необхідні для того щоб обрати який з багатьох розроблених розкладів більш доцільний для користувача [13].

Нам потрібно декілька базових функцій, за допомогою яких можливо представити більш складні, враховуючи те, що однією з цілей оптимізації є оптимізація ресурсів, тож буде доцільно представити оцінку використаних ресурсів розкладу у вигляді функції.

Окрім того нам вже відомі деякі теоретичні відомості, які виходять з логіки роботи з програмним забезпеченням, наприклад те, що у системі буде використовуватись деякий час така машина, яка може обробити саму ресурсовитратнуу задачу [14].

Також слід пам’ятати, що задача оптимізації ресурсів мають різний контекст, який на них впливає, наприклад - якщо машини користувача є його власністю та він повністю ними розпоряджається.нша ситуація, яка стає дедалі поширеною, це оптимізація коштів на використання ресурсів на різних платформах, наприклад у хмарному сервісі, який має різни типи та види ресурсів, та комбінація у часі цих ресурсів не завжди очевидна. Тому доцільно для цього ввести додаткову функцію оцінки - функцію, що буде оцінювати кошти, витрачені на обробку за планом розкладу, оптимізація такої функції дасть нам можливість при мінімальних коштах [15].

2.5 Алгоритми та методи оптимізації

Основною особливістю оптимізації cron-задач є те, що за одну з умовностей береться твердження, що cron-запис є тільки рекомендованим часом запуску, тобто замість строгого виконання у заданий час ми маємо частоту, яку можна на проміжках часу здвигати, розтягувати та стягнути [16].

Це значить, що оптимізація cron завдань буде не на рівні cron-формату [5], а не на рівні тих дат, що з них генеруються.

Зверху схема оптимізації виглядає у такій послідовності:

- збір cron-завдань;
- трансформування вхідних даних;
- оптимізація;

– аналіз;

– впровадження.

Збір даних буде виконувати один із компонентів, що пізніше буде розроблено та описано у цій роботі. Все що потрібно знати, це те що він просканує систему та його результати будуть вихідними даними для подальших кроків. З точки зору математики та алгоритмів нас цікавить трансформування даних, аналіз та оптимізація.

Трансформування даних, це один із необхідних процесів, що полегшує подальшу роботу над даними, після збору cron-записів вони представлені у вигляді списку (див. рис. 2.2). У секцію для трансформації входять такі кроки, як, групування cron-завдань за часом [5]. У цій роботі було використано наступні групи часу: 1 час, 24ч, місяць. Вибір таких груп обумовлений тим, що зсув задачі має межі зсуву, для якого була обрана функція log x . Це значить, що задача яка рекомендована до запуску кожні 15 хвилин, може бути зсунута на 3 хвилини назад чи вперед, а задачі які запускаються кожні 1-2 хвилини зсуву не підлягають, кожні 3-5 хвилин доступний зсув на 1 хвилину, а для 6-14 хвилин на 2 хвилини, і так далі.

```
*/3 * * * *
*/5 * * * *
*/7 * * * *
*/15 * * * *
*/5 * * * *
*/5 * * * *
*/15 * * * *
*/15 * * * *
*/15 * * * *
*/20 * * * *
*/15 * * * *
*/2 * * * *
*/5 * * * *
*/5 * * * *
```

Рисунок 2.2 - Список cron задач в межах одного часу

Після цього, дотримуючись класичній теорії розкладів, для оптимізації потрібна стандартна математична модель, у якій є дві осі: час та та цільова змінна

що буде оптимізуватися. Для прикладу буде взято кількість задач які виконуються у взятій точці часу. Метод трансформації доволі простий, за основу було використане готове рішення, яке на вхід бере cron-вираз, а на виході список часових точок, у яких задача будет запущена. Приклад результату обробки даних наведено на рис. 2.3. Кількість задач на момент часу - це доволі проста, можливо сказати навіть примітивна модель, але вона дуже зручна для наглядних цілей та актуальності того, що є простір для ускладнення моделі оптимізації. Як бачимо на рисунку є моменти часу, коли кількість задач доволі значна у порівнянні з іншими точками часу, також слід зазначити, що в списку відсутні деякі точки часу, в яких могли б виконуватись будь які задачі, так наприклад між 2021-04-22 04:10:00 та 2021-04-22 04:12:00 відсутня точка 2021-04-22 04:11:00.

```
2021-04-22 04:04:00 - 1
2021-04-22 04:05:00 - 5
2021-04-22 04:06:00 - 2
2021-04-22 04:07:00 - 1
2021-04-22 04:08:00 - 1
2021-04-22 04:09:00 - 1
2021-04-22 04:10:00 - 6
2021-04-22 04:12:00 - 2
2021-04-22 04:14:00 - 2
2021-04-22 04:15:00 - 11
2021-04-22 04:16:00 - 1
```

Рисунок 2.3 - Результат трансформації cron-виразів

Тепер, після того як були отримані трансформовані дані, можемо описати як саме оптимізувати таку модель. По перше потрібно відштовхуватись від очевидних моментів та фактів, а зараз вони такі:

- задачу можна зсунути на час який дорівнює 2 * log x, де x - час між двома наступними послідовними запусками;
- точки часу, що немає у розкладу дорівнюють точкам часу в яких задач немає, та будуть використані у процесі оптимізації;

– один варіант розкладу кращий за інший, якщо в першому кількість задач більш рівномірно за часом чим в іншому.

Слід додати, що можливий зсув точки часу можливий і за другими формулами, але логарифм росте дуже повільно, що дозволяє його використовувати на групі задач, які запускаються раз у годину. Для інших груп задач за часом краще використовувати інші функції оцінки зсуву.

Також слід зауважити, що пошук точок часу за замовчуванням був обран з інтервалом одну хвилину, для інших груп задач слід розглянути інші варіанти інтервалів, це дозволить спростити пошук та складність алгоритма.

Останній пункт залежить від цілі оптимізації, але для наглядності було обрано вже згадану просту модель розкладу.

Сам розроблений алгоритм, має наступні кроки:

– з списку моделі знаходиться часова точка, у якій найбільша кількість задач;
– з списку задач, в знайденій точці з пункту 1, береться задача, що має найдальший довжину зсуву;
– знаходиться така сусідня точка часу, що має найменшу кількість задач, та має мінімум на дві задачі менше ніж обрана точка часу;
– вибрана задача з пункту 2 зсувається на точку з пункту 3, длина зсуву зменшується на довжину зсуву що була зроблена;
– знов виконується пункт 2, поки усі сусідні точки часу не перестануть задовольняти пункт 3;
– знов виконується пункт 1, поки кількість заданих ітерацій не вийде за встановлені межі.

У випадку, якщо цільова функція оптимізації буде ускладнена, наприклад, результатом, повинен бути здійснений розподіл задач, що буде самим оптимальним за використанням ресурсів CPU. Алгоритм буде виглядати наступним чином:

– з списку моделі знаходиться часова точка, у якій найбільший показник завантаження CPU;

– з списку задач, в знайденій точці з пункту 1, береться задача, що має найдальший довжину зсуву;
– знаходиться така сусідня точка часу, що має найменшу завантаженість за CPU, та має мінімум на дві задачі менше ніж обрана точка часу;
– вибрана задача з пункту 2 зсувається на точку з пункту 3, довжина зсуву зменшується на довжину зсуву що була зроблена;
– знов виконується пункт 2, поки усі сусідні точки часу не перестануть задовольняти пункт 3;
– знов виконується пункт 1, поки кількість заданих ітерацій не вийде за встановлені межі.

Якщо ціль оптимізації моделі вказати не CPU, а використання пам'яті на кластері то алгоритм для оптимізації буде виглядати так:

– з списку моделі знаходиться часова точка, у якій найбільший показник завантаження пам’яті;
– з списку задач, в знайденій точці з пункту 1, береться задача, що має найбільшу довжину зсуву;
– знаходиться така сусідня точка часу, що має найменшу завантаженість за пам’яті, та має мінімум на дві задачі менше ніж обрана точка часу;
– вибрана задача з пункту 2 зсувається на точку з пункту 3, довжина зсуву зменшується на довжину зсуву що була зроблена;
– знов виконується пункт 2, поки усі сусідні точки часу не перестануть задовольняти пункт 3;
– знов виконується пункт 1, поки кількість заданих ітерацій не вийде за встановлені межі

Ускладнення цілі за якою буде оптимізована модель не завжди веде до ускладнення самого алгоритму оптимізації, так наприклад якщо використовувати оптимізацію за декілької показників, слід скомбінувати ці параметри до одного, тобто розробити функцію, що буде оцінювати два показники та віддавати один, наприклад алгоритм оптимізація за CPU та використаною пам'яттю машини буде виглядати так:

- з списку моделі знаходиться часова точка, у якій найбільший показник складеної оцінки CPU та пам’яті;
- з списку задач, в знайденій точці з пункту 1, береться задача, що має найдальший довжину зсуву;
- знаходиться така сусідня точка часу, що має найменшу завантаженість за складеної оцінки CPU та пам’яті, та має мінімум на дві задачі менше ніж обрана точка часу;
- вибрана задача з пункту 2 зсувається на точку з пункту 3, довжина зсуву зменшується на довжину зсуву що була зроблена;
- знов виконується пункт 2, поки усі сусідні точки часу не перестануть задовольняти пункт 3;
- знов виконується пункт 1, поки кількість заданих ітерацій не вийде за встановлені межі.

Складнощі виникають коли ціль оптимізації є бізнес цілі, наприклад оптимізація за коштами, тут мають місце наступні умови, а саме час за який потрібно оплачувати сервіс, якщо кошти потрібно витрачати тільки на час який використовують то алгоритм не дуже змінюється. Так наприклад, якщо сервіс підтримує оплату кластера захвилинами роботи, то алгоритм буде наступний:

- з списку моделі знаходиться часова точка, у якій найбільший показник часу виконання на кластері;
- з списку задач, в знайденій точці з пункту 1, береться задача, що має найдовший час виконання;
- знаходиться така сусідня точка часу, що має найменшу ціну виконання, та має мінімум на дві задачі менше ніж обрана точка часу;
- вибрана задача з пункту 2 зсувається на точку з пункту 3, довжина зсуву зменшується на довжину зсуву що була зроблена;
- знов виконується пункт 2, поки усі сусідні точки часу не перестануть задовольняти пункт 3;
- знов виконується пункт 1, поки кількість заданих ітерацій не вийде за встановлені межі.

Треба зазначити, що показники витрачених коштів оптимізуються якщо додати до моделей параметр, який відповідає характеристикам машин, на якій вони запущені. Модель має оцінювати як само потрібно запускати та розміщувати час та виконання на конкретних машинах, задача ускладнюється ще й тим що звичний алгоритм оптимізації не призведе до значних покращень, адже можливо й таке що за коштамами розклад стане оптимальним, але більш не стабільним. Очевидно, що кращим результатом моделі з ціллю економії коштів буде розклад з мінімальною кількістю витрат часів на запуск, що призведе до зриву використовування ресурсів у деяких точках часу.

В очевідь потрібно обминати таку сліпу модель оптимізації, найкращим варіантом буде зібрати модель та оптимізувати розклад за звичайними, так би мовити “прямими” параметрами, такі параметри були вже згадані в цьому пункті, а саме CPU, пам’ять, кількість задач на точку часу. Якщо спочатку оптимізувати за цими параметрами, а потім розробити модель що робить пропуски часу в яких кластер не буде використовуватись, то така гібридна модель може буде оптимальним рішенням між інфоструктурной та бізнес ціллю оптимізації.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Основні вимоги до компонентів системи

Перш за все слід визначити основні компоненти системи для того, щоб розробка продукту була зручною та ефективною, адже розподілення на частини та компоненти дає системі гнучкість, простоту, визначеність, чіткі та цілісні вимоги, адаптивність до інших можливих систем. Компоненти продукту у своїй взаємодії створюють систему, що і має виконувати поставлені цілі. Використовуючи прагматичний та оснований на здравому розуму підхід до проєктування компонентів - означає те, що ці компоненти можуть функціонувати окремо одне від одного , використовуватись для менших цілей. Кожен компонент системи повинен відповідати наступним вимогам [19]:

– незалежність від інших компонентів;
– прості та очевидні інтерфейси роботи з компонентом;
– стандартний потік вводу та виводу;
– декларований та незмінний формат вхідних та вихідних даних;
– функціональна незалежність частин компоненту;
– покриття тестами та документація.

3.2 Опис основних компонентів системи

Система складається з трьох основних компонентів:

– парсер cron задач;
– аналізатор розкладу;
– оптимізатор розкладу.

Парсер cron задач, буде збирати дані з системи, що використовує cron нотацію, трансформувати до єдиного формату розкладу, який може бути використаний для оптимізації або аналізу розкладу.

Аналізатор розкладу, це компонент який зображує у зручному для людини форматі - метрики, графіки, тощо. На вход він отримує дані від парсеру cron задач.

Оптимізатор розкладу - це компонент який використовуючи різні алгоритми та цілі оптимізації розкладу, крім того що цей компонент можу існувати окремо від інших компонентів, його буде зручно інтегрувати з іншими існуючими та популярними системами. Він отримає на вхід розклад у якому вказано всю інформацію що можливо зібрати з системи та проаналізувати, на виході буде отримано рішення у вигляді того ж розкладу але більш оптимального та оптимізовано згідно цілями оптимізації.

Проста та базова взаємодія компонентів представлена на рис. 3.1.

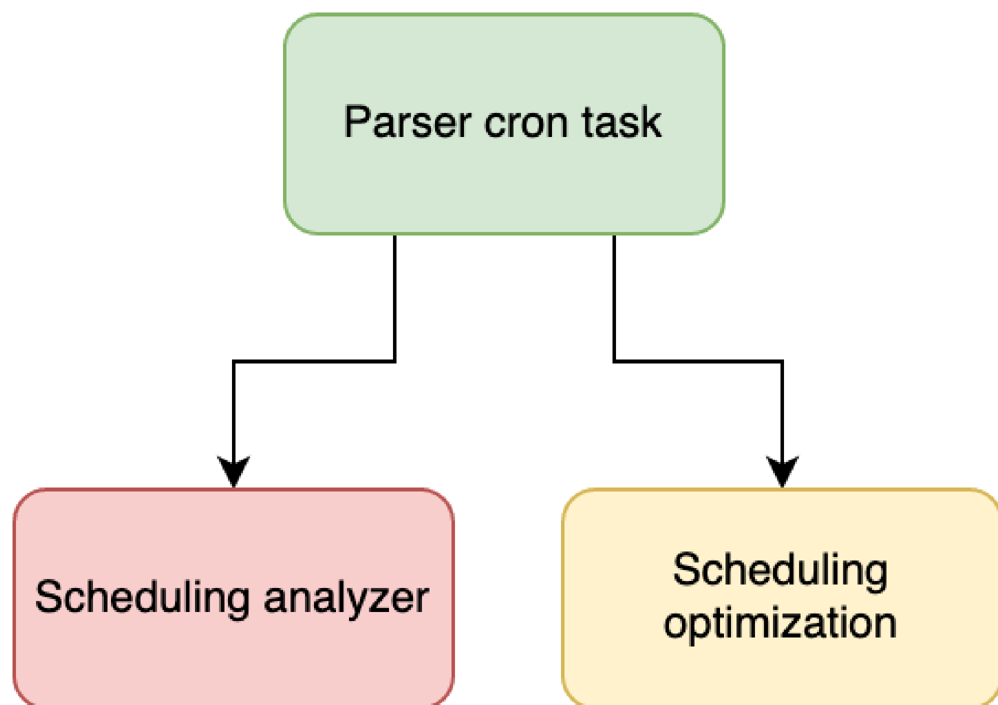


Рисунок 3.1 - Базова взаємодія компонентів системи

Аналізатор розкладу можливо використовувати для порівняння того як саме та що саме було зроблено оптимізатором у розкладі (див. рис. 3.2). Під аналізатором мається на увазі комплекс виводу та графіки, що відображається у ході огляду існуючого та ще не оптимізованого розкладу з моделью та основними показниками, та результатом рішення, з новими показниками.

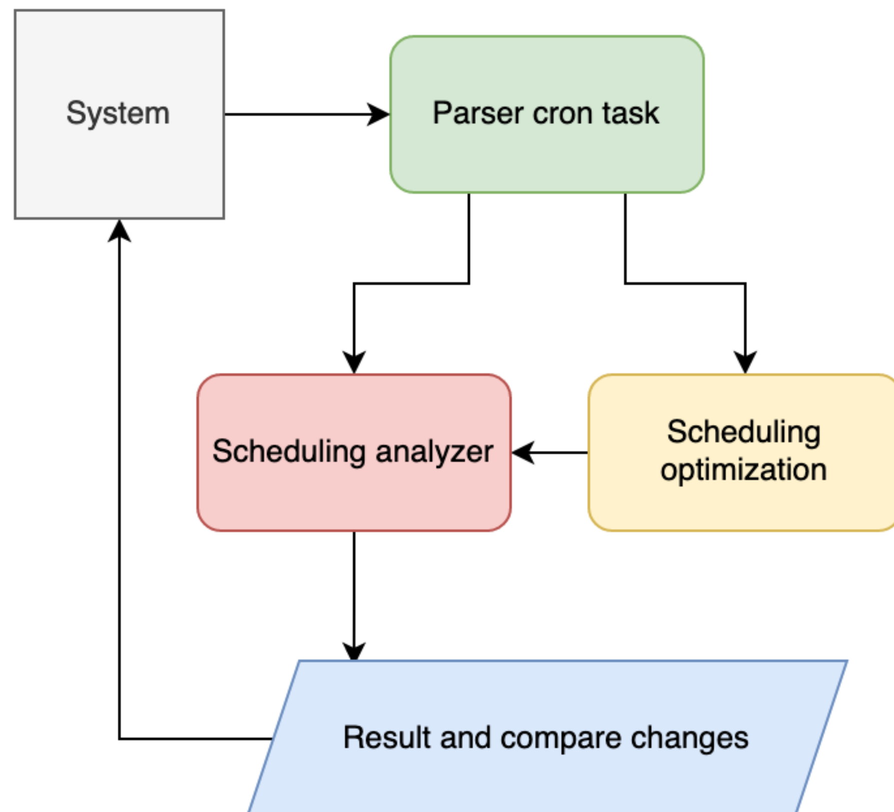
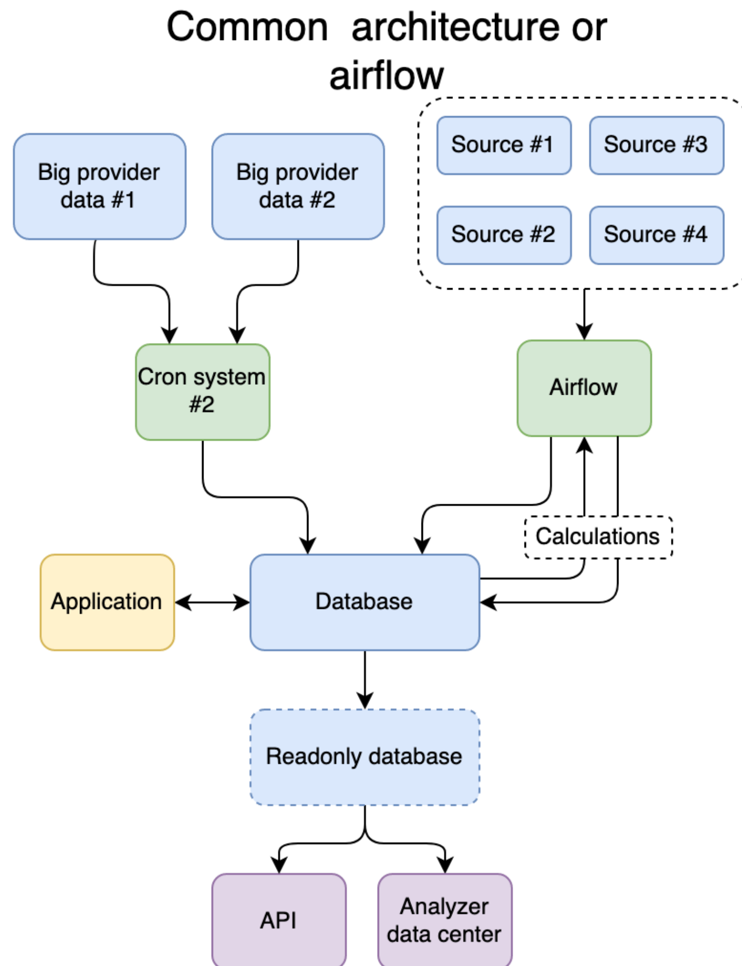


Рисунок 3.2 - Компоненти системи у взаємодії з системою розкладу

Компоненти розроблені таким чином, щоб бути використані як цикл досліду та впровадженню в основну систему.

3.3 Поширена схема застосування cron систем

Для того щоб зрозуміти проблему на практиці, потрібно намалювати схеми компонентів та їх взаємодію. Стане зрозуміло які компоненти та зв'язки потрібно оптимізувати. Далі наведено приклад архітектури, де використовують airflow як один з основних чи побічних систем розподілення задач за розкладом (див. рис. 3.3).



Малюнок 3.3 - Звичайна архітектура взаємодії з airflow

Як бачимо у системі присутні звичні компоненти та різні механізми поставки даних з яких збирається чи виконується якась взаємодія.

3.4 Інфраструктура системи

Для швидкого розгортання системи було використано декілька сервісів з хмарного сервісу AWS [17]. Усі компоненти мають свій простір виконання, де вони працюють. Сам сервіс airflow та його системні компоненти розміщені на одному AWS EC2 рішенні, це рішення є дуже простим та зручним, але має деякі недоліки, а саме керування для динамічного розгортання додаткових компонентів, якщо систему потрібно буде розширяти по горизонталі. Простір для Kubernetes [7] контейнерів було обрано для того щоб використовувати AWS EKS [17], це пакет

послуг який є дуже простим та гнучким інструментом для роботи з Kubernetes кластером (див. рис. 3.4).

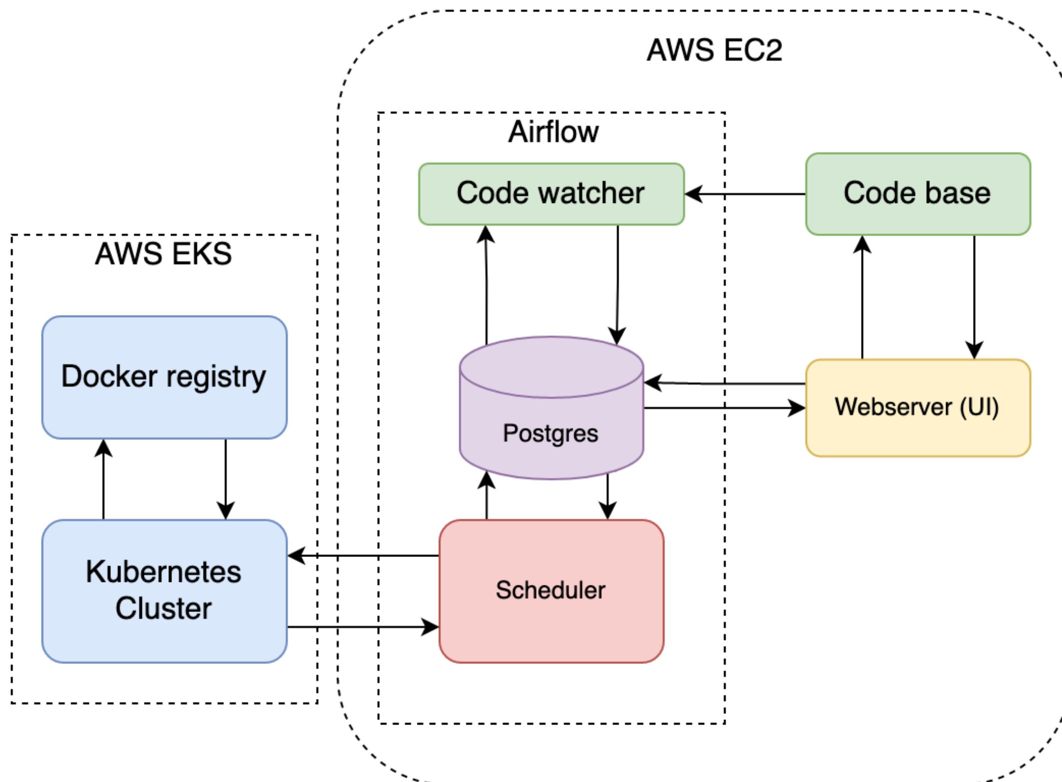


Рисунок 3.4 - Інфраструктура компонентів airflow

Так на рисунку ми бачимо, що усі системні компоненти виконуються у просторі AWS EC2, а контейнери [18] у просторі AWS EKS.

3.5 Взаємодія внутрішніх компонентів airflow

Apache Airflow має внутрішню структуру взаємодії компонентів, частину яких було показано раніше. Детальний аналіз документації та структури airflow показує, що в нього є основні компоненти які виконують різні цілі, для початку подивіться схему компонентів для подальшого роз'яснення (див. рис. 3.5).

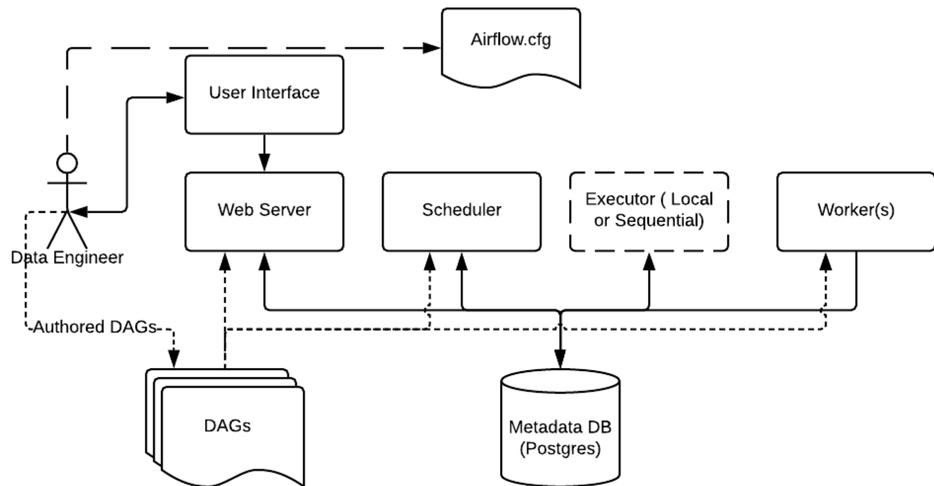


Рисунок 3.5 - Схема взаємодії компонентів airflow

Слід виділити такі компоненти як: WebServer, Scheduler, Executor, Worker, DAGs, Metadatabase. Кожний компонент виконується у окремому процессорному просторі (процес або потік), та має гнучку взаємодію з іншими компонентами, що дозволяє при необхідності розгорнути компоненти на різних пристроях.

DAGs - це компонент, зазвичай це звичайна директорія у коді проекта, на ній рекомендовано використовувати систему контролю версія для того щоб мати можливість виправити порушення в роботі системи, або подивитись історію змін коду кожного дагу, якщо над системою задач працюють більше одного розробника (див. рис. 3.6).

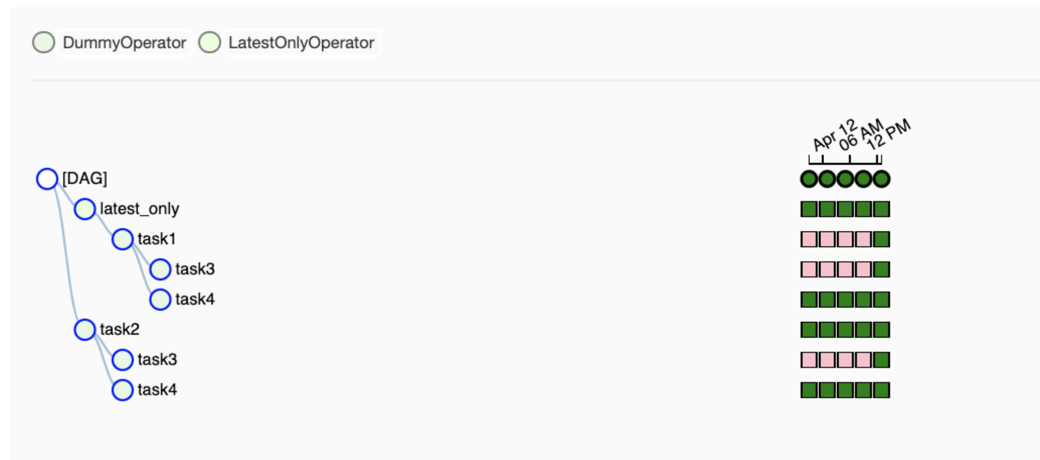


Рисунок 3.6 - Граф дагу в airflow

Але якщо потребуються нестандартні рішення зберігання коду завдань то можно завантажувати їх динамічно в глобальний простір інтерпретатора Python [6], він автоматично появиться в системі, але такий спосіб використовувати не рекомендується. Зазвичай вони виглядають у вигляді графу в той системі що використовуються (див. рис. 3.7).

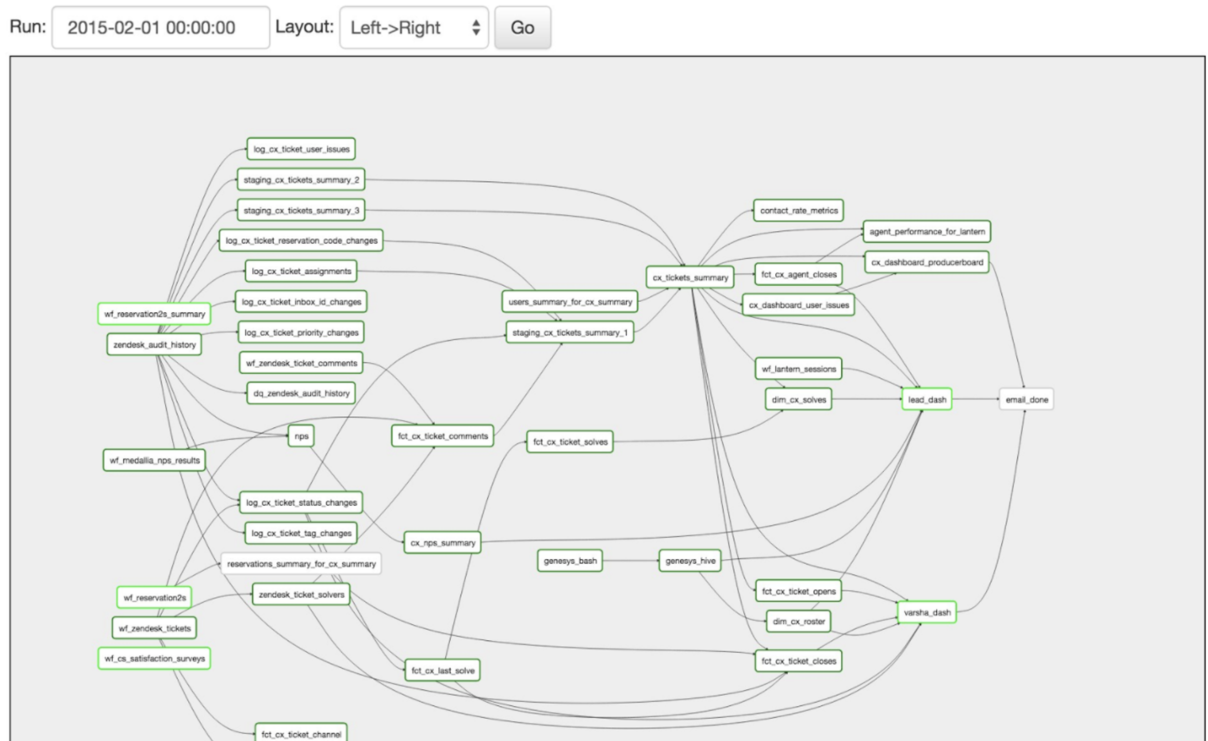


Рисунок 3.7 - Відображення багатьох зв'язків між задачами

Metadata DB - це компонент, що зберігає системну та інколи чутливу інформацію для всій системи airflow, у ньому ж зберігається інформація про логи та час запуску та інша інформація по дагам та таскам [19]. Це сховище потребує детальної уваги з боку DevOps команди, потрібно забезпечити резервне копіювання даних, окреме ізольоване місце виконання, та контроль доступу до нього. Окрім цього цей компонент потребує додаткової автоматизації, через те що зберігає доволі довгу історію логів та інформацію для обміну між задачами, що з часом займе багато дискового простору, рекомендується розробити скрипт, що буде видаляти інформацію біль ніж останні три місяці. Також потребується моніторинг або нотифікатор, що сповіст о критичних станах системи. Якщо

слідувати всім рекомендаціям, то робота цього компонента буде дуже стабільна та майже не займати уваги DevOps команди, або інших розробників (див. рис. 3.8).

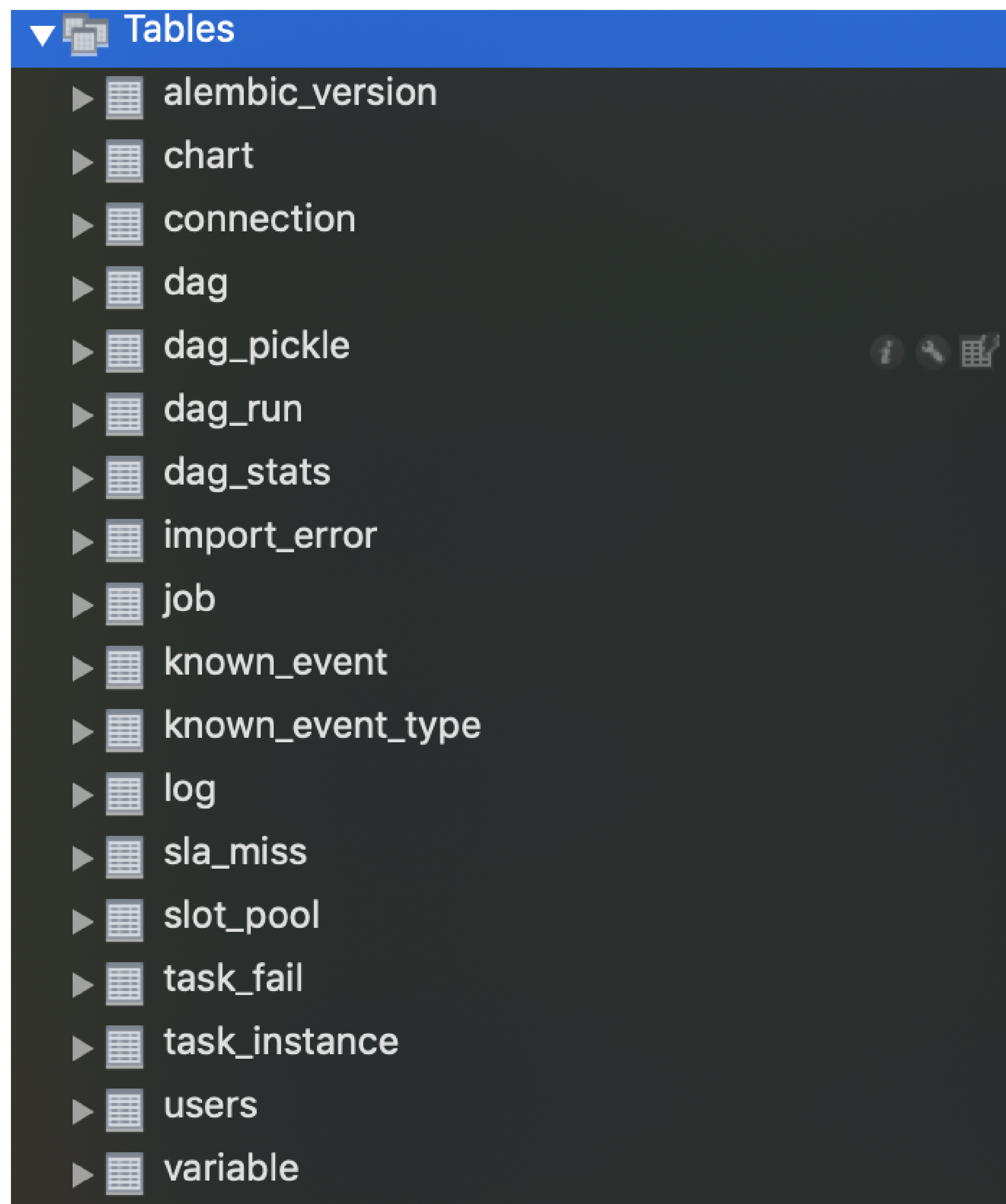


Рисунок 3.8 - Структура службової бази для Airflow

Scheduler - це компонент, що реалізує cron подібний функціонал для всій системі airflow, також у нього входить реалізація модуля, що сканує папку DAGs, на предмет дагів (нових або оновлених) час від часу (приблизно 1-2 кожні 5 секунд), після цього записує інформацію у свою базу даних Metadata DB. Рекомендується слідкувати за процесорним часом що займає цей компонент, це багатопроцесорний компонент, що робить його доволі гнучким, але чим більше

задач за одиницю потрібно виконувати тим більше ресурсів потрібно Scheduler для того щоб за ним слідкувати, тому що саме він зчитує логі з запущених задач та зберігає їх у Metadata DB. Також він відповідальний за моніторинг статусу завдань, тобто якщо задача завершилась з якимость статусом, переслідують наступні заплановані дії, з яких сповіщення та запис стану задачі у базу даних (див. рис. 3.9).

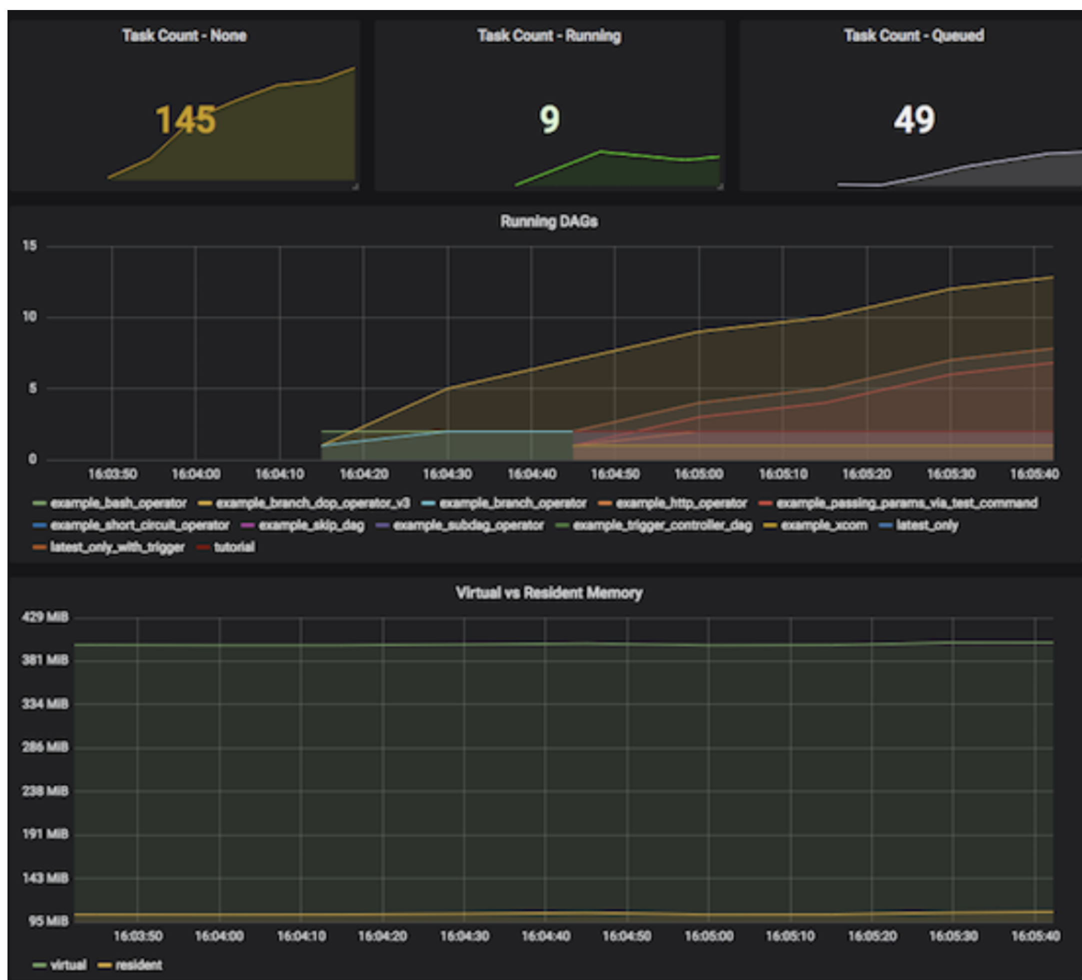


Рисунок 3.9 - Моніторинг за Scheduler та дагами

WebServer - це компонент airflow, що відображає існуючий функціонал взаємодії з іншими компонентами системи у зручному графічному вигляді. В самому airflow він налагоджений та працює через фреймворк flask, завдяки цьому веб-фреймворку реалізована API та UI частина для взаємодією з інтерфейсом airflow. Він не потребує моніторингу або нагляду бо кількість користувачів

зазвичай не перевищує розміру команди розробників. Повністю автономний модуль, що можно підключити та просто користуватись, не думаючи про щось ще. Приклад інтерфейса airflow (див. рис. 3.10). На головній сторінці після логіну відображається таблиця дагів яка дозволяє швидко шукати та просматривать задачі, що виконуються та їх статус. Також з верхньої панелі доступні такі функції розробника як “змінні” та “з'єднання”, вони доступні у вкладці “Адмін”. Це дозволяє налагоджувати ті параметри системи, що можуть бути доступні усім задачам одночасно або частично. Ці параметри записуються, зберігаються та зчитуються через Metadata DB.

Для перегляду логів слід натиснути на один з дагів, перейти у граф виконання, натиснути на потрібний крок задачі та вибрати відображення логів.

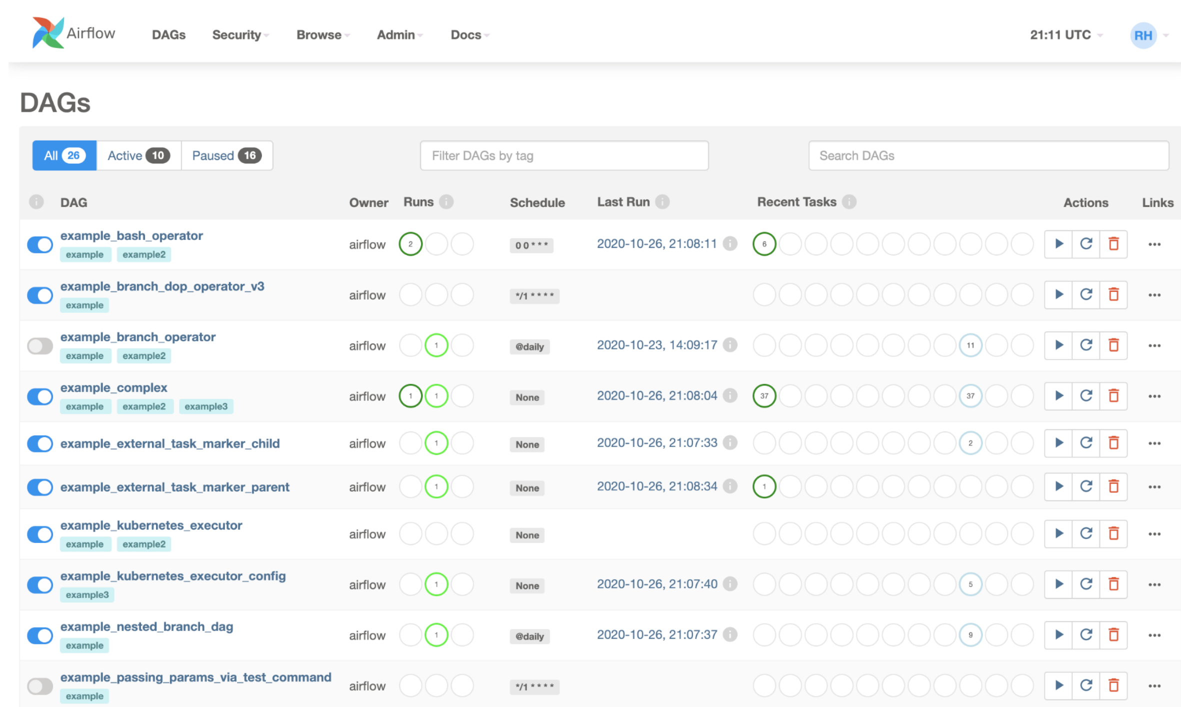


Рисунок 3.10 - Інтерфейс airflow webserver

Executor - це компонент системи який відповідає за простір виконання завдань, цей компонент може бути модифікован та адаптован під потреби бізнесу. Зазвичай використовують конкретні реалізації, такі як: Celery Executor, Kubernetes

Executor. В цїї роботі буде використан Kubernetes Executor. Взаємодія внутрішній компонентів airflow (див. рис. 3.11).

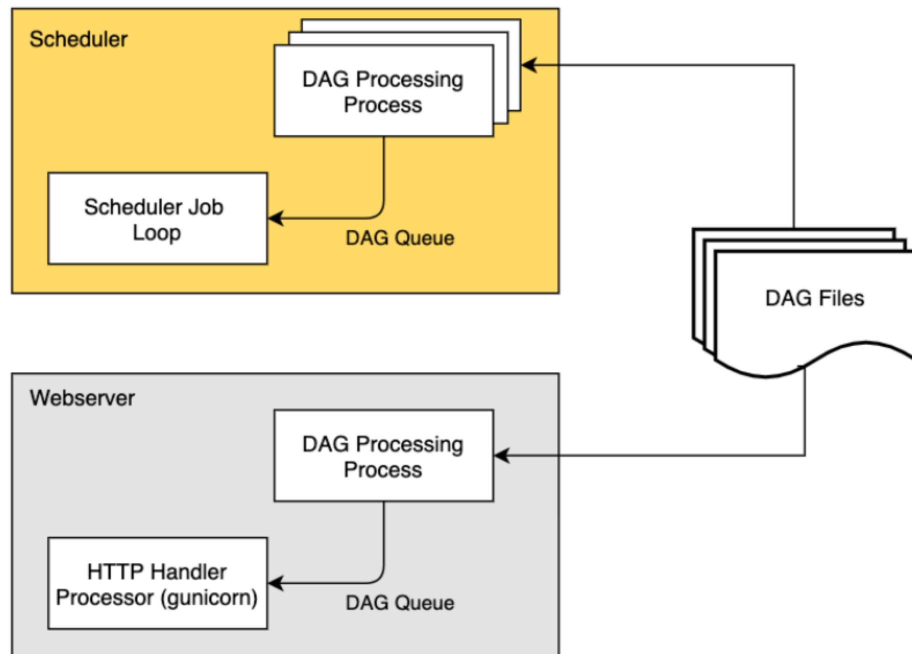


Рисунок 3.11 - Взаємодія внутрішній компонентів airflow

Worker - це компонент, який відповідає за один з вузлів виконання Executor, це верхня абстракція, що дозволяє слідкувати за процесом виконання завдання.

Розробник найчастіше взаємодіє з WebServer, та DAGs, тому їх налагодити потрібно в першу чергу.

3.6 Архітектура компонентів у кластері

Окрім того що сам Apache Airflow не вимогливий до машини на якій буде працювати, мінімальні вимоги для нього існують, а саме потребується машина з оперативною пам'яттю у 8 гігабайт та 4-й ядерний процесор. У цих вимог є дуже логічне обгрунтування, а саме те,що окрім Apache Airflow на ньому будуть використовуватись інші системи що будуть моніторити стан та логи самой системи та відслідковувати аномальні чи ризикові стани системи, кількість ядрів

потрібна для зручного та швидкого використання компоненту Scheduler. Слід також уточнити, що самі компоненти можуть бути використані окремо на різних машинах, що дозволяє виконувати сам компонент Apache Airflow як один із контейнерів в кластері Kubernetes [7]. Архітектуру використання Airflow у Kubernetes (див. рис. 3.12).

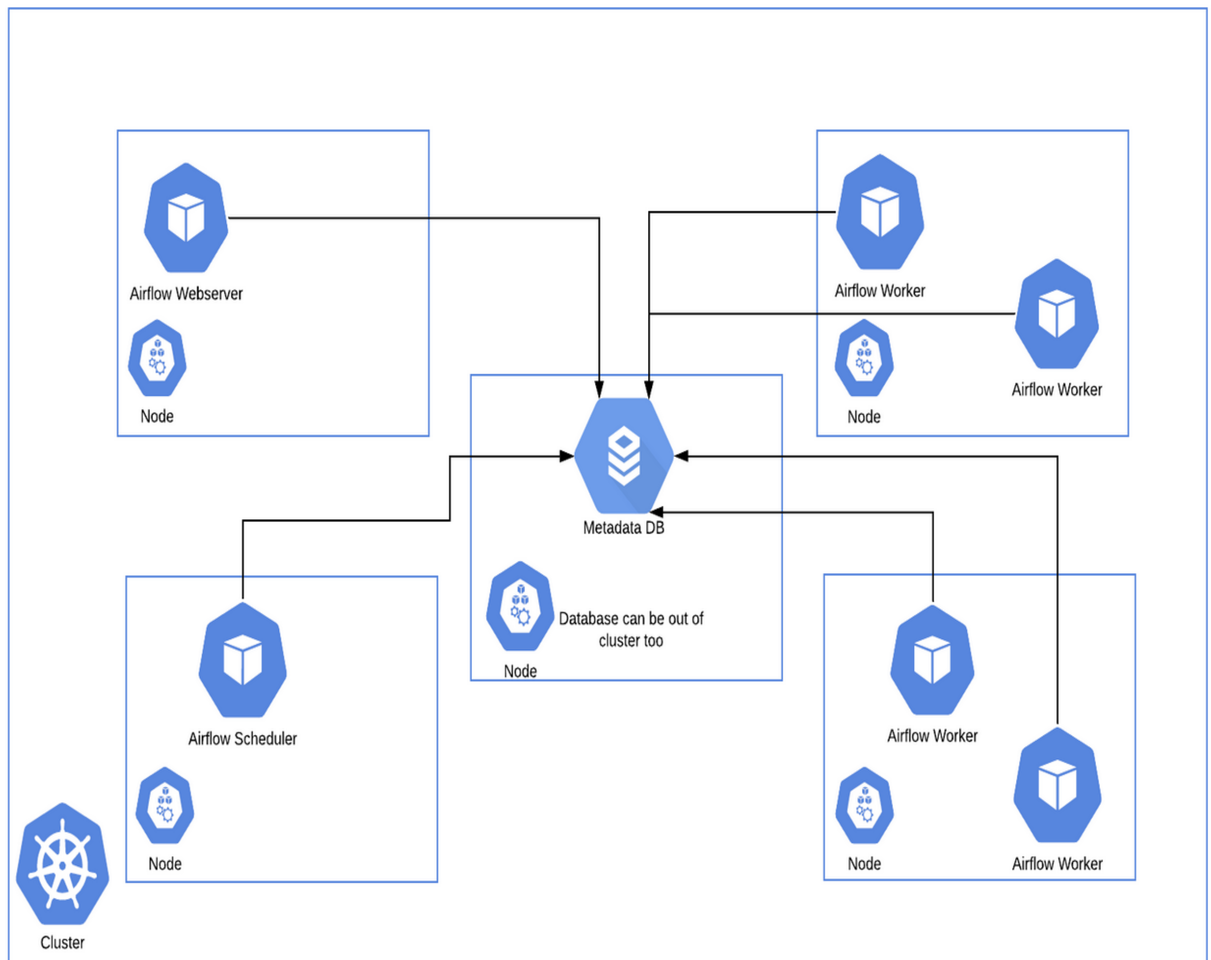


Рисунок 3.13 - Архітектура Airflow розгорнутої у Kubernetes

Як бачимо кожний компонент реалізовано контейнером, окрім того він виконується на окремій групі машин, що називаються нодами (вузлами), кожен компонент може бути горизонтально у разі необхідності масштабов, у цьому разі робота компонента більш стабільна та у разі збоїв одного фрагмента буде виконуватись інший. Усі компоненти під’єднані до однієї системи Metadata DB,

що теж розташована в окремій групі контейнерів на вузлу. Схема взаємодії компонентів (див. рис. 3.14).

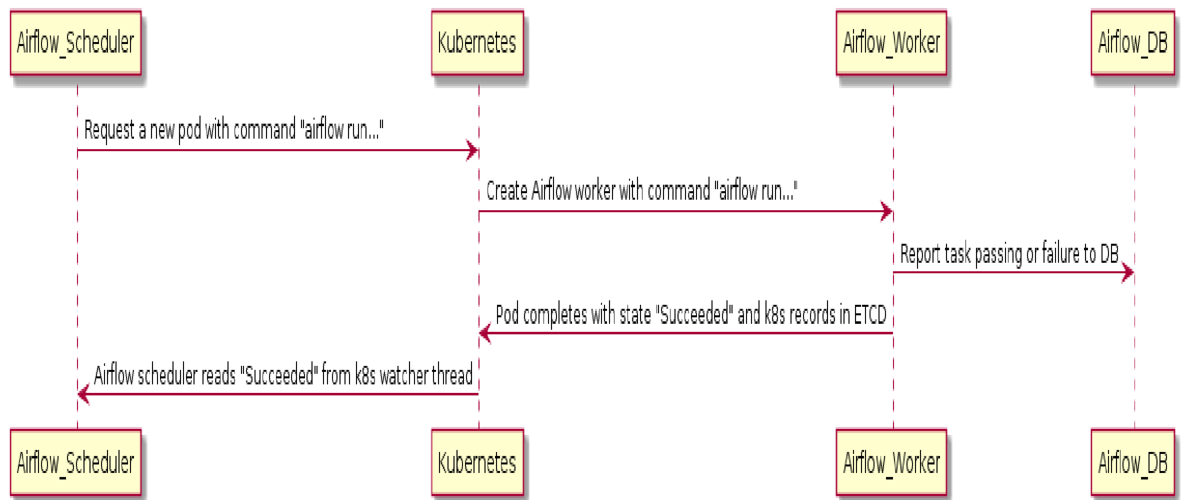


Рисунок 3.14 - Цикл життя команди у системі

Як бачимо, спочатку Scheduler відправляє команду на запуск ноди у Kubernetes, затим Kubernetes відправляє команду до Airflow Worker, після виконання результату, статус завершення та додаткова інформація буде записана до Metadata DB, після цього результат відправляється обратно до Airflow Worker, затим до Kubernetes та Airflow Scheduler. У разі невиконання завдання цикл спрощений (див. рис. 3.15).

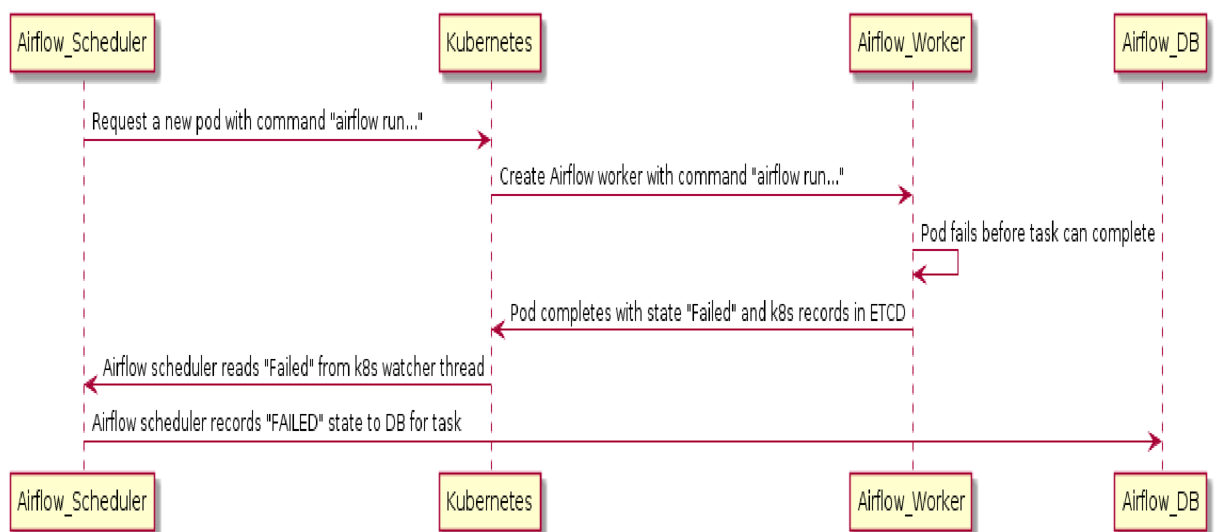


Рисунок 3.15 - Цикл команди у разі невиконання задачі

У разі невиконання результат у базу Metadata DB не записується.

Для роботи систем логування та моніторингу компоненти системи будуть виглядати наступним чином (див. рис. 3.16).

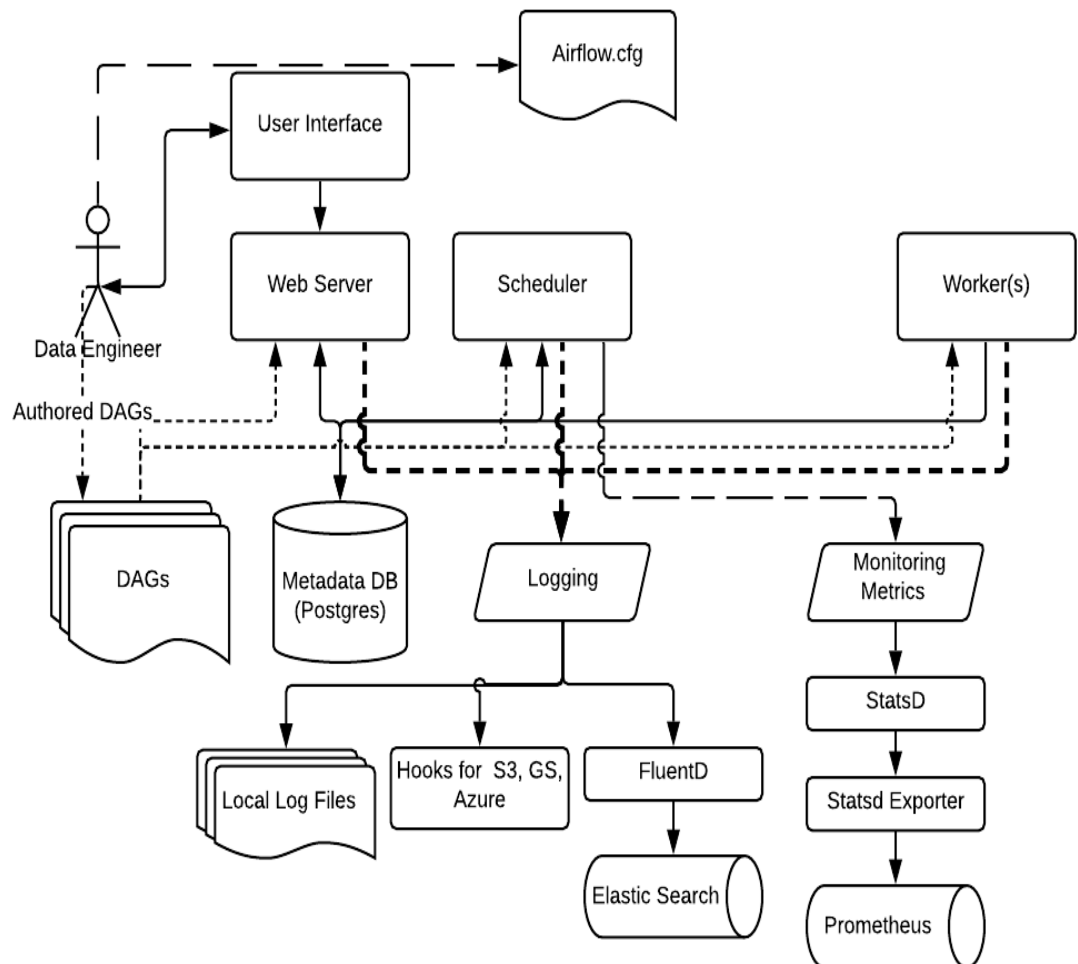


Рисунок 3.16 - Компоненти моніторингу та логу

Кожний компонент має виходи та тимчасові сховища [19], де зберігаються показники та історія виконання задач, запусків, та деякі дії які виконує користувач. Для цього в Apache Airflow э гнучкі системи адаптації для збору показників та логів, завдяки спеціалізованим сховищем в AWS, такім як S3, Prometheus, Elasticsearch можливо зібрати систему з постійно поновлюючих графіки за моніторингом всієї системи.

Немаловажним чином відіграє роль інструменти, що можуть просканувати готовність зовнішніх та внутрішніх показників на готовність для виконання ланцюгу завдань, так наприклад один із модулей під назвою “sensor”, реалізован саме для цих цілей. Такий класс жде написаною розробником ситуацію в якій доцільно запустити на виконання задачу, схему у виконанні яку виконує сенсор (див. рис. 3.17).

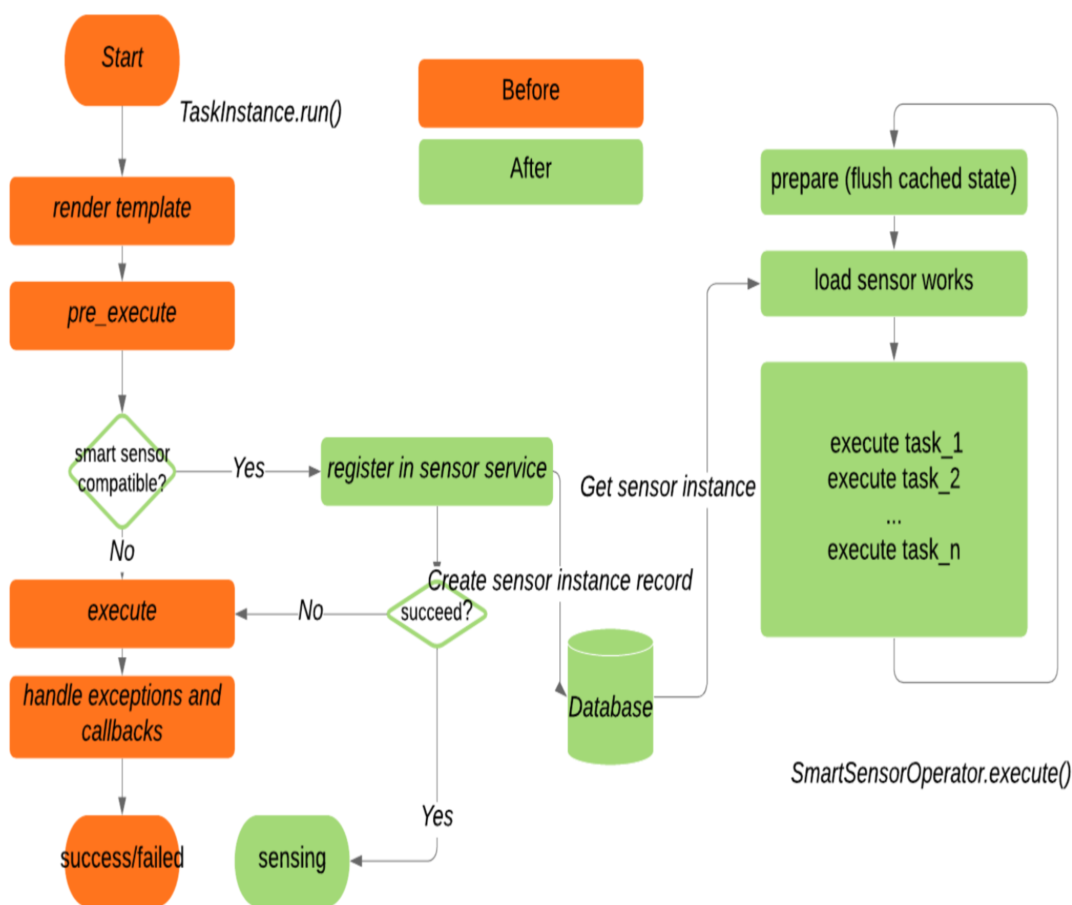


Рисунок 3.17 - Схема використання сенсорів у Airflow Dag

Як бачимо, за схемою перед самим виконанням задачі, якщо в дагу є сенсор то спочатку йде перевірка на те що всі умови, які були задані користувачем чи розробником досягнуті, якщо умови за встановлений час не були знайдені, то виконання дагу завершується без помилки. Якщо умови були досягнуті, але задача не була виконана програмно, то завершується с викликом помилки та відправляється на перезапуск за встановлений час.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1 Структура коду проекту

Так як сама система cron-завдань потребує реалізації, то слід визначити та декларувати деякі положення для розробників які будуть використовувати систему. Корінь проекту airflow (див. рис. 4.1).

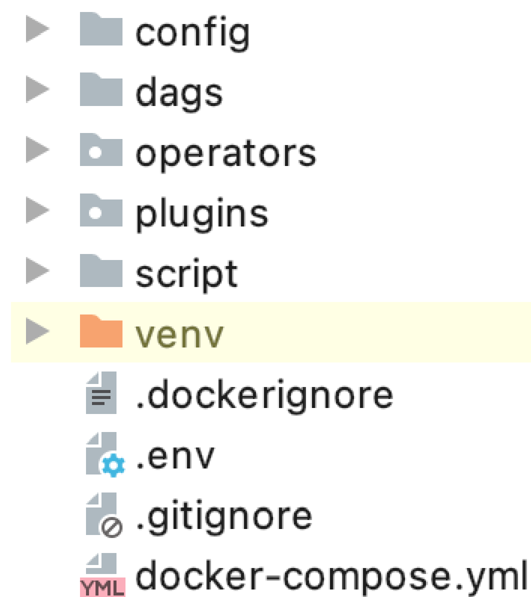


Рисунок 4.1 - Структура проекту airflow

Крім багатьох системних файлів та папок є такі що, потребують уваги. А саме:

– dags;
– plugins;
– config;
– operators;
– script.

Папка plugins - реалізує спеціальні компоненти системи для взаємодії з іншими системами, та в загалом для розширення функції, що може виконувати система. Airflow під’єднную цей модуль під час запуску та він доступний для використання як і звичайний Python модуль.

Папка dags - у цьому місці зібрані всі задачі та інструкції до їх виконання в особливому вигляді, зокрема за допомогою airflow системі та cron-нотації. Назва цієї папки теж має значення, а саме Directed Acyclic Graph, тобто направлений ациклічний граф. Це значить, що при розробці та планування автоматизованих задач потрібно слідкувати за тим, щоб задачі мали чітку послідовність та не возникало циклічних зв’язків між задачами, бо такі задачі можуть ніколи не завершитись. Приклад вигляду задач за dag принципом (див. рис. 4.2).

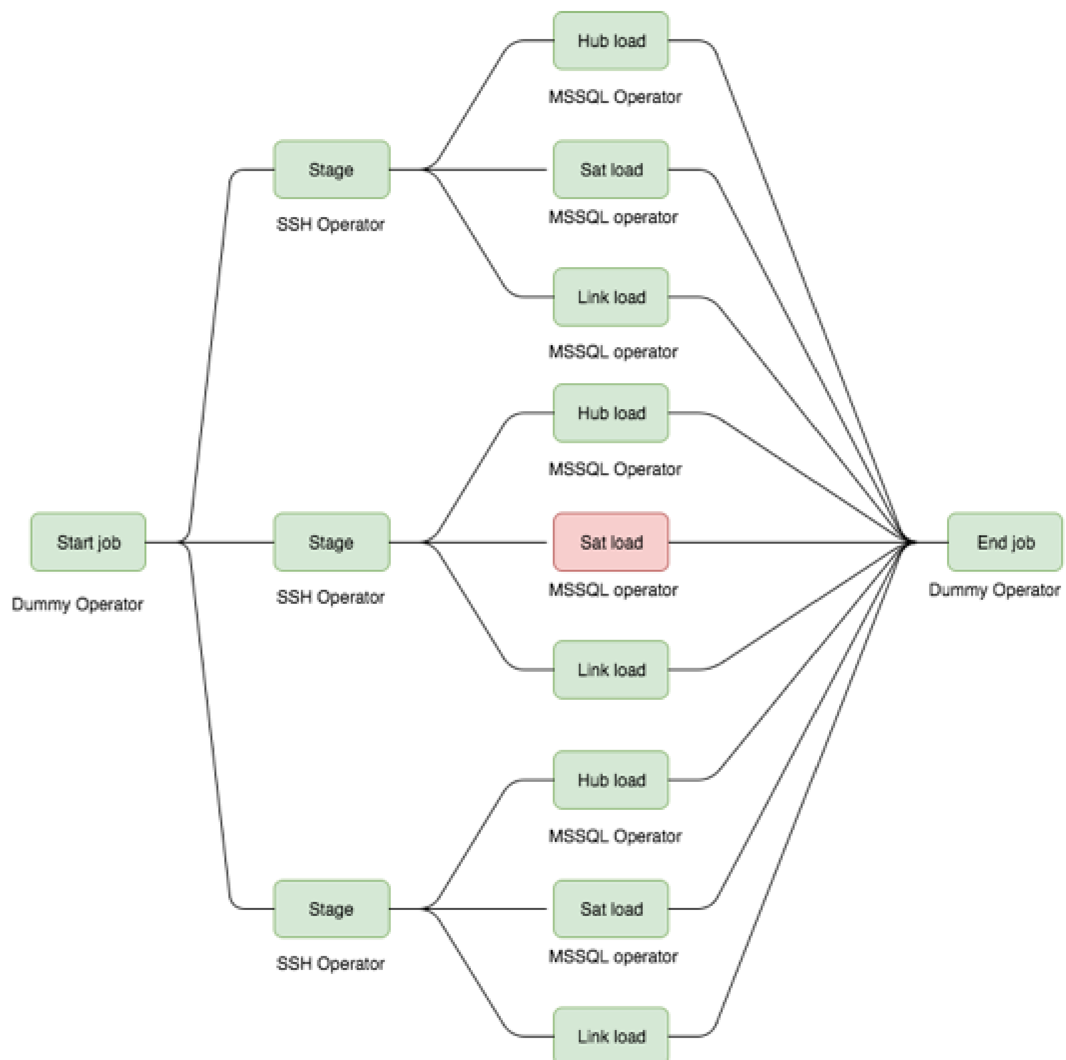


Рисунок 4.2 - Послідовність задач у airflow dag

Папка config - це місце у якому зібрані усі конфігурації до системи airflow, зокрема як будуть зберігатися, записувати та зчитувати логи системи, а також

підключення до службової бази даних, а також шаблони з яких будуть створюватися конфигурации з чутливою інформацією.

Папка script - це місце для системних та службових файлів які потребує система для единоразового використовування у разі першого налагодження системи на будь якої машині. Наприклад:

– створення користувача у службовій базі даних;
– перевірка цілісності системи
– запит до логів та систем моніторингу.
– безпечне видалення кешу та тимчасових файлі.
– запуск у тестовому режимі усієї системи.
– запуск у режимі debug усієї системи.

Папка operators - це найважливіший компонент, в ньому зберігається реалізація завдань які будуть запущені самою системою. Цей компонент має подвійну структуру, як і у багатьох проектів є частини, що використовуються усіма її частинами, тому вони виносятся на рівень вище, для них більша увага у разі змінення та вимоги для проходження тестів. Тобто перший (верхній рівень) цього компоненти це базові пакети та модулі які будуть використовувати всі учасники проекту. Другий рівень (тобто ніжний рівень), це рівень прикладної розробки, напряму зв'язаний з бізнес логікою. Розробники системи найчастіше працюють на цьому рівні. Між верхнім та нижнім рівнем розробки також є логіка, нижчі рівні, у подальшому модулі групуються за ціллю або способом виконання. Так виникає локальні рівні, що використовуються групою якийсь модулів, це дозволяє розділити модулі, та зробити їх більш незалежними для розробників, адже чим менше потрібно пам'ятати та концентруватися на не бізнес задачах тим краще і швидше буде виконана робота розробника.

4.2 Реалізація cron-завдань у системі airflow

За допомогою системі плагінів було реалізовано зручну систему операторів, що зберегли час розробника та користувачів сістеми. Так було реалізовано

зручний класс оператора для роботи з Kubernetes. Частина реалізації якого наведено (див. рис. 4.3). Airflow самостійно після запуску під'єднує всі додаткові реалізації класів, після завантаження вони доступни для імпорту як один із модулей Airflow, а саме airflow.operators.module_name, де module_name це назва модуля який був розташований в директорії з плагінами.

```
try:
    (final_state, result) = launcher.run_pod(
        pod,
        startup_timeout=self.startup_timeout_seconds,
        get_logs=self.get_logs)
finally:
    if self.is_delete_operator_pod:
        launcher.delete_pod(pod)

if final_state != State.SUCCESS:
    raise AirflowException(
        'Pod returned a failure: {state}'.format(state=final_state)
    )
if self.do_xcom_push:
    return result
```

Рисунок 4.3 - Частина реалізації оператора Kubernetes

Приклад використання реалізації нового оператору (див. рис. 4.4).

```
def generate_pod_task(task_id, parameters, **kwargs):
    return PodOperator(
        task_id=task_id,
        image_name="task_v1",
        parameters=parameters,
        volume=[volume],
        volume_mount=[volume_mount],
        **kwargs
    )
```

Рисунок 4.4 - Використання нового оператору для Kubernetes

На основі цієї функції буде реалізовані декілька десятків airflow дагів. Приклад реалізації airflow дага (див. рис. 4.5).

```
with DAG(dag_id='dag_first',
         catchup=False,
         default_args=default_args,
         schedule_interval='0 */15 * * *',
         max_active_runs=1) as dag_first:
    load_data_task = generate_pod_task("load_data", ['db_connection'])

    calc_data_task = generate_pod_task("calc_data", ['etl_db_connection'])

    load_data_task >> calc_data_task
```

Рисунок 4.5 - Приклад реалізації airflow даг

Як бачимо, даг є компонентом одного з Python модулів, що має на вхід назву яка буде ідентифікувати його в системі, ця назва повинна бути унікальною в системі. Cron нотація використовується щоб зазначити коли потрібно запускати dag, а саме в параметр schedule_interval, також можуть бути вказані запобіжні параметри, наприклад max_active_runs, що не дозволяє дагу дублікуватись у часі, та тільки один dag має змогу працювати в одиницю часу. Після цього у просторі dag, генеруються нові задачі та зв'язуються за допомогою символа “>>” (див. рис. 4.6).

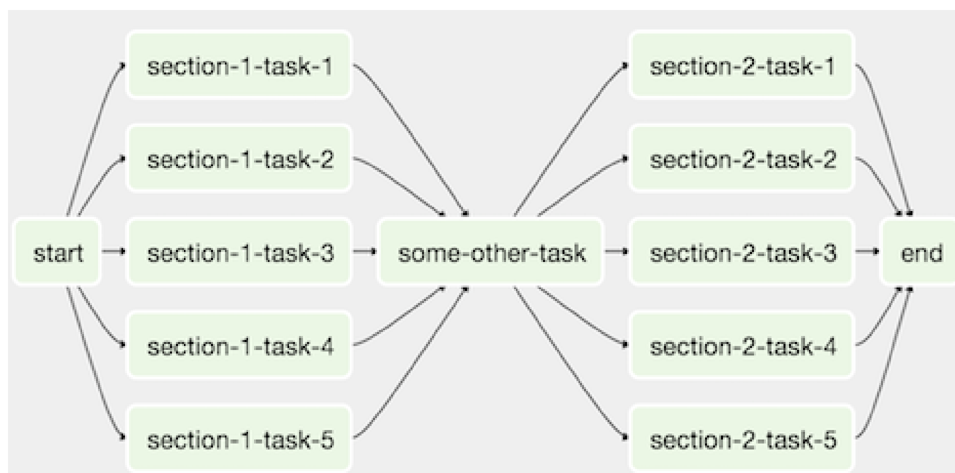


Рисунок 4.6 - Вигляд дагу в airflow UI

Даг також приймає параметри по замовчуванню. В загалом це система конфігурація того як перезапускати даг, кого сповіщати о невдалого запуску дага. Та як довго даг може працювати. Приклад такої конфігурацію наведено (див. рис. 4.7).

```
default_args: Dict[str, Any] = {
    'owner': 'kirill',
    'email': ['kirill.afendi@nure.ua'],
    'email_on_failure': False,
    'email_on_retry': False,
    'start_date': datetime(2021, 6, 13),
}
```

Рисунок 4.7 - Параметри дага по замовчуванню

Також написаний для Kubernetes оператор може приймати ресурси, а точніше потребуємо кількість ресурсів, у вигляді відсотка від процессору, та пам'яті у мегабайтах. Формат ресурсів наведено (див. рис. 4.8).

```
default_pod_task_resource = dict(
    request_cpu='210m',
    request_memory='236Mi',
    limit_cpu='404m',
    limit_memory='522Mi'
)
```

Рисунок 4.8 - Ресурси для Kubernetes задачі

Задання ресурсів поділяється на оперативну пам’ять та CPU, вказується граничні показники споживання цих параметрів системи.

4.3 Парсер для airflow системи

Як раніше зазначалось, парсер - це компонент який взаємодіє з системою cron-задач, дістає задачі та інформацію про них, що буде корисна для подальшого аналізу та оптимізації. В загалом не так багато можно взяти з cron систем, в дуже примітивному варіанті вказують тільки час запуску, у найкращому випадку можно дістати середній час виконання задачі, кількість потрібних ресурсів, кількість пропусків у разі невдалого запуску. Структура проекта парсера (див. рис. 4.9).

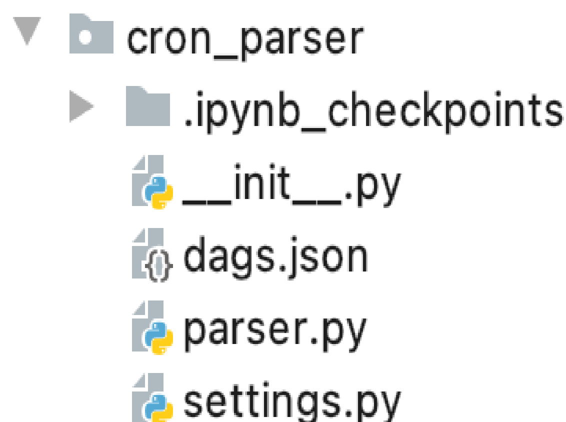


Рисунок 4.9 - Структура файлів парсера

Проєкт має стандартну структуру Python модулю, що дозволяє його без додаткових маніпуляції використовувати у інших проектів.

Реалізація парсеру заснована на роботі класичної системи airflow, а точніше її компоненту DagBag, цей компонент дозволяє сканувати директорії та збирати всю інформацію про задачі з системи.

Файл parser.py - відповадає за збір данних з системи, путь до системи зазначається у файлі settings.py. Результат зберігається у dags.json файлі, де зібрані всі даги з системи airflow. Приклад цієї структури (див. рис. 4.10).

Як бачимо, це JSON формат, у якому ключ відповідає номеру дагу, а значення відповідають інформацію, що з нього вдалось зібрати.

```
"1": {
    "schedule_interval": "*/3 * * * *",
    "tasks": {
        "0": {
            "execution_timeout": 180,
            "job_operator_type": "PodOperator",
            "resources": {
                "limit_cpu": "300m",
                "limit_memory": "512Mi",
                "request_cpu": "150m",
                "request_memory": "256Mi"
            }
        }
    }
},
```

Рисунок 4.10 - Структура результату роботи парсеру

Система працює таким чином, щоб зібрати всю доступну інформацію по cron-задачам. Процесс обробки файлів (див. рис. 4.11).

```
- 1_dag - */3 * * * *
---_job_0_task(PodOperator): cpu: 150m, memory: 256Mi - 0:03:00

- 2_dag - 0 */8 * * *
---_job_0_task(PodOperator): cpu: 150m, memory: 256Mi - 0:15:00

- 3_dag - 0 3 * * *
---_job_0_task(PodOperator): cpu: 200m, memory: 256Mi - 2:00:00
---_job_1_task(PodOperator): cpu: 200m, memory: 256Mi - 2:00:00

- 4_dag - 30 3 * * *
---_job_0_task(PodOperator): cpu: 200m, memory: 256Mi - 2:00:00

- 5_dag - */5 * * * *
---_job_0_task(PodOperator): cpu: 150m, memory: 256Mi - 2:00:00

- 6_dag - 0 10 * * *
---_job_0_task(PodOperator): cpu: 100m, memory: 256Mi - 0:10:00
---_job_1_task(PodOperator): cpu: 100m, memory: 256Mi - 0:10:00
```

Рисунок 4.11 - Процесс парсингу airflow системи

Пропуски між записами показують кінець одного дага та початок іншого, наступний рівень показує послідовність визову задач в кластері.

4.4 Аналізатор та оптимізатор розкладу

Як зазначалось в теоретичній частині роботі, перед оптимізацією чи аналізом розкладу cron завдання трансформуються у класичну модель теорії розкладів, де конкретно вказується час запуску задачі. Для таких траформації ми будемо використовувати Python бібліотеку croniter, фрагмент коду для трансформації cron-нотації (див. рис. 4.12).

```
def make_runs_calc(dags, start_date: datetime, end_date: datetime):
    def make_records_untill(dag, start_date, end_date):
        cron_expr = dag['schedule_interval']
        citer = croniter(cron_expr, start_date)
        cur_date = start_date
        while cur_date < end_date:
            cur_date: datetime = citer.get_next(datetime)
            yield cur_date, dag
    for i, dag in dags:
        for date, d in make_records_untill(dag, start_date, end_date):
            yield i, date, d
```

Рисунок 4.12 - Фрагмент коду трансформації cron-виразів

Наступним кроком є групування результату трансформації за частотою запусків. Для цього буде використана вже згадана бібліотека croniter, для цього ми вираховуємо час між наступними двома запусками. Для прикладу будуть знайдені всі задачі, що запускаються частіше ніж один раз на час, фрагмент коду (див. рис. 4.13).

```
def get_dags_by_timedelta_run(dags, td_min: timedelta, td_max: timedelta):
    date = datetime.now()
    for i, dag in dags.items():
        cron_expr = dag['schedule_interval']
        citer = croniter(cron_expr, date)
        first_next: datetime = citer.get_next(datetime)
        second_next: datetime = citer.get_next(datetime)
        dag_td = second_next - first_next
        if td_min <= dag_td < td_max:
            yield i, dag, dag_td
```

Рисунок 4.13 - Функція для групування за часом

Результати роботи цієї функції вже були презентовані (див. рис. 2.3). Аналізатор графично відобразить у вигляді списка та графіка нагруженість за часом (див. рис. 4.14).

```
2021-04-22 02:42:00 - 3
2021-04-22 02:44:00 - 1
2021-04-22 02:45:00 - 11
2021-04-22 02:46:00 - 1
2021-04-22 02:48:00 - 2
2021-04-22 02:49:00 - 1
2021-04-22 02:50:00 - 6
2021-04-22 02:51:00 - 1
2021-04-22 02:52:00 - 1
2021-04-22 02:54:00 - 2
2021-04-22 02:55:00 - 5
```

Рисунок 4.14 - Кількість задач на точку часу

Як бачимо, навіть за списком наглядно видно не збалансованість. Для наглядності виведемо графік, що буде відображати [20] цей самий список тільки для зручності у графічному вигляді (див. рис. 4.15).

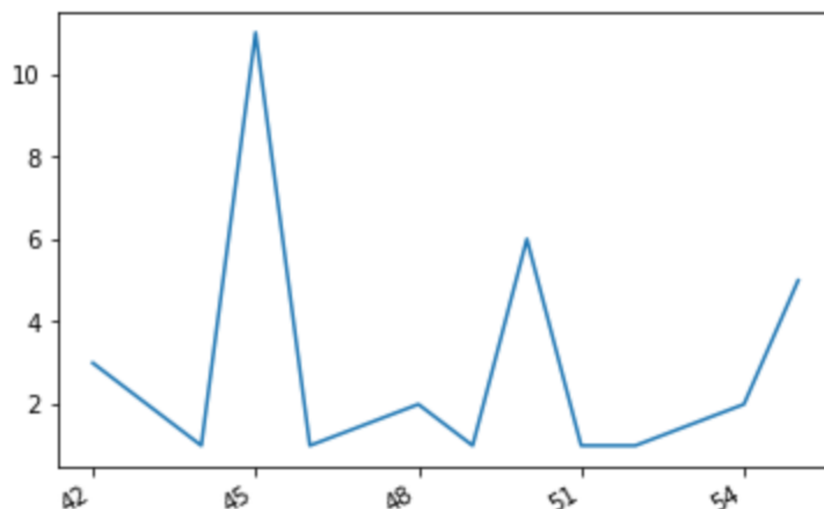


Рисунок 4.15 - Розподіл Задач за часом.

Теперь э все для пропуску трансформованої групи даних для оптимізації. Як було описано в пункті другого розділу 2.5, головний цикл функцій на мові Python [21] представлен (див. рис. 4.16).

```
def optimaze(dags):
    i = 100
    dates = [d for d in dags.keys()]
    first, last = dates[0], dates[-1]
    while i > 0:
        cur_max_indx = search_max_num_tasks(dags)
        print(f"Found = {len(dags[cur_max_indx])}")
        dags = optimaze_empty_task_time(dags, cur_max_indx, first, last)
        dags = base_optimaze_task_time(dags, cur_max_indx)
        i -= 1
    return dags
```

Рисунок 4.16 - Головний цикл оптимізації

Функція для пошуку часу з максимальною кількістю задач (див. рис. 4.17). Як бачимо, функція приймає на вхід даги, що розподілені за часом, перебираючи усі елементи точок часу знаходиться така що має малу кількість задач, та змінна що дорівнює останній мінімальної кількості задач буде змінена на новий показник, і так поки не дійде до кінця.

```
def search_max_num_tasks(dags):
    cur_len_v = 0
    indx = None
    for date, v in dags.items():
        if cur_len_v < len(v):
            cur_len_v = len(v)
            indx = date
    return indx
```

Рисунок 4.17 - Пошук точку часу за максимальної кількістю задач

Після знаходження точки з найбільшою кількістю задач, індекс точки передається у наступні функції, які знаходять можливі сусідні точки, що є пустими. Код пошуку пустих сусідніх точок (див. рис. 4.18).

```
def search_epmty_time_places(dags, indx: datetime, dag_id, dag_info):
    offset_space = dag_info['offset_available']
    cur_len_indx = len(dags[indx])
    for i in [x for x in range(-1 * offset_space, offset_space + 1) if x != 0]:
        n_indx = indx + timedelta(seconds=60 * i)
        if n_indx not in dags.keys():
            return n_indx
    return None
```

Рисунок 4.18 - Пошук пустих сусідніх точок часу

Наступник кроком є пошук сісудніх точок, на які можно розподілити задачі. Зображено функцію для пошуку таких сусідніх точок (див. рис. 4.19).

```
def base_optimaze_task_time(dags, indx):
    dag_ids = [dag_id for dag_id in dags[indx].keys()]
    for dag_id in dag_ids:
        dag_info = dags[indx][dag_id]
        offset_space = dag_info['offset_available']
        for i in [x for x in range(-1 * offset_space, offset_space + 1) if x != 0]:
            n_indx = indx + timedelta(seconds=60 * i)
            if not dags.get(n_indx, False):
                continue
            if abs(len(dags[n_indx].keys()) - len(dag_ids)) <= 1:
                continue
            if dag_id in dags[n_indx].keys():
                continue
            dags[n_indx][dag_id] = dict()
            to_move = dags[indx].pop(dag_id)
            dags[n_indx][dag_id] = to_move
            break
    return dags
```

Рисунок 4.19 - Пошук та сдвиг задач на сусідні точки

Після завершення роботи алгоритму розклад виглядає наступним чином (див. рис. 4.20).

```
2021-04-22 02:42:00 - 4
2021-04-22 02:44:00 - 3
2021-04-22 02:46:00 - 4
2021-04-22 02:48:00 - 2
2021-04-22 02:49:00 - 4
2021-04-22 02:51:00 - 1
2021-04-22 02:52:00 - 1
2021-04-22 02:54:00 - 3
2021-04-22 02:55:00 - 1
2021-04-22 02:43:00 - 4
2021-04-22 02:47:00 - 4
2021-04-22 02:53:00 - 3
```

Рисунок 4.20 - Результат розподілу задач за часом

У графічному вигляді результат (див. рис. 4.21).

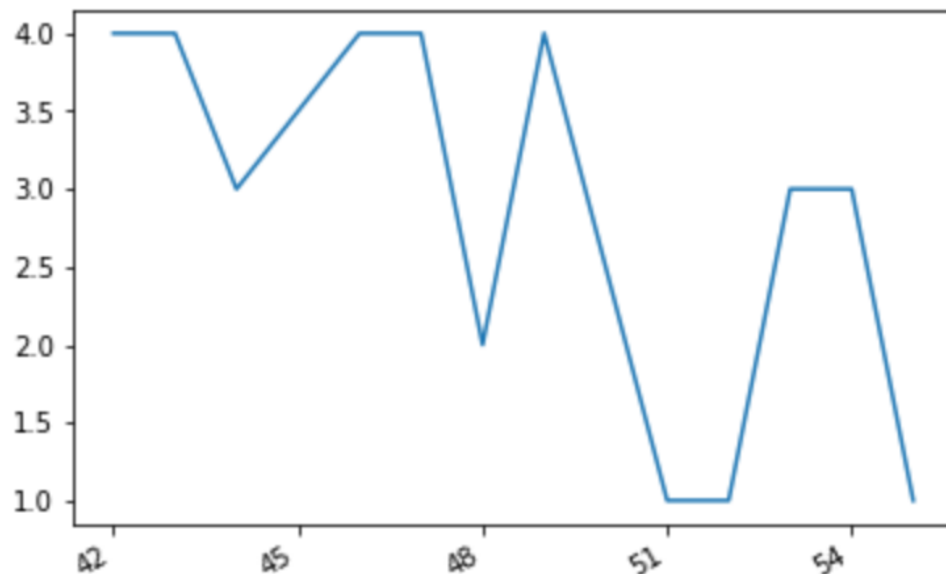


Рисунок 4.21 - Графік розподілу задач за часом

Як видно за графіком, розподіл часу став більш рівномірним, великих різниць в кількісті задач більше немає, для відображення було використано функцію (див. рис. 4.22).

```
def plot_schedule(dags):
    x, y = [], []
    for date, v in sorted(dags.items(), key=lambda v: v[0]):
        x.append(date)
        y.append(len(v))
    fig, ax = plt.subplots()
    fmt_month = mdates.MinuteLocator(interval=3)
    ax.xaxis.set_major_locator(fmt_month)
    ax.xaxis.set_major_formatter(mdates.DateFormatter('%M'))
    ax.plot(x, y)
    fig.autofmt_xdate()
```

Рисунок 4.22 - Функція для відображення графіку

Реалізація алгоритму для оптимізації за використанням CPU не потребує великих змін в алгоритмі, достатньо змінити функцію оцінки точки часу, за реалізації вона знаходиться у функції пошуку максимально навантаженої точкі часу. Приведен алгоритм для оцінки функції за CPU (див. рис. 4.23).

```
def search_max_num_tasks(dags):
    def dag_time_cpu(dags):
        for i, v in dags.items():
            for i, task in v['tasks'].items():
                yield task['resources']['request_cpu']
    cur_len_v = 0
    indx = None
    for date, v in dags.items():
        cpu_load = sum(dag_time_cpu(v))
        if cur_len_v < cpu_load:
            cur_len_v = cpu_load
            indx = date
    return indx
```

Рисунок 4.23 - Функкія пошуку максимально навантажень на точку часу

Функкія пошуку максимально навантажень CPU на точку часу проходить всі точкам часу, після цього вираховує для кожного дагу, в якому слід скласти використання всіх задач за одним показником. Як бачимо, функція приймає на вхід даги, що розподілені за часом, перебираючи усі елементи точок часу знаходиться така що має максимальну навантаженість задачами за показником CPU, та змінна що дорівнює останній мінімальної кількості задач буде змінена на новий показник, і так поки не дійде до кінця.

Приведен результат генерації послідовності часу за використанням CPU (див. рис. 4.24).

```
2021-04-21 02:42:00 - 390m
2021-04-21 02:44:00 - 120m
2021-04-21 02:45:00 - 2538m
2021-04-21 02:46:00 - 120m
2021-04-21 02:48:00 - 270m
2021-04-21 02:49:00 - 120m
2021-04-21 02:50:00 - 870m
2021-04-21 02:51:00 - 150m
2021-04-21 02:52:00 - 120m
2021-04-21 02:54:00 - 270m
2021-04-21 02:55:00 - 750m
2021-04-21 02:56:00 - 240m
2021-04-21 02:57:00 - 150m
2021-04-21 02:58:00 - 120m
2021-04-21 03:00:00 - 2928m
```

Рисунок 4.24 - Розподіл використання CPU за часом

До оптимізації спостерігався вибух використання CPU за однією точкою часу (див. рис. 4.25).

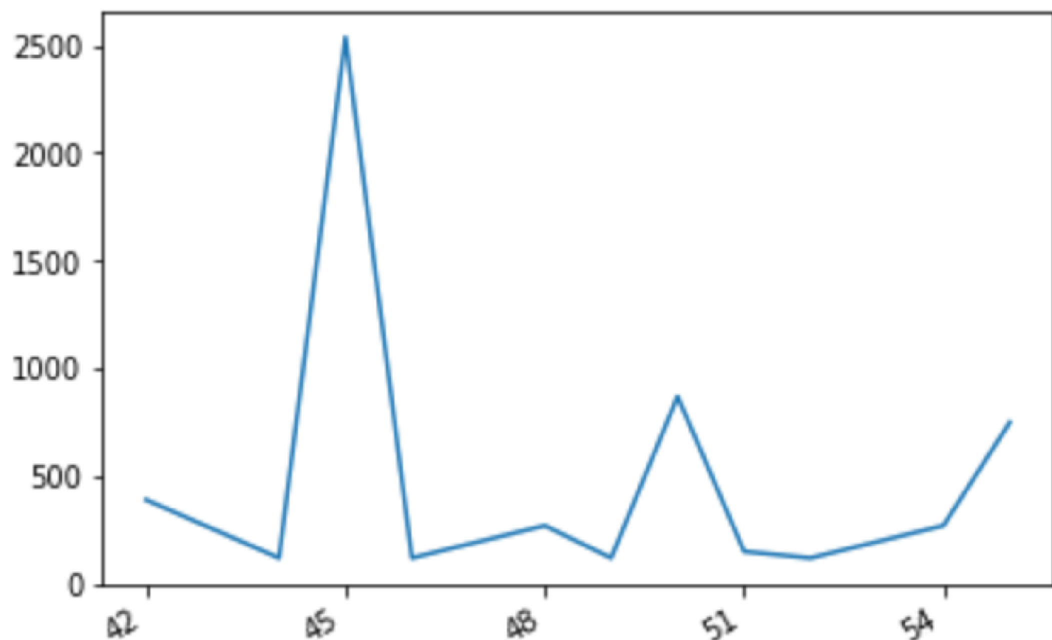


Рисунок 4.25 - Не оптимізована за споживанням CPU

Після модернізації алгоритму пошуку не оптимальних точок часу, результат оптимізації значно покращив розклад (див. рис. 4.26).

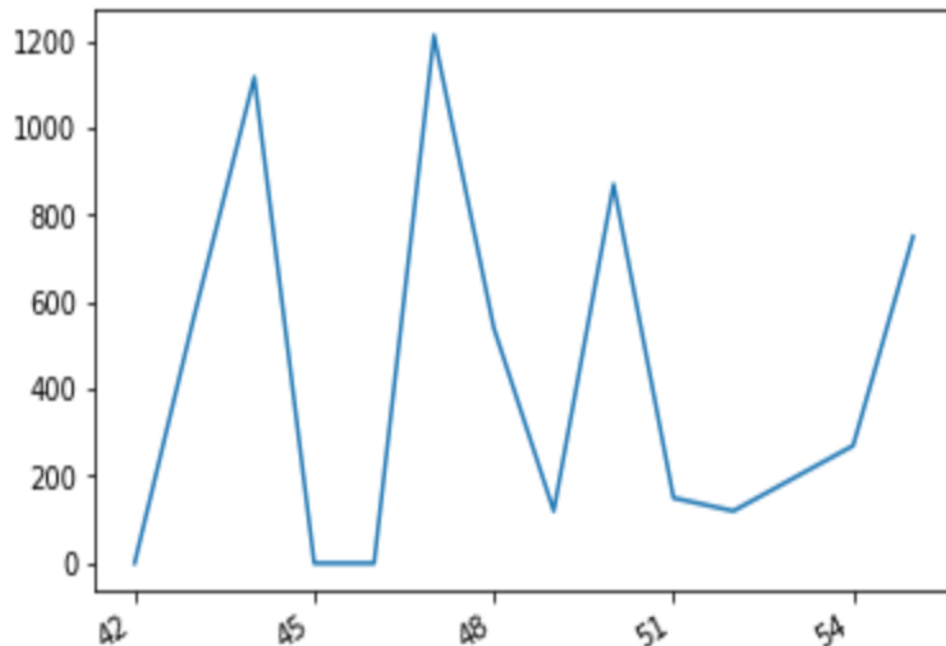


Рисунок 4.26 - Результат оптимізації за CPU

Оптимізація за використанням пам’яті на пристрої теж проводиться завдяки незначним змінам в функції оцінки (див. рис. 4.27).

```
def search_max_num_tasks(dags):
    def dag_time_memory(dags):
        for i, v in dags.items():
            for i, task in v['tasks'].items():
                yield task['resources']['request_memory']
    cur_len_v = 0
    indx = None
    for date, v in dags.items():
        memory_load = sum(dag_time_memory(v))
        if cur_len_v < memory_load:
            cur_len_v = memory_load
            indx = date
    return indx
```

Рисунок 4.27 - Функція пошуку найбільш навантажень за пам'яттю

Змінення алгоритму для оптимізації за використанням пам'яті в кластері теж не призвело до значних змін у функції оцінки (див. рис. 4.28).

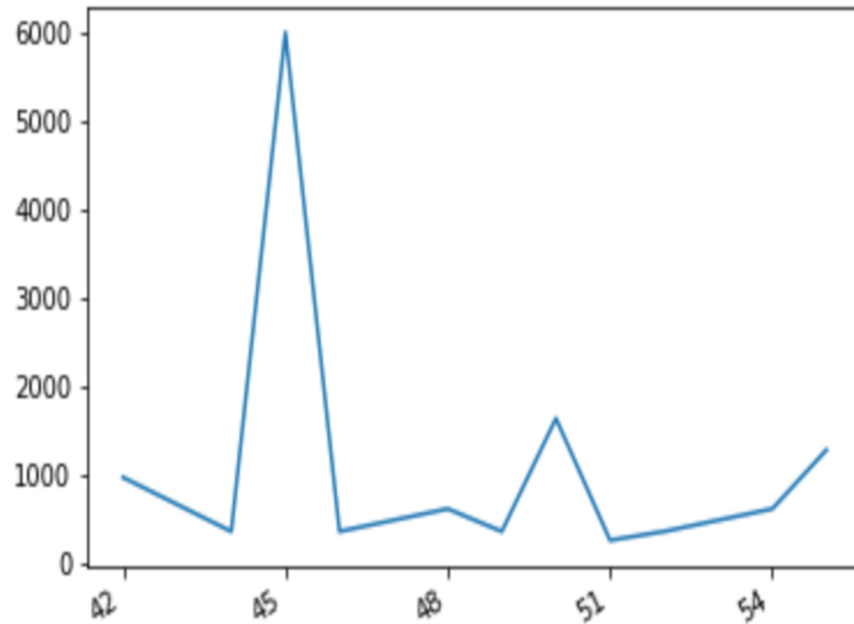


Рисунок 4.28 - Не оптимізоване використання пам'яті

До оптимізації спостерігався вибух використання пам'яті за однією точкою часу. Після оптимізації (див. рис. 4.29).

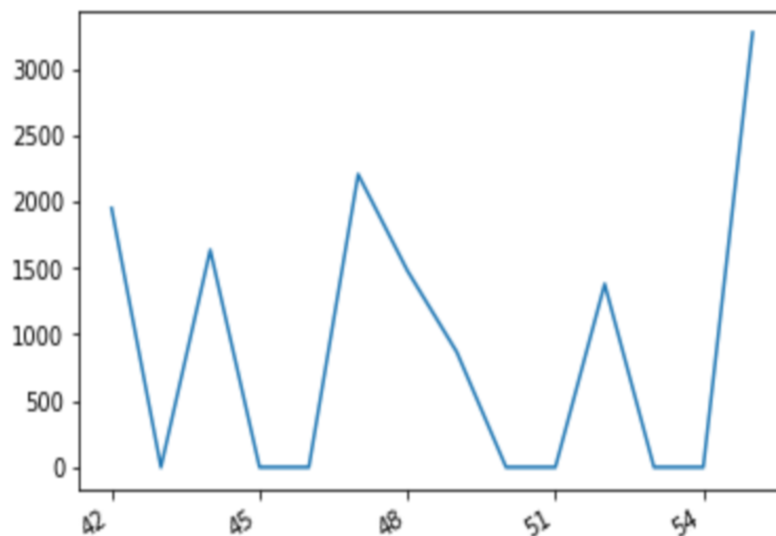


Рисунок 4.29 - Використання пам'яті після оптимізації

Треба зазначити, що екстремальних показників іноді зовсім не можливо позбавитись, але в такій ситуації можливо одне з рішень, це використовувати тимчасово інші машини, що будуть виконувати всі потреби системи у разі потреби високого навантаження на систему. Це стосується не тільки CPU, але й використання оперативної пам’яті. Комбінування показників теж може покращити використання існуючих ресурсів, окремо стоїть питання, як саме можна оптимізувати тимчасову автоматизовану оренду машин для виконання у разі високого навантаження у системі якого на жаль майже не уникнути, ці досліди виходять трохи за границю цієї роботи та залишаються на доопрацювання, так як це оптимізації самого використання машин, а не оптимізація розкладу.

4.5 Використані програмні забезпечення, бібліотеки та програмні продукти

У даній роботі було використано наступні програмні продукти з наступними версіями:

– система Apache Airflow версія 1.10.8;
– python версія 3.7;
– docker версією 20.10.8 та вище;
– docker-compose версією 1.28.4 та вище;
– kubernetes версія 1.18.10;
– postgreSQL версія 9.6 та вище;
– pip версією 19.2.4 та вище;
– pip пакет “croniter” версія 0.3.37;
– pip пакет “jupyterlab” версія 3.0.14;
– pip пакет “matplotlib” версія 3.4.1;
– pip пакет “boto3” версія 1.10.21;
– pip пакет “SQLAlchemy” версія 1.3.15;
– pip пакет “psycopg2-binary”;
– pip пакет “Flask” версія 1.1.2.

ВИСНОВКИ

Результатом кваліфікаційної роботи є робочий прототип, в якому було використано розроблену спрощену модель для оптимізації cron-задач. У результаті теоретичного аналізу було виявлено, що частотні системи запуску автоматизованих задач, такі як cron, можно звести до звичайної моделі оптимізації задач розкладу. Ця модель є системою координат, перша вісь якої є вісь часу. На цій вісі розміщені точки, кожна з яких відповідає часу запуску той чи іншої задачі. Така модель є класичною для теорії розкладів, зручною та гнучкою для різних методів та алгоритмів, окрім цього може ускладнюватись, замість точок можуть бути відрізки які позначають початок та кінець якоїсь задачі. Також були виявлені тенденції для багатьох областей програмного забезпечення, що використовують cron-подібні вирази, а саме для машинного навчання, великих даних, роботи кластерів, інфраструктурні системи, збір даних, транспортування, аналіз, трансформування даних. Були розглянуті програмні продукти: Celery, Ignes, Apache Airflow, та інші.

На основі однієї з cron-систем, що були перечислені вище, а саме Apache Airflow було створено тестове середовище, був розгорнут кластер для виконання docker контейнерів Kubernetes. Завдяки зручної інтеграції було легко налагоджена інтеграція Apache Airflow [4] з Kubernetes [7] кластером. Приклад такої системи повністю моделює складні системи автоматичного виконання завдань.

Результат аналізу теоретичної частини, що була взята з теорії розкладів, а також її моделі та методи довели, для використання над cron-системами. Було розроблено на основі класичної моделі алгоритм для перерозподілу задач рівномірним методом по кількістю задач на точку часу. Також результат процесу був реалізований у вигляді невеликої системи, що може інтегрувати з основною системою Apache Airflow, таким чином виконуючи повний цикл по збору, аналізу, оптимізації та впровадженню змін до розкладу.

Були досліджені різні підходи до оптимізації роботи таких систем - за часом, за ресурсами, за коштами. Було проведено експериментальне дослідження та отримані графіки роботи системи до оптимізації та після, які наглядно демонструють переваги запропонованої моделі оптимізації cron завдань.

Результати роботи були опубліковані ________________________

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. L. A. Vlasenko, A.G. Rutkas, , V. Semenets, *Stochastic Optimal Control of a Descriptor System Cybernetics and Systems Analysis*, 2020, 56(2), с. 204-212. DOI: 10.1007/s10559-020-00236-7.

2. O. Haitan, O. Nazarov, *Hybrid approach to solving of the automated timetabling problem in higher educational institution,* Системи управління, навігації та зв'язку. Збірник наукових праць 2 (60), с. 60-69, DOI: 10.26906/SUNZ.2020.2.060.

3. Шмелёв В. В. . *Динамічні задачі календарного планування.* Автоматика та телемеханіка, 1997.

4. Apache Airflow. Документація. URL: https://airflow.apache.org/docs/ (дата звернення: 03.03.2021).

5. Cron. Документація. URL: https://crontab.guru/crontab.5.html (дата звернення: 03.03.2021).

6. Python. Документація. URL: https://docs.python.org/3.7/ (дата звернення: 03.03.2021).

7. Kubernetes. Документація. URL: https://kubernetes.io/docs/concepts/ (дата звернення: 03.03.2021).

8. Конвей Р. В., Максвелл В. Л., Миллер Л. В. (1975). *Теория расписаний*. М.: «Наука».

9. Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), *Introduction to Algorithms (2nd ed.)*, MIT Press & McGraw-Hill, ISBN 0-262-03293-7 . pp. 327-8.

10. Gerard Sierksma; Yori Zwols. *Linear and Integer Optimization: Theory and Practice.* CRC Press, 2015, ISBN 978-1-498-71016-9.

11. John Dwyer . *An Introduction to Discrete Mathematics for Business & Computing*. 2019, ISBN 978-1-907934-00-1.

12. Hammer P. L., Johnson E. L., Korte B. H. .*Discrete Optimization II*, Annals of Discrete Mathematics, 5, Elsevier, pp. 427–453.

13. Fleischer Rudolf. *Algorithms - ESA*, 2000, Berlin, Heidelberg: Springer. pp. 202–210, doi:10.1007/3-540-45253-2_19.

14. Liu M., Xu Y., Chu, C., Zheng, F. .*Online scheduling on two uniform machines to minimize the makespan*, Theoret, Comput, Sci, 2019, 410 (21–23): 2099–2109. doi:10.1016/j.tcs.2009.01.007

15. Babar M.A., Dingsøyr T., Lago P., Vliet H. van . *Software Architecture Knowledge Management:Theory and Practice, First Edition*. Springer, 2009, ISBN 978-3-642-02373-6.

16. B. Len, P. Clements, R. Kazman. *Software Architecture in Practice, Third Edition. Boston: Addison-Wesley*, 2012, ISBN 978-0-321-81573-6.

17. AWS. Документація. URL: https://aws.amazon.com (дата звернення: 03.03.2021).

18. Docker. Документація. URL: https://docs.docker.com/ (дата звернення: 03.03.2021).

19. Postgres. Документація. URL: https://www.postgresql.org/docs/ (дата звернення: 03.03.2021).

20. Matplotlib. Документація. URL: https://matplotlib.org/docs (дата звернення: 20.03.2021).

21. Jupyter. Документація. URL: https://jupyter.org/documentation (дата звернення: 25.03.2021).