УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Петар Ђорђевић

РАЗВОЈ ЕФИКАСНОГ И СКАЛАБИЛНОГ ТОКА ПОДАТАКА ПОМОЋУ APACHE AIRFLOW-A И APACHE SPARK-A

мастер рад

Београд, 2024.

Ментор:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

др Мирјана Маљковић Ружичић, доцент
Универзитет у Београду, Математички факултет

Датум одбране:

Наслов мастер рада: Развој ефикасног и скалабилног тока података помоћу Apache Airflow-a и Apache Spark-a

Резиме: Инжењерство података је грана програмирања која се бави дизајнирањем, развојем и одржавањем система за прикупљање, чување и обраду великих количина података. Оно укључује креирање инфраструктуре која омогућава да подаци буду доступни и употребљиви за анализу и доношење одлука. Програмирање дистрибуираних система подразумева развој софтвера који функционише на више међусобно повезаних рачунарских јединица, чинећи их способним да раде као јединствена целина. Системи за обраду података данас функционишу дистрибуирано како би омогућили обраду великих количина података у реалном времену, побољшали скалабилност и осигурали високу доступност и отпорност на отказе. У овом раду имплементиран је систем чији су главни циљеви брза продукционализација, подршка рапидном развоју и ефикасно управљање великим токовима података. За изградњу дистрибуираног система за обраду података заснованог на програмском језику *Python* коришћени су алати као што су *Apache Airflow* за оркестрацију радних токова и *Apache Spark* за обраду података у великом обиму, као и *Apache Iceberg* за управљање табеларним подацима. Допунски алати отвореног кода, укључујући *Docker*, *MLflow* и *MinIO*, такође су коришћени како би употпунили и побољшали систем. Систем је осмишљен да буде прилагодљив и проширив, што омогућава будуће надоградње и лаку интеграцију нових функционалности.

Кључне речи: рачунарство, инжењерство података, токови података, дистрибуирани системи

Садржај

1 Увод **1**

2 Алати **4**

2.1 Docker 5
2.2 Apache Airflow 8
2.3 Apache Spark 16
2.4 Складишта података 20

3 Имплементација **24**

3.1 Микросервисна архитектура 24
3.2 Оркестрација 28
3.3 Обрада података 30
3.4 Регистар модела машинског учења 33

4 Пример тока података **34**

4.1 Чекање на податке 34
4.2 Обрада података 37

5 Дискусија **44**

6 Закључак **49**

Библиографија **50**

Глава 1

Увод

Историја инжењерства података прати развој технологија за управљање и обраду великих количина података од раних дана електронских рачунара [38]. Док термин „инжењерство података" није постојао пре 2000. године, основни принципи ове дисциплине почели су да се формирају са појавом првих релационих база података 1970-их, које су омогућиле структурирано чување и дохватање података, као и гаранције атомичности, конзистентности, изолованости и издржљивости (енг. *Atomicity, Consistency, Isolation, Durability* - *ACID*) [30]. Инжењерство података значајно је напредовало појавом технологија које омогућавају управљање и обраду великих количина података. Поред структуираних табеларних података, почеле су да се чувају велике количине неструктуираних и необрађених података, што је проузроковало да се развију језера података [25] и кофе [13]. Дошло је до експлозије различитих типова складишта података, од којих је свако било оптимизовано за специфичне потребе, што је довело до постојања великог броја разноврсних система које је било потребно ускладити. Почевши од основних принципа структуираног чувања и извлачења података, развој метода као што су процеси екстракције, трансформације и чувања података (енг. *Extract, Transform, Load* - *ETL*) [28] представљао је први корак у интеграцији података из различитих извора. Увођење дистрибуираних система и концепата као што су дистрибуирано складиштење података и обрада у великом обиму омогућило је обраду све већих количина података. У последњих неколико година, нагласак на обради података у реалном времену и увођење нових концепата као што су хибридни и федеративни системи додатно су утицали на развој области, која данас нуди разноврсне методе и принципе за ефикасно управљање подацима.

Историјски ток података (енг. *Data Pipeline*) је представљао процес у ком се сирови подаци прикупљају из различитих извора података, трансформишу, и затим смештају у складишта података ради даље употребе [29]. Данас се овај термин користи у општијем облику па се не односи само на припрему и складиштење података већ и на аутоматизацију саме обраде, надгледање процеса, генерисање извештаја и прослеђивање података другим процесима и токовима. Пре него што подаци уђу у складиште, они обично пролазе кроз одређену обраду, укључујући трансформације као што су филтрирање, маскирање и агрегација, чиме се обезбеђује одговарајућа интеграција и стандардизација података. Подаци се могу добијати из разноврсних извора, као што су програмски интерфејси апликација, релационе и нерелационе базе података, датотеке и други, али они ретко када буду спремни за непосредну употребу. Задаци припреме података и токова података обично представљају одговорност инжењера података (енг. *Data Engineer*) [38]. Постоје различити типови токова података, укључујући токове података за пакетну обраду, токове података у реалном времену, интеграционе токове података, као и токове података који су специфично дизајнирани за рад у облаку. Важно је напоменути да је ETL подкатегорија токова података.

У овом раду имплементиран је сложен дистрибуирани систем за обраду података у виду микросервиса, са циљем да се испита и демонстрира могућност интеграције савремених технологија и алата у један ефикасан и скалабилан систем. Овај систем служи као доказ концепта, који омогућава брзу и поуздану продукционализацију података, подржава рапидни развој апликација, и побољшава управљање подацима. Рад није фокусиран на решавање конкретног проблема, већ на истраживање и анализу изазова у повезивању различитих алата у целину, као и на документацију процеса, што би могло бити корисно у будућим пројектима. Систем је изграђен на програмском језику *Python*. Кључни алати који су коришћени укључују *Apache Airflow* за оркестрацију радних токова, *Apache Spark* за паралелну и дистрибуирану обраду података у великом обиму, и *Apache Iceberg* за управљање табеларним подацима у складишту података. Допунски алати отвореног кода, као што су *Docker* за паковање и изолацију зависности, *MLflow* за праћење експеримената у машинском учењу и *MinIO* за складиштење неструктуираних и полуструктуираних података, интегрисани су како би систем био комплетан и флексибилан. Овај приступ омогућава не само интеграцију различитих

алата и технологија, већ и будућу проширивост система, чиме се осигурава дугорочна одрживост и релевантност убрзаним технолошким променама.

У оквиру овог рада имплементиран је и пример тока података који демонстрира функционалност система. Овај пример служи као илустрација модерног тока обраде података и показује како се могу конструисати сложени токови са бројним компонентама и зависностима. Имплементирани ток података обухвата чекање на долазак података, њихову обраду и примену аналитичких метода, чување у различитим складиштима, као и обуку модела машинског учења на обрађеним подацима. Овај пример података истиче једноставност конструкције комплексних токова у имплементираном систему, као и могућност визуализације и надгледања тих токова.

У поглављу 2 су описани алати коришћени у раду, историјски контекст њиховог настанка, архитектура и значајне функционалности. Архитектура система за токове података и његове карактеристике описани су у поглављу 3. У поглављу 4 представљен је ток података који је имплементиран у систему као пример употребе. Поглавље 5 описује сфере система које би могле да се другачије имплементирају и унапреде, као и алате који би могли да буду алтернатива алатима коришћеним у раду. На крају је дат главни закључак рада.

Глава 2

Алати

У даљем тексту разматрамо алате који су коришћени у имплементацији пројекта описаног у овом раду. Ови алати представљају основу за развој и управљање сложеним системима, радним токовима и токовима података, као и за унапређење процеса у развоју софтвера и обради података. Сваки од описаних алата игра кључну улогу у обезбеђивању ефикасности, скалабилности и флексибилности система, и дозвољава безбедан и једноставан рапидан развој када је време кључни фактор у планирању и потребан нам је употребљив производ у кратком временском року.

У овом контексту, представићемо кључне технологије које су се показале као одлична решења за управљање радним токовима и контејнеризацију. *Docker*, као платформа за контејнеризацију, омогућава изолацију апликација и њихових зависности, што олакшава преносивост и управљање различитим окружењима. Са друге стране, *Apache Airflow* пружа моћан и флексибилан систем за оркестрацију радних токова, који је од суштинске важности за управљање комплексним процесима обраде података и других задатака у дистрибуираном окружењу. *Apache Spark*, као моћан систем за обраду података у меморији, нуди брзо и ефикасно решење за обраду великих количина података, што га чини кључним за аналитичке и обрађивачке задатке у модерном окружењу података. Поред горе наведених алата, описани су и додатни алати који, иако нису централни у систему, играју кључну улогу у обезбеђивању глатког функционисања у оквиру микросервисне архитектуре. Ови алати су неопходни за интеграцију и оптимизацију рада свих компоненти система.

2.1 Docker

Docker[1] је платформа отвореног кода која омогућава развој, испоруку и покретање апликација у изолованом окружењу званом контејнер [31]. Помоћу њега се може поједноставити процес управљања софтвером тако што омогућава развојним тимовима да пакују апликације са свим њиховим зависностима у целину која се може лако преносити и покретати на било ком систему који подржава *Docker*. Ово смањује проблеме са компатибилношћу између различитих окружења, омогућава бољу контролу над апликацијом, унапређује скалабилност репликацијом сервиса и олакшава континуирану интеграцију и испоруку. *Docker* је написан на програмском језику *Go*, и убрзо након свог представљања постао је један од најважнијих алата у свету *DevOps*-а [32], омогућавајући револуцију у начину на који се апликације развијају, тестирају и испоручују.

Историја

Docker је настао као одговор на потребу за лакшим и ефикаснијим начином управљања апликацијама у различитим окружењима. Пре његовог појављивања, једна од главних метода за изолацију апликација и њихово покретање у различитим окружењима била је употреба виртуелних машина. Виртуелне машине омогућавале су извршавање више апликација на једном физичком серверу, свака у свом виртуелизованом окружењу са сопственим оперативним системом. Међутим, овај приступ имао је значајне недостатке, укључујући високу потрошњу ресурса и споро покретање, због чега је био неефикасан за брзу испоруку и скалирање апликација. Контејнеризација је постојала и пре *Docker*-а, са алатима као што су *LXC*, *OpenVZ*, и *Solaris Zones*, који су пружали одређени ниво изолације и могућност покретања више апликација на једном језгру оперативног система [36]. Међутим, ови алати су били сложени за коришћење и захтевали су дубоко техничко знање за њихову имплементацију, што је ограничавало њихову ширу примену. *Docker* је настао да би решио ове проблеме пружајући лакши и стандардизован начин за паковање апликација и њихово извршавање у изолованим контејнерима. Главни циљ је био поједноставити процес креирања, распоређивања и управљања апликацијама тако да оне могу да се извршавају непромењене на различитим

[1] Званична веб страница: `https://www.docker.com`

инфраструктурама, било да је реч о развојном, тестном или продукционом окружењу.

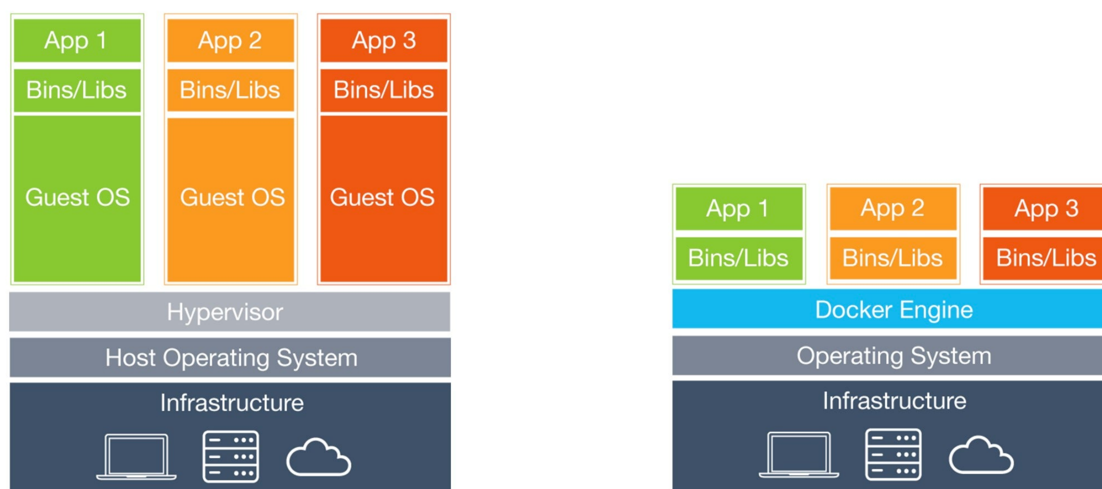
Docker је први пут представљен 2013. године од стране Соломона Хајкса, који је компанију основао као *DotCloud* 2008. године у Паризу [33, 36]. Иако је компанија првобитно била фокусирана на платформу као услугу, Хајкс је током конференције *PyCon* у марту 2013. године представио *Docker* као отворени пројекат намењен поједностављивању употребе технологије контејнера коју је његова платформа користила. *Docker* је брзо постао популаран међу програмерима због своје способности да поједностави рад са контејнерима и обезбеди комплетан екосистем за управљање њима.

Иницијално, *Docker* је користио *LXC* за управљање контејнерима, али је касније развио сопствену библиотеку, *libcontainer*, која је омогућила још бољу контролу и флексибилност [36]. *Docker* се убрзо одвојио од конкуренције тако што је понудио цео екосистем за управљање контејнерима, укључујући алате за изградњу, дељење и оркестрацију контејнера. Захваљујући својој иновативности и отвореном приступу, *Docker* је привукао пажњу великих технолошких компанија као што су *Microsoft*, *IBM* и *Red Hat*, као и значајне инвестиције од стране ризичних капитала [5]. *Docker* је данас кључни алат у *DevOps*-у, широко коришћен за континуирану интеграцију и испоруку, развој и тестирање апликација у облаку, као и за оркестрацију контејнера, а и даље се активно развија и усавршава.

Архитектура

Docker контејнеризација омогућава изолацију апликација и њихових зависности у лаганим, преносивим јединицама које могу да раде доследно на различитим окружењима. За разлику од виртуелних машина, које укључују цео оперативни систем и захтевају хипервизор за управљање [39], *Docker* контејнери деле исти кернел оперативног система машине на којој се извршавају (хост систем), што их чини много ефикаснијим у погледу потрошње ресурса. Овај ниво апстракције омогућава контејнерима да буду много мањи и бржи за покретање у поређењу са виртуелним машинама. Контејнери садрже само апликацију и њене директне зависности, док користе основне ресурсе и услуге хост система. Ово смањује оптерећење и омогућава бољу скалабилност, јер више контејнера може да ради на истом хардверу. Осим тога, пошто се контејнери ослањају на слојевиту архитектуру слика, они могу брзо и ефикасно

да се дистрибуирају и инстанцирају, што додатно убрзава развој и имплементацију апликација. На слици 2.1 приказана је разлика између контејнера и виртуалних машина.



Слика 2.1: Разлика између виртуалне машине и контејнера [33]

Један од кључних механизама које *Docker* користи су контролне групе [7], које омогућавају ограничавање, приоритизацију и изолацију употребе ресурса као што су процесор, меморија, диск У/И и мрежни проток између различитих контејнера. На овај начин, *Docker* осигурава да један контејнер не може монополисати ресурсе система, чиме се постиже стабилно и предвидиво понашање апликација. Поред контролних група, *Docker* користи просторе имена како би изоловао процесе, мрежне интерфејсе, тачке монтирања и друге системе унутар контејнера. Простори имена омогућавају да сваки контејнер има сопствени скуп ресурса, као што су *PID*-ови процеса или мрежне картице, што ствара илузију да сваки контејнер ради на сопственом систему, иако сви деле исти кернел оперативног система домаћина.

Docker користи концепт слика и слојева како би обезбедио ефикасно складиштење и извршавање апликација [12]. Слика је неизмењив шаблон који укључује све потребне зависности за покретање апликације. Када се контејнер креира из слике, *Docker* користи слојеве за чување промена, што омогућава лако креирање нових слика и минимизацију потрошње ресурса.

Основни елемент *Docker*-а је систем, који се састоји од три главне компоненте: демон, интерфејс командне линије и програмски интерфејс апликације.

Docker демон је процес који управља свим аспектима *Docker*-а на систему [34]. Он је одговоран за креирање, покретање и управљање контејнерима, као и за повезивање са регистром, где се чувају *Docker* слике. *Docker*-ов интерфејс командне линије омогућава корисницима да комуницирају са демоном користећи једноставне команде. Програмски интерфејс апликације *Docker* омогућава програмима да комуницирају са *Docker*-ом, што омогућава аутоматизацију и интеграцију са другим алатима и сервисима.

Docker Swarm

Docker Swarm је уграђени алат за оркестрацију контејнера који омогућава креирање и управљање кластерима контејнера на више сервера [11]. *Swarm* претвара групу *Docker* инстанци у један логички кластер, што омогућава скалабилност и високу доступност апликација. Коришћењем *Swarm*-а, корисници могу дефинисати и покренути сервисе који се састоје од више реплика контејнера.

Docker Compose

Docker Compose је алат који омогућава дефинисање и управљање више-контенјерским апликацијама у *Docker*-у [10]. Користећи једноставан формат *YAML*, корисници могу дефинисати све сервисе, мреже и логичке дискове потребне за покретање апликације. *Docker Compose* омогућава покретање свих дефинисаних компоненти са једноставном командом, а такође пружа и могућности за скалирање, ажурирање и управљање животним циклусом апликације. Овај алат је посебно користан за развојне тимове који раде на апликацијама које се састоје од више микросервиса, јер омогућава лако постављање и тестирање целокупног окружења на локалним машинама.

2.2 Apache Airflow

Apache Airflow[2] је платформа за управљање радним токовима и токовима података [22, 16] која се претежно користи у инжењерству података, мада има употребе и у другим сферама дистрибуираног програмирања. Овај алат омогућава креирање и заказивање радних токова програмски у језику *Python*,

[2] Званична веб страница: `https://airflow.apache.org`

као и надгледање путем уграђеног графичког корисничког интерфејса и програмског интерфејса апликације. Од самог почетка, пројекат је био отвореног кода, а 2019. године постаје *Apache* пројекат приоритетне важности [18].

Историја

Airflow је настао због потребе за алатом који може да скалира и координише комплексне процесе у великим компанијама. Пре појаве *Airflow*-а, коришћени су алати као што су *Cron* и *Jenkins* за аутоматизацију задатака. Међутим, ови алати су имали ограничења када је било потребно управљати сложеним процесима обраде података. *Cron*, на пример, је једноставан систем за заказивање задатака на нивоу оперативног система [6]. Иако је користан за основне задатке као што су покретање скрипти у одређено време, *Cron* нема могућност да управља зависностима између задатака, прати њихово извршење или скалира на ниво који је потребан за комплексне радне токове у великим организацијама. Са друге стране, *Jenkins* је алат за континуирану интеграцију и испоруку који нуди одређени ниво координације задатака [1]. Иако омогућава дефинисање и покретање сложенијих радних токова, његов фокус је углавном на аутоматизацији процеса изградње и тестирања софтвера, те му недостаје флексибилност и скалабилност потребна за оркестрацију великих и хетерогених система. Један од главних изазова био је управљање сложеним зависностима између задатака и интеграција са разноврсним системима и технологијама које су коришћене у процесима обраде података. Ни *Cron* ни *Jenkins* нису нудили могућност динамичког управљања овим зависностима, као ни погодан интерфејс за визуелизацију и праћење процеса. Због свих ових ограничења, појавила се потреба за алатом попут *Airflow*-а који је у потпуности фокусиран на координацију комплексних радних токова.

Airflow је развијен унутар компаније *Airbnb* у октобру 2014. године [22]. Првобитно га је развио Максим Бошмен, инжењер података у *Airbnb*-у, са циљем да омогући лакше планирање, оркестрацију и надгледање сложених процеса обраде података. *Airflow* је брзо постао популаран унутар компаније због своје флексибилности и моћних могућности за планирање и надгледање задатака. Као резултат, *Airbnb* је одлучио да овај алат отвори за јавност, омогућавајући и другим компанијама да га користе и доприносе његовом ра-

звоју. *Airflow* је објављен на *GitHub*-у[3] као пројекат отвореног кода. Веома брзо је привукао пажњу шире заједнице захваљујући својој архитектури заснованој на језику *Python* и подршци за усмерене ацикличке графове (енг. *Directed Acyclical Graph - DAG*), који поједностављују опис сложених процеса у облику кода. *DAG*-ови омогућавају корисницима да дефинишу низ задатака и њихове међусобне зависности на начин који је лако визуализовати и прати. Ово чини управљање комплексним радним токовима много ефикаснијим и омогућава лако праћење и отклањање грешака.

Пошто је више од 30 организација већ користило *Airflow* за управљање процесима обраде података, а преко 100 доприносиоца је обезбеђивало интеграције са системима трећих страна, направљен је предлог за прихватање *Airflow*-а у фондацију *Apache* [16]. *Airflow* је у то време развијан од стране четири инжењера: Максима Бошмена, Сидарта Ананда, Болке де Бруина и Криса Рикоминија, који су имали искуство у различитим *Apache* пројектима, што је довело до тога да се пројекат прихвати у инкубацију. Након три године, 2019. године, пројекат излази из инкубације и постаје један од приоритетних пројеката *Apache* фондације [18]. *Airflow* добија значајно ажурирање са верзијом 2.0, која доноси бројне побољшања као што су боље перформансе, побољшана сигурност и нови начин писања *DAG*-ова. Пројекат и даље има редовна ажурирања и активну заједницу.

Архитектура

Airflow је дистрибуирана платформа која омогућава оркестрацију радних токова кроз неколико главних компоненти [20]:

- **Распоређивач**: Распоређивач (енг. *Scheduler*) је централна компонента која управља извршавањем задатака. Његов главни задатак је да чита и обрађује дефиниције *DAG*-ова и одређује када ће се сваки задатак извршити. Он поставља задатке у редове чекања у складу са временским распоредом и зависностима које су дефинисане у *DAG*-у. Распоређивач се ослања на релациону базу података за чување стања и метаподатака радних токова.

- **Покретач**: Покретач (енг. *Triggerer*) је новија компонента уведена у *Airflow* 2.0 која омогућава ефикасно управљање асинхроним задацима

[3] GitHub: `https://github.com/apache/airflow`

и задацима који чекају на одређене услове пре него што се изврше. Ова компонента је креирана како би се смањило оптерећење на главним радницима и распореднику, и омогућила већа скалабилност приликом руковања великим бројем асинхроних задатака.

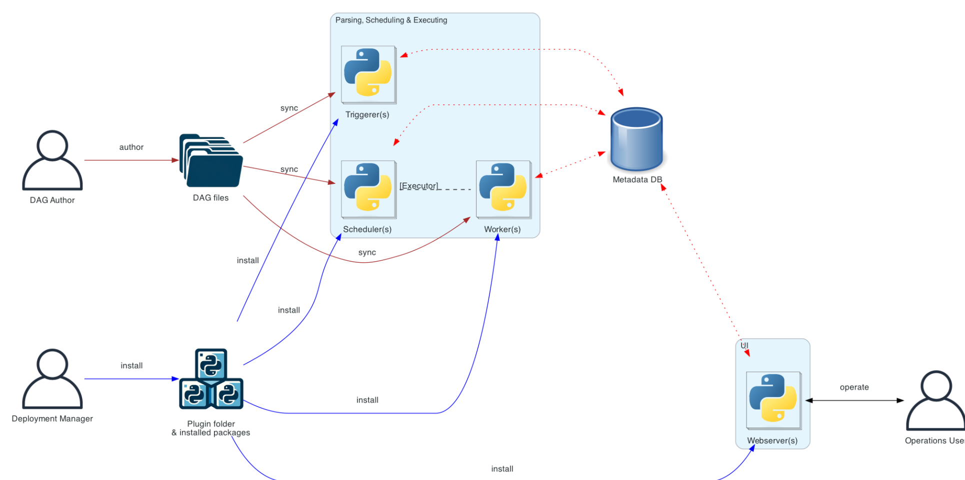
- **Веб сервер**: Веб сервер је компонента која пружа графички кориснички интерфејс за *Airflow*. Корисници могу преко овог интерфејса да прегледају статус својих *DAG*-ова, надгледају задатке, конфигуришу системе, започињу нове задатке и прегледају логове. Веб сервер омогућава преглед визуелизације *DAG*-ова, што олакшава праћење сложених радних токова.

- **Радници**: Радници су процеси који обављају конкретне задатке у оквиру радних токова. Распоређивач додељује задатке радницима, а они их извршавају у складу са задатом конфигурацијом. У зависности од коришћеног типа извршитеља, радници могу бити организовани локално на једном серверу, распоређени на више сервера, или скалирани унутар кластер оквира као што је Kubernetes.

- **Релациона база података**: Релациона база (обично *MySQL* или *PostgreSQL*) служи за складиштење метаподатака система. Као централни извор истине система, остале компоненте се високо ослањају на ову базу за своје функционисање. Ова база нема никакве везе са подацима који се обрађују и као таква може бити независна од остатка тока података.

Важно је нагласити да се све компоненте осим релационе базе могу реплицирати. Могуће је имати више распоређивача, и на основу врсте или захтевности задатка проследити га различитим распоређивача предвиђеним за другачије ситуације. Интеракција ових компоненти приказана је на слици 2.2.

Извршитељи (енг. *Executor*) у *Airflow*-у су компоненте које управљају извршавањем задатака у радним токовима [21]. Као саставни део распоређивача и радника, они апстрахују начин на који се задаци покрећу и извршавају, омогућавајући различите приступе у зависности од захтева за перформансама и ресурсима. Иако је комбиновање извршитеља могуће, унутар једног распоређивача и његових радника се препоручује строго опредељени приступ.



Слика 2.2: Визуализација архитектуре Airflow-а [20]

- Секвенцијални извршитељ је подразумевани извршитељ у *Airflow*-у. Он извршава задатке један по један унутар истог процеса. Његова једноставност омогућава лаку конфигурацију и коришћење, што га чини идеалним за тестирање и развој. Такође је једини извршитељ који може да користи *SQLite* као базу метаподатака, што га чини идеалним за локално покретање. Међутим, због тога што обавља само један задатак у исто време, секвенцијални извршитељ има ниске перформансе и није погодан за велике радне токове, где је потребна већа скалабилност. Уместо покретања радника, задатак се извршава унутар самог распоређивача.

- Локални извршитељ користи вишенитно извршавање за паралелизацију задатака на истом серверу. Овај извршитељ омогућава боље искоришћење ресурса и покретање више задатака одједном, што га чини погодним за мање и средње радне токове. Међутим, његова ограничења произлазе из капацитета једног сервера, што може довести до проблема са перформансама ако се обрађује велике количине података. Радници представљају различите нити покренуте од стране распоређивача.

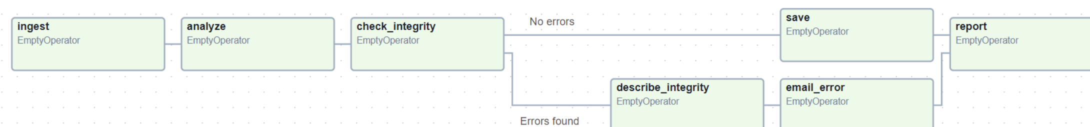
- Извршитељ *Celery* користи *Celery*, дистрибуирани систем за извршавање задатака, који омогућава распоређивање задатака на више рад-

ника распоређених на различитим серверима. Овај извршитељ омогућава високу скалабилност и флексибилност у распоређивању задатака. Међутим, конфигурација и управљање су комплекснији у поређењу са другим извршитељима, а потребне су додатне компоненте као што су *RabbitMQ* или *Redis* за управљање редовима чекања задатака.

- Извршитељ *Kubernetes* динамички креира раднике унутар кластера заснованим на алату *Kubernetes* за сваки задатак, што омогућава високу скалабилност и изолацију задатака. Овај извршитељ је погодан за велике организације са комплексним радним токовима који се обављају у облаку, пружајући флексибилност у руковању ресурсима. Међутим, конфигурација и управљање овим извршитељем су сложенији и захтевају добро познавање инфраструктуре *Kubernetes*-а.

Радни ток

DAG представља структуру која дефинише радни ток у *Airflow*-у. Сваки *DAG* се састоји од низа задатака, где сваки задатак може имати зависности од других задатака. Овај граф омогућава корисницима да јасно дефинишу редослед извршавања задатака, зависности међу њима и тачно време када треба да се изврше. Важност *DAG*-ова лежи у њиховој способности да управљају сложеним радним токовима и омогуће контролу над свим аспектима њиховог извршавања, као и интуитивна визуализација послова који треба да се изврше. На слици 2.3 је приказан пример *DAG*-а.



Слика 2.3: Пример визуализације *DAG*-а [20]

Оператори

У *Airflow*-у, оператори представљају основне градивне блокове за дефинисање и управљање задацима у радним токовима. Оператори су објекти који

извршавају конкретне операције унутар радног тока, омогућавајући корисницима да дефинишу шта ће се радити и како ће се задаци извршавати.

BashOperator је један од основних оператора у *Airflow*-у и омогућава извршавање команди унутар језика *Bash* као део радног тока. Овај оператор је идеалан за покретање скрипти, обављање системских задатака и интеракцију са оперативним системом. Захваљујући својој једноставности, *BashOperator* је користан у ситуацијама где је потребно извршити команде на нивоу система.

PythonOperator је специјализован за извршавање програма и функција у језику *Python* у оквиру радног тока. Овај оператор омогућава корисницима да пишу и извршавају произвољан *Python* код, што је корисно за обраду података, извршавање сложених израчунавања или интеграцију са *Python* библиотекама. *PythonOperator* пружа велику флексибилност у креирању логике задатака, пошто корисници могу директно користити *Python* код.

Корисницима је такође омогућено да креирају сопствене операторе који су специфични за њихове потребе [23]. Ови оператори могу бити развијени за рад са различитим сервисима и технологијама, као што су *AWS*, *Google Cloud*, или базе података, као и дефинисање задатака у било ком програмском језику. Корисник дефинише оператор или наслеђивањем *BaseOperatora* где онда можемо *Python* кодом дефинисати како се оператор извршава тако што дефинишемо метод *execute* или наслеђивањем било ког другог већ дефинисаног оператора. На пример, могли бисмо да дефинишемо *HaskellOperator* који извршава *Haskell* код тако што наследимо *BashOperator* и унутар њега дефинишемо да се прослеђени задатак извршава тако што се код проследи наредби *ghci*. Прилагођени оператори пружају потпуну флексибилност у креирању радних токова који укључују специфичне интеграције или пословну логику која није покривена основним операторима. *Airflow* већ има велики скуп уграђених оператора који покривају широк спектар функција и услуга. Ови укључују операторе за рад са базама података, сервисима у облаку, системима за управљање редовима порука, као и многе друге алате и технологије, што омогућава корисницима да лако интегришу различите компоненте у своје радне токове.

Сензори

Airflow сензори су посебна врста оператора који служе за праћење одређених стања или догађаја у систему пре него што покрену наредни корак у

току посла [4]. Сензори се користе да провере да ли је, на пример, датотека стигла на одређену локацију, да ли је база података ажурирана, или да ли је завршена нека спољашња операција. Они могу радити у два режима: континуирано проверавање (енг. *poke*) и одложено проверавање (енг. *reschedule*). У режиму континуираног проверавања, сензор ће у редовним интервалима проверавати стање све док услов није задовољен, што може дуго држати ресурсе заузетим. Насупрот томе, у режиму одложеног проверавања, сензор ће се активирати на одређени временски период и затим „успавати”, тј. ослободити заузете ресурсе до наредне провере. Примери сензора укључују сензор који прати постојање датотеке у одређеном директоријуму, сензор који чека да други ток посла заврши свој задатак пре него што настави са радом и сензор који прати доступност датотеке у неком складишту, нпр. *Amazon S3*.

XCom

XCom је механизам у *Airflow*-у који омогућава размену података између различитих задатака унутар истог *DAG*-а [2]. *XCom* омогућава да један задатак остави податке у глобалном простору, а други задатак их преузме и користи. Ово је корисно у ситуацијама где је неопходно прослеђивање резултата или информација између задатака који нису директно повезани. Поред тога, *XCom* подржава могућност дефинисања различитих поступака за чување података, што омогућава да се подаци чувају у различитим системима, као што су базе података или складишта у облаку, чиме се постиже већа флексибилност у раду са осетљивим информацијама.

Динамички задаци

Динамички задаци су флексибилан механизам у *Airflow*-у који омогућава динамичко генерисање задатака у току извршавања *DAG*-а, на основу тренутних података или услова. Проблем који решавају динамички задаци је потреба за унапред дефинисаним бројем или редоследом задатака, као и недостатак циклуса, што може бити неефикасно или непрактично у сложеним системима где се број задатака може мењати у зависности од контекста или улазних података. Са динамичким задацима, програмери могу креирати логичке петље, условна гранања или прилагодљиве токове задатака, чиме се постиже већа скалабилност и ефикасност у управљању сложеним процесима.

Конфигурација окружења

Везе, променљиве и резервоари су основни концепти у *Airflow*-у који омогућавају конфигурисање и управљање разним аспектима извршавања радних токова.

- Везе (енг. *connection*) представљају начин за конфигурисање приступа екстерним системима и сервисима, као што су базе података, програмски интерфејси апликација или услуге у облаку. Везе омогућавају задацима да лако приступе потребним ресурсима без потребе за дефинисањем детаља приступа у самом коду.

- Променљиве (енг. *variable*) служе за чување глобалних вредности које могу бити коришћене у више различитих *DAG*-ова или задатака. Ово је корисно за конфигурације које се могу мењати или које треба да буду доступне у више различитих контекста без мењања кода.

- Резервоари (енг. *pool*) омогућавају контролу над бројем паралелних задатака који могу бити извршени у оквиру одређеног *DAG*-а или у целом систему. Ово је корисно када постоје ограничења у погледу ресурса или када је потребно контролисати оптерећење одређених сервиса.

2.3 Apache Spark

Apache Spark[4] је платформа отвореног кода за обраду података која пружа једноставан и ефикасан начин за обраду велике количине података [27]. Основна сврха *Spark*-а је да омогући брзу обраду података користећи паралелно извршавање на великом броју чворова у кластеру. Његов дизајн је фокусиран на минимизацију времена потребног за обраду и анализу података, што га чини идеалним за различите аналитичке задатке, укључујући обраду података у реалном времену, машинско учење и интерактивну анализу података. Apache Spark је написан на програмском језику *Scala*, а 2014. године постаје *Apache* пројекат приоритетне важности [14].

[4]Званична веб страница: `https://spark.apache.org`

Историја

Apache Spark је настао као одговор на ограничења која су постојала у претходним алатима за обраду података великог обима, попут *Hadoop MapReduce*-а [26]. Иако је *MapReduce* омогућио обраду огромних количина података дистрибуирањем задатака на више машина, имао је значајне недостатке, посебно када је у питању брзина обраде и комплексност програмирања. *MapReduce* је био оријентисан ка обради података у серијама, што га је чинило спорим за апликације које захтевају интерактивну анализу или обраду у реалном времену. Алати као што су *Apache Flink* и *Apache Pig* имали су своје предности, али нису били у стању да обезбеде одговарајућу брзину и скалабилност. *Spark* је решио ове проблеме тиме што је омогућио убрзање обраде података захваљујући обради у меморији, чиме је постао и до 100 пута бржи од *MapReduce*-а у одређеним задацима [14]. Поред тога, *Spark* је поједноставио програмирање нудећи богатији програмски интерфејс апликације и подршку за сложене обраде података као што су итеративни алгоритми и обраде графова, што га чини знатно приступачнијим и флексибилнијим у односу на претходне алате.

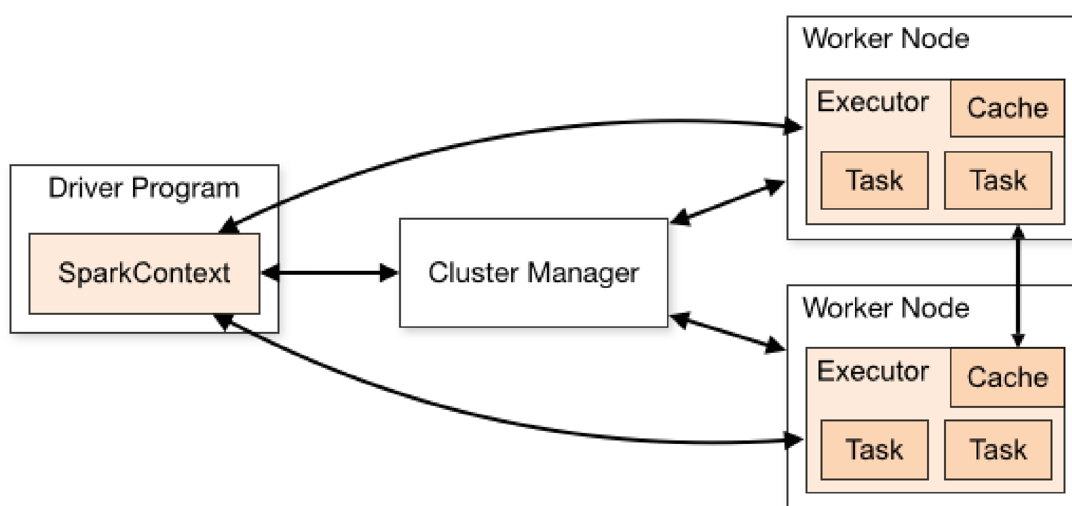
Настао је 2009. године у Лабораторији *AMPLab* на Универзитету у Калифорнији Беркли, под вођством Матеја Захарије [14], који је уједно и творац *MLflow*-а и тренутно обавља функцију главног техничког директора у компанији *Databricks*. Пројекат је брзо привукао пажњу заједнице и ушао у инкубатор фазу *Apache* фондације у јуну 2013. године. Током инкубације, *Spark* је стекао значајну подршку од више од 120 програмера из 25 различитих организација [14]. Захваљујући овом доприносу и активној заједници која га је подржавала, *Spark* је у фебруару 2014. године проглашен за приоритетни пројекат софтверске фондације *Apache*, што је потврдило његову зрелост и значај у оквиру система за обраду великих података. Захваљујући својој универзалности и перформансама, *Spark* је постао доминантан избор за обраду великих података, замењујући старије алате и постајући нови стандард у индустрији.

Архитектура

Apache Spark је дистрибуирани систем за обраду података, који је дизајниран са фокусом на брзу и ефикасну обраду великих скупова података директно у радној меморији [15]. Овај приступ значајно убрзава анализу

података у поређењу са традиционалним методама које се ослањају на складиштење података на диску.

Архитектура *Spark*-а је заснована на моделу „руководилац-радник“. Сервер руководилац управља кластером и координира задатке, док раднички сервере извршавају задатке који су им додељени. Ова подела омогућава *Spark*-у да паралелно обрађује велике количине података, распоређујући их на више радничких сервера, што додатно повећава брзину обраде. Ово је приказано на слици 2.4.



Слика 2.4: Визуализација архитектуре *Spark*-а [15]

Када корисник поднесе задатак *Spark*-у, међу радничким серверима се алоцирају ресурси за један контролни чвор који у себи чува информације о контексту рада и који је одговоран за покретање апликације и координацију извршавања задатака на осталим алоцираним радничким чворовима. Један чвор не мора заузети цео раднички сервер, већ може располагати само делом његових ресурса, и један раднички сервер унутар себе може имати више алоцираних чворова унутар себе. Контролни чвор распоређује податке и инструкције на више радничких чворова, који обрађују те податке у паралелним токовима. Раднички чворови затим враћају резултате назад контролном чвору, који их комбинује и представља као коначан резултат и сервер руководилац обавештава да је задатак завршен.

Обрада у радној меморији

Отпорни дистрибуирани скупови података (енг. *Resilient Distributed Datasets* - *RDD*) су основни програмски модел у *Spark*-у који омогућава рад са великим подацима расподељеним преко више чворова у кластеру [9]. *RDD* је непроменљива колекција објеката која се може паралелно обрађивати на више машина. Једна од кључних карактеристика *RDD*-а је његова отпорност на грешке, што значи да ако се део података изгуби или не успе да се успешно обради, *RDD* може аутоматски реконструисати те податке из првобитних трансформација. Иако је *RDD* раније био основа за развој *Spark* апликација, данас се све више прелази на употребу таблица података (енг. *DataFrame*), који омогућавају ефикаснији и једноставнији начин писања *Spark* кода, као и боље оптимизације кроз *Spark SQL*.

Једна од предности овакве архитектуре је и брже освежавање процеса у меморији. Како *Spark* ради са подацима директно у радној меморији, ажурирања и обрада података се извршавају без потребе за учитавањем и складиштењем података на диску, што значајно смањује време обраде и повећава ефикасност рада. Ова особина је кључна за рад са великим скуповима података где је брзина критичан фактор.

Процес размењивања података међу чворовима у *Spark*-у (енг. *Data Shuffle*) представља потенцијалну ману приступа обраде у радној меморији, јер подразумева пренос великих количина података између радничких чворова током фазе груписања и сортирања [3]. Овај процес може значајно успорити обраду, јер захтева дељење података преко мреже и привремено складиштење на диску, чиме се губи предност брзине рада у радној меморији чвора. Због тога програмери често настоје да избегну или минимизују размењивање података при писању токова обраде, користећи оптимизоване стратегије за рад са подацима како би задржали обраду у меморији и смањили време извршења.

Spark Connect

Spark Connect је нова компонента уведена у *Spark*-у верзије 3.4.0, осмишљена да реши проблем повезивања клијентских апликација са *Spark* кластерима на начин који омогућава већу флексибилност и скалабилност [17]. *Spark Connect* раздваја *Spark* програмски интерфејс апликације од извршног про-

грама, што омогућава да се корисници повежу са кластером преко лаганих клијентских апликација, без потребе за пуном инсталацијом *Spark*-а на клијентској страни. Овај приступ побољшава корисничко искуство и омогућава развој апликација у различитим окружењима и програмским језицима. Иако обећава значајне предности, *Spark Connect* је и даље у експерименталној фази и подложан је променама у будућим верзијама.

2.4 Складишта података

У контексту комплексних система и микросервисне архитектуре, важно је имати алате који подржавају различите аспекте управљања подацима и процесима. Док су горе наведени алати кључни за основне функције система, постоје и други важни алати који обезбеђују перзистентност података и подршку за различите врсте складишта. Ови алати укључују складишта података, базе података и формате података који осигуравају дугорочно чување информација и омогућавају боље управљање подацима. Сваки од ових алата је дизајниран да одговара специфичним потребама и типовима података, пружајући функционалности као што су напредно управљање верзијама, флексибилност у складиштењу и оптимизација перформанси.

MinIO

MinIO[5] је алат отвореног кода који је настао 2015. године, са циљем да обезбеди високо доступно складиштење података у окружењима која захтевају брзину и скалабилност. Написан је у програмском језику *Go*, што му омогућава високе перформансе и једноставну интеграцију у различите софтверске системе. *MinIO* је дизајниран да буде лак за коришћење и инсталацију, уз минималне захтеве за одржавање, што га чини погодним за предузећа свих величина.

Оно што MinIO чини посебно привлачним је његова компатибилност са програмским интерфејсом *Amazon S3*, што значи да апликације које већ користе *S3* могу без проблема да се пребаце на *MinIO*, без потребе за изменама у коду. Поред тога, *MinIO* подржава напредне функционалности као што су интеграција са базама података, логичка репликација података и извршавање

[5] Званична веб страница: `https://min.io`

функција вођених догађајима, што омогућава сложене операције директно на складиштеним подацима. Ове функције чине *MinIO* идеалним за примену у различитим сценаријима, укључујући складиштење великих количина података, аналитику, и обраду података у реалном времену.

Apache Iceberg

Apache Iceberg[6] је настао унутар компаније *Netflix* 2018. године, као одговор на изазове са којима су се суочавали при раду са великим количинама података. Овај формат табела отвореног кода осмишљен је да олакша рад са великим скуповима података, пружајући бољу контролу верзионисања и управљања партицијама у језерима података.

Iceberg омогућава трансакциони рад и *ACID* гаранције за операције на табелама [35], што је кључно за одржавање интегритета података у великим системима. Такође је оптимизован за брзо ажурирање података, што је остварено кроз његов иновативни начин управљања метаподацима и избегавање скупих операција на целим табелама. Ово га чини посебно погодним за сценарије где се подаци често мењају, а брзина и ефикасност су критични. Такође, подржава лако додавање и одузимање података без утицаја на перформансе и нуди бољу подршку за партиционисање у поређењу са претходним форматима, што га чини изузетно моћним форматом података када се ради о великим количинама података (петабајти).

Iceberg користи метаподатке да прати историју и промене у табелама правећи снимке у одређеним тачкама времена. Сваки снимак садржи потпун опис шеме табеле, информације о партицијама и фајловима. Ова структура омогућава брзе и ефикасне измене без потребе за редефинисањем свих датотека, што је идеално за рад са великим подацима. Историја се чува у специјализованом формату *Iceberg*, омогућавајући враћање на било коју тачку у прошлости, док каталози метаподатака осигуравају конзистентност резултата.

[6] Званична веб страница: `https://iceberg.apache.org`

MLFlow

MLflow[7] је алат отвореног кода за управљање животним циклусом модела машинског учења, развијен 2018. године од стране компаније *Databricks*, коју је основао Матеј Захарија, творац *Apache Spark*-а. Написан је на програмском језику *Python*, али подржава више језика, укључујући *R* и *Java*. Основна сврха *MLflow*-а је да омогући развојним тимовима да прате и управљају експериментима у машинском учењу, чувају артефакте, и олакшају процес развоја и испоручивања модела. *MLflow* је изузетно користан због своје могућности да прати читав животни циклус модела, од експерименталне фазе до продукције. Једна од кључних могућности је способност праћења експеримената, где се сви параметри, метрике, и артефакти чувају и могу се лако референсирати и поредити. Поред тога, *MLflow* подржава фазе као што су припрема и продукција, омогућавајући контролисан и поуздан прелазак модела између различитих фаза развоја и употребе. Што се тиче перзистентности података, *MLflow* омогућава складиштење података на различитим типовима складишта, као што су *MinIO*, релационе базе података, или било које друго складиште података које подржава овај алат. Ово омогућава флексибилност и прилагођавање потребама корисника, чинећи *MLflow* моћним алатом за развој и управљање моделима машинског учења.

Redis

Redis[8] је систем за управљање базама података у радној меморији, који се користи као кеш, брокер порука и база података. Настао је 2009. године као пројекат отвореног кода који је развио Салваторе Санфилипо, првобитно за потребе једне од његових компанија. *Redis* је написан у програмском језику *C* и дизајниран је да пружи високу брзину и ефикасност у обради података.

Оно што га чини посебним је његова способност да ради у радној меморији, што омогућава изузетно брз приступ подацима. Подржава разне структуре података као што су ниске, листе, скупови, хешови и сортирани скупови, што га чини веома флексибилним за различите типове апликација. Такође је познат по својој подршци за атомичне операције, што значи да се операције извршавају у једном кораку без могућности прекида, што је корисно за

[7] Званична веб страница: `https://mlflow.org`

[8] Званична веб страница: `https://redis.io`

одржавање конзистентности података.

Неке од корисних могућности *Redis*-а укључују репликацију података, што омогућава прављење резервних копија у реалном времену, и подршку за израду снимка стања базе података. Осим тога, Redis подржава трансакције и покретање скрипти писаних у језику *Lua*, што омогућава сложеније операције над подацима. *Redis*-ова једноставност, брзина и флексибилност чине га изузетно популарним алатом у развоју модерних веб апликација.

PostgreSQL

PostgreSQL[9] је настао 1986. године на Универзитету Калифорније у Берклију, као део пројекта који је предводио професор Мајкл Стоунбрекер. Првобитно назван *POSTGRES* (скраћено од *POST Ingres*), овај систем је изграђен као наследник ранијег система за управљање базама података, *Ingres*, са циљем да се превазиђу ограничења традиционалних релационих база података.

Проблем који је *PostgreSQL* решавао односио се на потребу за управљањем комплексним типовима података и великом количином информација у окружењу где су захтеви за флексибилношћу и проширивошћу били високи. Његова подршка за напредне функције као што су трансакције, интегритет података и проширивост, као и способност да се ради са различитим типовима података, чини га веома популарним избором међу програмерима и администраторима база података.

PostgreSQL је популаран због своје стабилности, проширивости и активне заједнице која константно доприноси његовом развоју. Осим тога, алати као што је *pgAdmin* омогућавају једноставно управљање и визуализацију базе података, чинећи рад још приступачнијим и ефикаснијим.

[9] Званична веб страница: `https://www.postgresql.org`

Глава 3

Имплементација

У оквиру мастер рада имплементиран је систем који обједињује више различитих компоненти, од микросервисне архитектуре до оркестрације и обраде података, као и регистар модела машинског учења [37]. Кроз употребу микросервиса осигурана је скалабилност и флексибилност система, док је оркестрација омогућила координацију извршавања различитих делова апликације и обраду сложених радних токова. Поред обраде података, *Apache Spark* је задужен за машинско учење, уз интеграцију са регистром модела за верзионисање и управљање моделима.

3.1 Микросервисна архитектура

Да би се остварио модеран и флексибилан приступ развоју софтвера, одлучено је да се у изради овог пројекта примени микросервисна архитектура. Ова архитектура је изабрана због својих бројних предности у односу на традиционалне монолитне архитектуре, које могу бити ограничене када су сложени и растући системи у питању. Један од кључних разлога за избор микросервиса је због високе скалабилности не само целог система већ и појединачних делова. Уместо да се цео систем прави као један монолит, микросервиси омогућавају да се свака компонента система развија, тестира и поставља независно. Ова модуларна структура не само да олакшава хоризонталну скалабилност, већ такође омогућава лакше управљање оптерећењем и усклађивање ресурса према специфичним потребама сваке компоненте. Још једна значајна предност микросервисне архитектуре је њена погодност за рад у облаку. Микросервиси су природно компатибилни са овим окружењем,

јер нуде флексибилност у управљању и распоређивању ресурса. Ово олакшава имплементацију и управљање јер се сваки микросервис може самостално развијати, што омогућава лакше прилагођавање променама у оптерећењу и евентуално новим потребама. Намера да се смањи спрегнутост система је још један кључни разлог за избор микросервиса. У монолитним системима, све компоненте су чврсто интегрисане у један код, што може довести до сложених зависности и проблема при ажурирању или променама. Наспрам томе, у микросервисној архитектури свака компонента функционише као независна јединица која комуницира преко дефинисаних интерфејса. Ово смањује зависности и омогућава лакше управљање и надоградњу компоненти без утицаја на целокупан систем.

Компоненте

Микросервисна архитектура за коју смо се одлучили захтевала је пажљиво одабране алате и приступе за њено ефикасно имплементирање. *Docker* је одиграо кључну улогу у овом процесу, пружајући решења која су омогућила лаку изолацију и управљање различитим компонентама система. Један од најзначајнијих аспеката *Docker*-а је његова способност да обезбеди конзистентност окружења за све сервисе. У пракси, већина коришћених сервиса имала је званичне Docker слике, што је значајно поједноставило процес постављања и управљања. Ове слике су, најчешће, већ биле оптимизоване за своје специфичне задатке и ретко су захтевале велике измене, што је смањило потребу за детаљним конфигурацијама. *Docker Compose* је у великој мери олакшао управљање читавим системом, омогућавајући дефинисање свих компоненти на једном месту и покретање целокупног система једном командом. Овај приступ је осигурао да систем буде доследан и лако репродуцибилан на различитим машинама и окружењима. Кључни параметри конфигурације *Docker Compose*-а, као што су *healthcheck*, *depends on*, и *deploy*, били су од суштинске важности.

- Коришћење параметра *healthcheck* омогућило је дефинисање провера исправности сваке компоненте, осигуравајући да се сви сервиси подижу само ако су успешно покренуте зависне компоненте.

- Параметар *depends on* је пружио начин за управљање зависностима између контејнера, чиме се избегло покретање сервиса пре него што њи-

хови предуслови буду испуњени, што је изразито било корисно у ситуацијама где је пре сервиса потребно да се подигне база његових метаподатака.

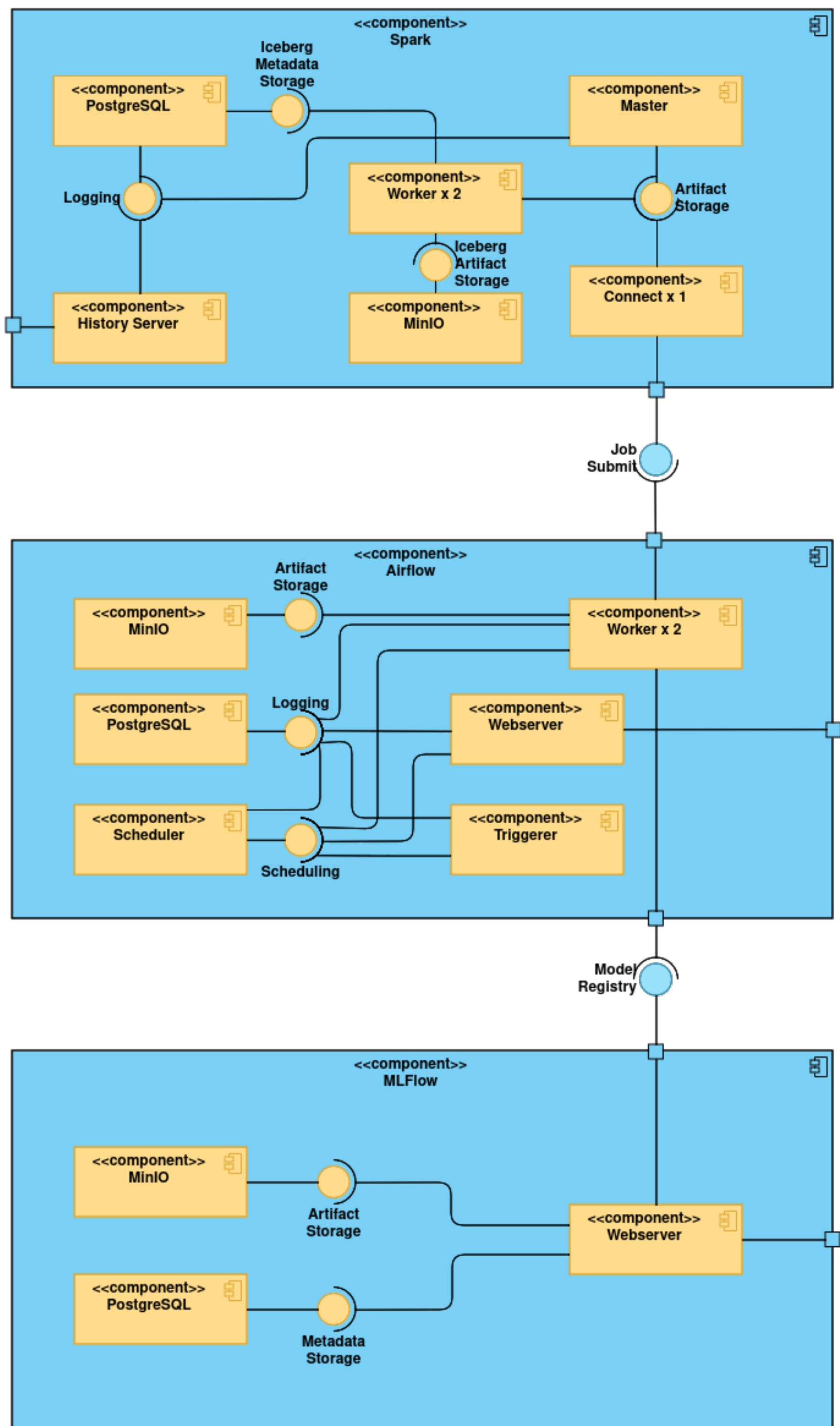
- Параметар *deploy mode replicated* је омогућио лако стварање и управљање репликама контејнера, што је било нарочито корисно за сервисе као што су *Airflow* и *Spark*, где је било потребно обезбедити већи број радника ради оптимизације перформанси и скалабилности.

```
spark-worker:
  build:
    context: spark
    dockerfile: Dockerfile
  depends_on:
    spark-master:
      condition: service_healthy
  environment:
    <<: *common-spark-env
    WORKLOAD: worker
    HEALTCHECK_PORT: 8081
  volumes:
    - ./spark/work-dir:/opt/spark/work-dir
  healthcheck:
    test: curl -f http://localhost:$$HEALTCHECK_PORT
    interval: 5s
    timeout: 5s
    retries: 5
  deploy:
    mode: replicated
    replicas: 2
```

Табела 3.1 пружа преглед коришћених слика у раду. На слици 3.1 приказан је дијаграм компоненти имплементираног система.

Алат	**Слика**	**Мењано**
Airflow	`quay.io/astronomer/astro-runtime:12.0.0`	Не
Spark	`spark:3.5.1-scala2.12-java17-python3-ubuntu`	Да
MLFlow	`ghcr.io/mlflow/mlflow:v2.15.1`	Да
Maeven	`maven:eclipse-temurin`	Да
MinIO	`quay.io/minio/minio:RELEASE.2024-08-17T01-24-54Z`	Да
PostgreSQL	`docker.io/postgres:16`	Не
pgAdmin	`dpage/pgadmin4:8`	Да
Redis	`redis:7`	Не

Табела 3.1: Табела компоненти, коришћених слика и да ли је оригинална слика мењана за потребе рада



Слика 3.1: Дијаграм компоненти.

Складишта

Одлучено је да се за сваку групу сервиса подигне посебно складиште, иако су у питању исте технологије као што су *PostgreSQL* и *MinIO*. Док је било могуће централизовати складиштење података, овај приступ би нарушио принципе микросервисне архитектуре. Незаобилазне предности овог решења су јасна енкапсулација података и системска независност. На овај начин, не само да се избегавају могући проблеми са цурењем података, већ се и поједностављује праћење ресурса, јер је лакше идентификовати и управљати оптерећењем на нивоу појединачних сервиса. Међутим, цена овога је покретање више система за управљање базама података, што смањује перформансе целокупног система. Овај приступ такође олакшава конфигурацију, јер већина база података нуди званичне слике које омогућавају дефинисање параметара путем променљивих окружења, чиме се значајно смањује сложеност и потреба за директном интервенцијом унутар самих сервиса.

3.2 Оркестрација

Оркестрација у модерним токовима података игра кључну улогу као централизовани извор истине, омогућавајући прегледност и управљање сложеним радним процесима. Она обезбеђује да сви задаци буду правилно синхронизовани и да се подаци крећу кроз систем на прави начин, што је од суштинске важности за стабилност и ефикасност система. Важно је нагласити да оркестратор не врши директну обраду података, већ само заказује и управља процесима који су задужени за ту обраду, осигуравајући да се све активности одвијају у правом редоследу и у складу са дефинисаним правилима. У овом раду, као оркестратор је изабран *Apache Airflow*.

Конфигурација алата

Airflow је имплементиран кроз микросервисну архитектуру, где је свака компонента система: распоређивач, покретач, веб сервер и радници, подигнута као засебан контејнер. Поред основних *Airflow* компоненти, подигнути су и посебни контејнери за *PostgreSQL*, *MinIO* и *Redis*, који служе као подршка за рад самог оркестратора. За извршитељ је изабран *Celery* јер омогућава хоризонталну скалабилност, што је кључно у овом контексту. Пошто

кластер *Kubernetes* није доступан, преостали извршитељи би захтевали консолидовање распоредника и радника у један контејнер. Овакав приступ би довео до потребе за вертикалним скалирањем у случају да су потребне боље перформансе, што значи да би било неопходно додати више ресурса појединачним контејнерима. Супротно томе, Celery омогућава лако додавање нових радничких контејнера, чиме се систем скалира хоризонтално, што је много флексибилније и ефикасније решење у овој архитектури. Из овог разлога наш оркестраторски подсистем садржи контејнер Redis. Такође, можемо подићи контејнер Flower за детаљне метрике о стању свих радника, који је доступан само у овом извршитељу.

У оквиру имплементације система Airflow, постоји и посебан контејнер чија је улога да иницијализује сам *Airflow* при подизању. Овај контејнер извршава важне задатке као што су креирање неопходних база података у *PostgreSQL*-у, конфигурација *MinIO* складишта и иницијализација основних сервиса, као што су распоређивач, радници и веб сервер. Осим тога, у овај процес је додата и скрипта која аутоматски иницијализује све потребне везе, променљиве и резервоаре читањем из датотеке у формату *YAML*. Овај приступ осигурава да систем остаје перзистентан и да се важни параметри и конфигурације лако одржавају. Иако смо свесни да би употреба *Secret Manager*-а или *Vault*-а представљала сигурније решење за чување осетљивих информација, безбедност није у фокусу овог рада, па је овај приступ изабран као довољно адекватан за тренутне потребе.

Комуникација међу задацима

Подразумевани поступак *XCom* у *Airflow*-у серијализује све артефакте у формат *pickle* и чува их у *PostgreSQL* бази података. Међутим, овакав приступ није идеалан за дуготрајне системе због неколико разлога [19]. Прво, серијализација таблица података у *pickle* доводи до губитка важних информација о типовима колона. Друго, *PostgreSQL* се оптерећује непотребним подацима који су често релевантни само током извршавања тока, а не након тога, што може узроковати проблеме са меморијом. Поред тога, артефакти могу бити велики, попут целих таблица података или модела, што даље оптерећује базу. На крају, *pickle* има познате безбедносне пропусте и није предвиђен за осетљив или поверљив садржај. Из ових разлога, имплементиран је нови *XCom* поступак који чува метаподатке о променљивима и путању до

артефакта у бази, али сам артефакт смешта у складиште *MinIO*. Таблице података се серијализују у формат *parquet*, *numpy* низови у њихов адекватни формат, док се све што је могуће чува као бајткод у формату *BSON*. Како би се задржала постојећа функционалност, артефакти који не могу бити другачије обрађени и даље се серијализују као *pickle*, при чему је остављена могућност одржаваоцима система да додатно дефинишу имплементацију за типове који су у њиховом случају важни.

Логови и метрике

Поред артефаката, у складишту *MinIO* се такође чувају системски логови *Airflow*-а, што омогућава централизовани приступ и лакше праћење, као и избегавање губитка логова у случају да се контејнер ресетује или уклони. Овај приступ обезбеђује трајнију и поузданију евиденцију извршавања процеса.

3.3 Обрада података

Обрада података обухвата све активности и процесе који се односе на сакупљање, трансформацију, и анализу података са циљем добијања корисних увида и информација. Важно је ову функцију издвојити у засебан сервис како би се осигурала већа флексибилност и ефикасност у управљању подацима, као и омогућило лакше скалирање и оптимизација процеса. За овај део пројекта је изабран *Apache Spark,* који ће такође бити коришћен за основне активности машинског учења посредством модула *MLlib*, иако машинско учење није главни фокус овог рада.

Конфигурација алата

Поред стандардних контејнера за главни чвор и радничке чворове, подигнути су и посебни контејнери за *Spark History Server* и *Spark Connect*. Ови контејнери служе за праћење и визуализацију историје извршења задатака, као и за омогућавање спољним системима да се повежу са кластером *Spark*. Такође, постављени су контејнери за *PostgreSQL* и *MinIO* који подржавају рад *Spark*-а, чиме се осигурава да је систем комплетан и интегрисан.

За *Spark* слику, било је неопходно извршити значајне измене у односу на подразумевану како би се интегрисали сви потребни додатни елементи.

Главни изазов је био како најефикасније додати *JAR*-ове који садрже битне класе за везу са *PostgreSQL*-ом, *MinIO*-ом, *Apache Iceberg*-ом и *Spark Connect*-ом у већ постојећу слику. Да би се ово постигло, направљен је процес који минимизује потребу за руковање самим контејнером и омогућава једноставну интеграцију. Процес почиње од основне слике *Maven*-а која се користи за преузимање зависности. У овом кораку, *pom.xml* датотека је копирана у слику, а затим су све зависности, укључујући и *JAR*-ове који су потребни, преузете и копиране у директоријум унутар слике. Процес интеграције потребних зависности у *Apache Spark* слику реализован је коришћењем *Docker*-ове вишестепене изградње, тако што се зависности, поред конфигурационих фајлова, ископирају у саму слику. У другој фази, користи се основна *Spark* слика, где се преузети *JAR*-ови из претходне фазе интегришу у *Spark* директоријум. Овај приступ осигурава да су сви потребни *JAR*-ови доступни током извршавања *Spark*-а, чиме се олакшава управљање зависностима и конфигурација окружења.

```
#Prikupljanje potrebnih JAR-ova
FROM maven:eclipse-temurin as jar-collector
COPY ./pom.xml .
RUN mvn dependency:copy-dependencies -e -DoutputDirectory=/jars

# Obrada osnovne slike
FROM spark:3.5.1-scala2.12-java17-python3-ubuntu
COPY --from=jar-collector /jars/* /opt/spark/jars/
COPY ./spark-defaults.conf /opt/spark/conf/
COPY ./entrypoint.sh /opt/spark/
WORKDIR /opt/spark
ENTRYPOINT [ "/opt/spark/entrypoint.sh" ]
CMD []
```

У класичним имплементацијама *Apache Spark*-а, *JAR*-ови који садрже неопходне класе и библиотеке морају бити присутни на свим чворовима који учествују у извршавању послова, укључујући и чвор који је поднео захтев да се задатак изврши. Ова потреба за присуством *JAR*-ова на свим чворовима може довести до повећане сложености и спрегнутости система, јер сваки чвор мора бити конзистентан у погледу библиотека и верзија које користи. Када је сам *Spark* у питању, то не представља проблем, јер цео подсистем свакако третирамо као целину, али се у нашем случају употребе задаци заказују из система који није део *Spark*-а - *Airflow*. У нашем случају, овај модел доводи до непотребне спрегнутости различитих компоненти које нису директно повезане, чиме се ствара сложенији систем за управљање и одржавање. Међутим, захваљујући компоненти *Spark Connect*, која иако експериментална,

игра кључну улогу у решавању овог проблема, успели смо да се ослободимо потребе за *JAR*-овима унутар *Airflow* контејнера. *Spark Connect* омогућава *Airflow*-у да се повеже на главни *Spark* чвор и тако управља извршавањем *Spark* послова, без потребе да се директно интегрише са компонентама и библиотекама на свим чворовима. Када *Airflow* користи оператор за *Spark*, он се не повезује директно са главним чвором за извршавање *Spark* послова. Уместо тога, он се повезује на *Spark Connect*, који се налази у нашем кластеру и који постаје контролни чвор нашег задатка, док раднички чворови бивају алоцирани на радничким серверима. Резултат овога је да се контролни чвор не мора алоцирати на радничким серверима, што значи да раднички сервери могу да се користе само за радничке чворове, али прави тачку загушења која се може решити репликацијом севера *Spark Connect*.

Логови и метрике

Сви системски логови и догађаји у нашем систему чувају се унутар *MinIO* складишта, што омогућава њихову перзистентност и сигурност. Овај приступ осигурава да се подаци не изгубе, чак и ако се систем поново покрене или ако дође до било каквих грешака. Једна од значајних предности овог решења је што *Spark History Server* може директно приступити и парсирати логове и догађаје из *MinIO*-а. Ова функционалност омогућава детаљну визуализацију и надгледање свих задатака, укључујући потрошњу ресурса на свим нивоима, као и стабло извршавања израза.

Формат података

Унутар самог *Spark*-а је имплементирана интеграција са *Apache Iceberg*-ом. Каталог метаподатака је дефинисан у *PostgreSQL* бази података, док су артефакти смештени у *MinIO* складишту. Креиран је каталог под именом *iceberg* који је конфигурисан да интегрише ове две компоненте. Кориснику је омогућено да врши упите као да ради са обичном релационом базом података и постојање нивоа апстракције је сакривено од њега, иако се у позадини користи комбинација метаподатака у *PostgreSQL*-у и артефаката у формату *parquet* у *MinIO* складишту.

3.4 Регистар модела машинског учења

Регистар модела машинског учења је кључна компонента у систему који подржава развој и продукцију машинских модела. Потреба за специјалним складиштем произлази из потребе да се сви експерименти, метрике и верзије модела чувају на једном месту. Ово омогућава лако верзионисање и репродуцибилност модела, једноставну промоцију модела у продукцију, и избегавање ситуација у којима различити инжењери креирају моделе одвојено и чувају податке о њима на неодговарајућим местима. У овом раду, за ове сврхе је коришћен *MLflow*.

Конфигурација алата

Имплементација *MLflow*-а у овом систему није захтевала подизање великог броја контејнера: довољан је један контејнер за *MLflow* и његова складишта. Овај контејнер је конфигурисан тако да се сви метаподаци о моделима, укључујући историју, метрике и локације артефаката, чувају у *PostgreSQL* бази, док су сами артефакти смештени у *MinIO* складишту. *MLflow* аутоматски води рачуна о организацији и локацији ових података, што је апстраховано од корисника и поједностављује процес управљања моделима. Слика је морала да се надогради да би подржавала додатне потребе складишта.

```
FROM ghcr.io/mlflow/mlflow:v2.15.1
RUN pip install \
  psycopg2-binary # PostgreSQL
  boto3           # MinIO
ENTRYPOINT []
```

Подразумевано, *MLflow* захтева од корисника да на свом рачунару има инсталиране све потребне библиотеке за складиштење артефаката. Међутим, у овом раду је конфигурисан параметар који омогућава *MLflow*-у да делује као прокси, примајући артефакте од корисника и складиштећи их на одговарајућу локацију. Ово значајно смањује зависности и поједностављује рад инжењера машинског учења, омогућавајући им да се фокусирају на развој модела. *MLflow* се може користити како из *Spark*-а, тако и из *Airflow*-а, те га је потребно третирати као специјализовано складиште за моделе машинског учења, иако ради много више од тога.

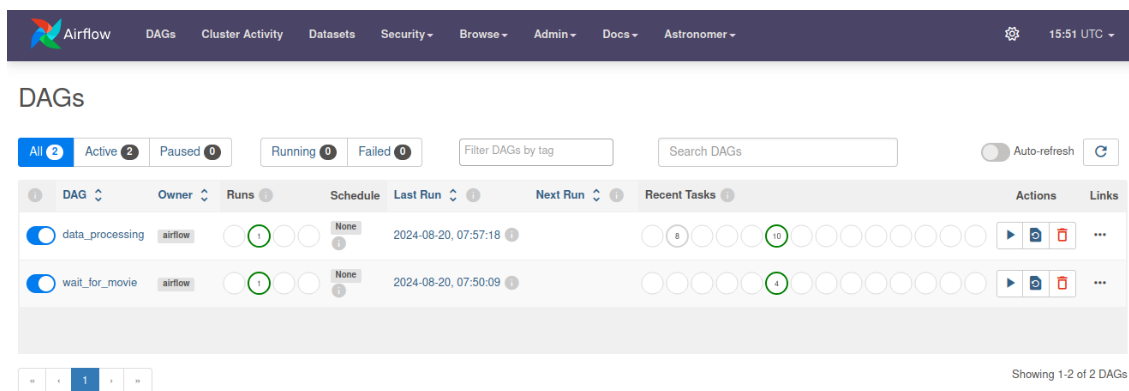
Глава 4

Пример тока података

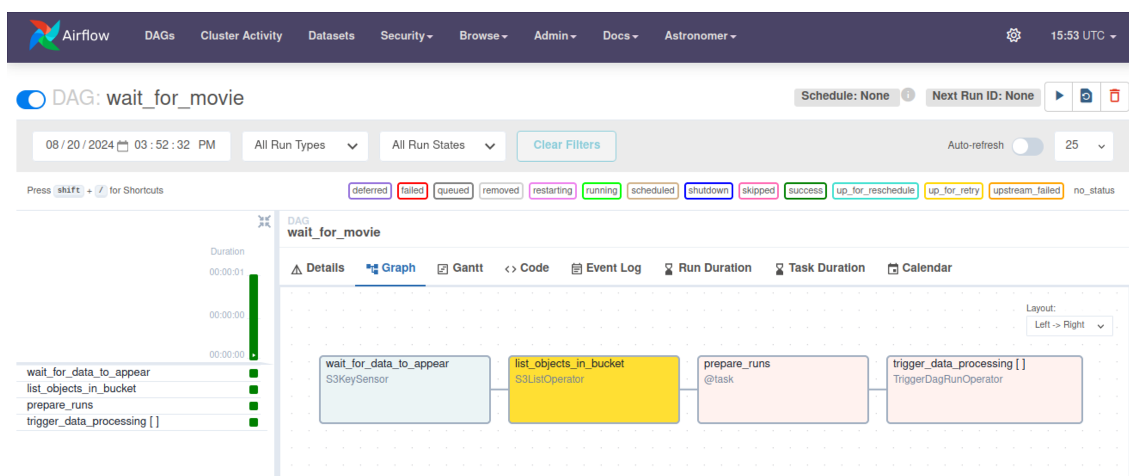
У овој секцији је описан пример тока података који је имплементиран у оквиру рада. Иако примарни циљ рада није сам ток, већ претходно описана архитектура, овај пример служи као илустрација како би модеран ток обраде података требао да изгледа. Ток није оптимизован за продукцијску употребу, већ је креиран са намером да демонстрира што више функционалности система. Конкретно, имплементирана су два *DAG*-а, *"wait for movie"* и *"data processing"*, који се ослањају један на други и могу се посматрати као јединствени ток. Први *DAG* чека на долазак података и, када подаци стигну, за сваку појединачну датотеку покреће други *DAG*. Други *DAG* врши основне трансформације, чишћење података и тренирање модела машинског учења. Циљ овог примера је да покаже колико је једноставно конструисати комплексне токове са бројним компонентама и зависностима у савременим системима. На слици 4.1 приказано је како они изгледају у графичком корисничком интерфејсу.

4.1 Чекање на податке

DAG назван *"wait for movie"* има једноставну али битну улогу: чека да подаци пристигну у кофу унутар складишта *MinIO* и за сваки пристигли *CSV* покрене инстанцу наредног *DAG*-а. На слици 4.2 приказано је како овај *DAG* изгледа у графичком корисничком интерфејсу.



Слика 4.1: *DAG*-ови *"wait for movie"* и *"data processing"* у графичком корисничком интерфејсу алата *Airflow*. Види се колико пута је *DAG* покренут, када је било последње покретање, успешност појединачних задатака, као и опција да се *DAG* ручно покрене, освежи или обрише.



Слика 4.2: *DAG "wait for movie"* у графичком корисничком интерфејсу алата *Airflow*. Види се статус свих задатака, као и однос зависности задатака у облику усмереног ацикличког графа оријентације са лева на десно.

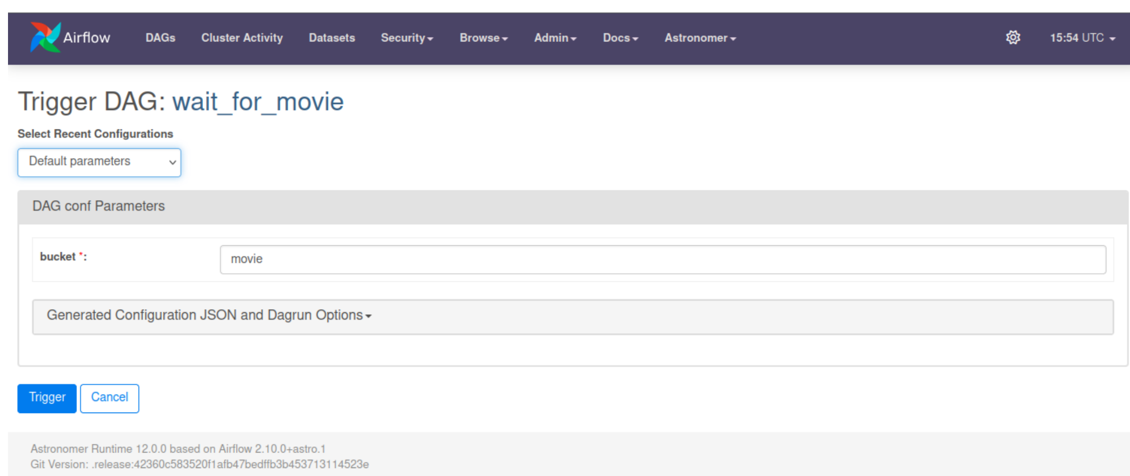
Параметри

DAG је параметризован промењивом *"bucket"* којом можемо одредити која се кофа у складишту посматра. Покренут је са подразумеваним параметрима, али је уведено да би се показало да пажљиво написани *DAG*-ови могу изнова да се користе, што смањује потребу за дупликацијом кода. Такође је угашено аутоматско покретање *DAG*-а у специфичним временским интервалима, мада би ово могло да се омогући прослеђивањем ниске у формату *Cron*

[24]. Наредни код показује како се све ово поставља применом специјализованог декоратора.

```
@dag(
  schedule=None,
  catchup=False,
  params={'bucket' : Param("movie", type="string")},
  render_template_as_native_obj=True
)
def wait_for_movie():
  # Ovde se opisu zadaci i njihove medjuzavisnosti
wait_for_movie()
```

На слици 4.3 приказано је како изгледа ручно покретање, обликовано дефинисаним параметрима *DAG*-а.



Слика 4.3: *DAG “wait for movie“* у графичком корисничком интерфејсу алата *Airflow* приликом ручног покретања. Сви параметри просеђени декоратору бивају графички приказани на страници.

Сензор

Само чекање врши оператор *wait for data to appear*. Користи се подразумевани оператор *S3KeySensor* који чека да у кофа стигне неопходни податак. Докле год је кофа празна, *DAG* неће наставити са радом што обезбеђује да се обрада података неће започети пре него што сви неопходни ресурси буду доступни. Овај приступ смањује могућност грешака услед недостатка података и оптимизује коришћење ресурса тако што спречава непотребно покретање других задатака или DAG-ова који зависе од тих података. Поред тога, омо-

гућава боље планирање и временску координацију у сложеним системима за обраду података.

```
S3KeySensor(
  task_id='wait_for_data_to_appear',
  bucket_key='s3://{{params.bucket}}/*',
  aws_conn_id=AWS_CONN_ID,
  wildcard_match=True,
  verify=False
)
```

Покретање следећег тока

Након што *S3KeySensor* открије нове податке, *S3ListOperator* листа све објекте у кофи. Овај оператор враћа листу путања до свих датотека који се налазе на дефинисаној локацији. Наредни оператор потом прилагоди ту листу, и направи листу конфигурација за покретање следећег *DAG*-а. Свака датотека се обрађује на исти начин, али у току писања кода ми не знамо колико датотека ће постојати у тренутку покретања. Пошто не постоје петље као концепт, ово би представљало проблем да не постоје динамички задаци. Листу конфигурација прослеђујемо *TriggerDagRunOperator*-у, који у методу *expand* дефинише који параметар оператора се мапира из које листе.

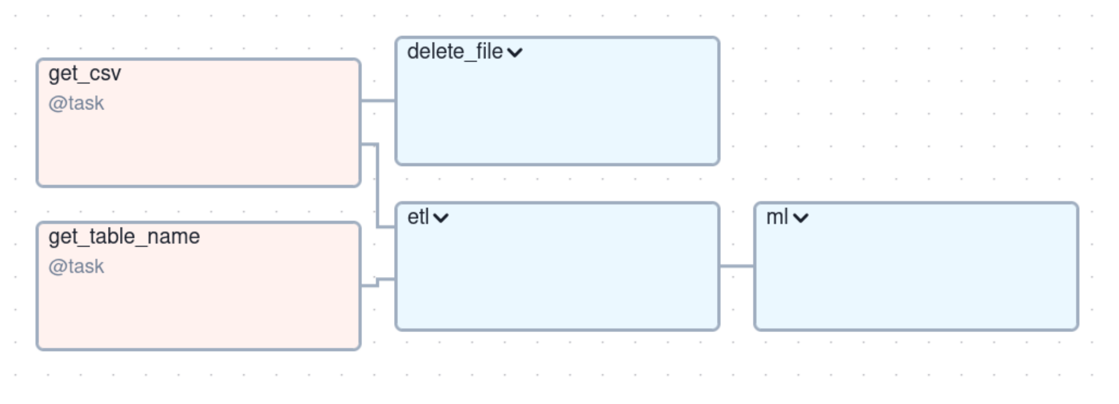
```
@task
def prepare_runs(bucket_name, paths):
  return [
    {"clear" : True, "bucket" : bucket_name, "path" : path}
    for path in paths
  ]
confs = prepare_runs("{{params.bucket}}", datasets.output)
TriggerDagRunOperator.partial(
   task_id = "trigger_data_processing",
   trigger_dag_id="data_processing",
   wait_for_completion=False
).expand(conf=confs)
```

Резултат овог кода је покретање независних инстанци *DAG*-а *"data processing"* за сваку датотеку у *CSV* формату која се налази у кофи *"movie"*.

4.2 Обрада података

DAG назван *"data processing"* врши потпуни циклус обраде података, укључујући њихово преузимање, трансформацију, складиштење и моделовање. Прво, преузима *CSV* датотеку из кофе, затим, након што добије име табеле из путање датотеке, врши опционално брисање датотеке из кофе ако је то

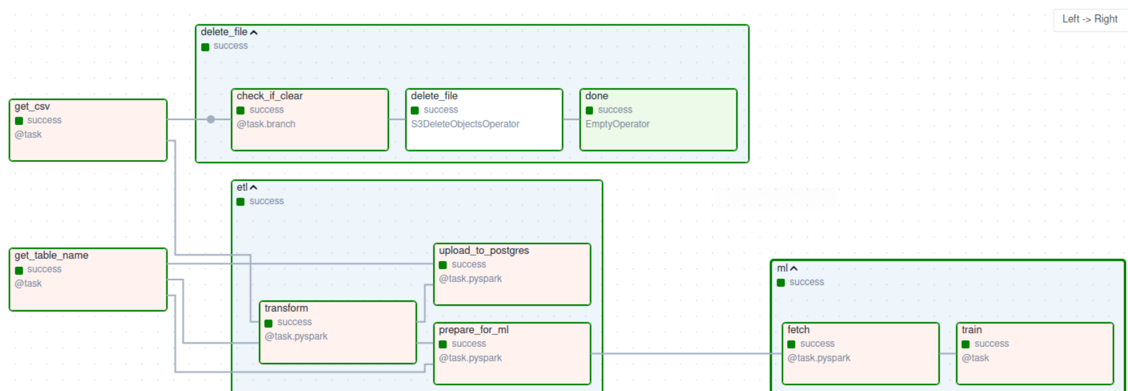
наглашено као параметар. Након тога, врши трансформације над подацима као што су прерада текстуалних вредности, конверзија датума, и убацивање прерађених података у *PostgreSQL* базу. Додатно, припрема податке за машинско учење кроз трансформацију категоричких вредности у нумеричке и уклањање непотребних колона. На крају, ови подаци се користе за тренирање модела линеарне регресије који се чува у *MLflow* систему за праћење експеримената. Пошто је у току података строго дефинисан низ зависности између задатака, могуће је извршавати више задатака у паралели, чиме се убрзава процес обраде и повећава ефикасност. На сликама 4.4 и 4.5 приказано је како овај *DAG* изгледа у графичком корисничком интерфејсу. На слици 4.6 приказано је како изгледа ручно покретање, обликовано дефинисаним параметрима *DAG*-а.



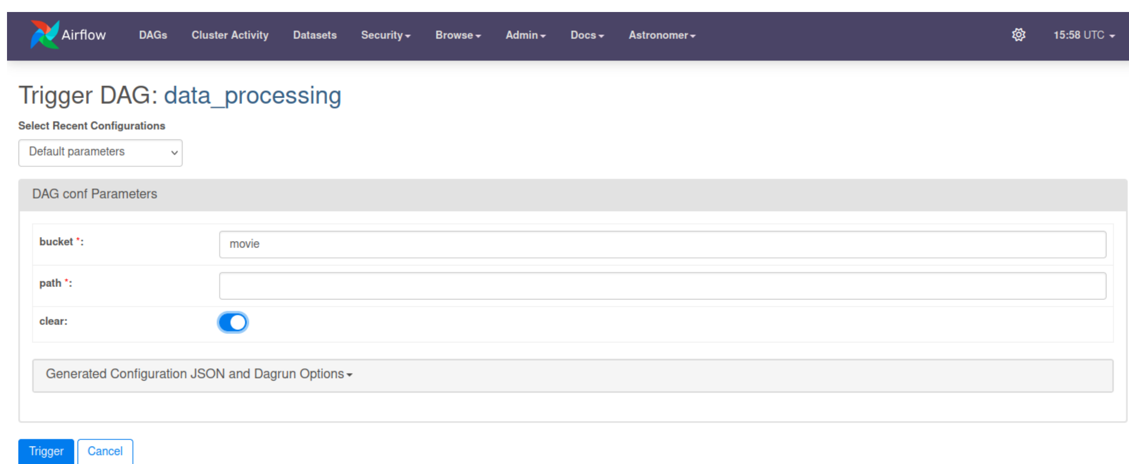
Слика 4.4: *DAG “data processing“* у графичком корисничком интерфејсу алата *Airflow*. Задаци су подељени у логичке групе да се не преоптерети поглед на ток детаљима који нису у датом тренутку битни.

Преузимање датотеке

Оператор *“get csv“* преузима датотеку из кофе и учитава је у глобалну меморију тока података као таблицу података. Овај приступ је ефикаснији од поновног читања датотеке директно из кофе јер избегава више захтева према удаљеном серверу, што може успорити процес обраде ако је кофа географски удаљена. Поред тога, неки сервиси за удаљено складиште наплаћују сваки приступ подацима, тако да овај начин штеди ресурсе. Такође, чувањем датотеке у меморији током извршавања *DAG*-а, смањује се ризик да се ток



Слика 4.5: *DAG "data processing"* у графичком корисничком интерфејсу алата *Airflow*. Групе задатака су проширене, и види се сваки задатак који се извршава у току, као и појединачне зависности између задатака.



Слика 4.6: *DAG "data processing"* у графичком корисничком интерфејсу алата *Airflow* приликом ручног покретања. Сви параметри просеђени декоратору бивају графички приказани на страници.

података ослони на датотеку која би могла бити промењена или избрисана током обраде, чиме се осигурава конзистентност података.

Брисање датотеке

Ова група задатака брише одређену датотеку из кофе након што је успешно учитана, или само ако је параметар *"clear"* подешен приликом покретања *DAG*-а. Ако је овај параметар активиран, датотека ће бити уклоњена, у супротном ће се овај корак прескочити. Ова опција омогућава флексибилност

у руковању датотекама, зависно од потреба корисника.

```
@task.branch(task_id="check_if_clear")
def branch_clear(clear: bool, delete_task_id: str):
  return delete_task_id if clear else None

branch_clear("{{params.clear}}", delete_file.task_id)
```

Обрада датотеке

Ова група задатака, названа *"etl"*, обавља различите трансформације и обраду података помоћу библиотеке *PySpark* која нам дозвољава да дефинишемо начин на који се подаци обрађују у програмском језику *Python*. Први задатак, *"transform"*, конвертује *CSV* податке у таблицу података, брише редове у којима се налазе недозвољене вредности, преименује колоне и врши разне трансформације типа података. Обрађени подаци се затим чувају у табели у формату *Iceberg*.

```
@task.pyspark(conn_id=SPARK_CONN_ID)
def transform(
  csv: pd.DataFrame,
  table_name: str,
  spark: SparkSession
):
  df = (
    spark.createDataFrame(csv)
    .replace("-", None)
    .withColumnsRenamed({
      "Production budget $" : "production_budget",
      "Domestic gross $" : "domestic_gross",
      "Worldwide gross $" : "worldwide_gross"
    }).withColumn(
      "genres", f.split("genres", ",")
    ).withColumn(
      "director_professions",
      f.split("director_professions", ",")
    ).withColumn(
      "production_date",
      f.col("production_date").cast(t.DateType())
    ).withColumn(
      "director_birthYear",
      f.col("director_birthYear").cast(t.IntegerType())
    )
  )
  table_name = f"{table_name}_raw"
  df.writeTo(f"iceberg.{table_name}").create()
  return table_name
```

Други задатак, *"upload to postgres"*, учитава податке из табеле у формату *Iceberg* у базу података *Postgres*. Користи се веза *JDBC* за овај поступак, а

подаци се уписују у табелу у јавном простору базе података, са подешавањем за преклапање постојећих података.

```
@task.pyspark(conn_id=SPARK_CONN_ID)
def upload_to_postgres(
  raw_table_name: str, table_name: str,
  postgres_conn_id, spark: SparkSession
):
  uri = PostgresHook(postgres_conn_id).get_uri()
  uri = urllib.parse.urlparse(uri)
  host = uri.hostname
  port = uri.port
  db   = uri.path[1:]
  jdbc_url = f"jdbc:postgresql://{host}:{port}/{db}"

  df = spark.read.table(f"iceberg.{raw_table_name}")
  df.write.jdbc(
    url=jdbc_url, table=f"public.{table_name}", mode="overwrite",
    properties={
      "user": uri.username,
      "password": uri.password,
      "driver": "org.postgresql.Driver"
    }
  )
  df.printSchema()
  return jdbc_url
```

Трећи задатак, *"prepare for ml"*, припрема податке за машинско учење. Овај задатак модификује табелу додавањем нових колона, претварањем категоријских података у бинарне колоне и уклањањем непотребних колона. Податке чува у *Iceberg*-у.

Машинско учење

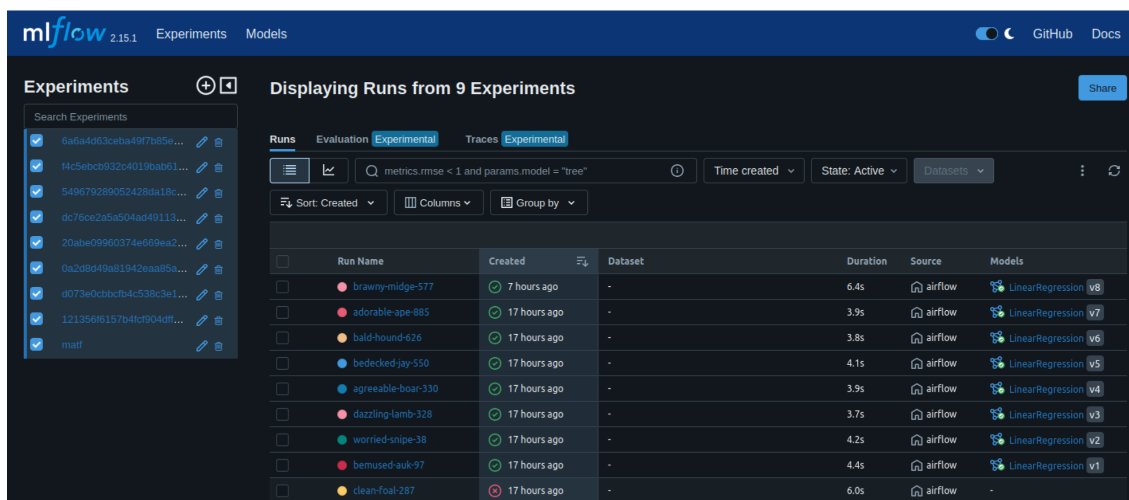
Ова група задатака обавља процес обуке модела машинског учења. Прво, у задатку *"fetch"* подаци се извлаче из *Iceberg*-а у глобалну меморију тока података. Након тога, у другом задатку, ови подаци се користе за обуку једноставног модела линеарне регресије регресије помоћу библиотеке *Scikit-learn*. Овај модел је једноставан, с обзиром на то да машинско учење није главни фокус рада. Резултати обуке, укључујући метрике као што су средња квадратна грешка (енг. *Mean Square Error* - MSE), средња апсолутна грешка (енг. *Mean Absolute Error* - MAE) и коефицијент одређености ($R2$), заједно са моделом, се чувају у *MLflow* за каснију анализу и евалуацију.

```
mlflow.set_experiment("matf")
with mlflow.start_run():
  mlflow.log_param("test_size", test_size)

  model = make_model()
  model.fit(X_train, y_train)
```

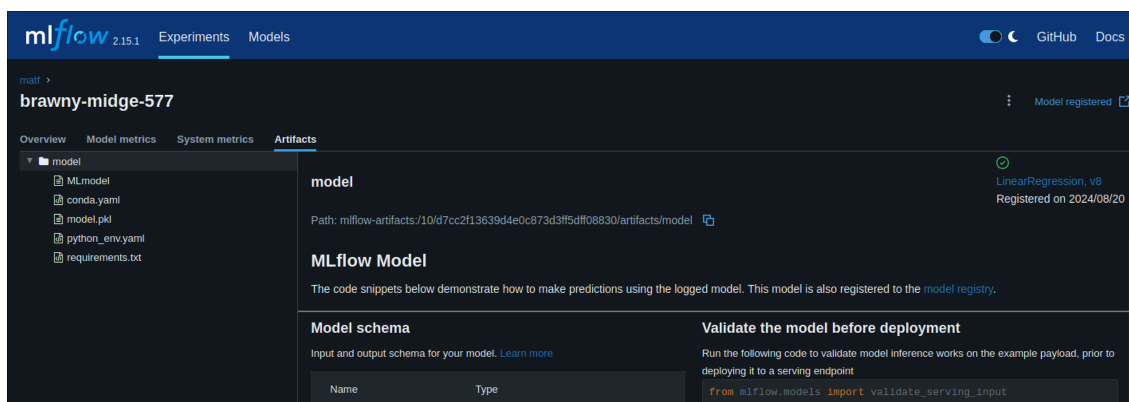
```
y_result = model.predict(X_test)

metrics = evaluate(y_test, y_result)
mlflow.log_metric("mse", metrics["mse"])
mlflow.log_metric("mae", metrics["mae"])
mlflow.log_metric("r2", metrics["r2"])

mlflow.sklearn.log_model(
  model, "model",
  registered_model_name="LinearRegression"
)
```

Коришћењем блока за праћење тренинга омогућава се надгледање трајања процеса и идентификација могућих грешака у извршавању. Унутар њега корисник сам дефинише параметре и метрике које би пратио и модел се аутоматски серијализује и чува у складишту ради касније употребе и анализе.

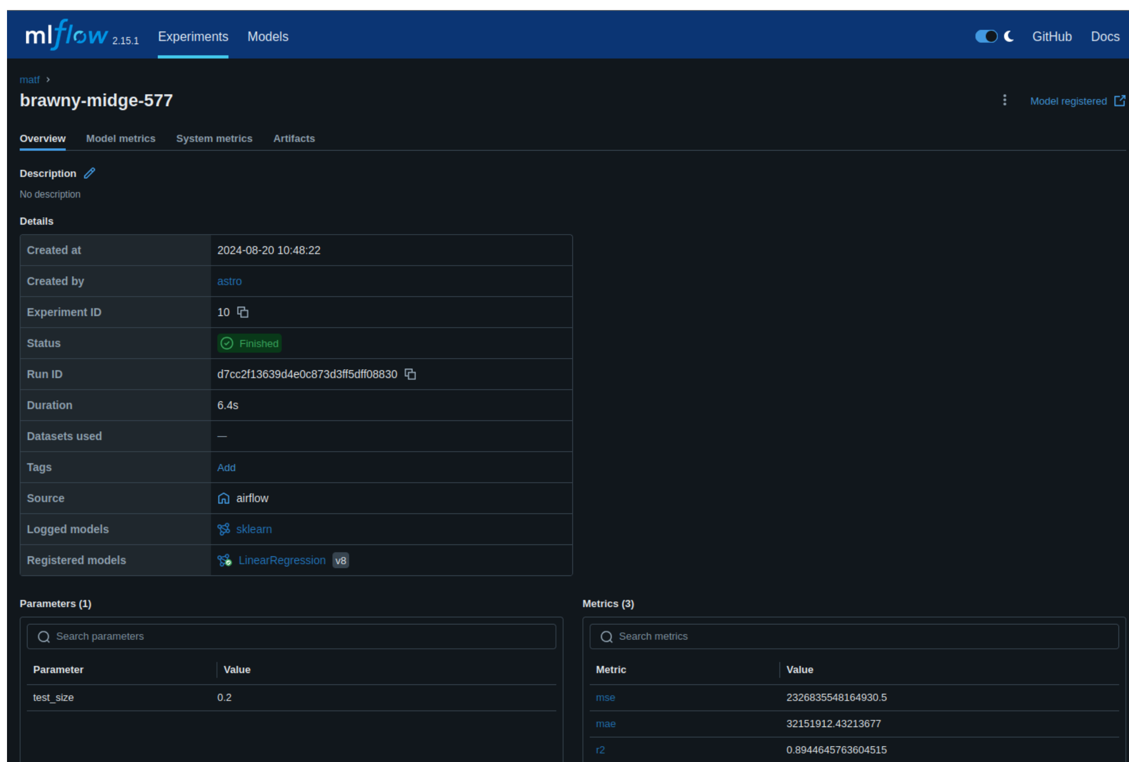
На слици 4.7 је приказано како експерименти изгледају у графичком корисничком интерфејсу алата *MLflow*, а на сликама 4.9 и 4.8 се могу видети метаподаци о специфичном моделу.



Слика 4.7: Преглед експеримената у графичком корисничком интерфејсу алата *MLflow*. Види се комплетна историја тренирања верзионисаног модела.



Слика 4.8: Преглед артефакта модела у графичком корисничком интерфејсу алата *MLflow*. Ове датотеке генерише програмски интерфејс апликације *MLflow* на основу окружења у којем се експеримент извршава, чинећи модел потпуно репродуцибилним.



Слика 4.9: Преглед параметара и метрика модела у графичком корисничком интерфејсу алата *MLflow* за специфични експеримент. Ови подаци могу да се претражују, дајући нам једноставан увид у стање модела.

Глава 5

Дискусија

Током израде рада, идентификовани су одређени аспекти система који могу представљати ограничења или недостаке у предложеној архитектури. У овом одељку биће размотрени ови недостаци и потенцијалне области за побољшање у неком будућем раду.

Ако је неопходно ажурирати систем или додати нове библиотеке, тај процес није ни брз ни једноставан. Потребно је искључити постојећи подсистем и покренути нову верзију, што подразумева период у коме систем није функционалан и може доћи до прекида у раду. Међу обрађеним алатима не постоји ниједан прилагођен за високу доступност. Још један недостакак је мањак подршке за аутоматско скалирање система, што значи да, иако је у питању само један параметар, неко мора ручно да изврши ову промену.

Spark Connect може представљати уско грло у систему, нарочито када се повећа број радника унутар *Airflow*-а. Са друге стране, чињеница да је у питању један контејнер који се лако може реплицирати значајно умањује овај проблем. Такође, пошто је *Spark Connect* преалоцирани контролни чвор, његова издвојеност значи да ресурси који би иначе били потрошени на контролни чвор у самом Спарк кластеру остају доступни за радничке чворове, тако да ресурси нису изгубљени. И поред свега наведеног, систем који би могао да га уклони без повећања спрегнутости имао би предност у односу на тренутно имплементирани систем, јер би имао једну мање компоненту о којој мора да се води рачуна.

Покренуто је посебно складиште метаподатака за сваки подсистем. Претпоставка је да појединачним складиштима није потребна велика количина ресурса иако се често користе јер је количина података која протиче кроз њих мала. У том случају, надгледање би заиста било лакше, јер би се посма-

трањем употребе радне меморије и процесора целог контејнера могло закључити да ли функционише исправно, с обзиром на то да мали домен употребе не оставља простор за неочекиване варијације у перформансама. Међутим, у окружењима у којима је ефикасна употреба ресурса од кључног значаја услед њиховог мањка ово можда не би био добар приступ проблему, јер се покреће више система за управљање база података а један би био довољан. Консолидовање свих складишта у једно би значајно уштедело ресурсе, али би то захтевало активно управљање схемама података. У случају проблема, дијагностика би постала сложенија и захтевала би напредније механизме за надгледање него што су обрађени у раду.

Безбедност информација, иако важна тема у развоју софтвера, није обрађена у овом раду. Везе и креденцијали се прослеђују у сировом тексу као промењиве окружења. Штавише, везе и креденцијали између компоненти система су фиксно дефинисани, што значи да би замена било које компоненте захтевала ручно ажурирање ових вредности. Коришћењем *Docker Compose*-а ови проблеми не могу да се превазиђу.

Постоје различити алати који би могли заменити тренутну архитектуру пројекта и значајно побољшати њену функционалност и ефикасност. Ови алати би омогућили боље управљање ресурсима, већу скалабилност, поузданост и поједноставили комплексне операције, што би довело до унапређеног перформанса и лакшег одржавања система. Ови алати нису искоришћени услед недостатка ресурса али их је важно поменути јер би драстично олакшали одржавање система описаног у овом раду.

Инфраструктура

Kubernetes[1] је алат за оркестрацију контејнера који би могао да донесе бројне бенефите у пројекат. За разлику од тренутног решења са *Docker Compose*-ом, где је рециклирање контејнера неопходно због ограничених ресурса, *Kubernetes* би захтевао алокацију ресурса за сваки контејнер, али би обезбедио боље управљање и скалабилност. Постоје готове имплементације целих система путем *Helm*-а[2], које укључују функционалности као што су балансирање оптерећења, аутоматско скалирање ресурса и интеграције са системима за надгледање кластера. Додатно, *Airflow* би могао да се пребаци на

[1] Званична веб страница: `https://kubernetes.io`
[2] Званична веб страница: `https://helm.sh`

извршитељ *Kubernetes*, што би омогућило да радници буду активни само по потреби, чиме би се смањила укупна потрошња ресурса, и компонента *Spark Connect* би из имплементације могао да се избаци без повећања спрегнутости.

Terraform[3] је алат који омогућава дефинисање инфраструктуре у облику кода. Он би довело до модуларнијег дефинисање целокупног система, где параметри за повезивање контејнера не би морали ручно да се уносе, већ би били аутоматски одређени. *Terraform* је изузетно користан у окружењима где постоји цео процес континуалне интеграције и испоруке, јер омогућава лако управљање и верзионисање инфраструктурних компоненти, што би поједноставило и убрзало развој и одржавање система.

Контрола тока

Инспирисани алатом *Apache Airflow*, развијени су алати *Mage*[4] и *Dagster*[5], који такође служе за оркестрацију и аутоматизацију сложених токова података. *Mage* нуди интерактивно корисничко искуство са нагласком на лакоћу употребе и брзо постављање експеримената, док *Dagster* уводи концепт дефинисања задатака заснован на сигурности типова, чиме побољшава управљање зависностима и обраду грешака. Оба алата препознала су и решили неке од пропуста које има *Airflow*, као што су комплексност конфигурације и ограничене могућности за тестирање. Ипак, због своје релативне младости и још увек активног развоја, тренутно нису довољно зрели да би се могли сматрати потпуно виабилним решењима у продукцијским окружењима.

Обрада података

DBT[6] (енг. *Data Build Tool)* је алат намењен за трансформацију и мапирање података у складиштима података, омогућавајући корисницима да једноставно пишу *SQL* код за извршавање трансформација, а затим га аутоматски претварају у низове корака који се могу лако пратити и верзионисати. *DBT* је постао веома популаран у свету обраде података захваљујући својој способности да трансформације учини транспарентнијим и лакшим за управљање. *Apache Airflow* има интеграцију са *DBT* пројектима кроз званични додатак назван *Cosmos*, који омогућава да се *DBT* токови визуализују као

[3] Званична веб страница: `https://www.terraform.io`
[4] Званична веб страница: `https://www.mage.ai`
[5] Званична веб страница: `https://dagster.io`
[6] Званична веб страница: `https://www.getdbt.com`

DAG-ови у *Airflow*-у, пружајући корисницима јасан увид у процес обраде података. Иако је *DBT* изузетно популаран и широко коришћен, за овај пројекат је одлучено да се користи само *Spark* како би се смањила комплексност, али у пракси се оба алата често користе упоредо за максималну ефикасност у трансформацији података.

Dask[7] је алат за паралелну и дистрибуирану обраду података у *Python*-у. Прилагођен је овом језику, што га чини идеалним за тимове који су већ навикли да програмирају у *Python*-у, омогућавајући им да лако интегришу постојеће библиотеке и алате. Због своје флексибилне архитектуре, лакше је додавати и управљати зависностима у пројектима, чиме се поједностављује рад у сложеним окружењима. У одређеним проблемима *Dask* показује боље перформансе од *Apache Spark*-а, због своје лакше тежине и фино подешених операција [8]. Ипак, треба имати на уму да је релативно нов алат, што значи да још увек може имати мање стабилности и подршке у односу на зрелије алате као што је [27], услед чега није обрађиван у овом раду.

Надгледање

Grafana[8] би служила као централизовани систем за надгледање свих процеса у систему. За разлику од тренутне архитектуре, где је сваки сервис задужен да обезбеди свој систем за надгледање процеса, *Grafana* може да парсира логове готово сваке апликације, омогућавајући детаљно праћење и анализу. Такође, могле би да се комбинују различити логови и догађаји, чиме би се лакше идентификовала и решавала проблематична понашања у систему.

Prometheus[9] је систем за надгледање и упозоравање отвореног кода који је дизајниран за прикупљање и обраду временских серија. Он се интегрише са многим алатима и платформама, укључујући *Kubernetes*, што га чини идеалним за праћење перформанси инфраструктуре и апликација у реалном времену. У комбинацији са *Grafana*-ом, *Prometheus* пружа моћан систем за визуелизацију и надгледање који омогућава тимовима да брзо детектују и реагују на потенцијалне проблеме у систему.

[7] Званична веб страница: `https://www.dask.org`
[8] Званична веб страница: `https://grafana.com`
[9] Званична веб страница: `https://prometheus.io`

Безбедност

Vault[10] је алат отвореног кода намењен за управљање осетљивим подацима као што су креденцијали, кључеви, сертификати и други тајни подаци. Омогућава безбедно похрањивање и динамичко генерисање тајни, што смањује ризик од неовлашћеног приступа. Он подржава аутоматску ротацију тајни и обезбеђује фино подешене контроле приступа, чиме се осигурава да само овлашћени корисници и сервиси имају приступ одређеним тајнама.

[10] Званична веб страница: `https://www.vaultproject.io`

Глава 6

Закључак

У овом раду описан је приступ изградњи дистрибуираног система за обраду података, са акцентом на употребу савремених алата и технологија. У раду су искоришћени *Apache Airflow* за оркестрацију, *Apache Spark* за обраду података и *Docker* за паковање зависности.

Основна идеја овог рада је изградња и имплементација дистрибуираног система за обраду података који користи савремене алате и технологије. Циљ је био развој система који омогућава ефикасно прикупљање, складиштење и обраду података у реалном времену, уз висок степен скалабилности и отпорности на отказе. Интеграција различитих алата и технологија отвореног кода допринела је постизању овог циља.

Ток података имплементиран у овом раду обухвата све фазе од прикупљања података до њихове обраде и анализе. Процес почиње чекањем на доступност тражених података који се затим транспортују и складиште у централизованом складишту података. Након тога, подаци се подвргавају различитим процесима обраде, укључујући филтрирање и трансформацију. Коначна фаза подразумева тренирање модела на обрађеним подацима и његово складиштење.

Постигнути резултати потврђују важност коришћења напредних алата и технологија у савременом инжењерству података. Будући развој овог система може укључити додатне оптимизације и интеграције нових алата, од којих су неки већ описани у раду, у циљу побољшања ефикасности и безбедности система у брзо променљивом окружењу података. Аутоматско скалирање значајно би смањило потребу за ручном интервенцијом у систему. Коришћење система за размену тајни допринело би безбедности и употребљивости система у продукционом окружењу.

Библиографија

[1] TechTarget Adam Riglian. Jenkins. https://www.techtarget.com/searchsoftwarequality/definition/Jenkins. Accessed: 27-Aug-2024.

[2] Apache Airflow. Xcoms. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/xcoms.html. Accessed: 27-Aug-2024.

[3] SelectFrom Ani. Spark optimization : Reducing shuffle. https://selectfrom.dev/spark-optimization-reducing-shuffle-9cb2c109e977. Accessed: 27-Aug-2024.

[4] Astronomer. Airflow sensors. https://www.astronomer.io/docs/learn/what-is-a-sensor. Accessed: 27-Aug-2024.

[5] Swapnil Bhartiya. Why docker is the favourite of the devops world? https://www.opensourceforu.com/2017/02/docker-favourite-devops-world/, 2017. Accessed: 15-Aug-2024.

[6] Comtronic Blog. Automation with cron job on centos 8. https://comtronic.com.au/automation-with-cron-job-on-centos-8. Accessed: 27-Aug-2024.

[7] Anuj Chourasia. Understanding how containers work behind the scenes. https://blog.kubesimplify.com/understanding-how-containers-work-behind-the-scenes, 2023. Accessed: 15-Aug-2024.

[8] Dask. Comparison to spark. https://docs.dask.org/en/latest/spark.html, 2021. Accessed: 15-Aug-2024.

[9] Databricks. What is a resilient distributed dataset (rdd)? https://www.databricks.com/glossary/what-is-rdd. Accessed: 27-Aug-2024.

[10] Docker. Docker compose overview. https://docs.docker.com/compose. Accessed: 27-Aug-2024.

[11] Docker. Swarm mode overview. https://docs.docker.com/engine/swarm. Accessed: 27-Aug-2024.

[12] Docker. What is an image? https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image. Accessed: 27-Aug-2024.

[13] Amazon AWS Documentation. What is amazon s3? https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html, 2021. Accessed: 30-Aug-2024.

[14] Apache Software Foundation. The apache software foundation announces apache™ spark™ as a top-level project. https://news.apache.org/foundation/entry/the_apache_software_foundation_announces50, 2014. Accessed: 15-Aug-2024.

[15] Apache Software Foundation. Spark cluster mode overview. https://spark.apache.org/docs/latest/cluster-overview.html, 2014. Accessed: 15-Aug-2024.

[16] Apache Software Foundation. Airflow proposal. https://cwiki.apache.org/confluence/display/incubator/AirflowProposal, 2016. Accessed: 15-08-2024.

[17] Apache Software Foundation. High-level spark connect architecture. https://spark.apache.org/spark-connect/, 2018. Accessed: 15-Aug-2024.

[18] Apache Software Foundation. The apache software foundation announces apache™ airflow™ as a top-level project. https://news.apache.org/foundation/entry/the-apache-software-foundation-announces44, 2019. Accessed: 15-Aug-2024.

[19] Apache Software Foundation. Airflow best practices. https://airflow.apache.org/docs/apache-airflow/stable/best-practices.html#communication, 2021. Accessed: 15-Aug-2024.

[20] Apache Software Foundation. Airflow core concepts. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html, 2024. Accessed: 15-Aug-2024.

[21] Apache Software Foundation. Airflow executors. https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/executor/index.html, 2024. Accessed: 15-Aug-2024.

[22] Apache Software Foundation. Apache airflow project overview. https://airflow.apache.org/docs/apache-airflow/stable/project.html, 2024. Accessed: 15-Aug-2024.

[23] Apache Software Foundation. Creating a custom operator. https://airflow.apache.org/docs/apache-airflow/stable/howto/custom-operator.html, 2024. Accessed: 15-Aug-2024.

[24] IBM. Unix cron format. https://www.ibm.com/docs/en/db2oc?topic=task-unix-cron-format. Accessed: 27-Aug-2024.

[25] IBM. What is a data lake? https://www.ibm.com/topics/data-lake. Accessed: 21-Aug-2024.

[26] IBM. What is apache mapreduce? https://www.ibm.com/topics/mapreduce. Accessed: 27-Aug-2024.

[27] IBM. What is apache spark? https://www.ibm.com/topics/apache-spark. Accessed: 21-Aug-2024.

[28] IBM. What is etl? https://www.ibm.com/topics/etl. Accessed: 21-Aug-2024.

[29] IBM. What is a data pipeline? https://www.ibm.com/topics/data-pipeline, 2021. Accessed: 30-Aug-2024.

[30] IBM. What is a relational database? https://www.ibm.com/topics/relational-databases, 2021. Accessed: 30-Aug-2024.

[31] IBM. What is docker? https://www.ibm.com/topics/docker, 2024. Accessed: 21-Aug-2024.

[32] IBM. What is docker? https://www.ibm.com/topics/devops, 2024. Accessed: 21-Aug-2024.

[33] Paul Krill. What is docker? the spark for the container revolution. https://www.infoworld.com/article/2253801/what-is-docker-the-spark-for-the-container-revolution.html, 2023. Accessed: 15-Aug-2024.

[34] Docker Labs. Everything you want to know about docker daemon. https://dockerlabs.collabnix.com/beginners/components/daemon. Accessed: 27-Aug-2024.

[35] MinIO Blog Matt Sarrel. Digging deeper into iceberg: Acid transactions on tables. https://blog.min.io/iceberg-acid-transactions. Accessed: 27-Aug-2024.

[36] Rani Osnat. A brief history of containers: From 1970s chroot to docker in 2016. https://www.aquasec.com/blog/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016/, 2016. Accessed: 15-Aug-2024.

[37] Petar Đorđević. Implementacija rada. https://gitlab.com/matf-bg-ac-rs/matf-petar-dordevic-master-rad, 2024.

[38] Monika Solanki, Bojan Božić, Christian Dirschl, and Rob Brennan. Towards a knowledge driven framework for bridging the gap between software and data engineering. *Journal of Systems and Software*, 149:476–484, 2019.

[39] VMWare. What is a hypervisor? https://www.vmware.com/topics/hypervisor. Accessed: 27-Aug-2024.

Биографија аутора

Петар Ж. Ђорђевић рођен је 09.10.1995. године у Београду. Завршио је основну школу “Иван Горан Ковачић“, као и природно-математички смер Прве београдске гимназије.

Смер “Информатика“ на Математичком факултету Универзитета у Београду уписује 2014. године. На наведеном смеру је дипломирао 2022. године са просечном оценом 8,46. Мастер студије уписује на истом факултету одмах након дипломирања.

Од јула 2022. до фебруара 2024. године радио је као инжењер података и инжењер машинског учења у фирми “Куцерис“, где је био одговоран за развој и имплементацију решења за управљање подацима и примену алгоритама машинског учења.

У октобру 2022. године бива изабран у звање “Сарадник у настави“ паралелно са мастер студијама и послом у фирми. Држи вежбе из курсева “Развој софтвера“ и “Алати за развој софтвера“ на четвртој години основних студија смера “Информатика“, као и вежбе из курса “Дизајн програмских језика“ на трећој години основних студија смера “Рачунарство и информатика“.

Област интересовања укључује пре свега развој софтвера и дистрибуирани системи.