МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра інформаційних технологій

«До захисту допущено»
В.о. завідувача кафедри

_____ Світлана ВАЩЕНКО

____ _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»,
освітньо-професійної програми «Інформаційні технології проектування»
на тему: Підсистема зберігання та обміну даними системи підтримки прийняття рішень при управлінні гібридною енергомережею

Здобувача групи __ІТм-24__ Горбатенко Олександр Олексійович_
(шифр групи) (прізвище, ім'я, по батькові)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Олександр ГОРБАТЕНКО
(підпис) (Ім'я та ПРІЗВИЩЕ здобувача)

Керівник професор кафедри ІТ, д.т.н., доцент Сергій ТИМЧУК _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Суми – 2023

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра інформаційних технологій

Спеціальність 122 «Комп'ютерні науки»

Освітньо-професійна програма «Інформаційні технології проектування»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІТ

Світлана ВАЩЕНКО

«___» _____ 2023 р.

ЗАВДАННЯ

на кваліфікаційну роботу магістра студентові

Горбатенко Олександр Олексійович

(прізвище, ім'я, по батькові)

1 Тема кваліфікаційної роботи Підсистема зберігання та обміну даними системи підтримки прийняття рішень при управлінні гібридною енергомережею

затверджена наказом по університету від «08» листопада 2023 р.№ 1249-VI

2 Термін здачі студентом кваліфікаційної роботи «_15_» ___грудня__ 2023 р.

3 Вхідні дані до кваліфікаційної роботи перелік вимог до підсистеми зберігання та обміну моделей, файли прогнозних моделей

4 Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) аналіз предметної області, постановка задачі та методи дослідження, проектування підсистеми зберігання та обміну прогнозних моделей для системи підтримки прийняття рішень при управлінні гібридною енергомережою, практична реалізація підсистеми зберігання та обміну прогнозних моделей

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових слайдів презентації) Актуальність, постановка задачі, задачі, аналіз типів хмарних сховищ даних, таблиця порівняння типів хмарних сховищ, схема вибору типу сховища, функціональні вимоги до підсистеми зберігання та обміну даними, інструменти реалізації, контекстна діаграма процесу отримання даних моделей системи підтримки прийняття рішень, декомпозиція діаграми процесу отримання метаданих, декомпозиція діаграми процесу отримання бінарних даних, діаграма варіантів використання підсистеми зберігання та обміну моделей, практична реалізація, демонстрація роботи, висновки.

6. Консультанти випускної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

Дата видачі завдання _____.

Керівник _____

(підпис)

Завдання прийняв до виконання _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Формування технічного завдання	13.10.23 – 18.10.23	
2	Аналіз методів зберігання даних	18.10.23 – 23.10.23	
3	Розробка функціональних вимог	23.10.23 – 30.10.23	
4	Проектування архітектури підсистеми	30.10.23 – 12.11.23	
6	Розробка підсистеми	12.11.23 – 30.11.23	
7	Розгортання підсистеми	30.11.23 – 01.12.23	
8	Тестування підсистеми	01.12.23 – 07.12.23	
9	Підготовка технічної документації	07.12.23 – 13.12.23	
10	Завершення та аналіз роботи	13.12.23 – 15.12.23	

Магістрант _____ **Олександр ГОРБАТЕНКО**

Керівник роботи _____ **д.т.н., доц. Сергій ТИМЧУК**

АНОТАЦІЯ

Кваліфікаційна робота магістра на тему «Підсистема зберігання та обміну даними системи підтримки прийняття рішень при управлінні гібридною енергомережею» є актуальною через зростаючу потребу в ефективному управлінні ресурсами в енергетичній галузі. Робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел на 42 найменувань, та додатків, загальним обсягом 83 сторінок, з яких 51 сторінка становить основний текст, 5 сторінки списку використаних джерел, 27 сторінок додатків.

Метою дослідження є розробка підсистеми зберігання та обміну моделями, яка інтегрується в систему підтримки прийняття рішень для управління гібридними енергомережами. Робота включає аналіз існуючих методів зберігання даних в хмарних сховищах, розробку концептуальної моделі, проектування та реалізацію підсистеми. Особлива увага приділяється організації зберігання прогнозованих моделей, які представляють собою як бінарні файли, так і метадані, такі як конфігурація моделей, а також залежності використовуваних при процесі прогнозування.

Одним із ключових результатів є проведення тестування підсистеми з використання реальних прогнозованих моделей різних типів. Розробка має практичне значення для використання у системах підтримки прийняття рішень, оскільки сприяє підвищенню ефективності управління моделями прогнозування.

Щодо впровадження, розроблена підсистема може бути інтегрована із системою підтримки прийняття рішень при управлінні енергомережами як локально, так з використанням провідних хмарних сервісів.

Ключові слова: система підтримки прийняття рішень, гібридна електромережа, хмарне сховище даних, S3 протокол, REST API, Spring Boot, мікросервіс, MinIO, контейнеризація, Docker.

ЗМІСТ

ВСТУП

Енергія життєво необхідна для стрімкого прогресу сучасного світу. Це важливий фактор, який відповідає за промисловий та сільськогосподарський розвиток. По всьому світові виробництво енергії є причиною приблизно сімдесяти відсотків викидів парникових газів. За відсутності змін у системі виробництва енергії, зміни клімату продовжать викликати незворотні зміни у навколишньому середовищі [1].

Відновлювана енергія (ВЕ) – енергія, що отримується від невичерпних, швидко поповнюваних джерел, які можуть відновлюватися природним способом. До таких джерел відносять сонячне світло, вітер, гідро-енергію (припливи та відпливи, течії). Використання цих джерел вигідне для країн та світу з екологічного та технологічного боку. Відновлювана енергія доступна для всіх країн і зокрема тих, що відчувають нестачу природної невідновлюваної сировини, такої як вугілля чи газ, які грають величезну роль на світовому ринку [2].

Наприкінці 2021 року в Європі відбулося невиправдано високе штучне підвищення цін на російські енергоносії, яке можна інтерпретувати як енергетичну агресію РФ. 24 лютого 2022 року Росія розпочала повномасштабну війну проти України, знищуючи населення, транспортну та житлову інфраструктуру.

Станом на 10 серпня 2022 р. було пошкоджено або зруйновано 300 об’єктів комунальної теплоенергетики, серед яких — 10 потужних ТЕЦ. Восени РФ перейшла до цілеспрямованого руйнування енергетики України. Внаслідок цього енергетичного тероризму з 10 жовтня по 10 листопада 2022 р. пошкоджено або зруйновано третину об’єктів енергетичної інфраструктури (близько 400 одиниць). У листопаді-грудні 2022 р. ракетні удари по об’єктах енергетики стали ще інтенсивнішими. Загалом станом на 10 лютого 2023 р. енергосистема України пережила 14 масштабних ракетних атак і ще більше бомбардувань дронами [3].

Важливою складовою економічної політики воєнного часу має бути розроблення стратегічного плану заходів щодо реконструкції галузевої інфраструктури та відновлення нормального функціонування енергетики в умовах післявоєнного розвитку країни. В енергетичному секторі України воєнного періоду першочерговим завданням є мобілізація та оптимізація використання наявних енергоресурсів для надійного енергопостачання за умови ефективного енергоменеджменту. Енергобезпека, енергостійкість, енергогнучкість, енергоживучість і підвищення енергоефективності стають основними імперативами і пріоритетами політики у світі, особливо в Європі і, безперечно, в Україні. Нагальною потребою є скорочення використання викопного палива і збільшення частки відновлюваних джерел енергії та водню [4].

Децентралізація енергопостачання, зелена енергетика, місцева енергетика — це тренд розвитку, зумовлений і посилений війною. Він доповнює світові зусилля з отримання чистої (низьковуглецевмісної або вуглецевонейтральної) енергії з метою мінімізації згубних наслідків глобального потепління клімату [5].

Світовий попит на енергію задовольняється використанням викопного палива, але оскільки воно наносить величезний шкідливий вплив, потрібно знайти альтернативу. Нове рішення повинно бути екологічно чистим джерелом вироблення енергії, щоб забезпечити «зелене» та незалежне (автономне) середовище. Джерелами, що відповідають наведеним характеристикам є відновлювані джерела енергії [6].

Актуальність. Для України сьогодні вкрай важливою є енергоефективно орієнтована інноваційна модернізація енергетики з проривним тактичним і стратегічним оновленням економіки [7].

Під тактичним оновленням розуміється використання вже розроблених і апробованих на практиці технологій, які гарантують помітний техніко-економічний ефект «уже зараз», а під стратегічним — проведення відповідних фундаментальних досліджень і розроблення за їх результатами нових проривних технологій, зокрема створення інтелектуальних мереж електро-, тепло- і водозабезпечення [8].

Перехід на використання зеленої енергії потребує встановлення спеціального устаткування, сонячних панелей, вітрових чи гідро-електростанцій. На їх роботу

впливають різні важливі фактори, як, наприклад, швидкість вітру для вітряків та довжина світового дня, хмарність, інші погодні умови та географічне положення для сонячних панелей. Ефективне управління енергоспоживанням потребує використання системи енергоменеджменту, що складається з комплексу програмних і апаратних засобів [9, 10]. Процеси прогнозування та оптимізації енерговитрат супроводжуються обробкою великих масивів даних, чим більше вибірка даних, тим точніша модель прогнозування, а отже правильніші прийняті управлінські рішення [11]. Для збереження і ефективної обробки моделей та інших системних даних потрібна ефективна система управління сховищем даних.

Об'єкт дослідження: процес обробки та збереження моделей при прийнятті рішень щодо управління життєвим циклом енергії.

Предмет дослідження: моделі та методи представлення та збереження даних у хмарних сховищах при управлінні гібридною енергомережею.

Гіпотеза: використання розробленого сервісу по зберіганню даних в S3-подібному об'єктному сховищі дозволить СППР керувати моделями, конфігураціями, залежностями та іншими метаданими для кожного клієнта із загальною доступністю в будь-який момент часу, під високим навантаженням, з низькою затримкою, високою стійкістю та гнучкістю.

Мета: розробити підсистему для зберігання моделей, конфігурацій та інших службових файлів системи підтримки прийняття рішень (СППР) управління гібридною енергомережею. Підсистема повинна бути окремим сервісом, який легко розгортається та масштабується з подальшою підтримкою хмарних технологій.

Практичне значення: Практичне значення роботи полягає у тому, що розроблена підсистема зберігання та обміну моделей побудована за мікросервісною архітектурою з використанням S3 протоколу для обміну даними з об'єктним сховищем, стане частиною системи підтримки прийняття рішень управління гібридними електромережами яку можна буде розгорнути в хмарному сервісі з мінімальними доопрацюваннями.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Ідентифікація проблеми

Обсяг генерованих, оброблюваних і збережених даних досяг безпрецедентного рівня. Навіть під час останньої економічної кризи не спостерігалося уповільнення або інформаційного спаду. Навпаки, потреба в обробці, переміщенні та зберіганні даних тільки зросла.

Протягом останнього десятиліття необхідність боротьби з надмірним обсягом даних і перехід апаратного забезпечення від підходів «scale-up» до «scale-out» призвели до появи нових систем зберігання великих даних, які відійшли від традиційних моделей реляційних баз даних [12]. Ці підходи зазвичай жертвують такими властивостями, як узгодженість даних, щоб підтримувати швидкі відповіді на запит із збільшенням обсягів даних [13]. Великі сховища даних використовуються подібно до традиційних систем керування реляційними базами даних, напр. для рішень онлайнової обробки транзакцій (OLTP) і сховищ даних над структурованими або напівструктурованими даними. Особливо сильною стороною є обробка неструктурованих і напівструктурованих даних у великому масштабі.

З появою та зростанням соціальних медіа та Інтернету речей (IoT) попит на розподіл комп'ютерних ресурсів зазнає великих змін. Враховуючи, що ера IoT наближається, це велика проблема, щоб задовольнити вимоги до обчислювальних ресурсів для користувачів у всьому світі. Персональним клієнтам і підприємствам потрібні більш безпечні, швидші, стабільні та зручні мережеві послуги [14]. Попередній метод розподілу ресурсів базується на локальній і статичній моделі. Можливості обчислень залежать лише від локальної фізичної машини, що обмежує масштабованість обчислювальних ресурсів і не може задовольнити вимоги щодо роботи з керуванням деякими ресурсами з чутливістю до часу. Таким чином, попередні рішення не можуть бути більш конкурентоспроможними та ефективними. Хмарні обчислення розглядалися як варіант для задоволення потреб у наданні послуг

Інтернету та збалансування вартості та продуктивності порівняно з традиційними локальними обчислення. З різними видами обчислювальних послуг і розвитком інформаційних технологій хмарні обчислення перетворюються на модель, яка становить послуги для надання користувачам, перетворюючись на потреби повсякденного життя [15]. Хмарні обчислення включають програми або послуги, що надаються через Інтернет, апаратні компоненти, системи зберігання та системне програмне забезпечення. Як віртуальний пул ресурсів із гнучкістю, хмарні обчислення обслуговують величезну кількість клієнтів і пропонують доступ до мережі для користувача, якому потрібен обчислювальний ресурс у зручному підході; ці обчислювальні ресурси можуть бути комбінацією мережі, сервера, програм або систем зберігання [16].

Із зростанням популярності хмарних обчислень зростає і їх вплив на великі дані. У той час як Amazon, Microsoft і Google створюють власні хмарні платформи, інші компанії, зокрема IBM, HP, Dell, Cisco, Rackspace тощо, будують свої пропозиції на основі OpenStack, платформи з відкритим кодом для побудови хмарних систем. За даними IDC [17], до 2020 року 40 % цифрового всесвіту «торкнуться хмарних обчисленнях», і «можливо, навіть 15 % будуть підтримуватися в хмарі». Хмара загалом, і зокрема хмарне сховище, можуть використовувати як підприємства, так і кінцеві користувачі. Для кінцевих користувачів зберігання їхніх даних у хмарі забезпечує надійний доступ звідусіль і з будь-якого пристрою. Крім того кінцеві користувачі можуть використовувати хмарне сховище як просте рішення для онлайн-резервного копіювання даних своїх робочих столів. Подібним чином для підприємств хмарне сховище забезпечує гнучкий доступ із багатьох місць і швидке й просте масштабування ємності, а також нижчі ціни на сховище та кращу підтримку на основі ефекту масштабу з особливо високою економічною ефективністю в середовищі, де корпоративне сховище потреби змінюються з часом вгору і вниз [18].

Блокове, об’єктне та хмарне сховище файлів — це три способи зберігання даних у хмарі, щоб користувачі та програми могли отримати до них віддалений доступ через мережеве з’єднання. Об’єктне сховище зберігає та керує всіма даними в

неструктурованому форматі та в одиницях, які називаються об’єктами. Блокове сховище приймає будь-які дані, наприклад файл або запис бази даних, і розділяє їх на блоки однакового розміру. Потім воно зберігає блок даних у фізичному сховищі, оптимізованому для швидкого доступу та пошуку. Хмарне сховище файлів — ще один метод зберігання даних, який надає серверам і програмам доступ до даних через спільні файлові системи. Кожен тип пропонує свої унікальні переваги для різних випадків використання.

Об’єктне сховище — це архітектура зберігання даних, де дані зберігаються в ізольованих контейнерах, які називаються об’єктами. Об’єктне сховище розділяє дані на окремі блоки, які містять унікальний ідентифікатор і метадані для опису даних і полегшують доступ до них і їх отримання, ніж інші типи сховищ.

Наприклад, порівнюючи сховище файлів і об’єктів, немає ієрархії папок або каталогів. Замість цього об’єкти зберігаються в плоскому середовищі даних або пулі зберігання. Об’єкти можна зберігати локально, але зазвичай вони зберігаються в хмарі, тому організації та команди можуть отримувати доступ до даних з будь-якого місця. Отримання доступу до об’єкта відбувається з використанням унікального ідентифікатора і метаданих.

Ця плоска модель пам’яті робить її ідеальною для роботи з великими обсягами неструктурованих даних, таких як вміст соціальних мереж, відео або дані датчиків, які часто важко зберігати в ієрархічному порядку.

Це також означає, що сховище об’єктів набагато легше масштабувати, ніж інші типи сховищ, оскільки дані організовані в одному глобальному пулі сховищ. Ви можете легко отримувати доступ до даних і керувати ними, навіть якщо вони зберігаються на кількох апаратних пристроях і в різних географічних розташуваннях.

Загальні випадки використання об’єктного сховища включають хмарні додатки, Інтернет речей (IoT), великі дані, зберігання та доставку мультимедійних даних, а також резервне копіювання та архівування.

1.1.1 Переваги зберігання об’єктів

Масштабованість. Зі сховищем об’єктів майже немає обмежень щодо масштабованості. Для того, щоб додати більше пам’яті за потреби, потрібно просто додати більше пристроїв.

Створено для великих даних. Об’єктне зберігання спрощує зберігання та керування великими обсягами неструктурованих даних, завдяки чому воно добре підходить для випадків використання великих даних, таких як штучний інтелект, машинне навчання та прогнозна аналітика [19].

Спрощене зберігання. У сховищі об’єктів немає папок або вказівників, що полегшує отримання даних, оскільки не потрібно мати точне місцезнаходження.

Доступне зберігання на вимогу. Зберігання об’єктів базується на споживанні. Користувач платить лише за потрібне сховище та може масштабувати його за потреби, зменшуючи витрати на зберігання всіх його даних.

Краща можливість пошуку. Сховищу об’єктів надає розширені можливості пошуку, які дозволяють шукати об’єкт на основі його метаданих, вмісту та інших атрибутів.

1.1.2 Недоліки зберігання об’єктів

Хоча зберігання об’єктів стає все більш популярним, особливо в хмарних сховищах, є деякі недоліки. Об’єктне сховище не є ідеальним для транзакційних даних, оскільки запис даних є дещо повільнішим процесом порівняно з файловим або блочним сховищем.

Неможливість зміни об’єкту після його створення. Для того, щоб оновити об’єкт в сховищі, доведеться створити його заново та завантажити.

Блокове зберігання — це архітектура зберігання даних, яка збирає дані та розбиває їх на блоки фіксованого розміру, які можна читати та записувати окремо. Кожному блоку присвоюється унікальний ідентифікатор, який потім зберігається на фізичному сервері. Система зберігання розміщує блоки там, де це ефективніше, тобто блоки можна розподіляти між різними системами та середовищами.

При запитуванні даних система зберігання блоків збирає відповідні блоки даних з будь-якого місця, де вони зберігаються, і повертає їх користувачу. Подібно

до сховища об’єктів, блочне сховище не покладається на єдиний шлях до даних, як сховище файлів.

Однак одна важлива відмінність, коли йдеться про блочне та об’єктне зберігання, полягає в тому, що метадані в блочному сховищі є більш обмеженими. Дозволяється включити лише основні атрибути файлу, тоді як у сховищі об’єктів можна налаштувати метадані, щоб включити більш детальну інформацію.

Завдяки можливостям детального контролю та оптимізації блочне сховище добре підходить для критично важливих робочих навантажень, які потребують низької затримки та частих змін. Загальні варіанти використання включають сховище для баз даних, контейнерів або транзакційних робочих навантажень, відтворення медіафайлів, аналітику масштабування, кешування та серверне сховище для віртуальних машин.

1.1.3 Переваги блокового зберігання.

Швидка продуктивність. Блокове сховище забезпечує кілька шляхів до даних і використовує обмежені метадані, зменшуючи передачу даних і забезпечуючи ефективне отримання даних.

Висока масштабованість. Можливість додавати нові блоки за потреби, оскільки потреба в ємності зростає, не впливаючи на продуктивність.

Легка модифікація. На відміну від сховища файлів або об’єктів, для даного типу сховища потрібно лише змінити певний блок або блоки, на які впливає зміна файлу в сховищі блоків.

1.1.4 Недоліки блокового зберігання.

Основним недоліком блокового зберігання є те, що воно дороге. Для блокового сховища потрібні мережі зберігання даних (SAN), що додає багато додаткових витрат на керування та обслуговування, а також доводиться платити за весь виділений простір для зберігання, навіть якщо він не використовується.

Крім того, обмежене використання метаданих може мати свої недоліки, особливо коли мова йде про обробку неструктурованих даних або операцій, які покладаються на метадані, наприклад пошуку або отримання даних.

Сховище файлів — це архітектура зберігання даних, яка використовує файли та папки для організації даних. Дані зберігаються у файлах, а потім упорядковуються в папках. Потім папки впорядковуються в підкаталогах у каталогах. Файлове сховище використовує імена файлів, тип даних у файлі (розширення) і конкретний шлях до розташування даних як унікальні ідентифікатори.

Воно має ту ж саму логіку як у фізичних системах файлів, які впорядковують документи в ієрархії. Зберігання файлів також є найвідомішою та широко використовуваною системою зберігання даних, яка використовується на персональному комп'ютері.

Зберігання файлів полегшує пошук і отримання окремих елементів даних і може використовуватися для зберігання майже будь-якого типу даних. Однак системі потрібно знати точний шлях до файлу, щоб знайти дані, включаючи підкаталог і ім'я файлу, а зберігання файлів може зайняти багато часу на керування та важко використовувати ефективно, оскільки обсяг даних зростає.

Завдяки своїй звичності сховище файлів залишається одним із найпопулярніших типів сховищ, які використовуються сьогодні. Загальні варіанти використання включають керування веб-вмістом, спільне зберігання файлів і документів для спільної роботи та невелике локальне сховище файлів.

1.1.5 Переваги файлового сховища

Просте, легке зберігання. Більшість людей, які мають базові навички роботи з комп'ютером, можуть легко використовувати системи зберігання файлів для пошуку інформації без необхідності додаткового навчання.

Зручне керування. За допомогою сховища файлів користувачі можуть легко створювати, керувати та видаляти власні файли без підтримки. Зберігання файлів також забезпечує легке керування доступом, щоб налаштувати, хто може отримувати доступ до файлів, переглядати їх і вносити в них зміни.

Знайомі протоколи. Системи зберігання файлів покладаються на стандартні обчислювальні протоколи, такі як мережева файлова система (NFS), загальна файлова система Інтернету (CIFS) або серверний блок повідомлень (SMB).

1.1.6 Недоліки файлового сховища

Як згадувалося вище, системи зберігання файлів працюють добре до того моменту, коли стає важко отримати доступ до даних і керувати ними. Чим більше файлів, папок і каталогів, тим важче стає знайти та отримати доступ до інформації. З часом можливості пошуку починають погіршуватися, і пошук потрібної інформації може стати досить повільним, що вплине на продуктивність співробітників.

Хоча сховище файлів може технічно обробляти неструктуровані дані, воно зазвичай не підходить для роботи з великими обсягами зберігання неструктурованих даних. Згодом це також стає дорогим, оскільки єдиний спосіб масштабування, коли досягнуто лімітів пам'яті, – придбати нові пристрої зберігання.

1.1.7 Ключові відмінності між об'єктним, блочним і файловим сховищами.

У таблиці 1.1 наведено порівняння основних типів хмарних сховищ за ключовими характеристиками. Об'єктне зберігання найкраще використовувати для великих обсягів неструктурованих даних. Це особливо вірно, коли довговічність, необмежений обсяг пам'яті, масштабованість і складне керування метаданими є важливими факторами для загальної продуктивності.

Блокове сховище забезпечує високу швидкість обробки даних, низьку затримку та високу продуктивність. Будь-яка служба, яка потребує швидкого доступу до даних, добре працює з блоковим сховищем. Наприклад, аналітика в режимі реального часу, високопродуктивні обчислення та системи з великою кількістю швидких транзакцій — усі вони виграють від блокового зберігання. Хмарне сховище файлів найкраще, коли користувачам потрібен одночасний доступ до спільної системи файлів. Крім того, контроль доступу на рівні файлів дозволяє налаштовувати дозволи та списки контролю доступу (ACL) для підвищення безпеки. Наприклад, робочі середовища для спільної роботи, які потребують спільного використання файлів між віддаленими командами, використовують сховище файлів.

Таблиця 1.1 – Порівняльна характеристика типів хмарних сховищ даних

	Об’єктне сховище	Блочне сховище	Файлове сховище
За типом зберігання	Об’єкти, що зберігаються в масштабованих сегментах	Блоки фіксованого розміру в жорсткому розташуванні	Файли організовані ієрархічно в папках і каталогах
За об’ємом даних	Підтримує великі обсяги даних	Підтримує великий обсяг даних	Краще для невеликих обсягів даних
За управління даними	Спеціальні метадані забезпечують легкий пошук	Більш обмежені можливості пошуку та аналітики	Ієрархічна структура добре працює для простіших менших наборів даних
За вартістю	Більш економічно ефективний	Більш дороге сховище, придбане у вигляді фіксованих блоків сховища	Дорожче, вимагає придбання нових пристроїв зберігання для масштабування
За моделлю оплати	«Сплачуй коли користуєшся»	Оплата за фіксований об’єм сховища (навіть, якщо не використовується)	Оплата за фіксований об’єм сховища (навіть, якщо не використовується)
За швидкодією	Нижча продуктивність, довший час обробки	Наднизька затримка та висока продуктивність	На продуктивність впливає більший обсяг даних
За масштабованістю	Висока масштабованість	Обмежена масштабованість	Обмежена масштабованість
Підходить для	Зберігання великих даних, статичні неструктуровані дані, аналітика	Транзакційні, структуровані дані, сховище для баз даних, диски для віртуальних машин і кешування	Спільне зберігання файлів, неструктуровані дані

Окрім характеристик різних типів хмарних сховищ, які були перелічені вище, можна скористатися рекомендацією від одного з провайдерів хмарних сервісів, Google Cloud Service (GCS), яка представлена у вигляді простої схеми (див. рис. 1.1). Основним критерієм при виборі типу сховища за вказаною схемою є протокол доступу до даних (у нашому випадку це RESTful).

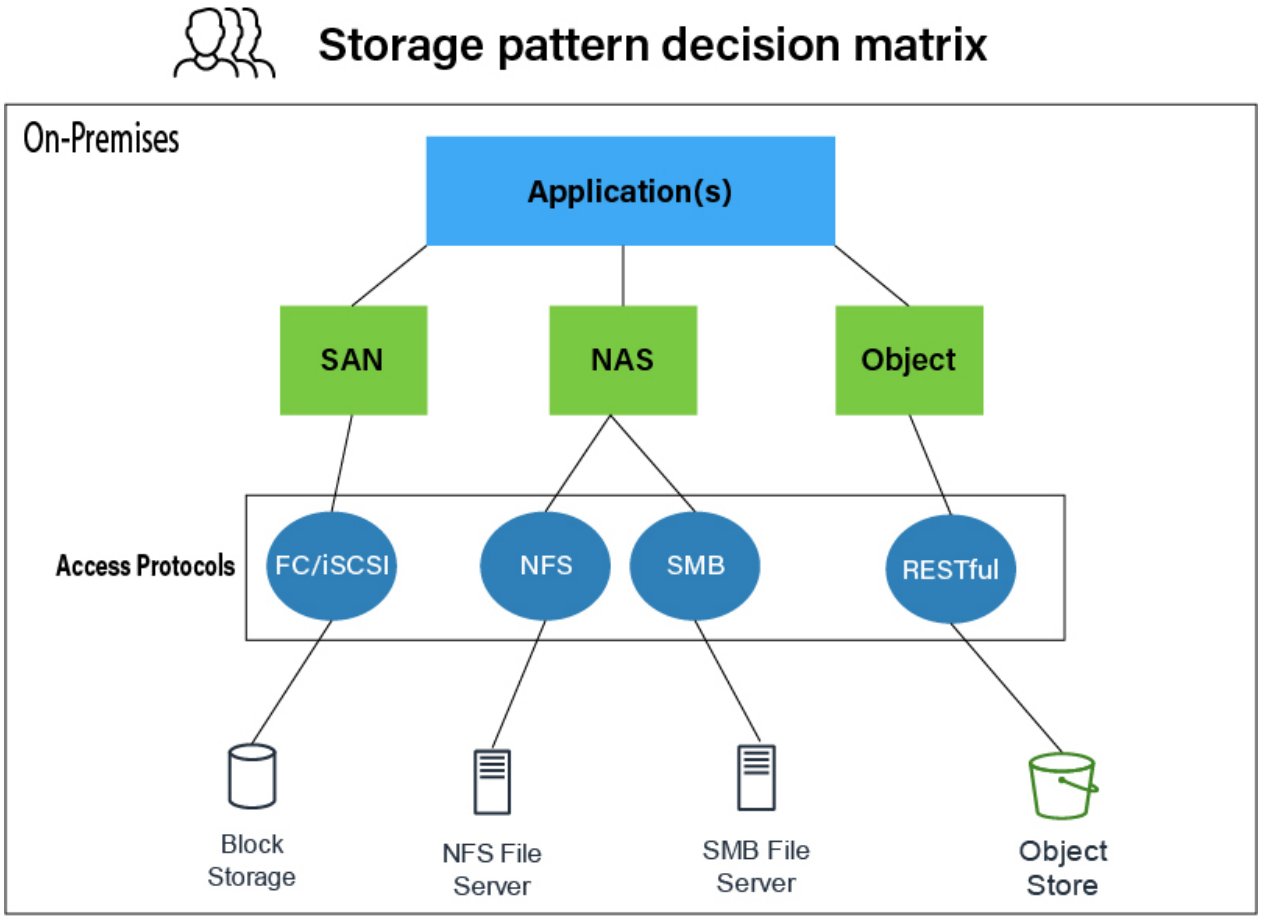

Рисунок 1.1 - Матриця прийняття рішень щодо шаблонів зберігання

Джерело: [21]

Таким чином, прийнявши до уваги, що об'єктне сховище окрім неструктурованих файлових даних, може також зберігати метадані (в нашому випадку конфігурацію моделей та список залежностей), таке сховище ще є більш економічно вигідне, при цьому маючи високу можливість масштабування, а також враховуючи рекомендації від GCS, найоптимальнішим типом хмарного сховища для вирішення поставлених задач є саме об'єктне сховище даних.

1.2 Аналіз методів проектування

З розвитком Інтернету веб-програми створили десятки мільярдів невеликих файлів; користувачі Інтернету завантажують величезну кількість фотографій, відео та музики, а люди щодня надсилають сотні мільярдів електронних листів. Згідно зі статистичними даними Інтернет-центру даних (IDC), кількість зросте в 44 рази протягом наступних десяти років, серед яких 80% складають неструктуровані дані, і більшість із них – неактивні дані [20]; однак, враховуючи такий величезний обсяг даних, блокова мережа зберігання даних (SAN) і файлове сховище (NAS) з можливостями розширення на рівні ПБ (пікобайт) безпорадні. Зазвичай ємність логічних блоків LUN у блочній мережі зберігання даних (SAN) становить лише кілька ТБ (терабайт). Кількість файлів, які підтримуються для оптимальної роботи однієї файлової системи, зазвичай обчислюється мільйонами. Людям потрібна система зберігання з новою архітектурою та високою масштабованістю, щоб задовольнити потреби користувачів у збільшенні ємності сховища від ТБ до ЕБ (ексабайт).

Як одне рішення, хмарним обчисленням вдалося зменшити побоювання про еластичність, розподіл ресурсів і вартість обчислювальних ресурсів, а процедура операцій системи спрощується протягом усього процесу; немає необхідності розуміти деталі технологій хмарної інфраструктури, що опосередковано знижує вартість навчання [22].

З різними типами клієнтів і вимогами сценарію мінімальна кількість великих постачальників хмарних обчислень надає різні рішення для публічного та приватного використання, що спрощує процес обслуговування та використання програмного забезпечення. Централізоване керування версіями також може підвищити безпеку обміну даними та підвищити ефективність співпраці [23].

Провідними постачальниками послуг хмарних обчислень є переважно компанії, які зосереджені на IT-індустрії та пропонують рішення та послуги хмарних обчислень, включаючи інфраструктури, центри обробки даних та багато

інструментів, пов'язаних із платформами хмарних обчислень. Такі як веб-сервіси Amazon (AWS), Microsoft Azure, Google Cloud Platform, RedHat і VMware.

Рисунок 1.2 – Провідні провайдери хмарних сервісів

Джерело: [24]

Кожний вендор хмари має свої характеристики зі своїми продуктами.

Amazon Web Services: Amazon Web Services (AWS) — це постачальник хмарних обчислень, який надає пов'язані з хмарою продукти та послуги, зокрема інфраструктури, сервери, обчислювальні процесори та центри обробки даних. Рішення та послуги AWS є надійними та гнучкими.

Microsoft Azure: використовуючи переваги всесвітнього поширення центрів обробки даних, Microsoft створила свою платформу хмарних служб на її основі та надає приватну та публічну хмару для свого клієнта для керування, тестування та публікації програм. Microsoft Azure базується віртуалізації робочих столів за допомогою гіпервізора.

Платформа Google Cloud: Google є одним із провідних виробників хмарних продуктів, перевагами Google є пошук, машинне навчання та аналіз даних; таким чином, хмарні сервіси Google орієнтовані на впровадження гілки даних і машинного навчання; крім того, Google надає послуги безпеки щодо зберігання даних із зобов'язаннями безпеки.

Хмарні послуги IBM: система хмарних обчислень IBM відрізняється від інших хмарних постачальників, основними хмарними службами від IBM є система як послуга та інфраструктура як послуга (IaaS); IBM надає своїм користувачам професійні інструменти для зберігання даних, мереж і обчислень на основі їх досвіду, з іншого боку, швидкий час відгуку та відносно низька вартість є їхніми перевагами в цій галузі.

З точки зору підходу до зберігання даних, обсяги даних, створених користувачем Інтернету, є значними, створюючи великі масиви даних і значне збільшення частки структурованих даних, дві тенденції, які DAS, SAN і NAS не можуть ефективно вирішити через їх власні технічні та архітектурні обмеження. Створення нової системи зберігання, яка адаптується до цієї нової тенденції щодо функцій даних, і розробка веб-сервісів Інтернету з високою ефективністю та безпекою є важливими. Вибір відповідного методу для зберігання даних став критичним розділом, коли користувачі намагаються побудувати програмну систему з високою ефективністю; під час цього прогресу, з різними атрибутами даних, такими як рівень важливості, частота використання та розмір об'єкта зберігання, клієнт повинен вибрати рішення щодо зберігання даних, яке залежить лише від конкретної ситуації, як зовсім інше рішення від традиційних, зберігання об'єктів вважається перспективною технологією зберігання в порівнянні зі зберіганням файлів і швидким зберіганням. Переваги зберігання об'єктів подібні до поточної тенденції програмно-визначеного зберігання на підйомі. Клієнти будуть використовувати можливості зберігання серверів, які в більшості випадків є комерційними; Постачальники апаратного забезпечення повинні додати більш глибоку модернізацію бізнесу дискового сервера замість того, щоб покладатися виключно на бізнес-рівень зберігання даних для продажу масивів або пристроїв зберігання файлів. Це залишає великий простір для фантазії нових постачальників програмно-визначених систем зберігання даних.

Крім того, щоб підвищити ефективність зберігання даних для прийняття поточної моделі додатків, особливо різних типів неструктурованих даних, порівняно з різноманітними сховищами системи з минулого, такі як блокове зберігання та

хмарне зберігання файлів, зберігання об’єктів стає все більш популярним і перспективним у різних сферах. Різні інструменти або системи керування даними користуються різними функціями; наприклад, очікується, що одна система зберігання має кращу продуктивність при обробці великої кількості даних, але не дуже добре працює при обробці невеликих об’єктів [25]. Таким чином, досягти оптимальної продуктивності всієї системи для усунення цих проблем є досить складним завданням, розробнику потрібно розглянути налаштування різноманітного програмного та апаратного забезпечення, їм потрібно провести експеримент і збалансувати продуктивність і вартість для отримання оптимального рішення, наскільки це можливо, таким чином створюючи оптимізовані системні рішення з високою продуктивністю щодо гнучкості, ефективності та масштабованості, що є ключовими вимогами для веб-додатків і галузей, які потребують обробки великих даних [26].

2 ПОСТАНОВКА ЗАДАЧІ

2.1 Мета та задачі дослідження

Метою роботи є розробка підсистеми зберігання моделей та конфігураційних даних, яка буде підтримувати систему прийняття рішень управління гібридною енергомережею. Як зазначається у праці [27], необхідність підтримки керування файловими даними та метаданими в середовищах великих даних була доведена випадками їх використання у системах прийняття рішень, коли ми маємо багато неструктурованих моделей прогнозування разом з їх конфігураціями та іншими параметрами.

Окрім того, проаналізувавши сучасні підходи до розробки та проектування інформаційних систем а також їх окремих частин було встановлено наступні такі функціональні вимоги до підсистеми зберігання та обміну моделей прогнозування:

- Підсистема повинна зберігати файли моделей разом з їх метаданими, такими як конфігурації та залежності.
- Підсистема повинна віддавати по запиту файли моделей, а також метадані.
- Підсистема повинна надавати швидкий доступ до найбільш використовуваних моделей (потрібно реалізувати кешування).
- Сховище даних повинно бути об’єктним, а також S3-сумісним (доступ до даних сховища здійснюється по S3-протоколу).
- Підсистема повинна підтримувати RESTful підхід доступу.
- Підсистема повинна мати загальну доступність у будь-який момент часу, під високим навантаженням, з низькою затримкою, високою стійкістю та гнучкістю.

У рамках даної роботи для показу і доведення ефективності концептуального підходу та методів дослідження і той же час спрощення роботи з розгортанням інфраструктури для сервісу було виокремлено деякі нефункціональні вимоги:

- Сервіс можна використовувати локально за допомогою механізму контейнеризації.

- S3 об’єктне сховище також повинно бути локальне і використовувати фізичні можливості локальної машини (диск, процесор, пам’ять). Також повинно запускатися у власному віртуальному контейнері.
- S3 сервіс провайдер запускати як окремий мікросервіс, а не як кластер сервісів.
- Повинен бути UI з адмінським доступом до S3 сховища для візуального перегляду його вмісту, а також адміністрування.

Реалізація поставлених задач дозволить системі підтримки прийняття рішень для управління гібридною електромережею ефективно управляти моделями, їх конфігураціями та іншими даними, підвищить швидкодію не блокуючи задачі процесами читання-запису файлів. Окрім того, використання REST API та S3 підходів до проектування сервісу дозволить його легке розширення а також доволі просту інтеграцію з хмарними сервісами.

2.2 Вибір інструментів та методів реалізації

На початку 2000-х років сервіс-орієнтована архітектура (SOA) з’явилася як парадигма розподілених обчислень, обробки електронного бізнесу та корпоративної інтеграції [28]. Швидко SOA та веб-сервіси стали предметом ажіотажу, і практично кожна організація намагалася їх прийняти, незважаючи на їх фактичну придатність. Що ще гірше, було майже стільки визначень SOA, скільки людей її прийняли. Це призвело до великої невдачі багатьох із цих спроб, оскільки вони намагалися змінити проблему, щоб вона відповідала розв’язку [29]. Сьогодні мікросервіси є новою зброєю для досягнення тих самих (і навіть більше) цілей, поставлених перед SOA багато років тому. Мікросервіси («SOA done right») описують особливий спосіб проектування програмних додатків як наборів незалежно розгорнутих сервісів, що забезпечують динамічність, модульність, розподілену розробку та інтеграцію гетерогенних систем. Однак нічого не дається безкоштовно: з’явилися нові (і старі)

виклики, включаючи дизайн і специфікацію сервісу, цілісність даних і управління узгодженістю.

Мікросервіси — це сучасний підхід до програмного забезпечення, за допомогою якого код програми доставляється невеликими керованими фрагментами, незалежними від інших. Їх невеликий масштаб і відносна ізоляція можуть призвести до багатьох додаткових переваг, таких як полегшення обслуговування, підвищення продуктивності, більша відмовостійкість, краще узгодження бізнесу тощо [30].

Фреймворк Spring - це дуже популярний і широко використовуваний фреймворк Java для створення веб- і корпоративних додатків. За своєю суттю Spring - це контейнер для ін'єкції залежностей, який забезпечує гнучкість конфігурації beans у різні способи, такі як XML, анотації та JavaConfig. З роками фреймворк Spring розвивався в геометричній прогресії, задовольняючи такі потреби сучасних бізнес-додатків, як безпека, підтримка сховищ даних NoSQL, обробка великих даних, пакетна обробка, інтеграція з іншими системами тощо. Spring, разом зі своїми підпроектами, став життєздатною платформою для створення корпоративних додатків [31].

Spring Boot побудований поверх звичайного Spring-фреймворку. Таким чином, він надає всі функції Spring і при цьому його простіше використовувати, ніж Spring.

З переваг використання Spring Boot можна віднести наступне:

- Скорочує час розробки та підвищує продуктивність додатку
- Дозволяє уникнути використання конфігурації XML, яка присутня у Spring
- Він включає в себе вбудований Tomcat-сервер
- Дуже просте розгортання, файли war і jar можна легко розгорнути на сервері Tomcat
- Забезпечує просте обслуговування та створення кінцевих точок REST
- Архітектура на основі мікросервісів

Docker — це комерційна платформа контейнеризації та середовище виконання, яке допомагає розробникам створювати, розгортати та запускати контейнери. Він

використовує архітектуру клієнт-сервер із простими командами та автоматизацією через єдиний API.

Контейнер — це стандартна одиниця програмного забезпечення, яка упаковує код і всі його залежності, щоб програма швидко й надійно запускалася з одного обчислювального середовища в інше. Образ контейнера Docker — це легкий автономний виконуваний пакет програмного забезпечення, який містить усе необхідне для запуску програми: код, час виконання, системні інструменти, системні бібліотеки та налаштування. Контейнер дозволяє запускати програми у власному ізольованому просторі пам’яті, зберігаючи доступ до загальних ресурсів, таких як файлові системи. Контейнери залишаються легкими, оскільки не вимагають інтенсивних процесорів систем, таких як підтримка графічного інтерфейсу користувача, драйверів пристроїв та інших складніших компонентів операційної системи. Контейнери створюються із зображень, які поєднуються з локальними ресурсами операційної системи хостингу [32].

Docker також надає набір інструментів, який зазвичай використовується для упаковки додатків у незмінні образи контейнерів шляхом написання файлу Docker і виконання відповідних команд для створення образу за допомогою сервера Docker. Розробники можуть створювати контейнери без Docker, але платформа Docker полегшує це. Потім ці образи контейнерів можна розгортати та запускати на будь-якій платформі, яка підтримує контейнери, наприклад Kubernetes, Docker Swarm, Mesos або HashiCorp Nomad.

S3 (Simple Storage Service) - це протокол обміну даними, який надає прості та надійні можливості зберігання та доступу до об'єктів. S3 використовує HTTP/S для взаємодії з клієнтами та підтримує такі операції, як створення, читання, запис, видалення та переміщення об'єктів [33].

S3 має ряд переваг, зокрема:

- Простота використання: S3 має простий інтерфейс, який легко освоїти.
- Надійність зберігання: S3 забезпечує надійне зберігання об'єктів, навіть у разі збоїв.
- Масштабованість: S3 може масштабуватися до мільйонів об'єктів.

- Безпека: S3 забезпечує безпеку об'єктів за допомогою таких засобів, як аутентифікація, авторизація та шифрування.

S3 широко застосовується в хмарних сервісах, зокрема Amazon Web Services.

MinIO — це високопродуктивне розподілене об’єктне сховище, розроблене для великомасштабних середовищ даних [34]. Це добре підходить для Amazon S3-сумісної заміни HDFS, особливо коли використовується для AI/машинного навчання, IoT та інших великих даних.

Об’єктне сховище MinIO включає сервер, додатковий клієнт і додаткові комплекти розробки програмного забезпечення (SDK):

Сервер MinIO — це розподілений сервер зберігання об’єктів, який включає в себе низку можливостей корпоративного рівня.

Клієнт MinIO («mc») — це сучасна альтернатива UNIX командам, які підтримують зберігання об’єктів у веб-масштабі розгортання.

MinIO Client SDK включає прості API для доступу до будь-якого сховища об’єктів, сумісних з Amazon S3.

MinIO — це рішення з відкритим вихідним кодом, яке пропонує кілька корпоративних можливостей для захисту даних, підтримки цілісності даних, посилення безпеки та максимальної гнучкості.

Таким чином рішення з використанням MinIO є «золотою серединою» як для стартапів так і корпоративних проектів, які в подальшому можуть легко інтегруватися у хмарні сервіси.

3 МОДЕЛЮВАННЯ ТА ПРОЕКТУВАННЯ

3.1 Структурно-функціональне моделювання процесу

З праць Перрі та Вольфу [35] відомо, що архітектура програмного забезпечення була центром досліджень програмної інженерії в 1990-х роках. Складність сучасних програмних систем зумовила необхідність більшого акценту на компонентних системах, де реалізація розділена на незалежні компоненти, які взаємодіють для виконання бажаного завдання. Дослідження архітектури програмного забезпечення досліджують методи визначення того, як найкраще розділити систему, як компоненти ідентифікують і взаємодіють один з одним, як передається інформація, як елементи системи можуть розвиватися незалежно, і як все вищезазначене можна описати за допомогою формальних і неформальних позначеннь. Хороша архітектура не створюється у вакуумі. Усі проектні рішення на архітектурному рівні повинні прийматися в контексті функціональних, поведінкових і соціальних вимог системи, що проектується, що є принципом, який однаково застосовується як до архітектури програмного забезпечення, так і до традиційної галузі архітектури будівель [36].

Базуючись на аналізі викладеному у попередніх розділах, найбільш вдалим архітектурним рішенням для вирішення поставлених задач є використання мікросервісів в поєднанні з S3-подібним об’єктним сховищем даних, яке можна розгорнути як локально (для простоти демонстрації функціоналу) за допомогою контейнеризації, так і в якості service-mesh. Використання технології мікросервісів дозволить скоротити час на розробку рішення, дозволить уніфікувати інтерфейс взаємодії з іншими системами та підсистемами. Також такий підхід дає змогу розробити платформо-незалежний додаток, що дозволить використовувати стек технологій відмінних від основної системи підтримки прийняття рішень, які будуть більше підходити для розв’язання поставлених задач.

Для моделювання процесів взаємодії системи зберігання та обміну даними з інших компонентів системи, було розроблено архітектурну модель, яку представлено

за допомогою нотації C4 (рис. 3.1), яка є не строгою, але більш візуально зрозумілішою. Це дозволяє зрозуміти зв'язки системи, проте для концептуального моделювання цього недостатньо, тому було також розроблено контекстну діаграму (рис. 3.2) та діаграму декомпозиції (рис. 3.3) процесу отримання метаданих, та файлових даних (рис. 3.4) в нотації IDEF0.

IDEF0 — це широко використовувана техніка для структурованого аналізу та проектування систем. Широко задокументовано його використання для підвищення продуктивності та комунікацій у комп'ютерно-інтегрованих виробничих системах, а останнім часом і як інструменту для реінжинірингу бізнес-процесів.

IDEF0, що означає Icam DEFinition for Function Modeling, — це методологія, яка використовується для функціонального моделювання та графічної нотації для опису бізнес-процесів, інформаційних систем та аналізу програмної інженерії. Він був розроблений у 1980-х роках ВПС США в рамках програми Integrated Computer-Aided Manufacturing (ICAM) [37].

IDEF0 використовує ієрархічну серію діаграм, щоб розбити систему на її складові функції та підфункції. Кожна діаграма, яка називається блоком IDEF0, представляє функцію як чорний ящик із входами, виходами, механізмами та елементами керування. Механізми — це ресурси, що використовуються для виконання функції, а елементи керування — це фактори, які впливають на виконання функції.

Рисунок 3.1 – Архітектура системи представлена в нотації C4

Рисунок 3.2 – Контекстна діаграма в нотації IDEF0

З контекстної діаграми, зображеної на рисунку 3.2, можна побачити, що основними входами для процесу отримання даних моделей системи підтримки прийняття рішень є запит на отримання метаданих та запит на отримання бінарних даних моделей.

Основними обмеженнями для даного процесу є REST API (обмеження стандарту REST), JSON формат, S3 протокол (протокол обміну даними з об'єктним сховищем), обмеження розміру метаданих (пов'язане з розміром хедера запиту), обмеження обсягу бінарних даних (встановлюється адміністратором підсистеми).

При цьому серед основних ресурсів та інструментів, які задіяні для виконання даного процесу є:

- JVM (Java Virtual Machine) – слугує для виконання Java додатку
- Веб-сервер – потрібен для обробки запитів та відправки результатів модулю активації моделей
- Docker контейнер - слугує для виконання Java додатку та Веб-серверу
- Redis – потрібен для кешування бінарних даних моделей

- MinIO S3 сервер – сервер, який обробляє запити на збереження та отримання S3 об’єктів (моделей)
- Підсистема зберігання та обміну – бізнес-логіка, яка контролює процес отримання моделей системи підтримки прийняття рішень

На виході нашого процесу отримує метадані та бінарні дані запитуваних моделей.

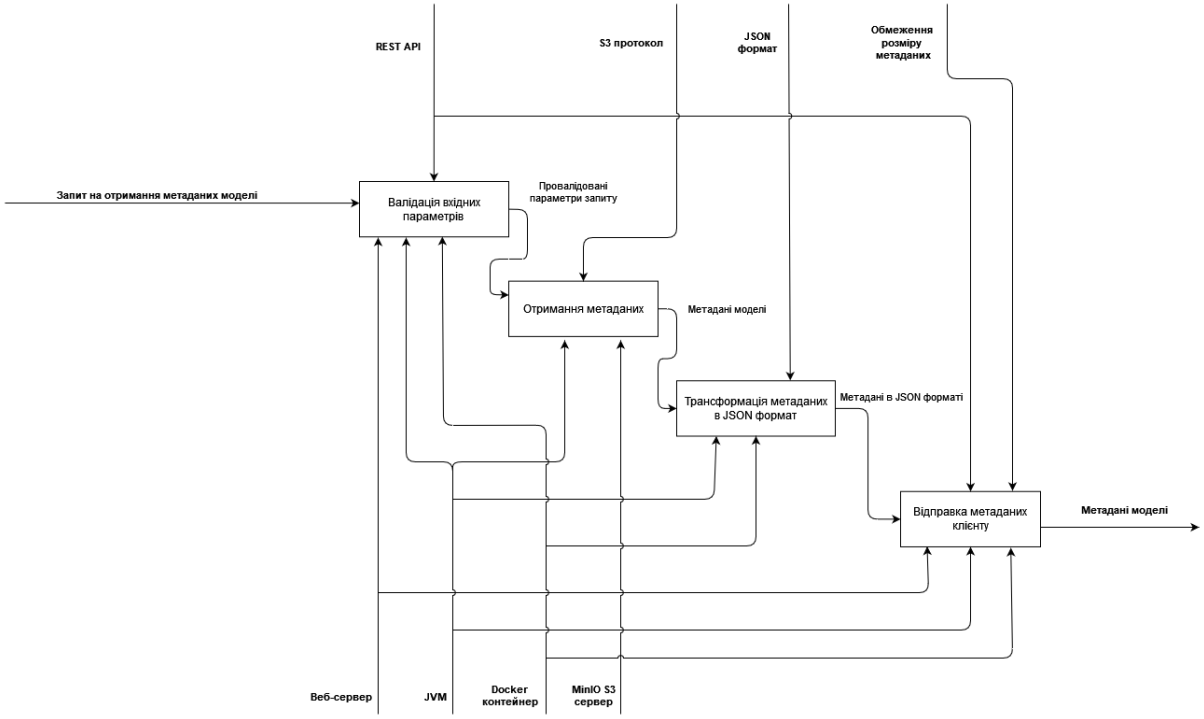

Рисунок 3.3 – Діаграма декомпозиції в нотації IDEF0 – Отримання метаданих

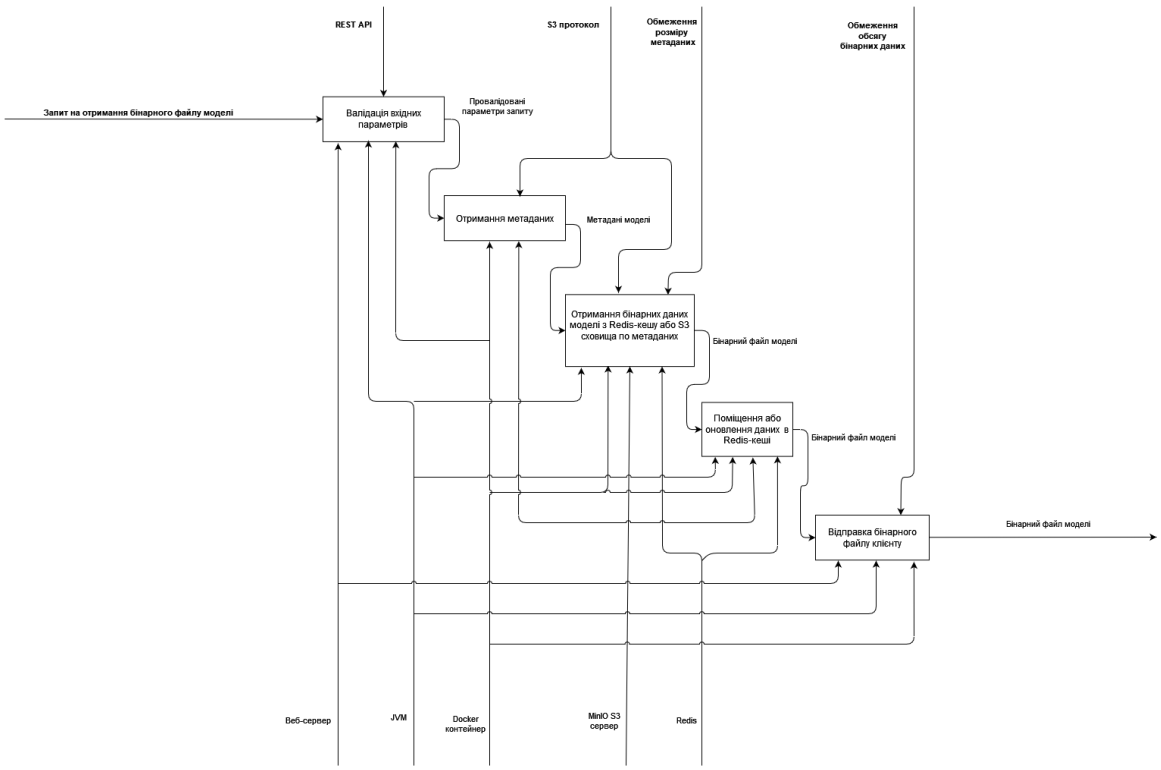

Рисунок 3.4 – Діаграма декомпозиції в нотації IDEF0 – Отримання файлових даних

Архітектура складається з наступних компонентів:

- Система підтримки прийняття рішень – представляє собою окремий сервіс, який запускає з певною періодичністю аналіз вхідних даних зібраних з електромереж різних типів та різних конфігурацій. Цей сервіс потребує доступу до згенерованих моделей отриманих при попередньому навчанні, а також збереженні таких моделей та інших даних.
- Підсистема зберігання та обміну даними – сервіс, який надає доступ до моделей з використанням REST API запитів. Інтерфейс доступу буде описано детально в розділі 3.2. Для реалізації цієї підсистеми використано фреймворк Spring Boot, а середовищем виконання обрано Docker контерйнер.
- Cache – для покращення швидкодії та доступу для частого використовуваних моделей використовується механізм in-memory кешування за допомогою інструментів Redis. Кожна модель після отримання з S3 сховища поміщається у сховище Redis, де зберігається протягом 30 хвилин, та у

випадку подальшого запиту на отримання даної моделі, буде зроблена перевірка співпадіння хеш суми файлу моделі в кеші та хеш суми об'єкту в S3 сховища, яка відповідає даному файлу. Якщо їх хеш суми співпадуть буде надано дані з кешу Redis і не буде робитися запит до MinIO. При повторному запиті час життя моделі в сховищі Redis автоматично подовжується на 30 хвилин.

- MinIO – сервіс-провайдер доступу до об'єктного сховища на базі протоколу S3. Даний сервіс також розгортається з використанням середовища Docker. MinIO дозволяє розгортання декількох мікросервісів об'єднаних у кластер, але в даній роботі вибрано більш простіший варіант розгортання даного продукту як окремого сервісу в одному Docker-контейнері.
    - Models Repository – S3-подібне об'єктне сховище даних. В роботі було використано Docker volumes в якості фізичного дискового простору. Але є можливість підключення зовнішніх дисків та пристроїв і використовувати їх як одне цілісне сховище.

3.2 Моделювання варіантів використання

Найважливішим етапом проектування інформаційної системи є розробка структури прикладного програмного забезпечення. Проектування інформаційної системи можна описати як багатоетапний процес створення та модернізації з використанням методичних та різноманітних інструментів [38].

Виділення етапу проектування інформаційної системи як окремого етапу можна розмістити між аналізом та розробкою. На практиці, однак, планування, яке формально починається з визначення цілей проекту, часто продовжується через фази тестування та впровадження, що робить чіткий розподіл на фази складним бо принципі неможливим [39].

UML – це стандартизована візуальна мова, яка використовується для розробки, документування та комунікації систем програмного забезпечення. По суті, це набір

діаграм із певними символами та правилами, які зображують різні аспекти системи, включаючи її структуру, поведінку та взаємодії. Цей інструментарій допомагає ідентифікувати, візуалізувати, створювати та документувати аспекти програмних систем для розробників інформаційних систем та програмного забезпечення. UML ефективний при моделюванні складних систем і в першу чергу є графічним інструментом при розробці об'єктно-орієнтованого програмного забезпечення та процесів розробки програмного забезпечення. Загалом, UML є потужним інструментом, який може значно покращити процес розробки, надаючи візуальний і стандартизований спосіб представлення програмного забезпечення та обміну даними [40].

В уніфікованій мові моделювання (UML) актор представляє зовнішню сутність, яка взаємодіє з системою, що моделюється.

Для більше детального опису процесів отримання та завантаження моделей в S3 сховище було розроблено діаграму використання (рис. 3.4), яка показує основних акторів та варіантів дій, які може виконати підсистема. На діаграмі послідовності (рис. 3.5), послідовність запитів які виконує система при запиті метаданих та файлових даних модулем активації прогнозних моделей.

Загалом для підсистеми зберігання та обміну даними можна виділити наступних акторів:

- Системний адміністратор – може завантажувати прогнозовані моделі в S3 сховище, керувати метаданими, налаштуванням доступу, створенням нових бакетів і т.д.
- Модуль активації – модуль, який робить запити на отримання метаданих та файлових даних моделі прогнозування. Також цей модуль має можливість завантажувати чи оновлювати моделі.
- MinIO S3 сервер – провайдер S3 об’єктного сховища.
- Redis – сховище, яке дає швидкий доступ до найбільш часто запитуваних моделей. Організовує кешування об’єктів.

Процес отримання даних буде виглядати наступним чином:

1. Зовнішня система (або користувач) надсилає запит на отримання моделі з параметрами *user_id*, *data_source_id*, *model_type, model_name.*
2. Система робить запит до Redis і перевіряє чи запитувана модель знаходиться в кеші.
    a. Якщо так, то відбувається запит до MinIO на отримання метаданих.
    b. Якщо метаданих за запитом не існує, то дані видаляються з кешу і користувачу повертається HTTP 404 Not Found.
    c. Якщо метадані існують, то вони повертаються користувачу (разом з тимчасовим посиланням на файл моделі).
    d. Якщо кеш не містить даних про модель, то система робить запит MinIO і повертає їх користувачу (разом з тимчасовим посиланням на файл моделі), а в разі їх відсутності повертає HTTP 404 Not Found.
3. Зовнішня система (модуль активації) робить запит на отримання файлу моделі за тимчасовим посиланням.

Процес завантаження даних буде виглядати наступним чином:

1. Зовнішня система (або користувач) надсилає запит на завантаження моделі з параметрами *user_id*, *data_source_id*, *model_type*, *model_name.*
2. Система надсилає запит до MinIO на створення метаданих і якщо вони були успішно поміщені в S3 сховище, то повертає їх користувачу разом з тимчасовим посилання на завантаження файлу моделі.
3. Зовнішня система (користувач) робить запит на завантаження файлу моделі за тимчасовим посиланням.
4. Система завантажує файл в MinIO і зв’язує її з створеними метаданими.
5. Система поміщає файл і дані моделі в кеш.

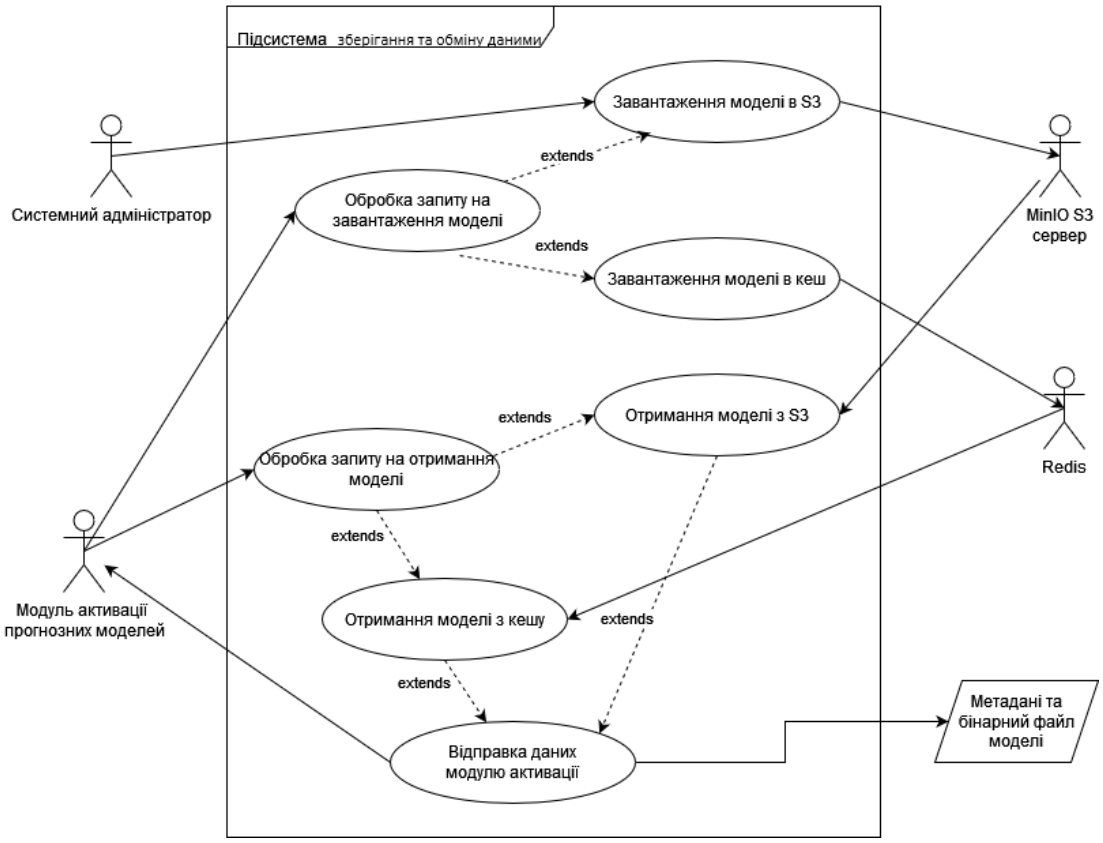

Рисунок 3.4 – Діаграма варіантів використання підсистеми

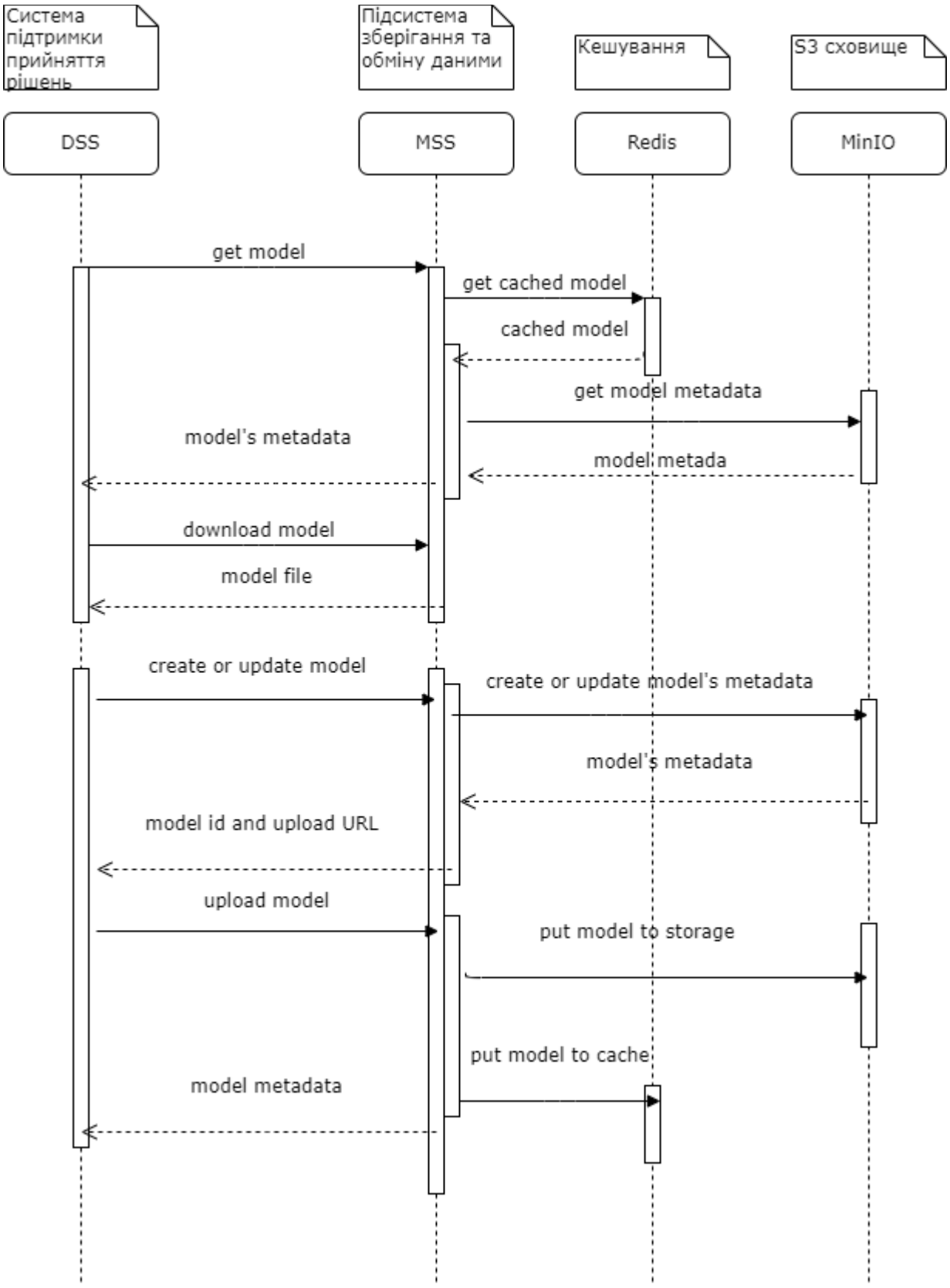

Рисунок 3.5 – Діаграма послідовності

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ПІДСИСТЕМИ ЗБЕРІГАННЯ ТА ОБМІНУ МОДЕЛЕЙ

4.1Налаштування середовища розробки та практична реалізація

Розробка мікросервісу проводилася в середовищі IntelliJ Idea. IntelliJ IDEA — це популярне інтегроване середовище розробки (IDE), призначене для максимального підвищення продуктивності розробників, особливо для тих, хто працює з Java і Kotlin. Воно розроблене компанією JetBrains і доступне як у безкоштовному виданні Community Edition, так і в платному Ultimate Edition з додатковими функціями [41].

IntelliJ дозволяє використовувати як встановлене на машині середовище виконання Java Development Kit (JDK), а також інструмент автоматизації побудови проекту, так і дозволяє установити ці інструменти під час створенню та налаштуванню проекту. Для створення нового проект з використанням Spring Boot, потрібно запустити IntelliJ, потім вибрати пункт меню «Fle -> Project…», в новому вікні потрібно вибрати пункт «Spring Initializr», де вказати ім’я проекту, вибрати версію JDK та інші параметри і натиснути «Next» (див. рис. 4.1).

Наступним етапом створення проекту є вибір версії Spring Boot, а також підключення потрібних модулів, які будуть потрібні для роботи. Мікросервіс, який було розроблено для підсистеми зберігання та обміну моделями, було розроблено з використанням Spring Boot версії 2.7, але IntelliJ Idea дозволяє вибрати лише версію Spring Boot не нижче 3.2. Це можна виправити вказавши потрібну версію Spring у файлах конфігурації (наведено в додатку Б.2). Альтернативним способом створення проекту є використання Spring Initializr на сайті Spring IO.

Як зазначалося у попередніх розділах, середовищем для розгортання підсистеми (мікросервісів) було обрано Docker. Для користувачів Windows доступний продукт Docker Desktop, який має UI для керуванням Docker-

контейнерами. Цей продукт є умовно безкоштовним, тому перед його використанням потрібно ознайомитися з умовами [42].

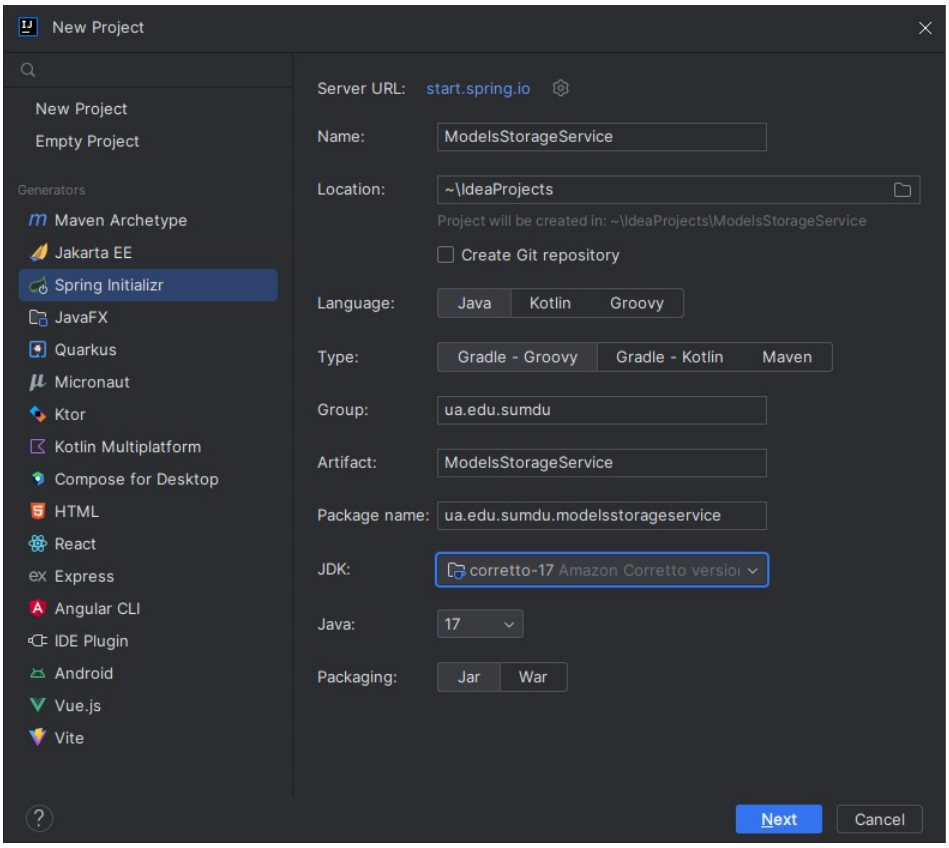

Рисунок 4.1 – Створення нового проекту

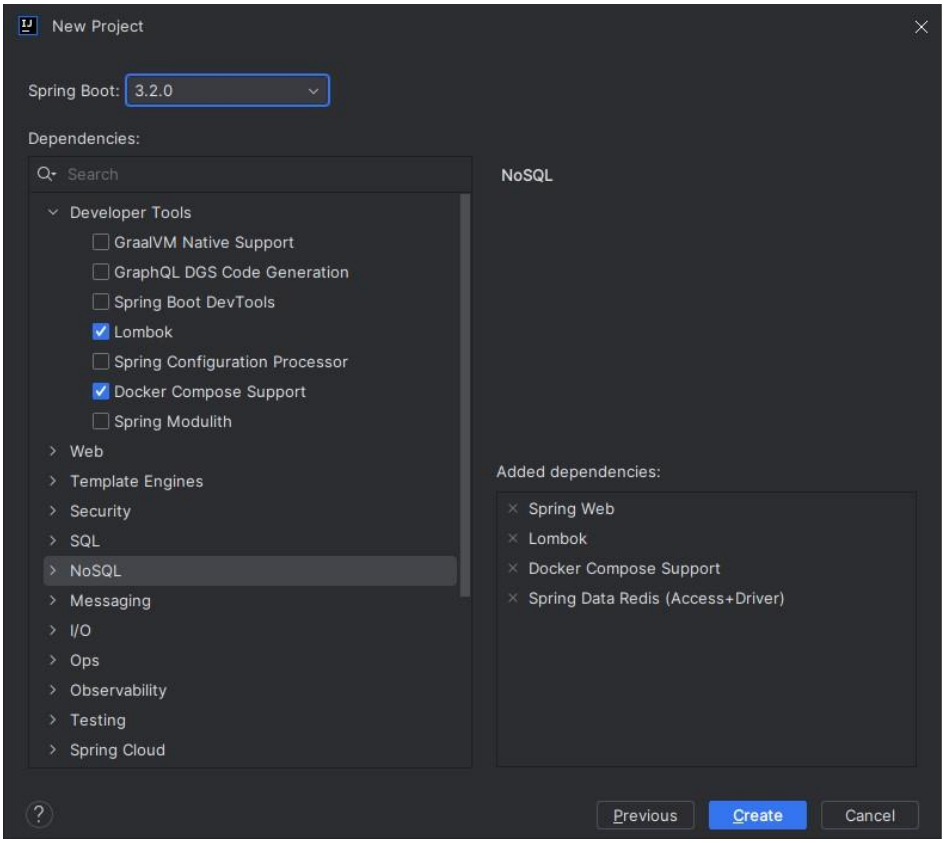

Рисунок 4.2 – Вибір версії Spring Boot та підключення модулів

Стартове вікно Docker Desktop має вигляд рис. 4.4.

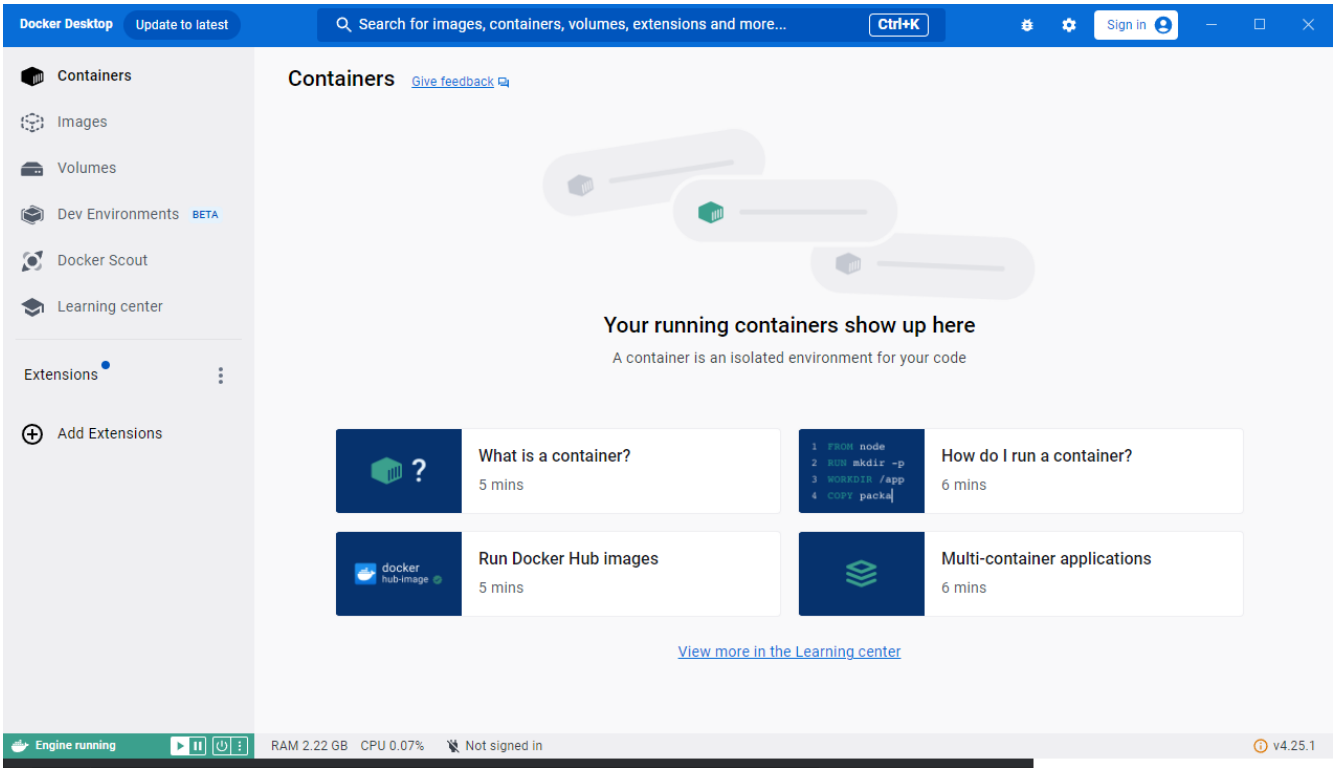

Рисунок 4.4 – Початковий екран Docker Desktop

В якості провайдера S3 об’єктного сховища в даній роботі використовується сервіс MinIO який доступний у вигляді Docker Image в публічному репозиторії. Інструменти Redis, який використовується для кешування файлових даних моделей, а також nginx, який використовується як проксі-сервер для Minio, також є в публічному доступі і готові для розгортання у середовищі Docker.

Для запуску перелічених сервісів скористаємося docker compose скриптом наведеним в додатку В. Після запуску docker compose скрипту в mingw-терміналі (див. рис. 4.5) перейдемо в середовище Docker Desktop і переконаємося, що всі потрібні нам контейнери запущені (рис. 4.6).

Перейдемо на сторінку входу для адміністрування MinIO об’єктного сховища (рис. 4.7) щоб створити потрібний нам Bucket для зберігання наших моделей прогнозування.

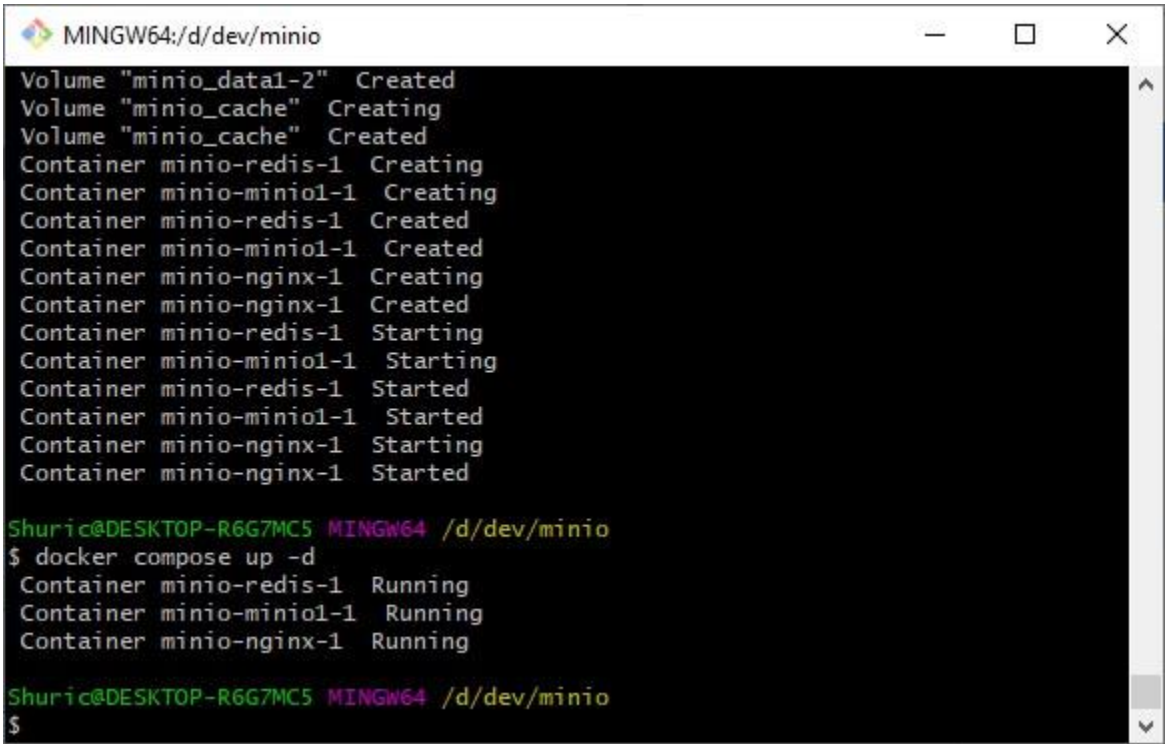

Рисунок 4.5 – Запуск MinIO та інших контейнерів за допомогою docker compose

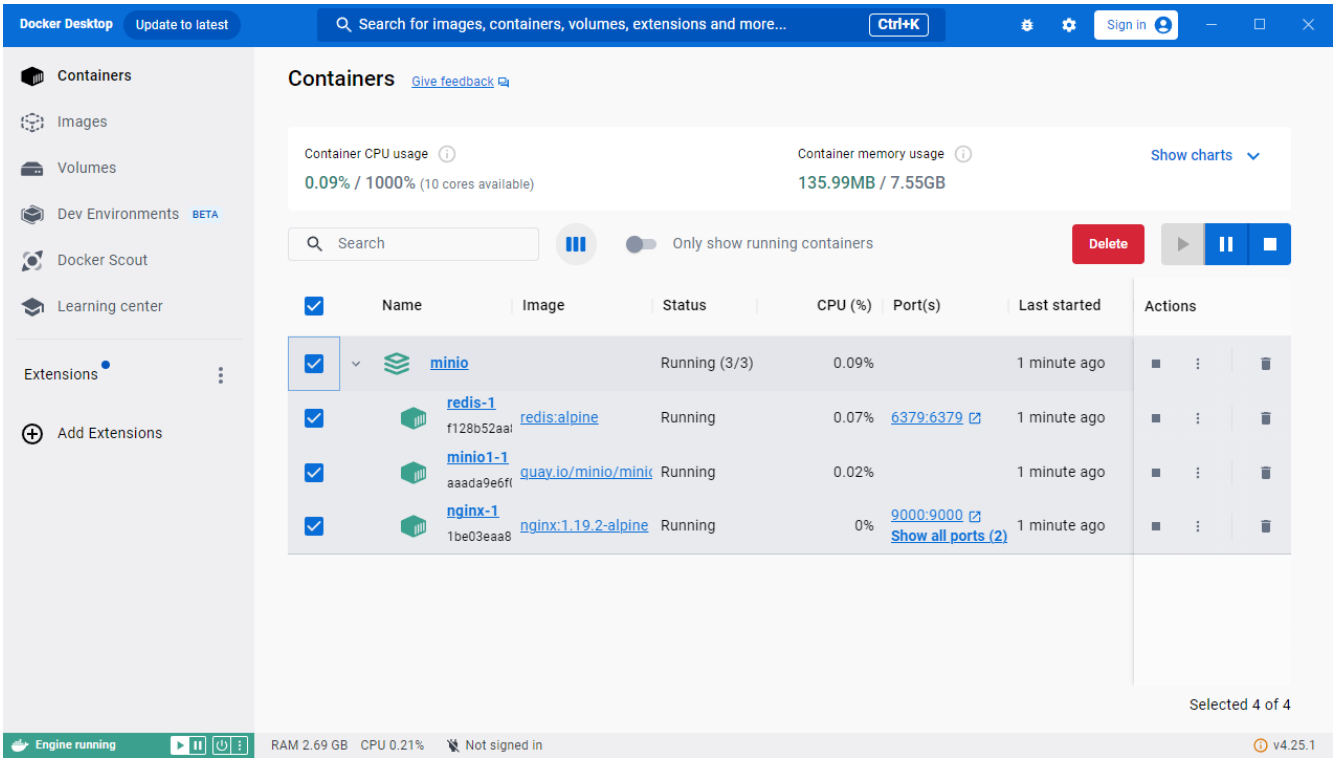

Рисунок 4.6 – Docker контейнери після запуску

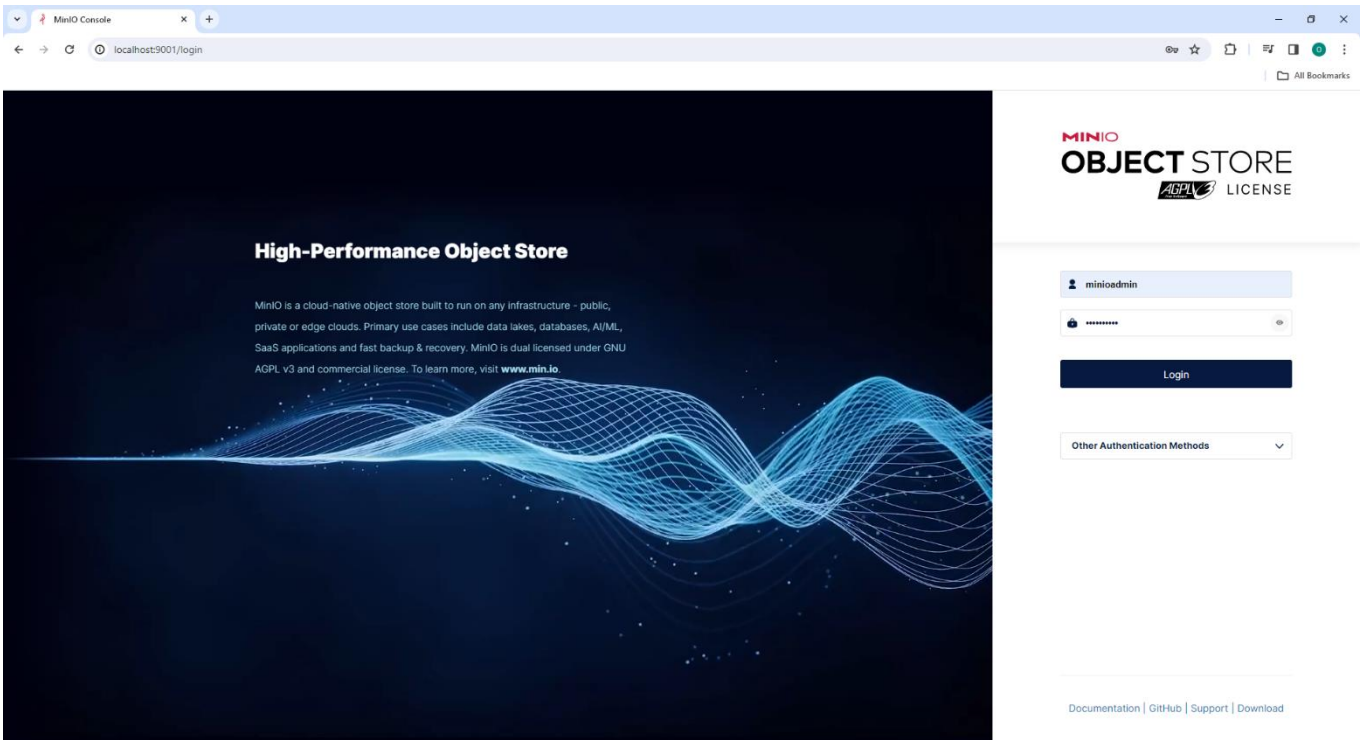

Рисунок 4.7 – Вхід до сторінки адміністрування MinIO

Процес створення нового Bucket-а виглядає як наведено на рисунку 4.8. В полі «Bucket Name» вводиму назву нашого Bucket-а «models-repository», залишаємо всі інші налаштування за замовчуванням і натискаємо кнопку «Create Bucket». Після додавання Bucket-а, його можна переглянути у вікні адміністрування MinIO (див. рис. 4.9). Тепер наше S3 об'єктне сховище готове для роботи.

Реалізація самого мікросервісу, який надає API для обробки запитів до підсистеми зберігання та обміну моделями наведена у вигляді лістингу основних Java-класів (додаток Б.1) та файлів конфігурації (додаток Б.2).

Нижче приведено опис основних Java-класів та їх призначення:

ModelsStorageServiceApplication – головний клас, що є початковою точкою запуску додатку.

ObjectsStorageController – клас, який реалізовує REST-контролер, що дозволяє обробляти запити на створення та отримання метаданих моделей.

FilesController – клас, який реалізовує REST-контролер, що дозволяє обробляти запити на завантажування та скачування бінарних файлів моделей.

ModelsStorageService – інтерфейс, який описує описує основні функції бізнес-логіки.

MinIoModelsStorageServiceImpl – клас, який реалізовує основну бізнес-логіку сервісу. У цьому класі здійснюється підключення до сервісу MinIO, трансформація метаданих у відповідності до стандарту S3, збереження та зчитування метаданих та файлових даних з використанням MinIO клієнта, кешування.

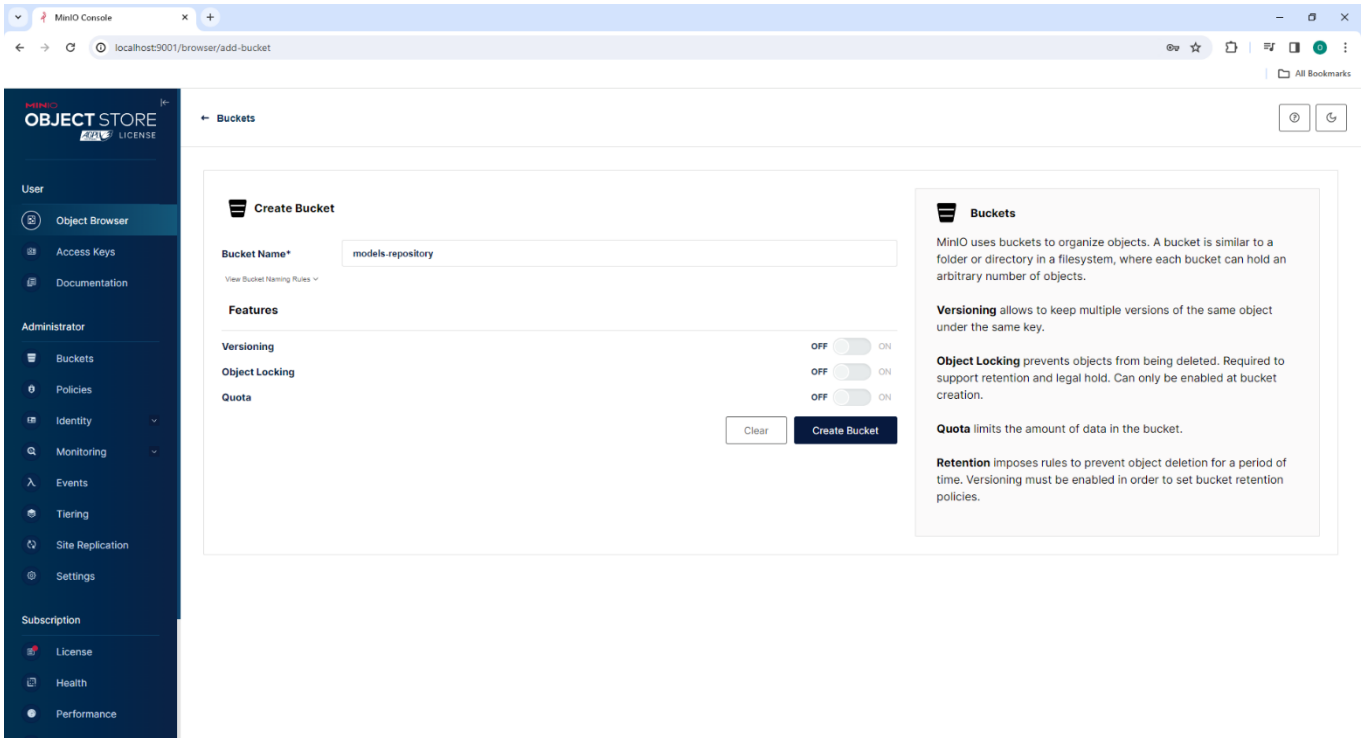

Рисунок 4.8 – Створення нового Bucket-а

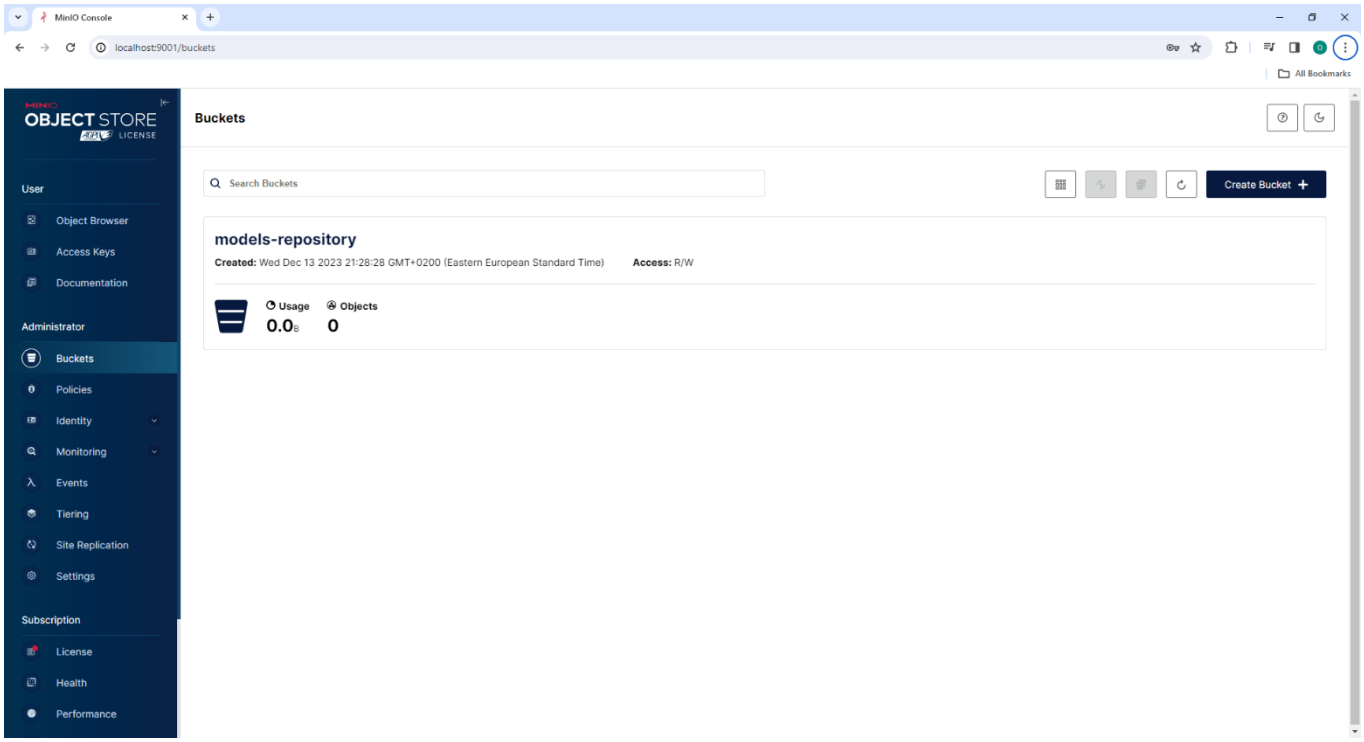

Рисунок 4.9 – Створений Bucket в об’єктному сховищі

4.2 Приклад використання підсистеми зберігання та обміну моделями

Після налаштування робочого середовища та практичної реалізації мікросервісу, який відповідає за обробку запитів до підсистеми зберігання та обміну моделей, перейдемо до запуску підсистеми та її тестування.

Для того, щоб запустити реалізований мікросервіс із середовища розробки IntelliJ Idea потрібно вибрати головний клас додатку ModelsStorageServiceApplication та виконати команду «Run». Процес запуску мікросервісу із середовища IntelliJ Idea показаний на рисунку 4.10.

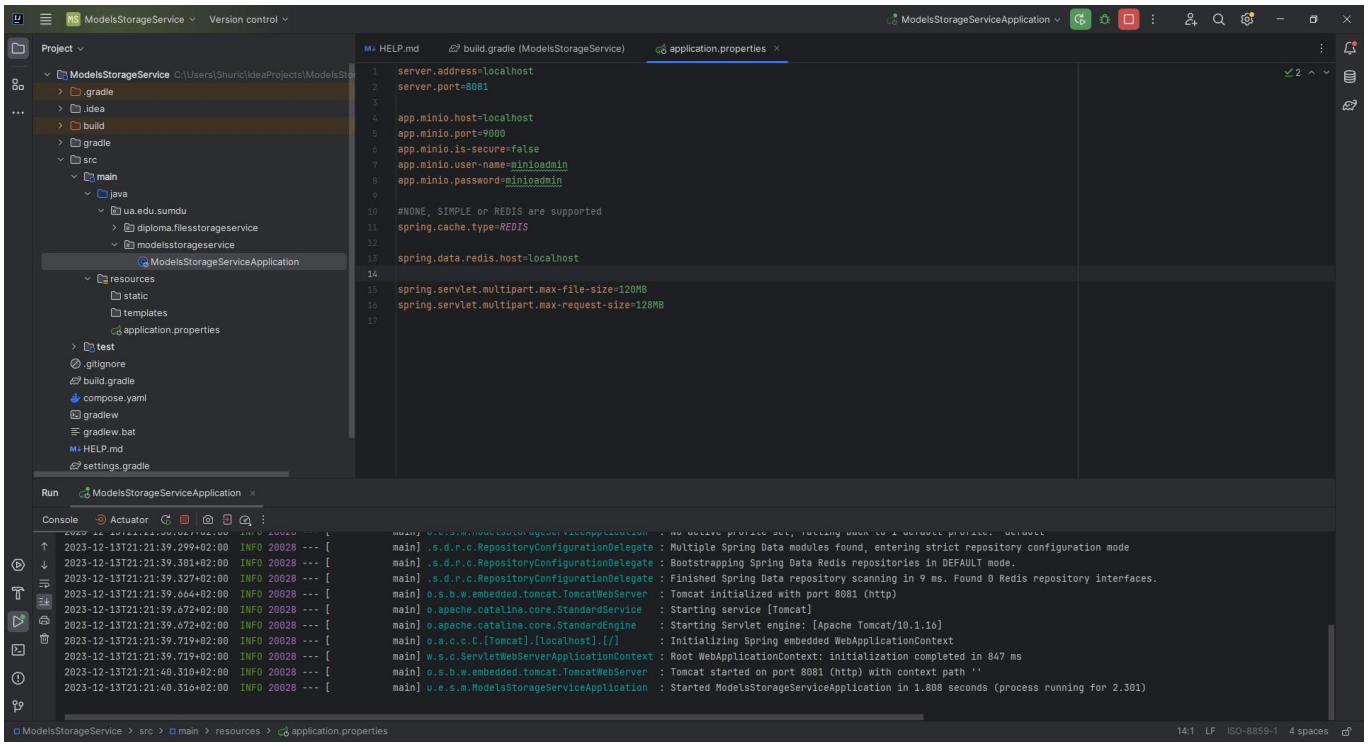

Рисунок 4.10 – Запуск мікросервісу із середовища IntelliJ Idea

Після запуску мікросервісу буде доступна сторінка перегляду документації сгенерованої інструментом Swagger (див. рис. 4.11), де показано список контролерів та можливі запити до підсистеми а також моделі підсистеми. Окрім того, самі тестові запити можна робити прямо з цієї сторінки.

Проте, в даній роботі для тестування нашої підсистеми використовувався інструмент Postman.

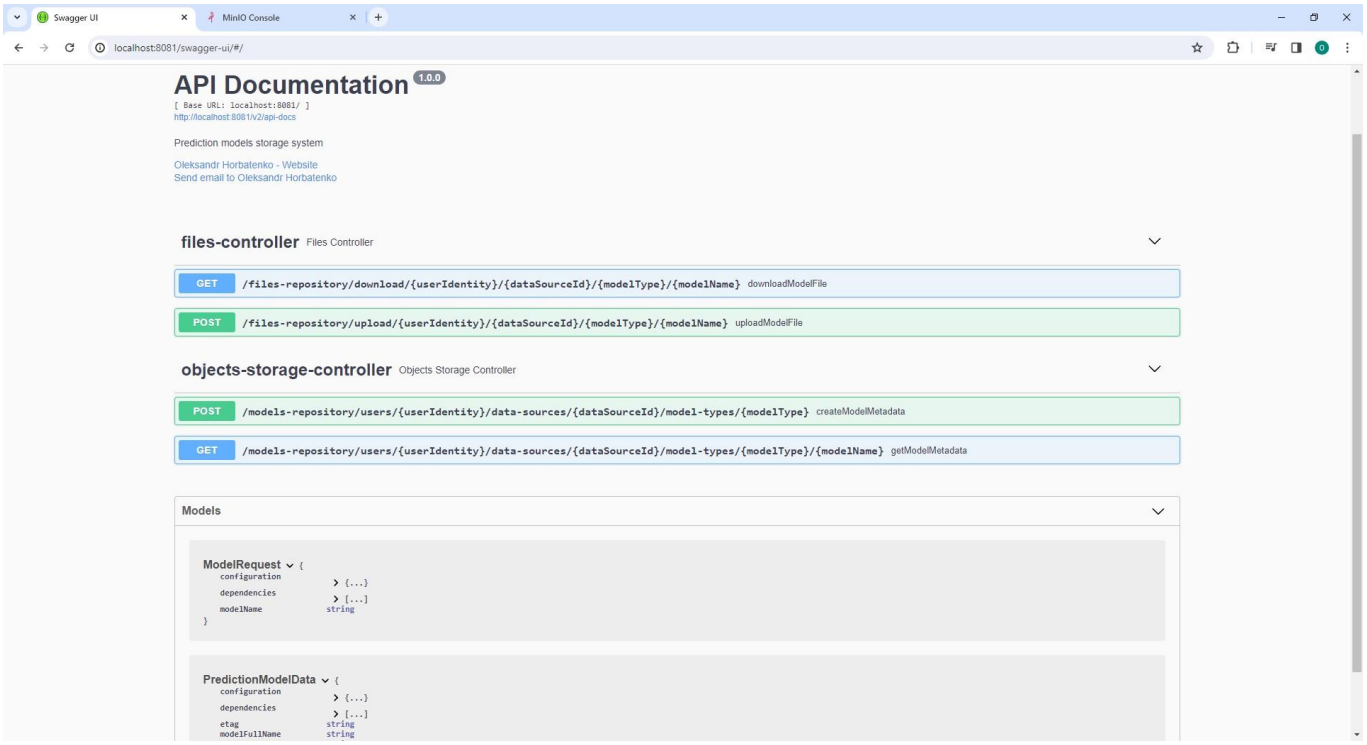

Рисунок 4.11 – API документація згенерована за допомогою інструменту Swagger

Щоб додати модель в об’єктне сховище S3, потрібно спочатку відправити запит на створення моделі (див. рис. 4.12). Якщо метаданих такої моделі не існує, то вони створяться в S3, якщо вони існують, то вони оновляться. Після успішного створення чи оновлення метаданих моделі повертається S3 ідентифікатор моделі (її повне ім’я).

Тепер можна відправити запит на завантаження бінарного файлу моделі (рис. 4.13). Після успішного завантаження файлу надсилається eTag ідентифікатор сховища S3.

Після успішно виконаних вищезазначених запитів модель як об’єкт буде доступна в S3 сховища, її можна переглянути в MinIO адміністративній консолі (див. рис. 4.14). Також тут же можна переглянути метадані моделі (див. рис. 4.15).

Для отримання метаданих завантаженої моделі виконаємо запит (рис. 4.16), а для отримання бінарного файлу моделі скористаємося запитом для скачування файлу моделі (рис. 4.17).

Після проведення тестування на декількох моделях було отримано результати часу завантаження бінарних файлів моделей в режимі з кешуванням та без кешування, які наведено в таблиці 4.1.

Таблиця 4.1 – Результати тестування підсистеми зберігання та обміну моделей

	№ запиту	Модель prediction_model_LSTM.joblib, 162.9 КБ	Модель prediction_model_Random_Forest.joblib, 32 МБ
З увімкненим кешуванням	1	126	688
	2 (відразу за 1-им)	9	440
	3 (після 1 хв. за 1-им)	10	455
Без увімкненого кешування	1	12	698
	2 (відразу за 1-им)	13	462
	3 (після 1 хв. за 1-им)	10	455

Як бачимо, перші запити на отримання моделей з включеним кешуванням є дещо повільнішим за запити з виключеним кешуванням. Це пов’язано з тим, що системі потрібен час на відправку даних в Redis. Подальші запити є дащо швидшими, але не дають велику виграшу по швидкодії. Це пов’язано в першу чергу з тим, що при локальному розгортанні підсистеми нехтуються втрати на транспортування даних мережею.

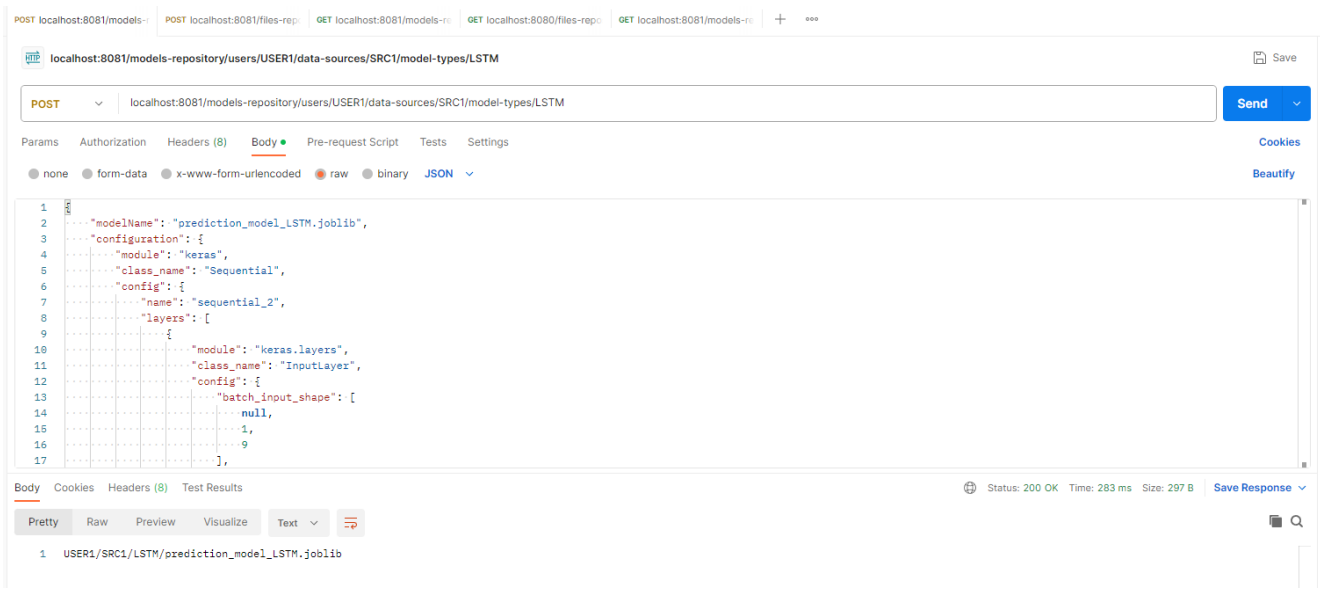

Рисунок 4.12 – Відправка запиту на збереження метаданих моделі

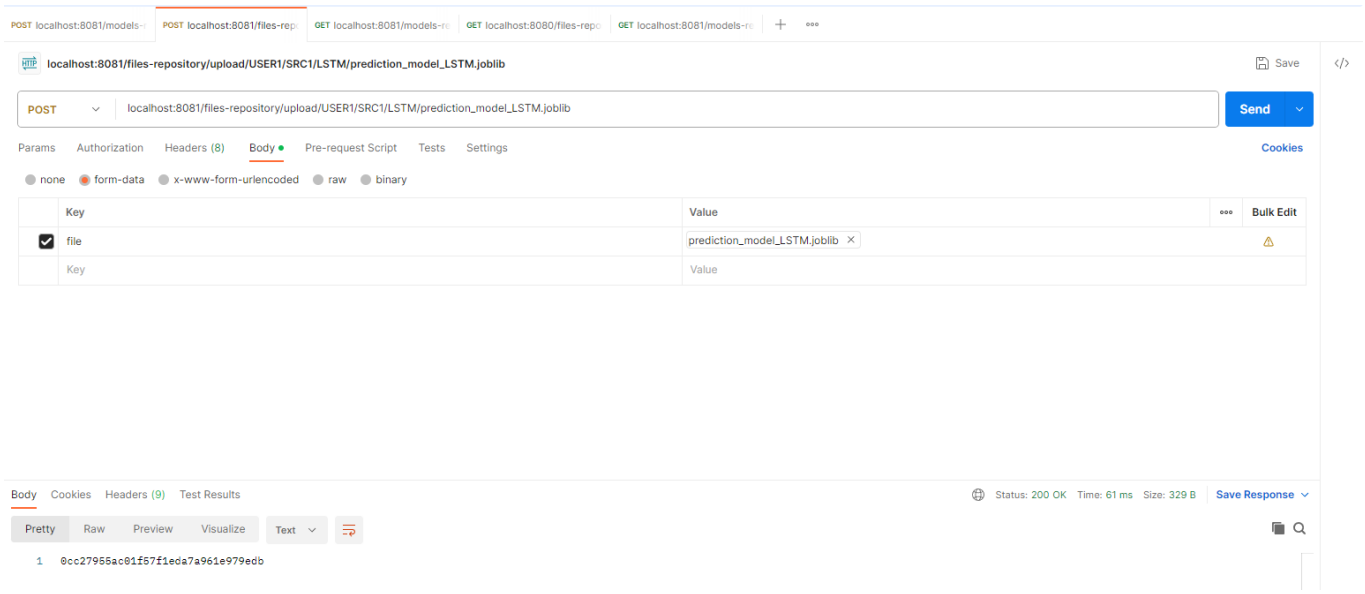

Рисунок 4.13 – Завантаження бінарного файлу моделі

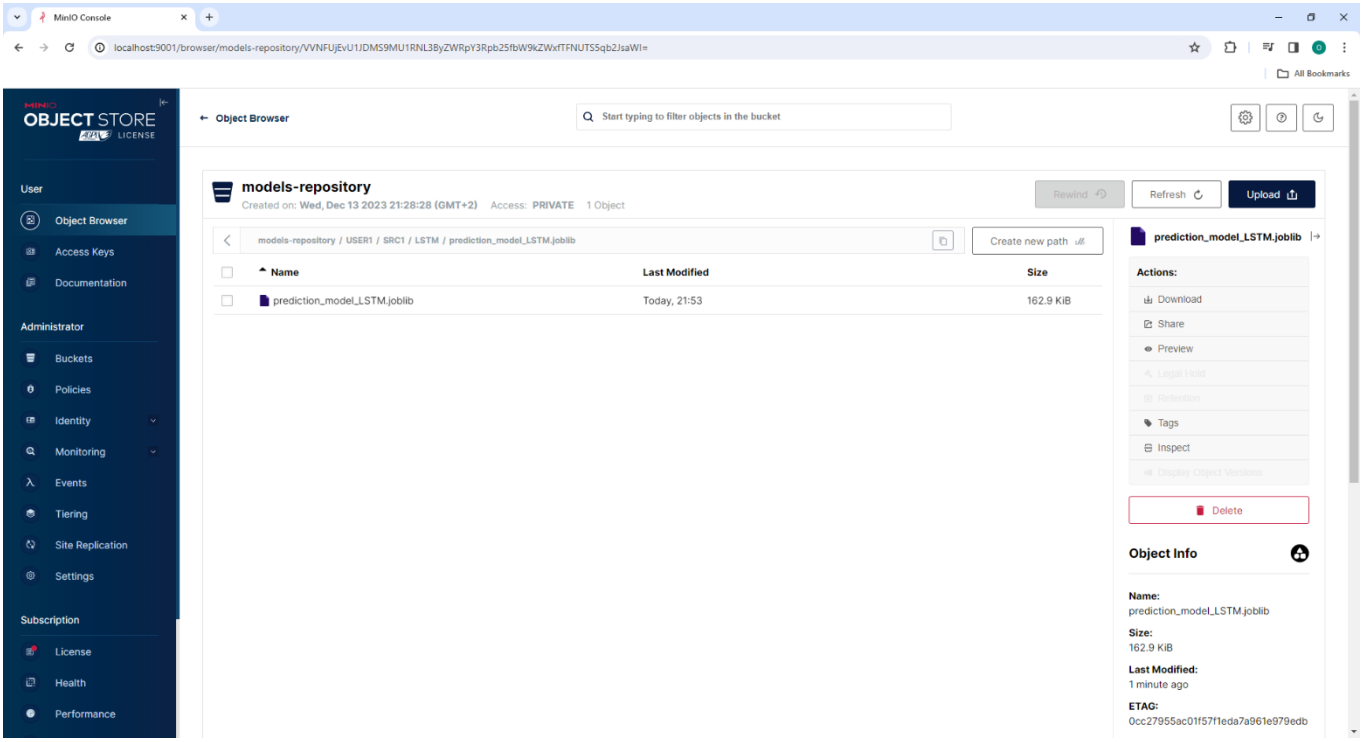

Рисунок 4.14 – Завантажена модель в об’єктному сховищі

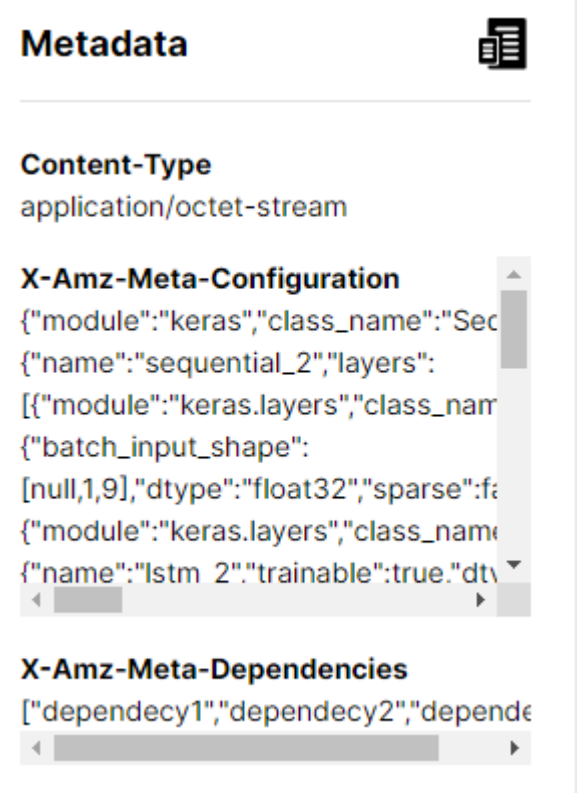

Рисунок 4.15 – Метадані моделі в об’єктному сховищі

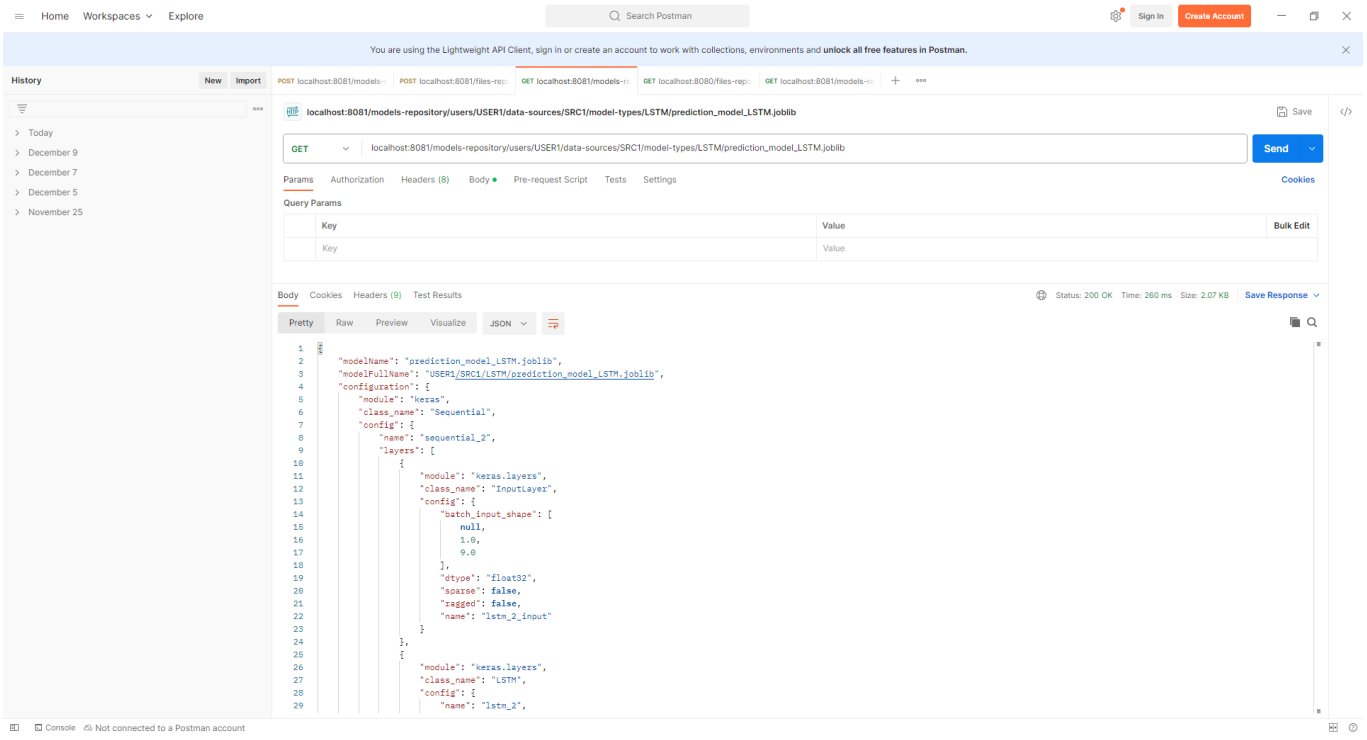

Рисунок 4.16 – Отримання метаданих моделі від підсистеми зберігання та обміну моделей

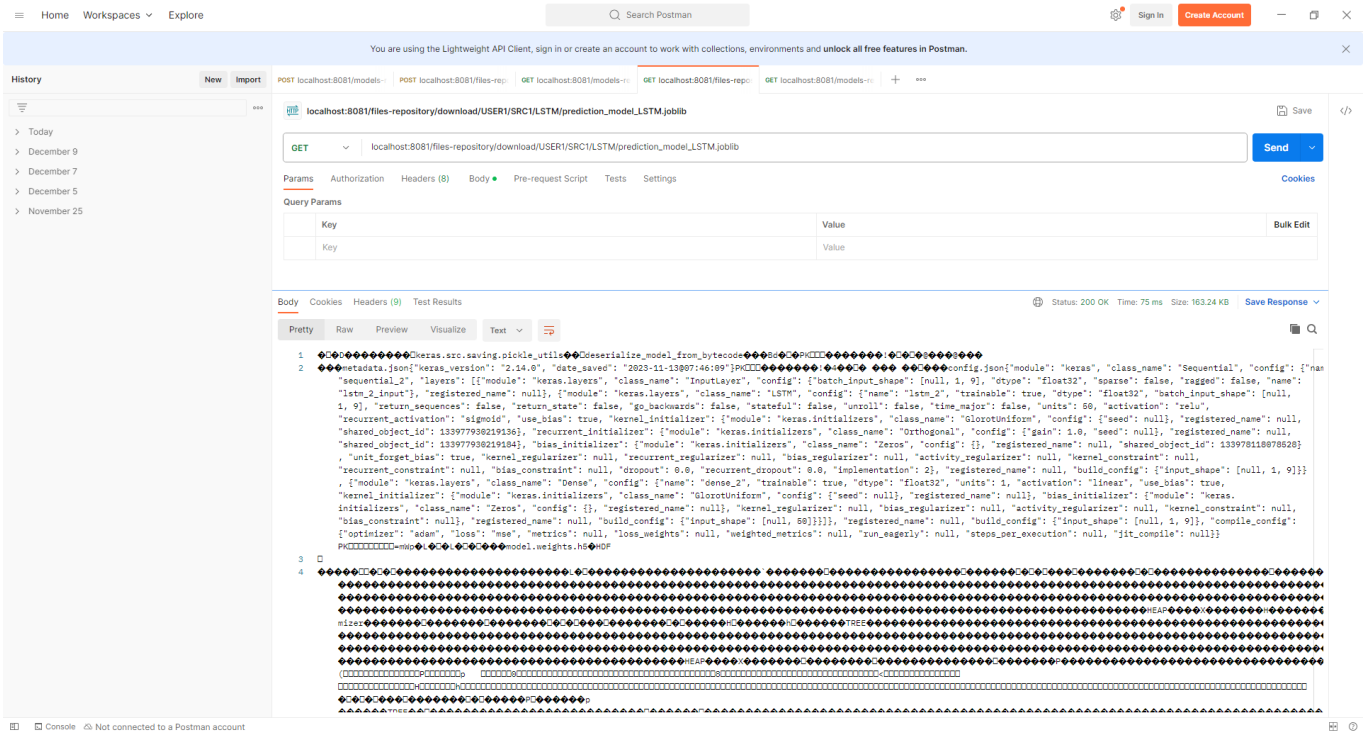

Рисунок 4.17 – Отримання бінарного файлу моделі від підсистеми зберігання та обміну моделей

Окрім запуску мікросервісу з середовища розробки IntelliJ Idea була також передбачена можливість запуску із середовища Docker. Для цього був написаний скрипт Docker наведений в додатку В.

Процес запуску мікросервісу за допомогою скрипту Docker файлу можна побачити на рисунку 4.18.

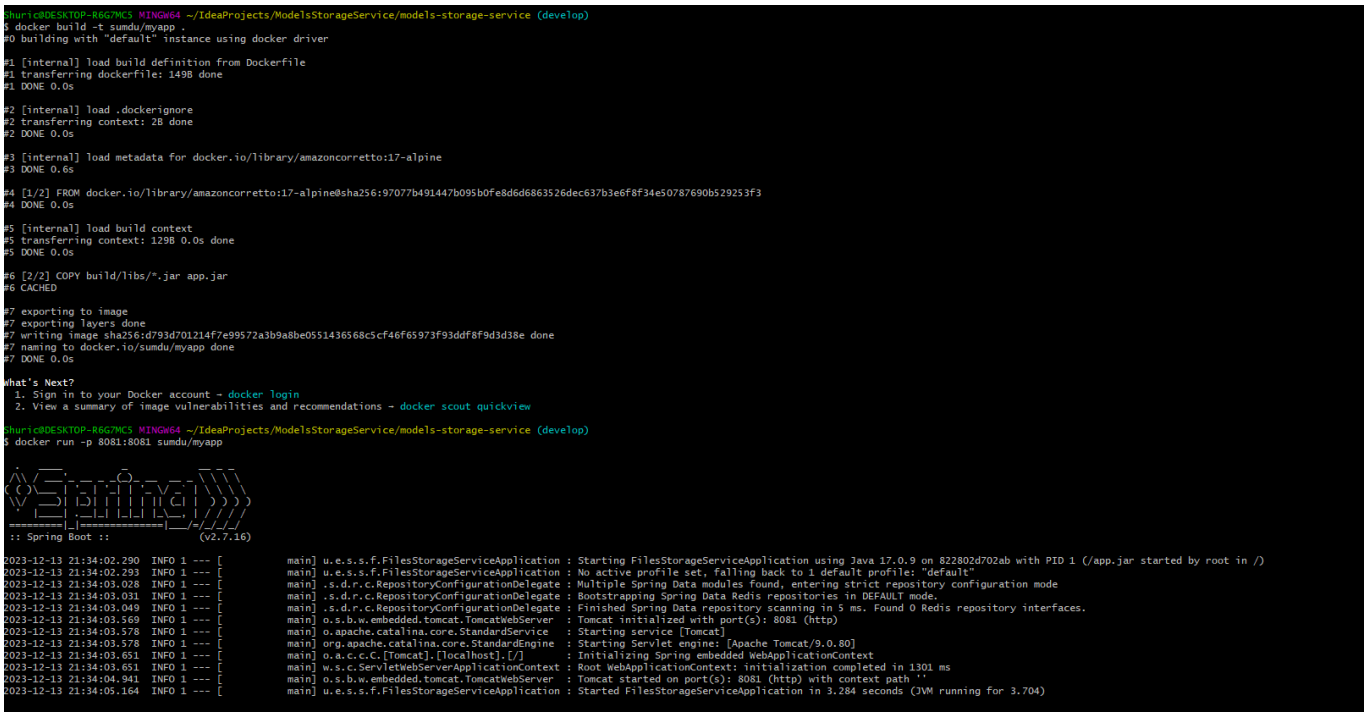

Рисунок 4.18 – Запуск мікросервісу за допомогою Docker-файлу

ВИСНОВКИ

В рамках кваліфікаційної роботи було виявлено та чітко поставлено задачі, розроблено графік виконання робіт, проведено аналіз ризиків та створено план та стратегію по їх мінімізації. Проведено аналіз існуючих рішень та технологій обміну та зберігання даних у хмарному сховищі необхідних для розробки підсистеми зберігання та обміну даними системи підтримки прийняття рішень при управлінні гібридною енергомережею. Також було формалізовано основні функціональні вимоги до цієї підсистеми.

В результаті порівняльного аналізу основних типів хмарних сховищ (об'єктне, блочне та файлове) було встановлено, що об'єктне сховище найбільше підходить для вирішення поставлених задач, оскільки воно забезпечує високу масштабованість, доступність та економічність. Крім того, об'єктне сховище дозволяє зберігати не тільки файлові дані моделей прогнозування, а також і набір метаданих, таких як конфігурація, залежності, тощо.

В ході роботи були розглянуті сучасні архітектурні підходи, фреймворки, технології та інструменти розробки. Було спроектовано архітектуру підсистеми з використанням RESTful архітектурного шаблону, Spring Boot мікросервісів та протоколу доступу до даних об'єктного сховища S3.

На основі вибраної архітектури та інструментів було розроблено концептуальну модель, включаючи діаграму варіантів використання та контекстні діаграми для підсистему зберігання та обміну моделей.

У результаті виконання кваліфікаційної роботи було виконано всі задачі, та реалізовано підсистему зберігання та обміну моделями, яку було розгорнуто локально, протестовано, оформлено документацію. За результатами аналізу тестування можна зробити висновки про ефективність вибраного методу організації зберігання та обміну моделей для системи підтримки прийняття рішень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Caroline E. Pavlowsky, Travis Gliedt: Geography and Sustainability, 2 (3): Individual and local scale interactions and adaptations to wind energy development: A case study of Oklahoma, USA / Elsevier, 2021. – 175-181 с.
2. Emmanuel P Agbo, Collins O Edet, Thomas O Magu, Armstrong O Njok, Chris M Ekpo, Hitler Louis: Heliyon, 7 (5): Solar energy: A panacea for the electricity generation crisis in Nigeria / Elsevier, 2021. – 21 с.
3. Pyrozhkov S.I. About the National Report of the NAS of Ukraine «National resilience of Ukraine: hybrid threats challenge response and prevention strategy». Visn. Nac. Akad. Nauk Ukr. 2022. (5): 45—55. https://doi.org/10.15407/visn2022.05.045
4. Heyets V.M. On the assessment of Ukraine's economic losses due to the armed aggression of the Russian Federation: Visn. Nac. Akad. Nauk Ukr. 2022. (5): 30—38. https://doi.org/10.15407/visn2022.05.030
5. From the Conference Hall of the Presidium of the NAS of Ukraine, March 30, 2022. Visn. Nac. Akad. Nauk Ukr. 2022. (5): 7—11.
6. Відновлювані джерела енергії / За ред. С.О. Кудрі. – Київ: Інститут відновлюваної енергетики НАНУ, 2020. – 392 с.
7. Kyrylenko O.V., Snezhkin Y.F., Basok B.I., Bazyeev Y.T. Ukraine's energy: probable scenarios of recovery and development. Visn. Nac. Akad. Nauk Ukr. 2022. (9): 22—37. https://doi.org/10.15407/visn2022.09.022
8. Obukhov S., Ibrahim A., Tolba M.A., M.El-Rifaie A. Power balance management of an autonomous hybrid energy system based on the dual-energy storage // Energies, 2019, v.12; doi:10.3390/en12244690.
9. John K. Kaldellis: Stand-alone and hybrid wind energy systems: Technology, energy storage and applications / Elsevier, 2010. – 554 с.
10. Yu Luo, Yixiang Shi, Ningsheng Cai: Hybrid Systems and Multi-energy Networks for the Future Energy Internet / Elsevier, 2010. – 240 с.

11. Шендрик С. О., Тимчук С. О. Аналіз предметної області прийняття рішень при управлінні гібридними енергомережами. Автоматика – 2017: матеріали XXIV Міжнародна конференція з автоматичного управління, Київ, 13–15 вересня 2017 р. Київ. 2017. С. 221-222
12. Joseph S. Valacich, Christoph Schneider, Matthew Hashim: Information systems today: managing in the digital world / Pearson Education, 2022. - 568 с.
13. Saiqa Aleem, Rabia Batool, Faheem Ahmed, Asad Khatak, and Raja Muhammad Ubaid Ullah. Architecture guidelines for saas development process. In Proceedings of the 2017 International Conference on Cloud and Big Data Computing, ICCBDC 2017, page 94–99, New York, NY, USA, 2017. Association for Computing Machinery.
14. Fei Hu: Big Data: Storage, Sharing, and Security / CRC Press, 2016. – 430 с.
15. Antoni Olivé: Conceptual modeling of information systems / Springer Berlin Heidelberg, 2007. – 455 с.
16. Roel J. Wieringa: Design science methodology: For information systems and software engineering / Springer Berlin Heidelberg, 2014. – 332 с.
17. Raul Gracia-Tinedo, Marc S´anchez Artigas, Adri´an Moreno-Mart´´ınez, Cristian Cotes, and Pedro Garc´ıa Lopez. Actively measuring personal cloud storage. In 2013 IEEE Sixth International Conference on Cloud Computing, pages 301–308, 2013.
18. Greg Schulz: Cloud and Virtual Data Storage Networking / Auerbach Publications, 2011. – 400 с.
19. Or Ozeri, Effi Ofer, and Ronen Kat. Object storage for deep learning frameworks. In Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning, DIDL '18, page 21–24, New York, NY, USA, 2018. Association for Computing Machinery.
20. Vlad Bucur, Catalin Dehelean, and Liviu Miclea. Object storage in the cloud and multi-cloud: State of the art and the research challenges. In 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), pages 1–6, 2018.
21. Google Cloud [електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/ (дата звернення: 10.11.2023)

22. Danny Burns: Systemic action research: A strategy for whole system change / Policy Press, 2007. – 194 с.
23. Aaron Wheeler, Michael Winburn: Cloud Storage Security: A Practical Guide / Elsevier, 2015. – 138 с.
24. Top 10 Cloud Provider Comparison 2023: VM Performance / Price [електронний ресурс] - Режим доступу до ресурсу: https://dev.to/dkechag/cloud-vm-performance-value-comparison-2023-perl-more-1kpp (дата звернення: 10.11.2023)
25. Yi Su, Dan Feng, Yu Hua, and Zhan Shi. Understanding the latency distribution of cloud object storage systems. Journal of Parallel and Distributed Computing, 128:71–83, 2019.
26. Dan C. Marinescu: Cloud Computing: Theory and Practice / Elsevier, 2017. – 566 с.
27. Smith, K.P., Seligman, L.J., Rosenthal, A., Kurcz, C., Greer, M., Macheret, C., Sexton, M., Eckstein, A., 2014. ”big metadata”: The need for principled metadata management in big data ecosystems, in: Katsifodimos, A., Tzoumas, K., Babu, S. (Eds.), Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference, ACM. pp. 13:1–13:4. URL: https://doi.org/10.1145/2627770.2627776, doi:10.1145/2627770.2627776.
28. Erl, T. (2005). Service-Oriented Architecture Concepts, Technology, and Design. Prentice Hall, Upper Saddle River, NJ, USA.
29. Erl, T. (2007). SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Prentice Hall, Upper Saddle River, NJ, USA.
30. Moisés Macero García: Learn Microservices with Spring Boot: A Practical Approach to RESTful Services Using an Event-Driven Architecture, Cloud-Native Patterns, and Containerization / Springer International Publishing, 2020. – 426 с. Moisés Macero García: Learn Microservices with Spring Boot: A Practical Approach to RESTful Services Using an Event-Driven Architecture, Cloud-Native Patterns, and Containerization / Springer International Publishing, 2020. – 426 с.

31.Hitesh Kumar Sharma, Anuj Kumar, Sangeeta Pant, Mangey Ram: DevOps: A journey from microservice to cloud based containerization / River Publishers, 2023. – 172 с.

32.James Turnbull: The Docker Book: Containerization is the new virtualization / Shroff Publishers, 2014. – 344 с.

33. William Penberthy, Steve Roberts: S3 Object Storage. In Pro .NET on Amazon Web Services: Guidance and Best Practices for Building and Deployment / Apress, 2022. – 269-301 с.

34. MinIO. High Performance Multi - Cloud Object Storage [електронний ресурс] - Режим доступу до ресурсу: https://min.io/resources/docs/MinIO-High-Performance-Multi-Cloud-Object-Storage.pdf (дата звернення: 01.12.2023)

35.Goodman SN, Schneeweiss S, Baiocchi M. Using Design Thinking to Differentiate Useful From Misleading Evidence in Observational Research. JAMA. 2017 Feb 21;317(7):705-707. doi: 10.1001/jama.2016.19970. PMID: 28241335.

36.D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4), Oct. 1992, pp. 40-52.

37.Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

38.Tanya Reilly: A Guide for Individual Contributors Navigating Growth and Change / O'Reilly Media, 2022. – 256 с.

39.Harold Abelson, Gerald Jay Sussman: Structure and Interpretation of Computer Programs / The MIT Press, 1996. – 657 с.

40. C.A. Costa, J.A. Harding, R.I.M. Young: The application of UML and an open distributed process framework to information system design /Computers in Industry, Volume 46, Issue 1, 2001 - Pages 33-48. URL: https://doi.org/10.1016/S0166-3615(01)00109-9, doi: 10.1016/S0166-3615(01)00109-9.

41.IntelliJ IDEA – the Leading Java and Kotlin IDE [електронний ресурс] - Режим доступу до ресурсу: https://www.jetbrains.com/idea/ (дата звернення: 09.11.2023)

42. Docker Subscription Service Agreement IDE [електронний ресурс] - Режим доступу до ресурсу: https://www.docker.com/legal/docker-subscription-service-agreement/ (дата звернення: 13.12.2023)

ДОДАТОК А

ПЛАНУВАННЯ РОБІТ
для розробки кваліфікаційної роботи магістра
«Підсистема зберігання та обміну даними системи підтримки прийняття рішень при управлінні гібридною енергомережею»

А.1 ІДЕНТИФІКАЦІЯ МЕТИ ПРОЕКТУ

У рамках розробки програмного модуля для активації прогнозних моделей в системі управління гібридною енергомережею, SMART-методологія дає можливість чітко визначити та структурувати цілі проекту. Розшифровка термінів SMART допомагає нам зосередитися на ключових аспектах мети проекту: вона має бути конкретною (Specific), вимірюваною (Measurable), досяжною (Achievable), реалістичною (Relevant) та обмеженою в часі (Time-framed).

- **S**: Мета дипломного проекту полягає у розробці сервісу, який забезпечуватиме доступ та збереження моделей для управління гібридною енергомережею.
- **M**: Ефективність сервісу можна вимірювати по тому скільки запитів на отримання моделей він може обробити одночасно (пропускна здатність). Критерії оцінки ефективності включають зниження помилок у прогнозах, покращення балансування навантаження та ефективність відповіді на зміни у мережі.
- **A**: В наявності є всі необхідні технології та ресурси для розробки цього сервісу, включаючи інструменти розробки а також середовище розгортання.
- **R**: Проект відповідає сучасним тенденціям в управлінні енергетичними системами та відображає поточні потреби галузі. Команда розробників має відповідні знання та досвід дляреалізаціїцього проекту.
- **T**: Проект має чітко визначені терміни виконання у відповідності до запропонованого графіку розробки та впровадження сервісу в систему управління гібридною енергомережею.

Таблиця А.1 – Деталізація мети методом SMART

Категорія SMART	Опис
Specific (конкретна)	Розробка сервісу зберігання та обміну даними моделей в енергомережі.
Measurable (вимірювана)	Оцінка ефективності модуля за точністю пропускною здатністю та кількістю одночасних запитів.
Achievable (досяжна)	Доступ до необхідних технологій та ресурсів.
Relevant (реалістична)	Проект відповідає актуальним потребам та стандартам галузі, команда має відповідні компетенції.
Time-framed (обмежена у часі)	Чітко визначені терміни для розробки та впровадження модуля.

А.2 ПЛАНУВАННЯ ЗМІСТУ СТРУКТУРИ РОБІТ ІНФОРМАЦІЙНОЇ СИСТЕМИ

У контексті розробки сервісу для обміну та збереження моделей в системі підтримки прийняття рішень гібридною енергомережею, структура розподілу робіт (Work Breakdown Structure, WBS) є ключовою для організації та планування проекту. WBS допомагає визначити ієрархію та залежності між різними етапамиробітісприяє чіткому розумінню обсягу та змісту проекту.

WBS організовує і визначає зміст всього проекту; верхній рівень WBS – це кінцевий продукт, програмний модуль. Кожен наступний рівень деталізує конкретні завдання та етапи, необхідні для досягнення цієї мети. Це допомагає визначити, які види діяльності включені в проект, а які ні.

Формування технічного завдання: Включає визначення функціональних та технічних вимог до сервісу, вибір технологій та платформ для розробки. Також на цьому етапі визначаються цілі та можливості проекту.

Планування проекту: Розробка OBS (Organizational Breakdown Structure) та матриці відповідальності, планування ресурсів, оцінка ризиків, розробка календарного плану (включаючи діаграму Ганта).

Реалізація проекту: Поділяється на кілька основних етапів, таких як проектування сервісу, його розробку, тестування та оптимізацію. Головна увага приділяється якості та ефективності роботи сервісу.

Впровадження та закриття проекту: Включає фінальне тестування, виправлення помилок,документуваннятавведеннямодуля в експлуатацію.

Діаграми WBS та OBS:

- Діаграма WBS: На рисунку А.1 наведено діаграму WBS, яка ілюструє ієрархічну структуру робіт в проекті.
- Діаграма OBS: На рисунку А.2 представлено OBS, що демонструє організаційну структуру проекту та розподіл відповідальності між учасниками.

Такий підхід до планування змісту структури робіт забезпечує чітке розуміння та ефективне управління проектом, від початкового етапу формування технічного завдання до завершального етапу реалізації та закриття проекту.

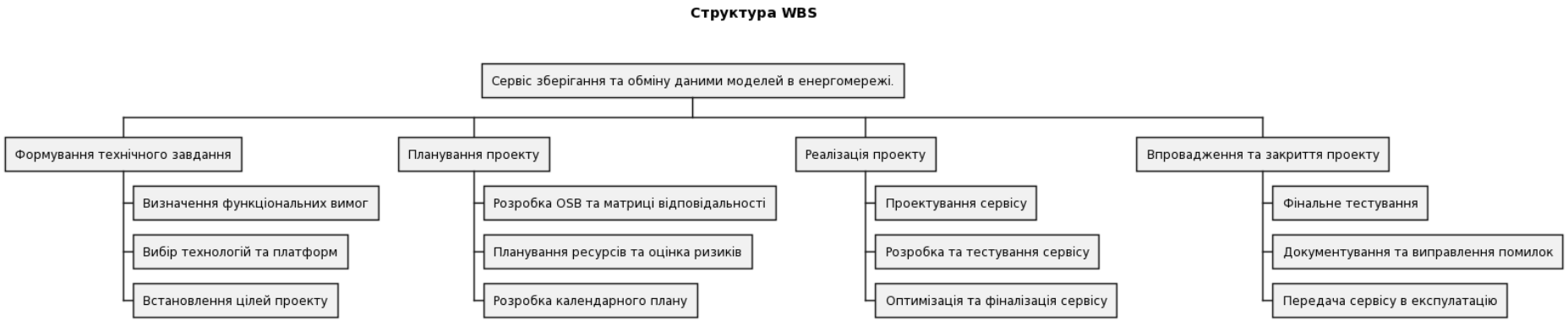

Рисунок А.1 – Структура WBS

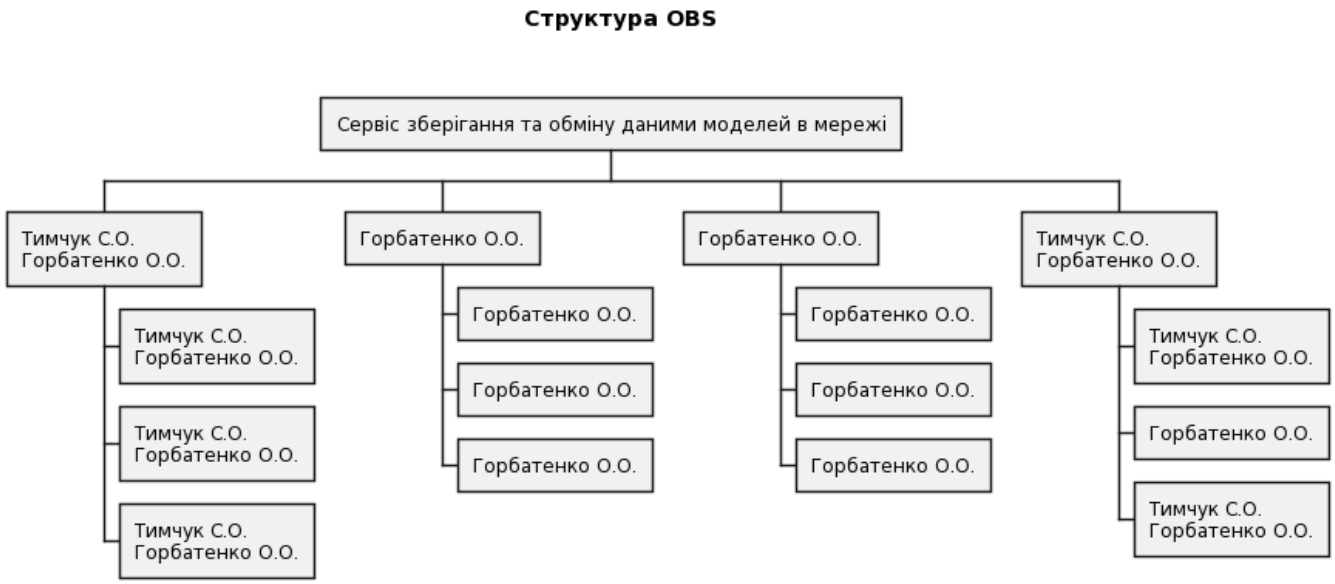

Рисунок А.2 – Структура OBS

А.3 ПОБУДОВА КАЛЕНДАРНОГО ГРАФІКУ ВИКОНАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

Для того, щоб мати уявлення про строки виконання робіт з урахуванням обмеженості у використанні ресурсів, а також мінімізації ризиків та затримок, будують календарний графік робіт.

Діаграма Ганта – горизонтальна лінійна діаграма, на якій задачі проекту представляються протяжними в часі відрізками, що характеризуються датами початку та закінчення, затримками і, можливо, іншими тимчасовими параметрами.

Кожен відрізок відповідає окремому завданню або підзадачі. Завдання і підзадачі, складові плану, розміщуються по вертикалі. Початок, кінець і довжина відрізка на шкалі часу відповідають початку, кінцю і тривалості завдання. На деяких діаграмах Ганта також показується залежність між завданнями.

На рисунку А.3 представлено побудовану діаграму Ганта розробки проекту.

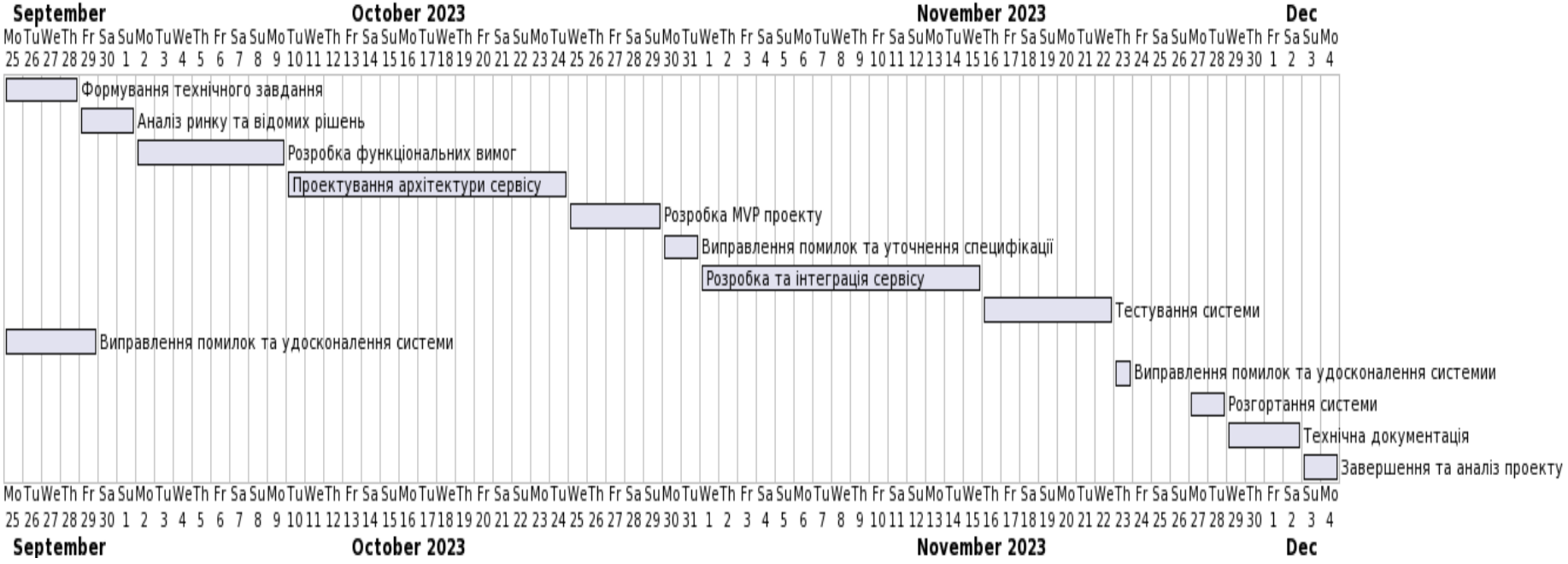

Рисунок А.3 – Діаграма Ганта

А.4 ПЛАНУВАННЯ РИЗИКІВ ПРОЕКТУ

Управління ризиками є критично важливим елементом у розробці програмного модуля для активації прогнозних моделей в системі управління гібридною енергомережею. Цей процес включає ідентифікацію потенційних ризиків, оцінку їх впливу на проект, розробку стратегій реагування та постійний моніторинг.

Процес управління ризиками включає:

- **Ідентифікація ризиків**: Виявлення потенційних загроз проекту, включаючи технічні, економічні, юридичні та екологічні ризики.
- **Оцінювання ризиків**: Використання методу експертних оцінок для визначення ймовірності та впливу кожного ризику.
- **Планування реагування на ризики**: Розробка заходів щодо попередження ризиків або мінімізації їх наслідків.
- **Моніторинг ризиків**: Постійне відстеження ідентифікованих ризиків та ефективності заходів реагування.

Таблиця А.2 – Ймовірність втрат

Ймовірність виникнення	Величина втрат
Слабоймовірно	Мінімальна
Малоймовірно	Низька
Ймовірно	Середня
Вельми ймовірно	Висока

Майже можливо	Максимальна

Таблиця А.3 – Класифікація за ступенем впливу та за рівнем ризику

Ризик	Ступінь впливу	Рівень ризику
Кібератаки	4	Ігноровані
Пошкодження даних	4	Ігноровані
Серверні проблеми	3	Незначні
Вихід з ладу обладнання	6	Незначні

План по усуненню ризиків:

- Підвищення безпеки системи: Впровадження сучасних технологій захисту даних та регулярний аудит безпеки.
- Резервне зберігання даних: Створення надійної системи резервного копіювання та відновлення даних.
- Оновлення та підтримка системи: Забезпечення своєчасного оновлення програмного забезпечення та апаратної частини.
- Забезпечення контингентності: Розробка плану дій на випадок виходу з ладу ключового обладнання, щоб забезпечити неперервність робіт над проектом.

Ці таблиці та плани дій допомагають команді проектів передбачати можливі проблеми та запобігати їх виникненню, що забезпечує стабільність проекту.

ДОДАТОК Б

Б.1 ЛІСТИНГ ПРОГРАМНОГО КОДУ

Головний клас програми ModelsStorageServiceApplication.java:

```
package ua.edu.sumdu.sppr.filesstorageservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ModelsStorageServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ModelsStorageServiceApplication.class, args);
    }

}
```

Контролер для управління метаданими ObjectsStorageController.java:

```
package ua.edu.sumdu.sppr.filesstorageservice.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;
import ua.edu.sumdu.sppr.filesstorageservice.application.model.ModelRequest;
import ua.edu.sumdu.sppr.filesstorageservice.domain.model.PredictionModelData;
import ua.edu.sumdu.sppr.filesstorageservice.domain.service.ModelsStorageService;

@CrossOrigin
```

```
@RestController
@RequestMapping("/models-repository/users/{userIdentity}/data-sources/{dataSourceId}/model-types/{modelType}")
public class ObjectsStorageController {

   private final ModelsStorageService storageService;

   @Autowired
   public ObjectsStorageController(ModelsStorageService storageService) {
      this.storageService = storageService;
   }

   @GetMapping(path ="/{modelName}",
         produces = {
            MediaType.APPLICATION_JSON_VALUE
   })
   public PredictionModelData getModelMetadata(@PathVariable String userIdentity,
                                    @PathVariable String dataSourceId,
                                    @PathVariable String modelType,
                                    @PathVariable String modelName) {

      String path = userIdentity + "/" + dataSourceId + "/" + modelType;

      return storageService.getModelMetadata(path, modelName);
   }

   @PostMapping(consumes = {
         MediaType.APPLICATION_JSON_VALUE
   })
   public String createModelMetadata(@PathVariable String userIdentity,
                        @PathVariable String dataSourceId,
```

```
                @PathVariable String modelType,
                @RequestBody ModelRequest request) {

    String path = userIdentity + "/" + dataSourceId + "/" + modelType + "/" +
request.getModelName();

    return storageService.storeModel(path, request.getConfiguration(),
request.getDependencies());
  }
}
```

Контролер для управління файловими даними FilesController.java:

```
package ua.edu.sumdu.sppr.filesstorageservice.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import ua.edu.sumdu.sppr.filesstorageservice.domain.service.ModelsStorageService;

import java.io.IOException;

@CrossOrigin
@RestController
@RequestMapping("/files-repository")
public class FilesController {

  private final ModelsStorageService storageService;

  @Autowired
  public FilesController(ModelsStorageService storageService) {
```

```
    this.storageService = storageService;
}

@PostMapping(
        path = "/upload/{userIdentity}/{dataSourceId}/{modelType}/{modelName}",
        consumes = {
            MediaType.MULTIPART_FORM_DATA_VALUE
        }
)
public ResponseEntity<String> uploadModelFile(
                        @PathVariable String userIdentity,
                        @PathVariable String dataSourceId,
                        @PathVariable String modelType,
                        @PathVariable String modelName,
                        @RequestParam("file") MultipartFile modelFile) {

    String path = userIdentity + "/" + dataSourceId + "/" + modelType + "/" + modelName;

    String eTag = null;
    try {
        eTag = storageService.uploadFile(path, modelFile.getInputStream(), modelFile.getSize());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    return ResponseEntity.ok(eTag);
}

@GetMapping(
        path = "/download/{userIdentity}/{dataSourceId}/{modelType}/{modelName}"
)
public @ResponseBody byte[] downloadModelFile(
```

```
                    @PathVariable String userIdentity,
                    @PathVariable String dataSourceId,
                    @PathVariable String modelType,
                    @PathVariable String modelName) {

        String path = userIdentity + "/" + dataSourceId + "/" + modelType + "/" + modelName;

        return storageService.downloadFile(path);
    }

}
```

Інтерфейс ModelsStorageService.java:

```
package ua.edu.sumdu.sppr.filesstorageservice.domain.service;

import ua.edu.sumdu.sppr.filesstorageservice.domain.model.PredictionModelData;

import java.io.InputStream;
import java.util.List;
import java.util.Map;

public interface ModelsStorageService {

    PredictionModelData getModelMetadata(String path, String modelName);
    String storeModel(String path, Map<String, Object> configuration, List<String> dependencies);

    String uploadFile(String path, InputStream file, long fileSize);

    byte[] downloadFile(String path);
}
```

Реалізація сервісу MinIoModelsStorageServiceImpl.java:

```
package ua.edu.sumdu.sppr.filesstorageservice.domain.service.impl;

import io.minio.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.Caching;
import org.springframework.stereotype.Service;
import ua.edu.sumdu.sppr.filesstorageservice.application.utils.JsonUtils;
import ua.edu.sumdu.sppr.filesstorageservice.domain.model.PredictionModelData;
import ua.edu.sumdu.sppr.filesstorageservice.domain.service.ModelsStorageService;

import java.io.ByteArrayInputStream;
import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Service("ModelsStorageService")
public class MinIoModelsStorageServiceImpl implements ModelsStorageService {

   private final MinioClient minioClient;

   public MinIoModelsStorageServiceImpl(
         @Value("${app.minio.host}") String minioHost,
         @Value("${app.minio.port}") int minioPort,
         @Value("${app.minio.is-secure}") boolean isSecureConnection,
         @Value("${app.minio.user-name}") String minioAccessKey,
         @Value("${app.minio.password}") String minioSecretKey
   ) {
      minioClient =
            MinioClient.builder()
                  .endpoint(minioHost, minioPort, isSecureConnection)
```

```
                .credentials(minioAccessKey, minioSecretKey)
                .build();
    }

    public PredictionModelData getModelMetadata(String path, String modelName) {
        try {
            String modelFullName = path + "/" + modelName;

            StatObjectResponse response = minioClient.statObject(StatObjectArgs
                    .builder()
                    .bucket("models-repository")
                    .object(modelFullName)
                    .build()
            );

            PredictionModelData modelData = new PredictionModelData(modelName);
            modelData.setModelFullName(modelFullName);

            modelData.setETag(response.etag());

            String configurationMetadata = response.userMetadata().get("configuration");
            if (configurationMetadata != null && !configurationMetadata.isBlank()) {
                modelData.setConfiguration(JsonUtils.getInstance().getMapFromJson(configurationMetadata));
            }

            String dependenciesMetadata = response.userMetadata().get("dependencies");
            if (dependenciesMetadata != null && !dependenciesMetadata.isBlank()) {
                modelData.setDependencies(JsonUtils.getInstance().getListFromJson(dependenciesMetadata));
            }

            return modelData;
        } catch (Exception e) {
```

```
        throw new RuntimeException(e);
    }
}

@Override
@Caching( cacheable = {
        @Cacheable(value = "modelFilesCache", unless="#result.length > 1024 * 1024")
    }
)
public byte[] downloadFile(String path) {

    try {
        return  minioClient.getObject(GetObjectArgs
                .builder()
                .bucket("models-repository")
                .object(path)
                .build()
        ).readAllBytes();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

@Override
public String storeModel(String path, Map<String, Object> parameters, List<String> dependencies) {

    Map<String, String> metadata = new HashMap<>();
    metadata.put("configuration", JsonUtils.getInstance().toJsonString(parameters));
    metadata.put("dependencies", JsonUtils.getInstance().toJsonString(dependencies));

    try {
        return minioClient.putObject(PutObjectArgs
```

```
            .builder()
            .bucket("models-repository")
            .object(path)
            .userMetadata(metadata)
            .stream(new ByteArrayInputStream(new byte[] {}), 0, -1)
              .build()
      ).object();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
}

@Override
public String uploadFile(String path, InputStream file, long fileSize) {
    try {
      Map<String, String> metadata = minioClient.statObject(StatObjectArgs
            .builder()
            .bucket("models-repository")
            .object(path)
            .build()).userMetadata();

      return minioClient.putObject(PutObjectArgs
                  .builder()
                  .bucket("models-repository")
                  .object(path)
                  .userMetadata(metadata)
                  .stream(file, fileSize, -1)
            .build()
      ).etag();
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
```

```
    }
}
```

Клас моделі PredictionModelData.java:

```
package ua.edu.sumdu.sppr.filesstorageservice.domain.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.Getter;
import lombok.Setter;

import java.util.List;
import java.util.Map;

@Getter
@Setter
public class PredictionModelData {

    public PredictionModelData(String modelName) {
        this.modelName = modelName;
    }

    private String modelName;
    private String modelFullName;
    private Map<String, String> configuration;
    private List<String> dependencies;

    private String eTag;

    @JsonIgnore
    private byte[] binaryData;
}
```

Б.2 КОНФІГУРАЦІЙНІ ФАЙЛИ ПРОГРАМИ

Основний файл конфігурації програми application.properties:

```
server.address=localhost
server.port=8081

app.minio.host=localhost
app.minio.port=9000
app.minio.is-secure=false
app.minio.user-name=minioadmin
app.minio.password=minioadmin

#NONE, SIMPLE or REDIS are supported
spring.cache.type=REDIS

spring.redis.host=localhost

spring.servlet.multipart.max-file-size=120MB
spring.servlet.multipart.max-request-size=128MB

#----------------
# Swagger
#----------------
spring.mvc.pathmatch.matching-strategy=ANT_PATH_MATCHER
```

Файл конфігурації збірки за допомогою інструменту Gradle (build.gradle):

```
plugins {
   id 'java'
```

```
    id 'org.springframework.boot' version '2.7.16'
    id 'io.spring.dependency-management' version '1.1.3'
}

group = 'ua.edu.sumdu.diploma'
version = '0.0.1-SNAPSHOT'

java {
    sourceCompatibility = '11'
}

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.google.code.gson:gson:2.10.1'

    implementation 'org.springframework.boot:spring-boot-starter-cache'
    implementation 'org.springframework.boot:spring-boot-starter-data-redis'
    implementation 'org.springframework.boot:spring-boot-starter-web'

    implementation 'redis.clients:jedis'

    implementation 'io.minio:minio:8.5.2'
```

```
    implementation 'io.springfox:springfox-boot-starter:3.0.0'

    compileOnly 'org.projectlombok:lombok'

    annotationProcessor 'org.projectlombok:lombok'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('bootBuildImage') {
    builder = 'paketobuildpacks/builder-jammy-base:latest'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

ДОДАТОК В

В.1 СКРИПТИ РОЗГОРТАННЯ

Docker файл основної програми:

```
FROM amazoncorretto:17-alpine

VOLUME /tmp

COPY build/libs/*.jar app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Docker compose файл для запуску скрвісу MinIO, nginx та Redis:

```
version: '3.7'


# Settings and configurations that are common for all containers

x-minio-common: &minio-common

  image: quay.io/minio/minio:latest

  command: server --console-address ":9001" http://minio1/data{1...2}

  expose:

    - "9000"

    - "9001"

  # environment:

    # MINIO_ROOT_USER: minioadmin

    # MINIO_ROOT_PASSWORD: minioadmin

  healthcheck:

    test: ["CMD", "mc", "ready", "local"]
```

```
    interval: 5s
    timeout: 5s
    retries: 5

# starts 4 docker containers running minio server instances.
# using nginx reverse proxy, load balancing, you can access
# it through port 9000.
services:
  redis:
    image: redis:alpine
    restart: always
    ports:
      - '6379:6379'
    volumes:
      - cache:/data

  minio1:
    <<: *minio-common
    hostname: minio1
    volumes:
      - data1-1:/data1
      - data1-2:/data2

  nginx:
```

```
    image: nginx:1.19.2-alpine
    hostname: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    ports:
      - "9000:9000"
      - "9001:9001"
    depends_on:
      - minio1

## By default this config uses default local driver,
## For custom volumes replace with volume driver configuration.
volumes:
  data1-1:
  data1-2:
  cache:
    driver: local
```