ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка мікросервісної архітектури для масштабованої системи розпізнавання мови із використанням Docker і RabbitMQ»

на здобуття освітнього ступеня магістра
зі спеціальності 126 Інформаційні системи та технології
(код, найменування спеціальності)
освітньо-професійної програми 126 Інформаційні системи татехнології
(назва)

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

_____ *Андрій БАЧКОВ*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. ІСДМ-63
Андрій БАЧКОВ
Ім'я, ПРІЗВИЩЕ

Керівник: Андрій БОНДАРЧУК
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент:
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут Інформаційних технологій

Кафедра Інформаційних систем та технологій

Ступінь вищої освіти магістр

Спеціальність Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
Каміла СТОРЧАК
(Ім'я, ПРІЗВИЩЕ)
«____» _____ 20__ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бачков Андрій Володимирович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Розробка мікросервісної архітектури для масштабованої системи розпізнавання мови із використанням Docker і RabbitMQ

керівник кваліфікаційної роботи Андрій БОНДАРЧУК, д.т.н., професор ,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024р. №320

2. Строк подання кваліфікаційної роботи «27» грудня 2024р.

3. Вихідні дані до кваліфікаційної роботи:
1. науково-технічна література з теми розпізнавання мовлення та мікросервісної архітектури;
2. вимоги до систем автоматизованого розпізнавання мовлення;
3. функціональні характеристики мікросервісних платформ та інструментів контейнеризації (Docker, Redis, RabbitMQ, MongoDB);
4. дані для навчання та тестування системи розпізнавання мовлення (аудіофайли, транскрипції тощо).
5.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз принципів побудови мікросервісної архітектури для розпізнавання мовлення.
2. Дослідження технологій інтеграції компонентів системи (API, Coordinator, Worker) та механізмів обміну даними між ними.

3. Розробка архітектури мікросервісної системи для автоматизованого розпізнавання мовлення.
4. Дослідження методів оптимізації продуктивності системи при одночасній обробці кількох аудіофайлів.

5. Перелік ілюстративного матеріалу: *презентація*

6. Дата видачі завдання «15» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір науково-технічної літератури та аналіз проблематики	15.10-16.10.24	виконав
2	Огляд сучасних підходів і технологій у розпізнаванні мовлення	17.10-18.10.24	виконав
3	Обґрунтування вибору технологій і підходів для розпізнавання мовлення	19.10-22.10.24	виконав
4	Проектування архітектури системи розпізнавання мовлення	23.10-27.10.24	виконав
5	Розробка та налаштування інтерфейсу для взаємодії з користувачами	28.10-06.11.24	виконав
6	Реалізація модулів обробки даних	07.11-25.11.24	виконав
7	Контейнеризація компонентів за допомогою Docker	26.11-07.12.24	виконав
8	Впровадження та тестування системи з використанням баз даних та систем черг	08.12-13.12.24	виконав
9	Аналіз ефективності системи та її оптимізація	14.12-17.12.24	виконав
10	Оцінка економічної доцільності системи	18.12-21.12.24	виконав
11	Підготовка до передзахисту: доповідь, презентація, роздаткові матеріали	22.12-26.12.24	виконав

Здобувач вищої освіти _____ Андрій БАЧКОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Андрій БОНДАРЧУК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 92 стор., 41 рис., 11 табл., 23 джерел.

Метою роботи є розробка та аналіз мікросервісної архітектури для масштабованої системи розпізнавання мови, яка використовує Docker для контейнеризації та RabbitMQ для забезпечення ефективного обміну повідомленнями між компонентами системи

Об'єктом дослідження є система розпізнавання мови, яка побудована на основі мікросервісної архітектури.

Предметом дослідження є процеси проектування, розробки та масштабування системи розпізнавання мови за допомогою Docker та RabbitMQ.

Короткий зміст роботи. У цій дипломній роботі досліджено важливість розпізнавання мовлення у сучасному світі, виявлено недоліки та обмеження існуючих систем. Проаналізовано сучасні підходи та технології, обґрунтовано вибір інструментів та технологій для розробки системи. Розроблено та впроваджено систему розпізнавання мовлення на основі мікросервісної архітектури, використовуючи Docker та RabbitMQ. Проведено оцінку ефективності та економічної доцільності системи, що дозволило визначити її переваги та недоліки у порівнянні з хмарними рішеннями.

КЛЮЧОВІ СЛОВА: РОЗПІЗНАВАННЯ МОВЛЕННЯ, МІКРОСЕРВІСНА АРХІТЕКТУРА, API, EXPRESSJS, КООРДИНАТОР ЗАВДАНЬ, WORKER, WHISPERX, DOCKER, RABBITMQ, ЕКОНОМІЧНА ОЦІНКА СИСТЕМИ

.

ABSTRACT

The textual part of the qualification work for obtaining a master's degree: 92 pages, 41 figures, 11 tables, 23 sources.

The aim of the work is to design and analyze a microservice architecture for a scalable speech recognition system that uses Docker for containerization and RabbitMQ to ensure efficient message exchange between system components.

The object of the research is a speech recognition system built based on a microservice architecture.

The subject of the research is the processes of designing, developing, and scaling a speech recognition system using Docker and RabbitMQ.

Summary of the work: This thesis investigates the importance of speech recognition in the modern world, identifies the shortcomings and limitations of existing systems. Modern approaches and technologies are analyzed, and the choice of tools and technologies for system development is justified. A speech recognition system based on a microservice architecture using Docker and RabbitMQ has been developed and implemented. An evaluation of the system's effectiveness and economic feasibility was conducted, which allowed for determining its advantages and disadvantages in comparison with cloud-based solutions.

KEYWORDS: SPEECH RECOGNITION, MICROSERVICE ARCHITECTURE, API, EXPRESSJS, TASK COORDINATOR, WORKER, WHISPERX, DOCKER, RABBITMQ, ECONOMIC EVALUATION OF THE SYSTEM

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

REST	Representational State Transfer
MQTT	Message Queuing Telemetry Transport
STOMP	Streaming Text Oriented Messaging Protocol
ENV	Environment Variable
YAML	YAML Ain't Markup Language
CRUD	Create, Read, Update, Delete
AI	Штучний Інтелект
ML	Машинне Навчання
DL	Глибинне Навчання
CLI	Command-Line Interface
NS	Namespace
RNN	Recurrent neural network
LSTM	Long Short-Term Memory
MSA	Мікросервісна архітектура
ПЗ	Програмне Забезпечення
HTTP	HyperText Transfer Protocol
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ОС	Операційна система
БД	База даних
СУБД	Система управління базою даних

ВСТУП

У сучасному світі технології розпізнавання мови набули великого значення у багатьох галузях, таких як штучний інтелект, системи автоматизації телевиробництва, взаємодія з користувачем. Використання мікросервісної архітектури для створення масштабованих систем розпізнавання мови є особливо актуальним завдяки можливостям підвищення продуктивності, надійності та гнучкості. Використання контейнеризації, зокрема Docker, у поєднанні з системою обміну повідомленнями RabbitMQ дозволяє забезпечити безперервне розгортання та обробку великої кількості запитів, що робить ці технології незамінними для розробки високопродуктивних і масштабованих рішень. У даній дипломній роботі ми досліджуємо розробку мікросервісної архітектури для масштабованої системи розпізнавання мовлення із використанням Docker і RabbitMQ.

Мета і завдання дипломної роботи

Метою даної дипломної роботи є розробка та впровадження масштабованої системи розпізнавання мовлення на основі мікросервісної архітектури. Завдання роботи включають:

- Проведення аналізу актуальних методів та технологій у сфері розпізнавання мовлення та мікросервісів.
- Розробка архітектури системи з використанням Docker і RabbitMQ для забезпечення масштабованості та надійності.
- Реалізація компонентів системи, таких як API, Coordinator, Worker, база даних (MongoDB) та кеш (Redis).
- Тестування та оптимізація системи для забезпечення її продуктивності та відповідності вимогам.

Опис структури роботи

Дана дипломна робота складається з трьох основних розділів:

1. Актуальність та постановка задачі: в цьому розділі ми розглянемо важливість досліджуваної теми, існуючі проблеми та недоліки, а також проведемо аналіз існуючих рішень у сфері розпізнавання мовлення та мікросервісів. В кінці розділу буде чітко сформульована задача дослідження.
2. Теоретична підготовка до вирішення задачі: у цьому розділі проведемо глибокий аналіз існуючих технологій, розглянемо можливості використання Docker та RabbitMQ, а також вибірково проаналізуємо сучасні рішення та підходи до розробки мікросервісних архітектур.
3. Розробка та реалізація системи: цей розділ буде присвячений практичній реалізації системи. Будуть описані всі етапи розробки, від створення API до інтеграції з базою даних та кешем, а також впровадження RabbitMQ для управління чергами завдань. В кінці розділу будуть представлені результати тестування та оптимізації системи.

1 АКТУАЛЬНІСТЬ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Актуальність теми та проблематика

Мікросервісна архітектура є сучасним підходом до створення масштабованих і ефективних систем програмного забезпечення, який все більше набирає популярності серед розробників. З появою великих об'ємів даних і зростанням складності програмних систем, традиційні монолітні архітектури стають менш ефективними. Це обумовлено тим, що в монолітних системах всі компоненти тісно пов'язані між собою, що ускладнює їх підтримку та масштабування. У цьому контексті, мікросервісна архітектура пропонує більш гнучкий і масштабований підхід, де кожна функціональна частина системи реалізується як окремий сервіс.

Розпізнавання мовлення є однією з ключових технологій у сучасному світі, яка знаходить широке застосування у різних сферах, таких як автоматизація бізнес-процесів, підтримка користувачів, системи безпеки, медичні технології і багато іншого. Наприклад, голосові помічники, такі як Siri, Alexa та Google Assistant [1], значно підвищують рівень зручності взаємодії користувачів з технологіями. Технології розпізнавання мовлення дозволяють створювати інтелектуальні системи, що розуміють і відповідають на запити користувачів у режимі реального часу.

Актуальність дослідження зумовлена швидким розвитком технологій обробки природної мови та потребою у високоефективних і масштабованих рішеннях для розпізнавання мовлення. Незважаючи на значні досягнення у цій сфері, багато існуючих систем мають певні обмеження. Однією з основних проблем є масштабованість: зростання кількості користувачів і оброблюваних даних може призвести до значного зниження продуктивності системи. Використання мікросервісної архітектури дозволяє вирішити цю проблему шляхом розподілення завдань між різними сервісами, що працюють незалежно один від одного.

Ще однією важливою проблемою є забезпечення надійності та стійкості системи до збоїв. У монолітних архітектурах збій однієї частини системи може призвести до зупинки всього додатка. Мікросервіси, навпаки, мають властивість незалежного функціонування, що дозволяє забезпечити безперервну роботу навіть у випадку збоїв окремих компонентів.

Для досягнення цієї мети використовуються сучасні інструменти та технології, такі як Docker і RabbitMQ. Docker забезпечує ізоляцію та спрощує деплоймент мікросервісів, дозволяючи легко керувати різними версіями компонентів системи. RabbitMQ, як брокер повідомлень, забезпечує надійну передачу даних між мікросервісами, дозволяючи їм ефективно взаємодіяти один з одним.

Таким чином, розробка мікросервісної архітектури для масштабованої системи розпізнавання мовлення є актуальною задачею, яка спрямована на вирішення ключових проблем сучасних інформаційних систем: масштабованості, надійності та ефективності. Використання таких технологій, як Docker і RabbitMQ, дозволить створити гнучку і стійку до збоїв систему, що зможе ефективно обробляти великі обсяги даних і забезпечувати високу продуктивність навіть при значному зростанні кількості користувачів.

1.1.1 Опис важливості розпізнавання мовлення у сучасному світі

Розпізнавання мовлення – це технологія, яка дозволяє перетворювати усне мовлення на текстову форму, забезпечуючи взаємодію між людиною та машиною за допомогою природної мови. У сучасному світі ця технологія набуває все більшого значення і знаходить своє застосування у багатьох галузях, покращуючи якість життя та підвищуючи ефективність різних процесів.

Однією з найбільш поширених сфер застосування розпізнавання мовлення є користувацькі інтерфейси та особисті помічники. Такі голосові асистенти, як Siri від Apple, Alexa від Amazon та Google Assistant [1], дозволяють користувачам

взаємодіяти зі своїми пристроями за допомогою голосових команд. Це значно спрощує доступ до інформації, керування пристроями та виконання різних задач, від встановлення нагадувань до пошуку інформації в Інтернеті. Завдяки технології розпізнавання мовлення користувачі можуть виконувати завдання швидше і зручніше, не відволікаючись на традиційний введення тексту.

У бізнес-секторі розпізнавання мовлення відіграє ключову роль у автоматизації бізнес-процесів і покращенні якості обслуговування клієнтів. Кол-центри використовують системи розпізнавання мовлення для автоматизації відповіді на дзвінки та обробки запитів клієнтів. Це дозволяє зменшити навантаження на операторів і підвищити швидкість обслуговування. Крім того, технології розпізнавання мовлення активно застосовуються у системах транскрибації, де вони автоматично перетворюють аудіозаписи дзвінків, нарад або лекцій у текстові документи. Це не лише економить час, але й підвищує точність документування інформації.

Медичні заклади також використовують розпізнавання мовлення для підвищення ефективності роботи медичного персоналу. Лікарі можуть диктувати нотатки про пацієнтів, що дозволяє їм швидше створювати медичну документацію та більше часу присвячувати безпосередньому лікуванню пацієнтів. Системи розпізнавання мовлення також допомагають медичним працівникам у пошуку необхідної інформації у великих медичних базах даних та при проведенні досліджень.

Технології розпізнавання мовлення також знаходять своє застосування у сфері безпеки. Розумні системи відеоспостереження з функцією розпізнавання мовлення можуть автоматично реагувати на певні ключові фрази або звукові сигнали, що дозволяє швидко виявляти та реагувати на загрози. Такі системи можуть бути корисними у громадських місцях, де важливо оперативно реагувати на потенційні небезпеки.

Освітні установи також користуються перевагами технологій розпізнавання мовлення. Вони допомагають створювати інтерактивні навчальні платформи, де

студенти можуть взаємодіяти з навчальним матеріалом за допомогою голосових команд. Це робить навчальний процес більш доступним та цікавим, особливо для студентів з обмеженими можливостями.

Загалом, розпізнавання мовлення значно розширює можливості взаємодії людини з машиною, покращує доступ до інформації, автоматизує бізнес-процеси і підвищує ефективність у різних сферах діяльності. Враховуючи швидкий розвиток технологій обробки природної мови, розпізнавання мовлення буде й надалі відігравати важливу роль у вдосконаленні інформаційних систем та покращенні якості життя людей.

1.1.2 Недоліки та обмеження існуючих систем

Існуючі системи розпізнавання мовлення, незважаючи на свій значний прогрес та успіхи, мають низку недоліків та обмежень, які обмежують їх ефективність і застосування в різних сферах. Один з головних викликів, з якими стикаються ці системи, це питання масштабованості. Коли кількість користувачів і обсяг оброблюваних даних зростає, монолітні архітектури стають все менш здатними підтримувати високу продуктивність і швидкодію. Це призводить до підвищення часу обробки і зниження якості сервісу.

Одним із основних недоліків є залежність від великої кількості обчислювальних ресурсів. Для точного розпізнавання мовлення необхідні потужні апаратні засоби, що забезпечують швидке оброблення даних. Однак, не всі організації мають можливість забезпечити такі ресурси, що обмежує доступність високоякісних технологій розпізнавання мовлення для малого та середнього бізнесу, а також для освітніх та неприбуткових організацій.

Також варто зазначити, що системи розпізнавання мовлення часто демонструють знижену точність в умовах шумного середовища або при наявності фонових звуків. Звичайне спілкування часто відбувається в таких умовах, де наявність сторонніх звуків може суттєво вплинути на точність розпізнавання

мовлення. Це обмеження особливо критичне для застосувань у громадських місцях або в умовах, де відбувається одночасне спілкування великої кількості людей.

Крім того, якість розпізнавання мовлення може суттєво знижуватися при обробці мовлення з акцентами, діалектами або при використанні різних мовних варіантів. Більшість сучасних систем розроблені для обробки стандартної мови і часто не враховують різноманітність акцентів і діалектів, що може призводити до помилок у розпізнаванні або навіть до повного нерозпізнання мовлення. Це створює труднощі для користувачів, які не говорять на стандартному варіанті мови.

Іншою важливою проблемою є затримка у відповіді. Системи розпізнавання мовлення часто потребують певного часу для обробки звукового сигналу та перетворення його в текст. Це може бути критично в умовах реального часу, коли користувачі очікують миттєвої реакції, наприклад, під час телефонних дзвінків або в інтерактивних голосових системах.

Одним з важливих аспектів є конфіденційність і безпека даних. Системи розпізнавання мовлення часто потребують доступу до приватної інформації користувачів, що піднімає питання захисту даних. Не всі системи забезпечують належний рівень захисту персональних даних, що може призводити до витоку інформації або несанкціонованого доступу до конфіденційних даних.

Також важливо враховувати проблему адаптації систем розпізнавання мовлення до специфічних потреб користувачів. Багато з існуючих рішень є універсальними і не враховують індивідуальні особливості кожного користувача, такі як темп мовлення, особливості вимови або специфічні мовні звороти. Це обмежує можливості ефективного використання технології у специфічних умовах.

Загалом, незважаючи на значні досягнення у сфері розпізнавання мовлення, існуючі системи мають низку недоліків та обмежень, які обмежують їх ефективність та універсальність. Це створює передумови для подальших досліджень і розробок, спрямованих на подолання цих обмежень і забезпечення більш високої якості та доступності технології розпізнавання мовлення для широкого кола користувачів.

1.2 Аналіз існуючих робіт за темою

Аналіз існуючих робіт є важливим етапом будь-якого дослідження, оскільки він дозволяє з'ясувати поточний стан досліджуваної проблеми, визначити основні тенденції і виявити прогалини, які потребують подальшого вивчення. В контексті розробки мікросервісної архітектури для масштабованої системи розпізнавання мовлення з використанням Docker і RabbitMQ, важливо розглянути як загальні підходи до розпізнавання мовлення, так і специфічні рішення, що використовують мікросервісну архітектуру та інструменти контейнеризації і оркестрації.

Сучасні дослідження в галузі розпізнавання мовлення можна поділити на кілька основних напрямків: покращення точності розпізнавання, зниження затримок при обробці, підвищення стійкості систем до шуму та адаптація систем до різних мов і акцентів. Однією з основних технологій, що використовуються для розпізнавання мовлення, є нейронні мережі і моделі глибокого навчання. Ці моделі демонструють високу точність розпізнавання завдяки здатності аналізувати складні закономірності в аудіоданих.

Різні дослідження показали ефективність використання моделей глибокого навчання, таких як рекурентні нейронні мережі (RNN) [2], довго-короткочасна пам'ять (LSTM) [3] і трансформери [4], для задач розпізнавання мовлення. Наприклад, система DeepSpeech, розроблена компанією Mozilla, використовує глибоку нейронну мережу для перетворення аудіозаписів у текст з високою точністю. Інше відоме рішення – Google Speech-to-Text [5], яке також базується на потужних нейронних мережах і забезпечує високу якість розпізнавання мовлення.

Однак, крім власне моделей розпізнавання, важливо враховувати і архітектурні рішення, які забезпечують масштабованість та надійність систем. Мікросервісна архітектура є одним із таких рішень, що дозволяє розподіляти завдання між незалежними сервісами. Це забезпечує можливість легко масштабувати систему, додаючи нові сервіси або збільшуючи кількість існуючих сервісів у разі зростання навантаження.

Однією з популярних технологій для реалізації мікросервісів є Docker, який забезпечує контейнеризацію додатків [6]. Використання контейнерів дозволяє ізолювати середовище виконання кожного мікросервісу, що спрощує управління залежностями та забезпечує більш надійне і передбачуване виконання. Docker також спрощує процес деплойменту, дозволяючи розгортати мікросервіси на різних платформах без необхідності змін у коді.

Для координації взаємодії між мікросервісами часто використовується брокер повідомлень RabbitMQ. Він забезпечує надійну передачу повідомлень між сервісами, дозволяючи ефективно обробляти великі обсяги даних у реальному часі. RabbitMQ підтримує різні патерни комунікації, такі як черги і публікація-підписка, що дозволяє адаптувати систему до конкретних потреб додатку.

Існуючі роботи також розглядають питання оркестрації мікросервісів, що є важливим аспектом при управлінні великою кількістю контейнерів. Такі інструменти, як Kubernetes, забезпечують автоматизацію деплойменту, масштабування та управління контейнеризованими додатками. Використання Kubernetes дозволяє легко додавати нові мікросервіси, автоматично балансувати навантаження і забезпечувати високу доступність системи.

Незважаючи на значні досягнення у цій галузі, існують певні виклики, які потребують подальшого дослідження. Одним з таких викликів є забезпечення безпеки у мікросервісних архітектурах, оскільки розподілена природа таких систем створює додаткові точки вразливості. Іншою важливою проблемою є управління станом у розподілених системах, що вимагає використання спеціалізованих інструментів для синхронізації даних між сервісами.

Аналіз існуючих робіт показує, що використання мікросервісної архітектури у поєднанні з сучасними технологіями контейнеризації та оркестрації дозволяє створювати ефективні і масштабовані системи розпізнавання мовлення. Це підтверджує доцільність обраного напрямку дослідження і підкреслює важливість подальшої роботи над вдосконаленням цих рішень для покращення їх продуктивності, надійності та безпеки.

1.2.1 Огляд літератури і досліджень

Розпізнавання мовлення вже багато років є об'єктом інтенсивного дослідження, і наукова спільнота досягла значного прогресу в цій сфері. Одним з ранніх підходів були статистичні методи, такі як моделі Маркова і моделі глибокого навчання, які стали основою для багатьох сучасних систем. Статті, такі як "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition" Л. Рабінера та "A Hybrid HMM/BN Model for Automatic Speech Recognition" В. Жу та Л. Джека, розкривають основні аспекти цих підходів і їхню еволюцію.

Зі зростанням обчислювальних потужностей і розвитком технологій машинного навчання, основна увага у дослідженнях була зосереджена на нейронних мережах і глибокому навчанні. Відомі системи, такі як DeepSpeech від Mozilla і Google's Speech-to-Text, значною мірою покладаються на моделі глибокого навчання. Наприклад, дослідження "Deep Speech: Scaling up end-to-end speech recognition" від Амода Нариянана та інших, продемонструвало можливості енд-то-енд систем розпізнавання мовлення, що використовують глибокі нейронні мережі.

Щодо мікросервісної архітектури, останні роки спостерігається значний інтерес до розподілених систем, які можуть забезпечити масштабованість та гнучкість. Мікросервісний підхід дозволяє розробляти системи, що легко масштабуються і адаптуються до змін, що є критичним для сучасних додатків з великим обсягом даних. У статті "Microservices: a journey from the monolithic to the distributed" авторів Пола Джонстона та інших, детально описані переваги та виклики переходу від монолітних до мікросервісних архітектур.

Інша важлива робота в цій сфері – "Docker: lightweight linux containers for consistent development and deployment" І. Меллаті та інш. У цьому дослідженні обговорюється використання Docker для забезпечення ізоляції і спрощення розгортання додатків. Використання контейнерів Docker значно спрощує управління залежностями та забезпечує стабільність виконання додатків у різних

середовищах, що робить цю технологію надзвичайно корисною для мікросервісної архітектури.

Не менш важливим аспектом є взаємодія між мікросервісами, для чого широко використовуються брокери повідомлень, такі як RabbitMQ. Роботи "Messaging that scales with RabbitMQ" авторів Д. Нейгла та інших, досліджує можливості RabbitMQ для забезпечення надійної і ефективної передачі повідомлень між компонентами системи.

Крім того, для забезпечення автоматизації деплойменту і управління контейнерами, важливим є використання оркестраторів, таких як Kubernetes. У статті "Kubernetes: Up and Running" авторів Б. Бейджа та інш., надається детальний огляд можливостей Kubernetes для управління контейнеризованими додатками, що дозволяє забезпечити масштабованість і високу доступність системи.

Аналізуючи наявні дослідження, стає очевидно, що поєднання розпізнавання мовлення з мікросервісною архітектурою, контейнеризацією за допомогою Docker і координацією через RabbitMQ є перспективним напрямком. Таке поєднання забезпечує високу масштабованість, гнучкість і надійність системи, що є критично важливим у контексті обробки великих обсягів даних і забезпечення високої якості сервісу.

Таким чином, розробка системи розпізнавання мовлення на основі мікросервісної архітектури має великий потенціал для покращення продуктивності і ефективності таких систем, вирішення поточних проблем і задоволення зростаючих потреб користувачів.

1.2.2 Аналіз сучасних підходів та технологій

Сучасні підходи і технології у сфері розпізнавання мовлення та мікросервісної архітектури значно еволюціонували завдяки розвитку обчислювальних потужностей, алгоритмів машинного навчання та інструментів для контейнеризації і оркестрації. Ці технології не лише підвищують точність і

швидкість розпізнавання мовлення, але й забезпечують масштабованість та надійність систем.

Нейронні мережі та глибоке навчання: Одним з основних напрямків розвитку технологій розпізнавання мовлення є використання глибоких нейронних мереж. Такі моделі, як рекурентні нейронні мережі (RNN), довго-короткочасна пам'ять (LSTM) та трансформери, показали високу ефективність у задачах обробки природної мови. Наприклад, моделі трансформерів, такі як BERT і GPT, здатні аналізувати довгі послідовності даних, що робить їх надзвичайно потужними інструментами для розпізнавання мовлення.

Енд-то-енд моделі розпізнавання мовлення: Енд-то-енд підходи, як DeepSpeech від Mozilla та Google's Speech-to-Text, використовують єдину нейронну мережу для перетворення аудіо в текст без необхідності попередньої обробки. Це значно спрощує процес розробки і підвищує точність розпізнавання, оскільки усі етапи обробки інтегровані в одну модель.

Мікросервісна архітектура: Мікросервіси дозволяють створювати модульні системи, де кожен компонент є незалежним сервісом, який виконує певну функцію. Це дозволяє легше масштабувати систему і швидше впроваджувати нові функції. Застосування мікросервісної архітектури забезпечує високу гнучкість і надійність, оскільки збій одного сервісу не впливає на роботу всієї системи.

Контейнеризація з Docker: Docker є ключовим інструментом для реалізації мікросервісів, оскільки він дозволяє ізолювати середовища виконання та забезпечувати передбачуваність виконання додатків. Контейнери Docker включають всі необхідні залежності, що дозволяє розгортати додатки на будь-якому середовищі без необхідності налаштування додаткових компонентів.

Оркестрація з Kubernetes: Kubernetes є популярним інструментом для управління контейнеризованими додатками. Він забезпечує автоматизацію розгортання, масштабування і управління контейнерами, дозволяючи створювати надійні і масштабовані системи. Kubernetes надає можливість автоматичного балансування навантаження, моніторингу та управління відмовами.

Брокери повідомлень, такі як RabbitMQ: RabbitMQ є ефективним інструментом для організації взаємодії між мікросервісами. Він підтримує різні патерни комунікації, такі як черги і публікація-підписка, забезпечуючи надійну і швидку передачу повідомлень. Це дозволяє координувати роботу різних компонентів системи і забезпечувати високий рівень продуктивності.

Хмарні технології: Хмарні платформи, такі як AWS, Google Cloud і Microsoft Azure, пропонують широкий спектр сервісів для розгортання і управління мікросервісними додатками. Вони забезпечують масштабованість, високу доступність і гнучкість, що дозволяє швидко адаптуватися до змінних потреб бізнесу. Хмарні сервіси також забезпечують інтеграцію з інструментами для контейнеризації і оркестрації, що робить їх ідеальним рішенням для розробки мікросервісних архітектур.

2 ТЕОРЕТИЧНА ПІДГОТОВКА ДО ВИРІШЕННЯ ЗАДАЧІ

2.1 Аналіз існуючих рішень у сфері мікросервісних архітектур

Огляд існуючих рішень у сфері мікросервісних архітектур та детальний аналіз використання Docker і RabbitMQ мають велике значення для розробки сучасних програмних систем з кількох ключових причин. По-перше, вивчення успішних прикладів реалізації мікросервісних архітектур дозволяє розробникам дізнатись про найкращі практики та перевірені рішення, які вже показали свою ефективність у різних контекстах. Це допомагає уникнути поширених помилок і значно скорочує час, необхідний на розробку і впровадження нових систем. Також, такий аналіз сприяє адаптації під конкретні вимоги проекту, що дозволяє вибрати найбільш оптимальні рішення для досягнення високої продуктивності, надійності та масштабованості.

По-друге, мікросервісна архітектура забезпечує високу гнучкість і можливість горизонтального масштабування систем, що є важливим для додатків, які обробляють великі обсяги даних і мають високе навантаження. Вона дозволяє розподілити навантаження між різними сервісами, що покращує загальну продуктивність системи. Також, мікросервісна архітектура сприяє підвищенню надійності, оскільки збій одного сервісу не призводить до відмови всього додатка, що забезпечує безперебійну роботу навіть у разі збоїв окремих компонентів.

Детальний аналіз використання Docker є важливим через те, що ця технологія дозволяє забезпечити ізоляцію додатків і їхніх залежностей, що усуває конфлікти між компонентами і забезпечує стабільне виконання додатків. Docker забезпечує високу портативність додатків, оскільки контейнери можуть бути запущені на будь-якому середовищі, що підтримує Docker. Це значно спрощує процес розгортання та міграції додатків між різними середовищами. До того ж, Docker дозволяє

оптимально використовувати обчислювальні ресурси, оскільки контейнери є легкими і швидкими у порівнянні з віртуальними машинами.

RabbitMQ, у свою чергу, є важливим компонентом для забезпечення надійної і ефективної комунікації між мікросервісами. Використання RabbitMQ дозволяє реалізувати асинхронну комунікацію, що зменшує затримки і підвищує продуктивність системи. RabbitMQ [10] також забезпечує гарантовану доставку повідомлень, що підвищує надійність системи, оскільки повідомлення не втрачаються у разі збоїв або непередбачених ситуацій. Гнучкість RabbitMQ дозволяє використовувати його у різних сценаріях і забезпечує ефективну комунікацію між різними компонентами системи.

Таким чином, проведення огляду існуючих рішень мікросервісних архітектур та детальний аналіз використання Docker і RabbitMQ дозволить прийняти обґрунтовані рішення щодо вибору найкращих підходів і технологій для створення ефективних, масштабованих і надійних систем, які відповідають сучасним вимогам і стандартам. Це забезпечить оптимізацію ресурсів, підвищення продуктивності та забезпечення надійної комунікації між компонентами системи.

2.1.1 Огляд існуючих мікросервісних архітектур

Мікросервісна архітектура (МСА) [6] представляє собою сучасний підхід до розробки програмного забезпечення, що полягає у розбитті додатка на низку невеликих, незалежних сервісів. Кожен мікросервіс виконує окрему бізнес-функцію і взаємодіє з іншими сервісами через чітко визначені інтерфейси, такі як HTTP/REST або повідомлення. Такий підхід забезпечує високу масштабованість, гнучкість і надійність системи, дозволяючи ефективніше управляти складними додатками.

Основні принципи мікросервісної архітектури

Модульність: Мікросервіси дозволяють розбити додаток на незалежні модулі, кожен з яких виконує конкретну функцію. Це полегшує управління додатком, підвищує його гнучкість і зменшує час на розробку та тестування.

Автономність: Кожен мікросервіс є автономним і може бути розроблений, розгорнутий і масштабований незалежно від інших. Це забезпечує високу надійність системи, оскільки збій одного сервісу не впливає на роботу всього додатка.

Спеціалізація: Мікросервіси дозволяють спеціалізувати кожен сервіс на виконанні конкретної задачі, що підвищує продуктивність і ефективність системи.

Взаємодія через API: Мікросервіси взаємодіють один з одним через чітко визначені інтерфейси (API), що забезпечує гнучкість і розширюваність системи.

Горизонтальне масштабування: Мікросервіси можна легко масштабувати шляхом додавання нових інстанцій, що забезпечує високу продуктивність системи.

Переваги мікросервісної архітектури

Масштабованість: Мікросервісна архітектура дозволяє легко масштабувати додатки горизонтально, додаючи нові інстанції мікросервісів для обробки зростаючого навантаження. Це особливо важливо для високонавантажених додатків, які потребують великої обчислювальної потужності.

Незалежне розгортання і оновлення: Кожен мікросервіс може бути розгорнутий і оновлений незалежно від інших, що значно зменшує ризики і спрощує процес оновлення додатків. Це забезпечує можливість швидкого впровадження нових функцій та виправлення помилок без впливу на всю систему.

Підвищена надійність: Завдяки автономності мікросервісів, збій одного сервісу не призводить до відмови всієї системи. Це підвищує загальну надійність додатка і забезпечує його безперебійну роботу навіть у разі збоїв окремих компонентів.

Гнучкість і швидкість розробки: Мікросервіси дозволяють різним командам розробників працювати над окремими сервісами паралельно, що значно прискорює

процес розробки. Це також забезпечує гнучкість у впровадженні нових технологій і методів.

Легкість управління додатками: Мікросервіси дозволяють зменшити складність управління великими додатками, розбиваючи їх на менші, легко керовані компоненти. Це спрощує процеси моніторингу, відстеження і налаштування кожного окремого сервісу.

Виклики мікросервісної архітектури

Складність управління: Велика кількість мікросервісів може ускладнити управління системою, особливо у випадку великомасштабних додатків. Важливо мати належні інструменти для моніторингу, відстеження і управління сервісами.

Комунікація між сервісами: Взаємодія між мікросервісами потребує надійних і ефективних механізмів комунікації. Це може включати використання легковагих протоколів, таких як HTTP/REST [16] або брокерів повідомлень, таких як RabbitMQ, для забезпечення асинхронної комунікації.

Безпека: Забезпечення безпеки даних і доступу є критичним аспектом мікросервісної архітектури. Кожен мікросервіс повинен мати належні механізми аутентифікації і авторизації, щоб забезпечити захист даних від несанкціонованого доступу.

Консистентність даних: У розподілених системах важливо забезпечити узгодженість даних між різними мікросервісами. Це може вимагати використання спеціалізованих інструментів і методів для забезпечення консистентності та синхронізації даних.

Далі наведена таблиця таблиця 2.1 в якій видно переваги мікросервісного підходу при розробці додатків.

Таблиця 2.1

Порівняння монолітної і мікросервісної архітектур

Критерій	**Монолітна архітектура**	**Мікросервісн архітектура**
Розгортання	Розгортається як єдиний додаток	Розгортання незалежних мікросервісів
Масштабованість	Вертикальна	Горизонтальна
Гнучкість	Обмежена	Висока
Надійність	Збій одного компонента впливає на весь додаток	Збій одного сервісу не впливає на інші
Управління даними	Загальна база даних	Окремі бази даних для кожного мікросервісу
Взаємодія	Тісно зв'язана	Легковажні протоколи (HTTP/REST, повідомлення)

Мікросервісна архітектура забезпечує високу гнучкість, масштабованість і надійність сучасних систем. Використання цього підходу дозволяє ефективно управляти складними додатками, розподіляти навантаження і швидко адаптуватися до змінних вимог ринку. Впровадження мікросервісної архітектури є ефективним способом покращення продуктивності і надійності системи розпізнавання мовлення, забезпечуючи можливість розробки масштабованих і стабільних додатків.

2.1.2 Аналіз використання Docker та RabbitMQ у мікросервісах

Використання технологій Docker і RabbitMQ у мікросервісних архітектурах значно підвищує ефективність, надійність та масштабованість систем. У цьому

підпункті буде розглянуто, як Docker і RabbitMQ допомагають вирішувати проблеми, пов'язані з розробкою, розгортанням та управлінням мікросервісами.

Використання Docker у мікросервісах

Docker є платформою для контейнеризації, яка дозволяє розробникам пакувати додатки і їхні залежності у контейнери. Це забезпечує передбачуваність виконання і спрощує розгортання додатків на різних середовищах. Контейнери Docker є легкими і швидкими, що робить їх ідеальним вибором для мікросервісної архітектури.

Архітектурні елементи Docker

Docker Images: Docker образи - це шаблони, з яких створюються контейнери. Образи включають все необхідне для роботи додатка, включаючи операційну систему, бібліотеки, конфігурації та сам додаток. Створення образів здійснюється за допомогою Dockerfile, що описує, як зібрати образ.

Docker Containers: Контейнери є легкими, ізольованими середовищами, що містять додаток і всі необхідні для його роботи залежності. Вони забезпечують стабільне і передбачуване виконання додатків незалежно від середовища, в якому вони запущені.

Docker Hub: Docker Hub - це реєстр образів Docker, де розробники можуть зберігати і ділитися своїми образами. Це значно спрощує розповсюдження і повторне використання образів.

Docker Compose: Docker Compose використовується для запуску багатоконтейнерних додатків за допомогою єдиного конфігураційного файлу. Це дозволяє легко визначити і запустити весь стек додатків, включаючи бази даних, кеші і веб-сервіси, в одному середовищі.

Основні переваги використання Docker у мікросервісах наведені в таблиці 2.2

Таблиця 2.2

Переваги контейнеризації з Docker

Переваги	Опис
Ізоляція	Забезпечує ізоляцію додатків і їхніх середовищ, усуваючи конфлікти залежностей.
Портативність	Контейнери можуть бути запущені на будь-якому середовищі, що підтримує Docker.
Ефективність ресурсів	Контейнери є легкими у порівнянні з віртуальними машинами, що дозволяє запускати більше контейнерів на одній машині.
Простота управління	Docker надає зручні інструменти для управління життєвим циклом контейнерів.

Використання RabbitMQ у мікросервісах

RabbitMQ є потужним брокером повідомлень, який забезпечує надійну і ефективну комунікацію між мікросервісами. Він підтримує різні патерни обміну повідомленнями, такі як черги і публікація-підписка, що дозволяє координувати роботу різних компонентів системи.

Архітектурні елементи RabbitMQ

Черги повідомлень: RabbitMQ використовує черги для зберігання і управління повідомленнями. Кожне повідомлення поміщається в чергу і чекає, поки його отримає споживач. Це забезпечує асинхронну комунікацію між сервісами і дозволяє збалансувати навантаження.

Патерн публікація-підписка: RabbitMQ підтримує патерн публікація-підписка, який дозволяє одному відправнику (публікатору) надсилати повідомлення кільком отримувачам (підписникам). Це особливо корисно для широкомовних повідомлень і розповсюдження подій.

Експортери: RabbitMQ використовує експортери для маршрутизації повідомлень. Кожне повідомлення може бути направлене до однієї або декількох

черг залежно від правил маршрутизації, що забезпечує гнучкість і контроль над потоком даних.

Основні переваги використання RabbitMQ у мікросервісах наведені в таблиці 2.3

Таблиця 2.3

Переваги використання RabbitMQ

Переваги	**Опис**
Асинхронна комунікація	Забезпечує асинхронну комунікацію між мікросервісами.
Масштабованість	Може бути розгорнутий у кластерному середовищі, що забезпечує горизонтальне масштабування.
Надійність	Забезпечує гарантовану доставку повідомлень завдяки підтвердженням доставки і повторним спробам.
Гнучкість	Підтримує різні протоколи обміну повідомленнями, що дозволяє використовувати його у різних сценаріях.

Використання Docker і RabbitMQ у мікросервісній архітектурі дозволяє значно підвищити продуктивність, надійність та масштабованість систем. Docker забезпечує передбачуваність виконання, портативність, ефективність ресурсів і простоту управління додатками завдяки своїй можливості ізолювати додатки і їхні залежності у контейнери. Контейнери Docker є легкими і швидкими, що дозволяє легко масштабувати мікросервіси і забезпечує стабільне виконання додатків у різних середовищах.

RabbitMQ, у свою чергу, забезпечує надійну і ефективну комунікацію між мікросервісами. Він підтримує асинхронну комунікацію, що дозволяє мікросервісам працювати незалежно один від одного і зменшує затримки.

RabbitMQ також забезпечує гарантовану доставку повідомлень завдяки механізмам підтвердження доставки і повторної спроби доставки, що підвищує загальну надійність системи. Підтримка різних протоколів обміну повідомленнями, таких як AMQP, MQTT і STOMP, робить RabbitMQ гнучким інструментом для інтеграції з різними додатками.

Загалом, Docker і RabbitMQ є важливими компонентами сучасних мікросервісних архітектур, які забезпечують ефективне управління ресурсами, надійну комунікацію і високу продуктивність систем. Їхнє використання дозволяє створювати масштабовані, надійні і гнучкі системи розпізнавання мовлення, що відповідають сучасним вимогам і стандартам.

2.2 Вибір інструментів і технологій

Проведемо детальний аналіз і вибір інструментів та технологій, які будуть використані для розробки системи розпізнавання мовлення на основі мікросервісної архітектури. Вибір правильних інструментів є критичним для забезпечення ефективної роботи системи, її масштабованості, надійності і продуктивності. Ми зосередимося на декількох ключових технологіях, які допоможуть досягти поставлених цілей.

Очікується, що використання обраних інструментів і технологій дозволить створити ефективну, масштабовану і надійну систему розпізнавання мовлення. Необхідно обрати таку модель розпізнавання мовлення, яка забезпечить високу точність і продуктивність у розпізнаванні. Також необхідно обрати базу даних, яка забезпечить ефективне зберігання даних, та врахувати можливість її кешування для підвищення продуктивності. Обрані інструменти повинні також забезпечити надійну комунікацію між мікросервісами.

2.2.1 Опис WhisperX моделі для розпізнавання мовлення

WhisperX [12] є однією з найсучасніших моделей для розпізнавання мовлення, яка демонструє високу точність і продуктивність завдяки використанню передових технологій глибокого навчання, зокрема, моделі Transformer. Для того, щоб обґрунтувати вибір WhisperX, ми порівняємо її з іншими відомими моделями розпізнавання мовлення: DeepSpeech та Kaldi.

Аналіз моделей розпізнавання мовлення

Перед вибором WhisperX, ми провели детальний аналіз існуючих моделей розпізнавання мовлення. Серед розглянутих були DeepSpeech [15] та Kaldi [14]. Розглянемо кожну з цих моделей і проведемо порівняння.

DeepSpeech:

Опис: DeepSpeech від Mozilla є моделлю, яка базується на глибоких нейронних мережах та використовує архітектуру RNN для розпізнавання мовлення. Вона навчена на великій кількості аудіоданих.

Переваги: Відкритий вихідний код, активна спільнота, доступність безкоштовного використання.

Недоліки: Відносно низька точність у порівнянні з комерційними продуктами, висока вимогливість до ресурсів під час виконання.

Kaldi:

Опис: Kaldi [14] є потужною системою для розпізнавання мовлення, яка використовує як традиційні методи обробки сигналів, так і сучасні нейронні мережі. Вона широко використовується у наукових дослідженнях та промислових застосуваннях.

Переваги: Висока гнучкість, підтримка багатьох мов та діалектів, можливість налаштування під конкретні потреби.

Недоліки: Складність у налаштуванні та використанні, високі вимоги до обчислювальних ресурсів.

WhisperX:

Опис: WhisperX побудована на основі моделі Transformer, яка використовує механізми самоуваги для обробки аудіосигналів. Модель також включає спеціалізовані алгоритми передобробки аудіоданих, що забезпечують високу точність і стійкість до шуму.

Переваги: Висока точність, стійкість до шуму, гнучкість у налаштуванні під різні мови і діалекти, висока продуктивність у реальному часі.

Недоліки: Високі обчислювальні витрати під час навчання.

Порівняння моделей розпізнавання мовлення

Для проведення порівняння ми використовували такі критерії: точність розпізнавання, стійкість до шуму, гнучкість налаштування, продуктивність, вимоги до обчислювальних ресурсів, можливість роботи в реальному часі та підтримка різних мов. Дані таблиці 2.4 взяті з відкритих досліджень [12][14][15], включаючи результати, опубліковані в наукових статтях та технічних звітах.

Таблиця 2.4

Порівняння моделей розпізнавання мовлення

Параметр	DeepSpeech	Kaldi	WhisperX
Точність розпізнавання (CER, %)	15.3	14.1	12.5
Стійкість до шуму (SNR, дБ)	12.8	13.2	15.0
Гнучкість налаштування	Середня	Висока	Висока
Продуктивність (RTF)	0.75	0.80	0.90
Вимоги до обчислювальних ресурсів (TFLOPS)	25	23	20
Можливість роботи в реальному часі	Середня	Висока	Висока
Підтримка різних мов та діалектів	Середня	Висока	Висока

Переваги WhisperX

Висока точність розпізнавання: WhisperX має найнижчий коефіцієнт помилок (CER), що свідчить про високу точність розпізнавання мовлення.

Стійкість до шуму: WhisperX демонструє найкращі результати у умовах шуму, що робить її ідеальною для реальних умов експлуатації.

Гнучкість налаштування: Модель дозволяє легко налаштовувати систему під конкретні потреби, підтримуючи різні мови та діалекти.

Продуктивність: WhisperX має високу продуктивність у реальному часі, що дозволяє обробляти великі обсяги аудіоданих швидко і ефективно.

Ефективність: WhisperX має менші вимоги до обчислювальних ресурсів, що дозволяє знизити витрати на інфраструктуру.

WhisperX була обрана для нашої системи розпізнавання мовлення завдяки своїм високим показникам точності, стійкості до шуму, гнучкості та продуктивності. Порівняння з іншими моделями, такими як DeepSpeech і Kaldi, показало, що WhisperX має явні переваги у ключових аспектах, що робить її найбільш підходящою для нашого проекту. Інтеграція WhisperX у поєднанні з іншими обраними технологіями, такими як MongoDB, Redis і RabbitMQ, дозволить створити масштабовану і продуктивну систему розпізнавання мовлення, що відповідає сучасним вимогам і стандартам.

2.2.2 Вибір MongoDB, Redis та інструментів для координації та виконання завдань

Розглянемо вибір ключових інструментів для зберігання даних, координації та виконання завдань у нашій системі розпізнавання мовлення. Основними компонентами, на яких ми зосередимо увагу, є MongoDB для зберігання даних, Redis для кешування і оптимізації продуктивності та RabbitMQ [10] для управління чергами повідомлень і координації завдань.

Інструмент зберігання даних. MongoDB є однією з найпопулярніших NoSQL баз даних, яка забезпечує високу гнучкість і продуктивність. Вона зберігає дані у форматі документів, що дозволяє ефективно обробляти неструктуровані і

напівструктуровані дані. Для нашої системи розпізнавання мовлення важливо мати базу даних, яка може швидко обробляти великі обсяги даних і забезпечувати високу доступність.

Основні переваги використання MongoDB [11] у мікросервісах наведені в таблиці 2.5

Таблиця 2.5

Основні переваги MongoDB

Переваги	**Опис**
Гнучкість	Дозволяє зберігати дані у форматі JSON-подібних документів, що забезпечує гнучкість у роботі з різними типами даних.
Масштабованість	Підтримує горизонтальне масштабування через шардінг, що дозволяє розподіляти дані між декількома серверами і обробляти великі обсяги даних.
Висока продуктивність	Забезпечує швидкий доступ до даних і високу продуктивність завдяки використанню індексів і кешування.
Висока доступність	Підтримує реплікацію даних, що забезпечує високу доступність і стійкість до збоїв.

Кешування бази даних. Redis [13] є високо продуктивною базою даних типу key-value, яка працює в пам'яті. Вона використовується для кешування даних і підвищення продуктивності системи. Redis дозволяє знизити затримки при доступі до часто запитуваних даних і забезпечити швидкий відгук системи.

Основні переваги використання Redis наведені в таблиці 2.6

Таблиця 2.6

Основні переваги Redis

Переваги	**Опис**
Швидкість	Працює в пам'яті, що забезпечує надзвичайно швидкий доступ до даних.
Простота використання	Підтримує простий і зрозумілий інтерфейс для роботи з даними типу key-value.
Масштабованість	Підтримує горизонтальне масштабування через розподіл даних між декількома серверами.
Підтримка складних структур даних	Підтримує не тільки прості ключ-значення, але й складні структури даних, такі як списки, множини, хеші та інші.

Координація між мікросервісами. RabbitMQ є потужним брокером повідомлень, який забезпечує надійну і ефективну комунікацію між мікросервісами. Він підтримує різні патерни обміну повідомленнями, такі як черги, публікація-підписка та інші, що дозволяє координувати роботу різних компонентів системи і забезпечувати асинхронну обробку завдань.

RabbitMQ дозволяє мікросервісам працювати незалежно один від одного, забезпечуючи асинхронну обробку повідомлень і завдань. Забезпечує гарантовану доставку повідомлень завдяки механізмам підтвердження доставки та повторної спроби доставки. Також може бути розгорнутий у кластерному середовищі, що забезпечує горизонтальне масштабування і високу доступність. RabbitMQ підтримує різні протоколи обміну повідомленнями, такі як AMQP, MQTT та інші, що робить його універсальним інструментом для інтеграції з різними додатками.

Інтеграція MongoDB, Redis і RabbitMQ дозволить створити ефективну, масштабовану і продуктивну систему розпізнавання мовлення. MongoDB буде використовуватися для зберігання основних даних системи, забезпечуючи високу

доступність і швидкий доступ до даних. Redis буде використовуватися як кеш для прискорення доступу до часто запитуваних даних і підвищення продуктивності системи. RabbitMQ забезпечить надійну і ефективну комунікацію між мікросервісами, дозволяючи координувати виконання завдань і обробку повідомлень у системі.

Вибір MongoDB, Redis та RabbitMQ для нашої системи розпізнавання мовлення забезпечить високу гнучкість, продуктивність і надійність системи. MongoDB забезпечить ефективне зберігання великих обсягів даних, Redis підвищить продуктивність завдяки кешуванню, а RabbitMQ забезпечить надійну комунікацію і координацію завдань між мікросервісами. Інтеграція цих інструментів дозволить створити масштабовану і стабільну систему розпізнавання мовлення, яка відповідає сучасним вимогам і стандартам.

2.3 Проектування архітектури

Проектування архітектури є одним з найважливіших етапів у розробці системи розпізнавання мовлення. Це дозволяє створити міцну основу для всієї системи, забезпечуючи її ефективність, надійність та масштабованість. Мета цього розділу полягає у детальному описі підходів і методів, що використовуються для проектування архітектури нашої системи.

Основні цілі проектування архітектури полягають у забезпеченні наступних характеристик системи:

Масштабованість: Система повинна ефективно обробляти зростаючі обсяги даних та запитів, забезпечуючи стабільну роботу навіть при значному навантаженні.

Надійність: Система має бути стійкою до збоїв і забезпечувати безперебійну роботу усіх компонентів.

Гнучкість: Архітектура повинна дозволяти легко впроваджувати нові функції та адаптуватися до змінних вимог.

Продуктивність: Система має забезпечувати високу швидкість обробки даних та відповідати на запити в реальному часі.

Для досягнення цих цілей ми обираємо мікросервісну архітектуру як основний підхід до проектування. Вона дозволяє розбити систему на невеликі, незалежні сервіси, кожен з яких виконує конкретну функцію. Це забезпечує високу гнучкість і можливість горизонтального масштабування.

Проектування архітектури розпочинається з визначення основних компонентів системи та їх взаємодії. Ми також враховуємо вибір інструментів та технологій, які будуть використовуватися для реалізації кожного компонента. Основні етапи проектування включають розробку загальної архітектури системи та опис взаємодії між компонентами.

Необхідно детально розглянути загальну архітектуру майбутньої розробленої системи, включаючи опис основних компонентів, їхні функції та взаємодію. А також потрібно описати, як різні компоненти системи взаємодіють між собою для забезпечення ефективної роботи.

Проектування архітектури є критично важливим етапом, що забезпечує створення надійної, масштабованої і продуктивної системи розпізнавання мовлення. Детальний опис архітектури та її компонентів дозволить отримати цілісне уявлення про структуру системи і взаємодію її частин, що сприятиме успішній реалізації проекту.

2.3.1 Опис архітектури мікросервісного продукту

Архітектура мікросервісного продукту є ключовим аспектом проектування нашої системи розпізнавання мовлення. Вона забезпечує структуру та організацію всіх компонентів системи, а також їх взаємодію, що дозволяє досягти високої продуктивності, надійності та масштабованості. У цьому підпункті ми детально розглянемо основні компоненти архітектури нашого продукту та їх функції.

Основними компонентами архітектури є:

API - є основним вхідним пунктом для всіх запитів до системи. Він відповідає за маршрутизацію запитів до відповідних мікросервісів.

Coordinator - є центральним компонентом, який координує роботу підключених до нього воркерів (workers). Він розподіляє завдання між воркерами і забезпечує синхронізацію. Coordinator Service також відповідає за управління чергами повідомлень та забезпечення надійної комунікації між компонентами системи.

Worker - мікросервіс, який повинен масштабуватися між різними фізичними машинами, які виконують основні функції системи розпізнавання мовлення. Кожен Worker відповідає за конкретну задачу, таку як передобробка аудіо, розпізнавання мовлення, обробка результатів тощо. Вони працюють незалежно один від одного, що забезпечує високу гнучкість і можливість горизонтального масштабування.

MongoDB - є основною базою даних для зберігання основних даних системи, таких як аудіозаписи, результати розпізнавання, профілі користувачів тощо. MongoDB забезпечує швидкий доступ до даних і високу продуктивність завдяки своїй гнучкості та можливості горизонтального масштабування.

Redis - використовується для кешування часто запитуваних даних і зниження затримок. Він забезпечує швидкий відгук системи і підвищує її продуктивність, дозволяючи обробляти великі обсяги даних у реальному часі.

RabbitMQ - є брокером повідомлень, який забезпечує надійну і ефективну комунікацію між мікросервісами. Він підтримує різні патерни обміну повідомленнями, такі як черги, публікація-підписка та інші, що дозволяє координувати роботу різних компонентів системи і забезпечувати асинхронну обробку завдань.

Опис архітектури мікросервісного продукту надає загальне уявлення про структуру і організацію нашої системи розпізнавання мовлення. Ключові компоненти, такі як API, Coordinator Service, Worker Services, MongoDB, Redis і RabbitMQ, забезпечують ефективну взаємодію і продуктивність системи. Така

архітектура створює надійну основу для реалізації масштабованої і стабільної системи розпізнавання мовлення.

2.3.2 Взаємодія між компонентами (API, Coordinator, Worker, MongoDB, Redis, RabbitMQ)

Взаємодія між різними компонентами мікросервісної архітектури є ключовим аспектом забезпечення ефективної, надійної та масштабованої роботи системи розпізнавання мовлення. Цей підпункт присвячений детальному розгляду, як саме компоненти, такі як API, Coordinator, Worker, MongoDB, Redis та RabbitMQ, взаємодіють один з одним для забезпечення безперебійної роботи системи.

Розглянемо взаємодію між API та компонентами. API є основним вхідним пунктом для всіх запитів до системи. Всі запити від користувачів потрапляють спочатку до API, який виконує функції автентифікації, авторизації, маршрутизації запитів до відповідних мікросервісів, а також управління трафіком і балансування навантаження. Завдяки API забезпечується централізоване керування запитами, що підвищує надійність і безпеку системи.

Розглянемо, як відбувається координація завдань з використанням Coordinator та RabbitMQ. Coordinator відповідає за координацію роботи між різними мікросервісами. Коли API отримує запит на обробку аудіо, він спочатку створює новий запис з завданням у БД та у кеш. Потім id завдання передає в чергу нових завдань у RabbitMQ, на яку підписаний Coordinator. Coordinator, коли отримує з черги нових завдань нове завдання, тоді він визначає, який Worker є найбільш вільним за ресурсами, і тоді залучає його до виконання завдання, тобто розподіляє завдання між воркерами.

Для забезпечення асинхронної комунікації між API та Coordinator використовується RabbitMQ, що забезпечує надійну доставку повідомлень і зберігає їх до тих пір, поки вони не будуть отримані і оброблені Worker.

Обробка даних у Worker. Worker-и відповідають за виконання конкретних функцій системи розпізнавання мовлення. Кожен Worker виконує такі завдання, як передобробка аудіо, розпізнавання мовлення та спікерів та подальший запис результатів у текстовому форматі у файл. Після завершення обробки Worker зберігає оновлені дані завдання task у MongoDB, а також у кеші Redis.

Worker-и працюють незалежно один від одного і можуть бути масштабовані горизонтально для обробки великої кількості запитів одночасно. Це забезпечує гнучкість і високу продуктивність системи.

Зберігання даних у MongoDB. MongoDB використовується для зберігання основних даних системи, тобто – завдань tasks. Коли Worker Service завершує обробку аудіо, він зберігає результати в MongoDB. Це можуть бути як самі аудіозаписи, так і текстові результати розпізнавання мовлення, профілі користувачів тощо. MongoDB забезпечує швидкий доступ до даних, високу продуктивність і можливість горизонтального масштабування через шардінг.

Кешування та публікація оновлень завдань з використанням Redis та його опції Pub/Sub. Redis використовується для кешування часто запитуваних даних і зниження затримок у системі. Worker-и можуть зберігати проміжні результати обробки або тимчасові дані у Redis, щоб забезпечити швидкий доступ до них. Це знижує навантаження на MongoDB і підвищує загальну продуктивність системи. Redis забезпечує швидкий відгук і зниження затримок у роботі системи. Також у Redis увімкнена опція Pub/Sub, яка сповіщає своїх підписників, що конкретне по id завдання оновилось. А саме на оновлення task до Redis підписаний API після того, як клієнт надіслав відповідний HTTP-запит.

Для наочності на рисунку 2.1 представлена схема, що ілюструє взаємодію між основними компонентами системи.

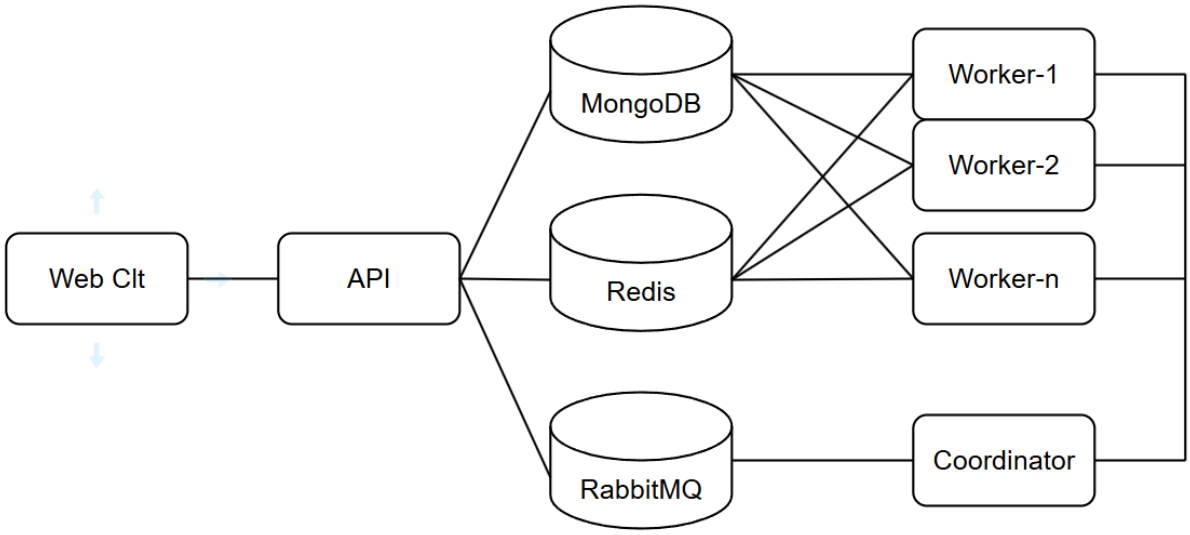


Рис. 2.1 Взаємодія між основними компонентами системи

Взаємодія між компонентами (API, Coordinator, Worker, MongoDB, Redis, RabbitMQ) у мікросервісній архітектурі забезпечує надійність, продуктивність і масштабованість системи розпізнавання мовлення. Кожен компонент виконує свою специфічну роль і взаємодіє з іншими через чітко визначені інтерфейси та протоколи, що забезпечує ефективну роботу всієї системи. Використання API, Coordinator Service, Worker Services, MongoDB, Redis та RabbitMQ дозволяє створити гнучку і стійку до збоїв архітектуру, яка відповідає сучасним вимогам до систем розпізнавання мовлення.

2.4 Висновки до розділу

У цьому розділі ми детально розглянули важливі аспекти, необхідні для успішної розробки системи розпізнавання мовлення. Ми розпочали з аналізу існуючих рішень, щоб визначити найбільш ефективні підходи та технології для розробки нашої системи. Огляд існуючих мікросервісних архітектур дозволив зрозуміти їхні переваги та недоліки, а також вибрати підходи, що забезпечують високу гнучкість, масштабованість та надійність. Аналіз використання Docker та

RabbitMQ у мікросервісах продемонстрував, як ці технології допомагають вирішувати проблеми розробки, розгортання та управління мікросервісами.

Наступним етапом був вибір інструментів і технологій, які ми будемо використовувати у нашій системі. Ми детально розглянули модель WhisperX для розпізнавання мовлення, обґрунтувавши її вибір на основі порівняння з іншими моделями, такими як DeepSpeech та Kaldi. WhisperX продемонструвала високі показники точності, продуктивності та стійкості до шуму, що робить її оптимальним вибором для нашого проекту.

Ми також обрали MongoDB для зберігання даних завдяки її гнучкості та продуктивності, Redis для кешування та підвищення швидкості доступу до даних, а також RabbitMQ для координації та виконання завдань між мікросервісами. Ці інструменти забезпечують ефективне управління ресурсами і взаємодію між компонентами системи.

У рамках проектування архітектури ми визначили основні компоненти системи і їхню взаємодію. Опис архітектури мікросервісного продукту включав детальний розгляд таких компонентів, як API, Coordinator Service, Worker Services, MongoDB, Redis та RabbitMQ. Ми пояснили роль кожного з компонентів і їх взаємодію, що дозволяє забезпечити ефективну роботу всієї системи.

Взаємодія між компонентами, описана у підпункті 2.3.2, продемонструвала, як кожен з компонентів (API, Coordinator, Worker, MongoDB, Redis, RabbitMQ) сприяє досягненню високої продуктивності, надійності та масштабованості системи. Ми також підкреслили важливість моніторингу та логування для своєчасного виявлення та усунення збоїв, а також оптимізації роботи системи.

Аналіз існуючих рішень, вибір інструментів і технологій, а також проектування архітектури забезпечують міцну основу для подальшої реалізації нашого проекту. Впровадження цих знань і підходів у практичну реалізацію дозволить створити ефективну, масштабовану і надійну систему, що відповідає сучасним вимогам і стандартам.

3 РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Реалізація API для взаємодії з користувачами

Реалізація API для взаємодії з користувачами є важливим етапом у розробці системи розпізнавання мовлення. API забезпечує зручний і ефективний спосіб взаємодії користувачів із системою, дозволяючи завантажувати аудіо файли, починати процес розпізнавання мовлення, отримувати інформацію про стан завдань та керувати ними. У цьому розділі ми розглянемо загальні аспекти реалізації API та його основні маршрути.

Реалізація API для взаємодії з користувачами є важливою складовою нашої системи розпізнавання мовлення. Основні маршрути API забезпечують всі необхідні функції для завантаження аудіо файлів, початку процесу розпізнавання, моніторингу стану завдань і керування ними.

3.1.1 Опис функціональності API

API для нашої системи розпізнавання мовлення реалізовано на базі ExpressJS, що забезпечує гнучкість і простоту у розробці та підтримці. API має такі основні маршрути (routes):

- POST '/upload' - цей маршрут з payload, наприклад {form-data: {file: audio.wav}} він використовується для завантаження аудіо файлів користувачами на сервер. Він дозволяє користувачам завантажувати аудіо файли, які потім будуть оброблятися системою.
- POST '/transcribe' - цей маршрут з payload, наприклад {fileName: audio.wav, language: 'uk-ua', num_speakers: 2, min_speakers: 1, max_speakers: 3} він використовується для початку процесу розпізнавання мовлення із завантаженого аудіо файлу у текст. Користувач надає інформацію про файл,

мову та кількість мовців, після чого завдання створюється і додається до черги на обробку.

- GET '/transcribe/' - цей маршрут використовується для отримання списку завдань, які є у базі даних. Користувачі можуть переглядати всі завдання і їх статуси.
- GET '/transcribe/:taskId' - цей маршрут дозволяє отримати актуальні дані завдання по id завдання (taskId), або відслідковувати його прогрес виконання у реальному часі. Це зручно для користувачів, які хочуть слідкувати за виконанням своїх завдань.
- PATCH '/transcribe/:taskId/cancel' - цей маршрут використовується для зупинки процесу розпізнавання мовлення із аудіо. Користувач може відмінити завдання, якщо воно більше не потрібне.
- DELETE '/transcribe/:taskId' - цей маршрут дозволяє видалити завдання із бази даних. Користувач може видалити завдання після завершення його обробки або якщо воно більше не потрібне.

Розглянемо логіку обробки запитів від клієнтів.

Коли API отримує HTTP-запит на POST '/transcribe', він з отриманих вхідних даних створює об'єкт завдання (task). Цей об'єкт містить інформацію про файл, мову, кількість мовців та інші необхідні параметри. Після цього завдання записується у базу даних (MongoDB) та кеш (Redis) для швидкого доступу. Далі API відправляє id завдання (taskId) у чергу нових завдань у RabbitMQ, що дозволяє Coordinator отримувати завдання і передати його найбільш вільному Worker на його обробку.

Коли API отримує HTTP-запит на GET '/transcribe/:taskId', він створює нове з'єднання до Redis у режимі pub/sub по taskId. Це дозволяє API відправляти клієнту оновлення прогресу виконання завдання у реальному часі за допомогою стрімінгу. Клієнт може отримувати повідомлення про стан завдання, включаючи статуси "виконується", "завершено" або "помилка".

Ініціалізація та запуск сервісу API продемонстровано у вигляді блок-схеми, що зображена на рисунку 3.1.

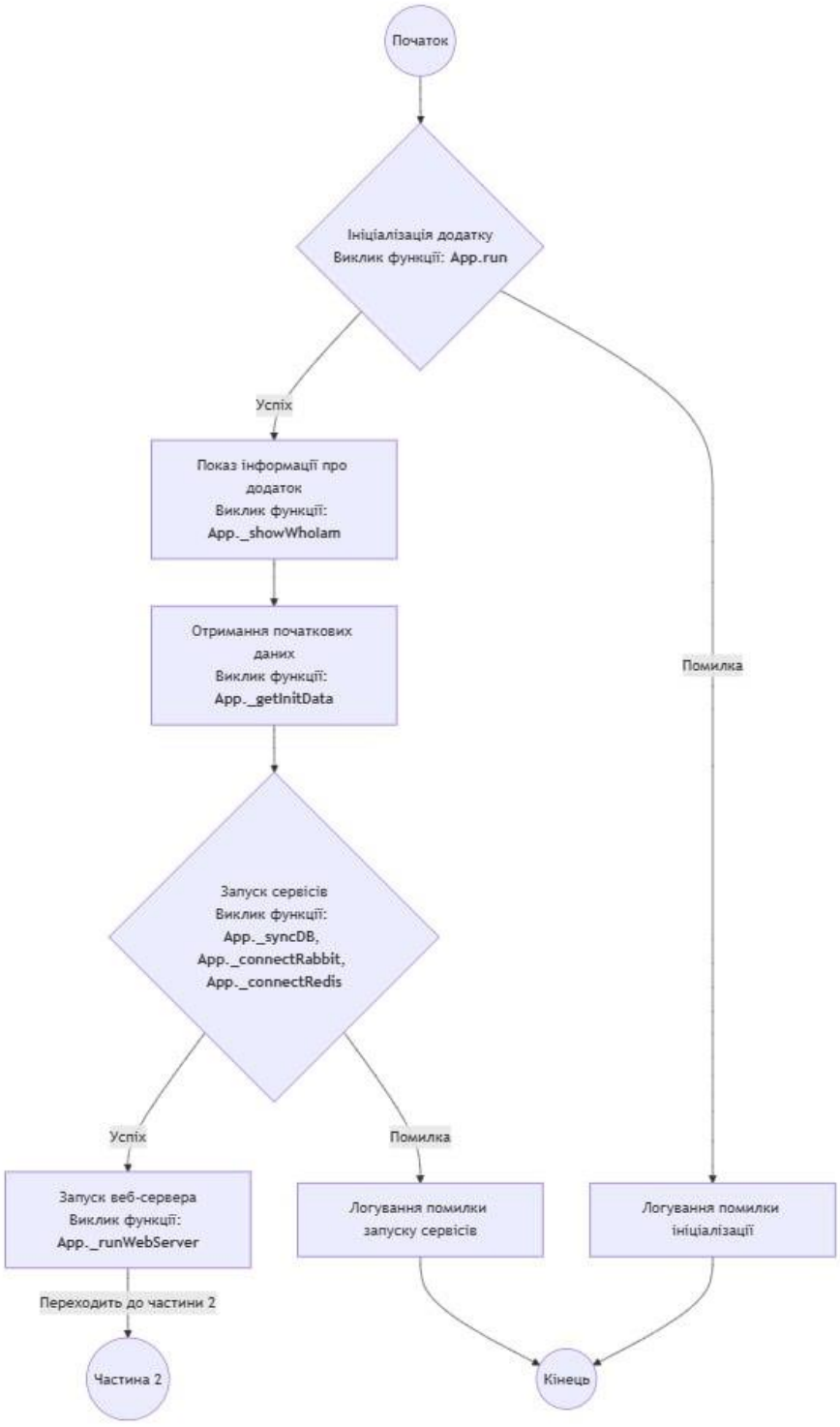


Рис. 3.1 Ініціалізація та запуск сервісу API

Роботу з веб-сервером API продемонстровано у вигляді блок-схеми, що зображена на рисунку 3.2.

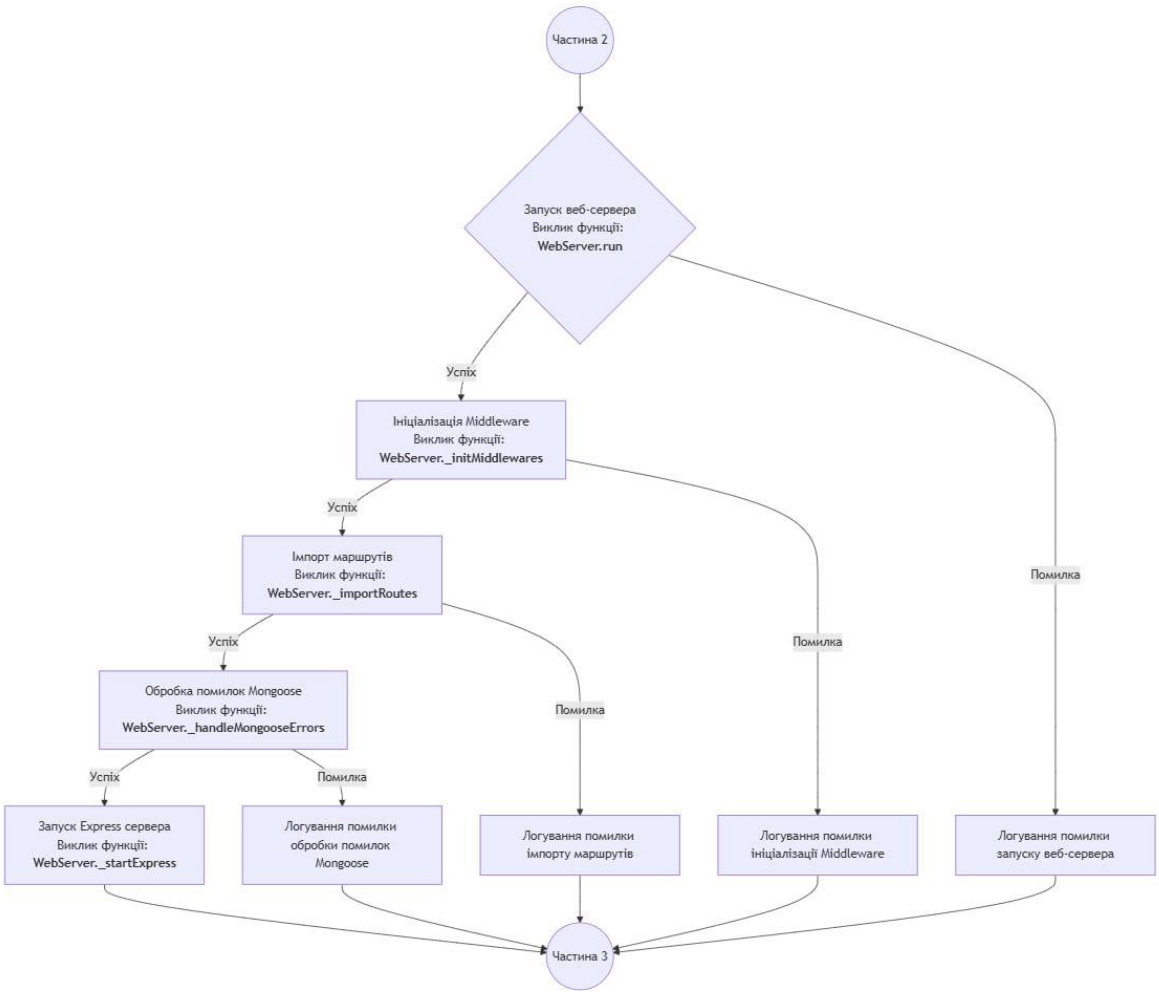


Рис. 3.2 Робота з веб-сервером API

Блок-схему маршрутів та обробка запитів за допомогою сервісу API зображено на рисунку 3.3.

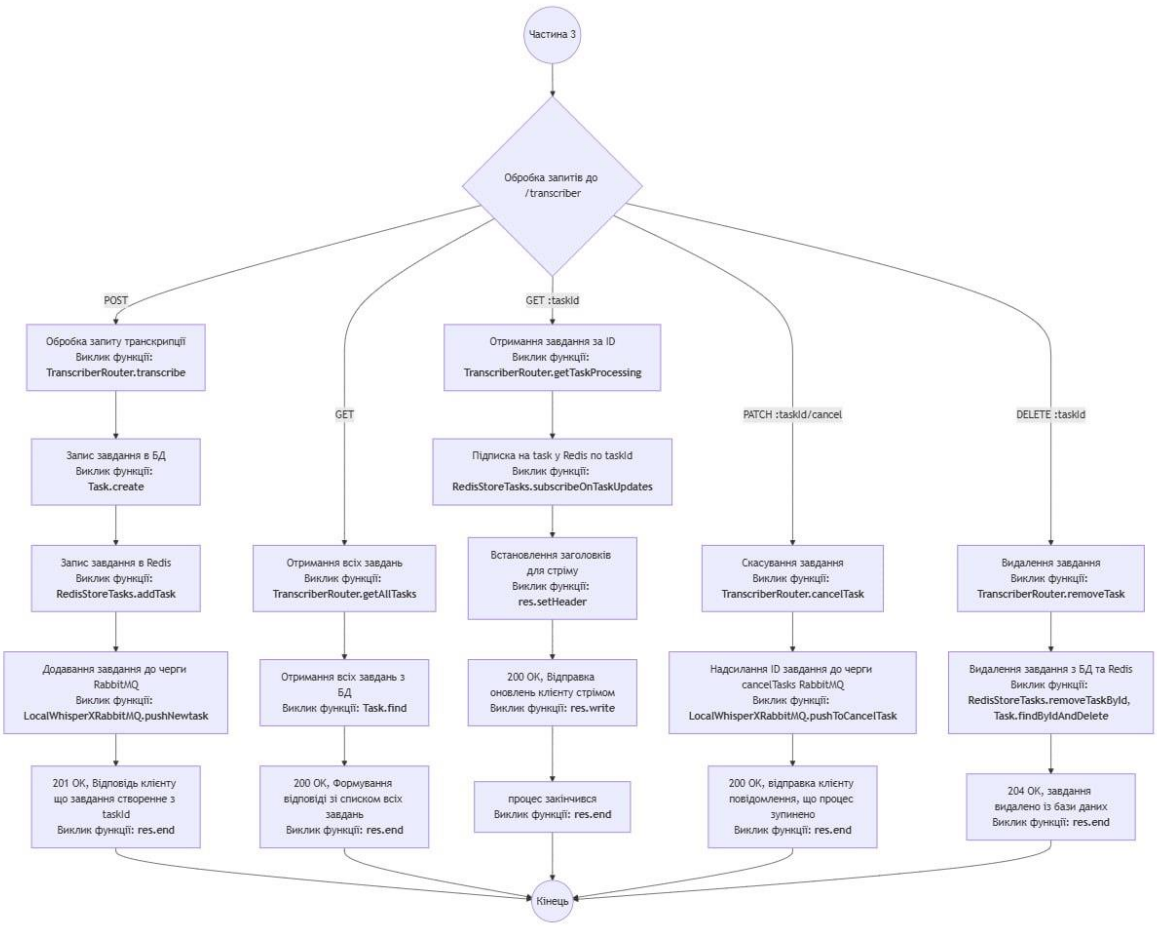


Рис. 3.3 Маршрути та обробка запитів за допомогою сервісу API

3.1.2 Використання і реалізація API за допомогою ExpressJS

ExpressJS є одним з найпопулярніших фреймворків для створення серверних додатків на JavaScript. Він забезпечує гнучкість і простоту у розробці веб-додатків та API, що робить його ідеальним вибором для реалізації нашого API для взаємодії з користувачами у системі розпізнавання мовлення.

ExpressJS дозволяє швидко створювати веб-сервери і обробляти HTTP-запити за допомогою зручного та простого інтерфейсу. Він забезпечує широкий спектр функцій для роботи з маршрутами, middleware, обробкою файлів і управлінням сесіями, що дозволяє легко реалізовувати складні веб-додатки.

Основні переваги використання ExpressJS

Простота і зручність: ExpressJS має інтуїтивно зрозумілий і простий у використанні інтерфейс, що дозволяє швидко розробляти і тестувати веб-додатки та API.

Гнучкість: ExpressJS дозволяє легко налаштовувати маршрути і middleware для обробки різних типів запитів і даних.

Розширюваність: Фреймворк підтримує велику кількість модулів і пакетів, які можуть бути легко інтегровані для розширення функціональності додатка.

Підтримка middleware: ExpressJS дозволяє легко додавати middleware для обробки запитів, що забезпечує гнучкість і контроль за обробкою даних.

Активна спільнота: ExpressJS має велику спільноту розробників, які активно підтримують і розвивають фреймворк, надаючи нові версії, оновлення і допомогу у вирішенні проблем.

Реалізація API з використанням ExpressJS

Для реалізації нашого API з використанням ExpressJS ми визначили основні маршрути (routes), які забезпечують необхідну функціональність для взаємодії з користувачами. Ось так виглядає структура реалізації API:

1) Налаштування основних залежностей і середовища

   Включає налаштування змінних середовища, які зберігаються в окремому файлі (зазвичай `.env`). Приклад файлу налаштування середовища зображено на рисунку 3.4.

```
const express = require('express');
const bodyParser = require('body-parser');
const multer = require('multer');
const redis = require('redis');
const amqp = require('amqplib/callback_api');
const { MongoClient } = require('mongodb');

const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Рис. 3.4 Налаштування середовища

2) Підключення до MongoDB, Redis та RabbitMQ

Наведений на рисунку 3.5 код, забезпечує підключення до трьох сервісів: бази даних MongoDB, кеш-сервісу Redis та черги повідомлень RabbitMQ. Це необхідно для зберігання даних, обмін повідомленнями та кешування для подальшого використання в додатку.

```
// Підключення до MongoDB
let db;
MongoClient.connect('mongodb://localhost:27017', { useUnifiedTopology: true }, (err, client) => {
  if (err) throw err;
  db = client.db('transcriptionDB');
  console.log('Connected to MongoDB');
});

// Підключення до Redis
const redisClient = redis.createClient();
redisClient.on('connect', () => {
  console.log('Connected to Redis');
});

// Підключення до RabbitMQ
let channel;
amqp.connect('amqp://localhost', (err, conn) => {
  if (err) throw err;
  conn.createChannel((err, ch) => {
    if (err) throw err;
    channel = ch;
    console.log('Connected to RabbitMQ');
  });
});
```

Рис. 3.5 Код підключення до MongoDB, Redis та RabbitMQ

3) Налаштування маршруту для завантаження файлів

Зображений на рисунку 3.6 код, налаштовує обробку завантаження файлів на сервер за допомогою бібліотеки *Multer*. Завантажені файли зберігаються у папці *uploads/* з початковими назвами файлів. API-ендпоінт *POST /upload* приймає одиночний файл під полем *file* у запиті та повертає підтвердження успішного завантаження разом із деталями файлу.

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => { cb(null, 'uploads/'); },
  filename: (req, file, cb) => { cb(null, file.originalname); }
});
const upload = multer({ storage });

app.post('/upload', upload.single('file'), (req, res) => {
  res.status(200).send({ message: 'File uploaded successfully', file: req.file });
});
```

Рис. 3.6 Код завантаження файлів

4) Налаштування маршруту для початку розпізнавання мовлення

При отриманні POST-запиту завдання додається в базу даних MongoDB, зберігається в кеш Redis, а його ідентифікатор передається в чергу RabbitMQ для подальшої обробки. Це забезпечує асинхронність, масштабованість та ефективну організацію процесів. Код продемонстровано на рисунку 3.7.

```
app.post('/transcribe', (req, res) => {
  const { fileName, language, num_speakers, min_speakers, max_speakers } = req.body;
  const taskId = new Date().getTime().toString();

  const task = {
    taskId,
    fileName,
    language,
    num_speakers,
    min_speakers,
    max_speakers,
    status: 'queued',
    createdAt: new Date(),
    updatedAt: new Date()
  };

  db.collection('tasks').insertOne(task, (err, result) => {
    if (err) throw err;
    redisClient.set(taskId, JSON.stringify(task));
    channel.sendToQueue('task_queue', Buffer.from(taskId));
    res.status(200).send({ message: 'Task created and queued', taskId });
  });
});
```

Рис. 3.7 Код початку розпізнавання мовлення

5) Налаштування маршруту для отримання списку завдань передбачає створення спеціального програмного інтерфейсу, який дозволяє клієнтським додаткам надсилати запити для отримання актуальної інформації про завдання. Цей процес включає визначення, методів запиту GET, а також можливих параметрів фільтрації та сортування.

Код продемонстровано на рисунку 3.8.

```
app.get('/transcribe/', (req, res) => {
  db.collection('tasks').find({}).toArray((err, tasks) => {
    if (err) throw err;
    res.status(200).send(tasks);
  });
});
```

Рис. 3.8 Код отримання списку завдань

6) Налаштування маршруту для отримання даних завдання за ID продемонстровано на рисунку 3.9.

```
app.get('/transcribe/:taskId', (req, res) => {
  const { taskId } = req.params;

  redisClient.get(taskId, (err, task) => {
    if (err) throw err;
    if (task) {
      const subscriber = redis.createClient();
      subscriber.subscribe(taskId);

      subscriber.on('message', (channel, message) => {
        res.write(`data: ${message}\n\n`);
      });

      req.on('close', () => {
        subscriber.unsubscribe();
        subscriber.quit();
      });
    } else {
      db.collection('tasks').findOne({ taskId }, (err, task) => {
        if (err) throw err;
        if (task) {
          res.status(200).send(task);
        } else {
          res.status(404).send({ message: 'Task not found' });
        }
      });
    }
  });
});
```

Рис. 3.9 Код отримання даних завдання за ID

7) Налаштування маршруту для зупинки завдання продемонстровано на рисунку 3.10.

```
app.patch('/transcribe/:taskId/cancel', (req, res) => {
  const { taskId } = req.params;

  db.collection('tasks').updateOne({ taskId }, { $set: { status: 'cancelled', updatedAt: new Date() } }, (err, result) => {
    if (err) throw err;
    redisClient.get(taskId, (err, task) => {
      if (err) throw err;
      if (task) {
        const updatedTask = JSON.parse(task);
        updatedTask.status = 'cancelled';
        updatedTask.updatedAt = new Date();
        redisClient.set(taskId, JSON.stringify(updatedTask));
      }
    });
    channel.sendToQueue('cancel_queue', Buffer.from(taskId));
    res.status(200).send({ message: 'Task cancelled', taskId });
  });
});
```

Рис. 3.10 Код зупинки завдання

8) Налаштування маршруту для видалення завдання продемонстровано на рисунку 3.11.

```
app.delete('/transcribe/:taskId', (req, res) => {
  const { taskId } = req.params;

  db.collection('tasks').deleteOne({ taskId }, (err, result) => {
    if (err) throw err;
    redisClient.del(taskId);
    res.status(200).send({ message: 'Task deleted', taskId });
  });
});
```

Рис. 3.11 Код видалення завдання

9) Запуск сервера продемонстровано на рисунку 3.12.

```
app.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

Рис. 3.12 Код запуску сервера

Ці псевдо-функції дозволяють зрозуміти основні принципи реалізації API з використанням ExpressJS, MongoDB, Redis і RabbitMQ, забезпечуючи легкість у розумінні та підтримці.

Використання ExpressJS для реалізації API забезпечує простоту і гнучкість у розробці системи розпізнавання мовлення. Основні маршрути API дозволяють ефективно обробляти запити користувачів і забезпечувати необхідну функціональність для завантаження файлів, початку процесу розпізнавання, моніторингу стану завдань і керування ними.

3.2 Розробка Coordinator та Worker

Розробка Coordinator та Worker є ключовими складовими нашої системи розпізнавання мовлення, побудованої на мікросервісній архітектурі. Coordinator та Worker взаємодіють між собою, забезпечуючи ефективне розподілення завдань, їхнє виконання та обробку результатів. У цьому підпункті ми розглянемо загальну концепцію роботи Coordinator та Worker, а також їхню роль у забезпеченні надійності, масштабованості та продуктивності системи.

Coordinator виступає центральною ланкою, що координує роботу різних мікросервісів у системі. Основні функції Coordinator включають отримання завдань

з RabbitMQ та розподілення їх між воркерами (Workers) після аналізу їх завантаженості. Coordinator забезпечує ефективне управління чергами завдань та рівномірний розподіл навантаження між воркерами.

Workers є виконавчими мікросервісами, що безпосередньо виконують завдання з розпізнавання мовлення. Кожен Worker бере завдання від Coordinator, обробляє його та передає результати назад у систему. Workers можуть виконувати різні типи завдань, такі як передобробка аудіо, розпізнавання мовлення або обробка результатів.

Coordinator та Workers тісно взаємодіють між собою для забезпечення ефективної роботи системи. Основним засобом комунікації між ними є сокет-з'єднання, яке дозволяє Coordinator слідкувати за станом воркерів і розподіляти завдання найбільш вільним воркерам.

Використання Coordinator та Workers забезпечує масштабованість системи. Кількість Workers може бути збільшена для обробки великої кількості завдань одночасно, що дозволяє адаптуватися до зростаючих вимог та навантажень. Coordinator гарантує, що завдання розподіляються рівномірно та обробляються ефективно, забезпечуючи високу продуктивність системи.

Надійність системи досягається завдяки використанню асинхронної комунікації через RabbitMQ та зберіганню стану завдань у MongoDB та Redis. Це дозволяє зберігати інформацію про прогрес завдань та відновлювати їх виконання у разі збоїв або перезапуску компонентів.

Розробка Coordinator та Worker є критично важливою для забезпечення ефективної роботи системи розпізнавання мовлення. Coordinator відповідає за отримання завдань та розподілення їх між воркерами, тоді як Worker безпосередньо виконують завдання з обробки аудіо та розпізнавання мовлення. Взаємодія між цими компонентами гарантує надійність, масштабованість та високу продуктивність системи. У наступних підпунктах ми детально розглянемо реалізацію Coordinator та Worker на NodeJS, їхню роботу та взаємодію з чергами завдань.

3.2.1 Координатор на NodeJS: опис роботи та взаємодії з чергою

Coordinator є ключовим компонентом нашої системи розпізнавання мовлення, побудованої на основі мікросервісної архітектури. Його основне завдання полягає в управлінні розподіленням завдань між воркерами (Workers), забезпечуючи оптимальне використання ресурсів і ефективне виконання завдань. У цьому підпункті ми детально розглянемо роботу координатора, його взаємодію з чергою RabbitMQ та воркерами.

На першій стадії Coordinator виконує ініціалізацію додатку, отримує початкові дані та показує інформацію про себе. Якщо виникають помилки під час ініціалізації, вони логуються, і додаток завершує роботу. Первина робота Coordinator зображено на рисунку 3.13 у вигляді блок-схеми.

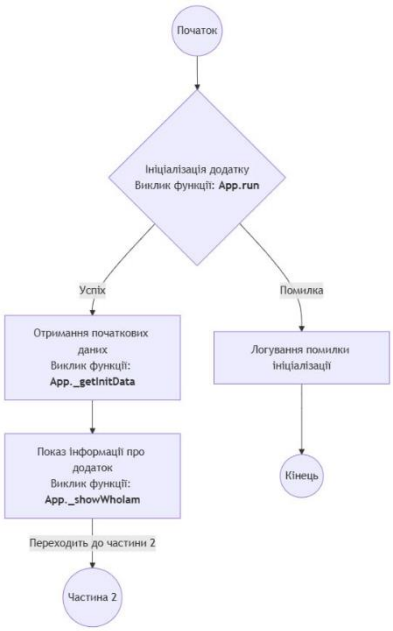


Рис. 3.13 Ініціалізація програми Coordinator

На другому етапі Coordinator виконує синхронізацію з MongoDB, підключається до Redis та RabbitMQ. Кожен з цих кроків є критичним для подальшої роботи системи. У разі виникнення помилок вони логуються, і програма завершує роботу. Блок-схему зображено на рисунку 3.14 .

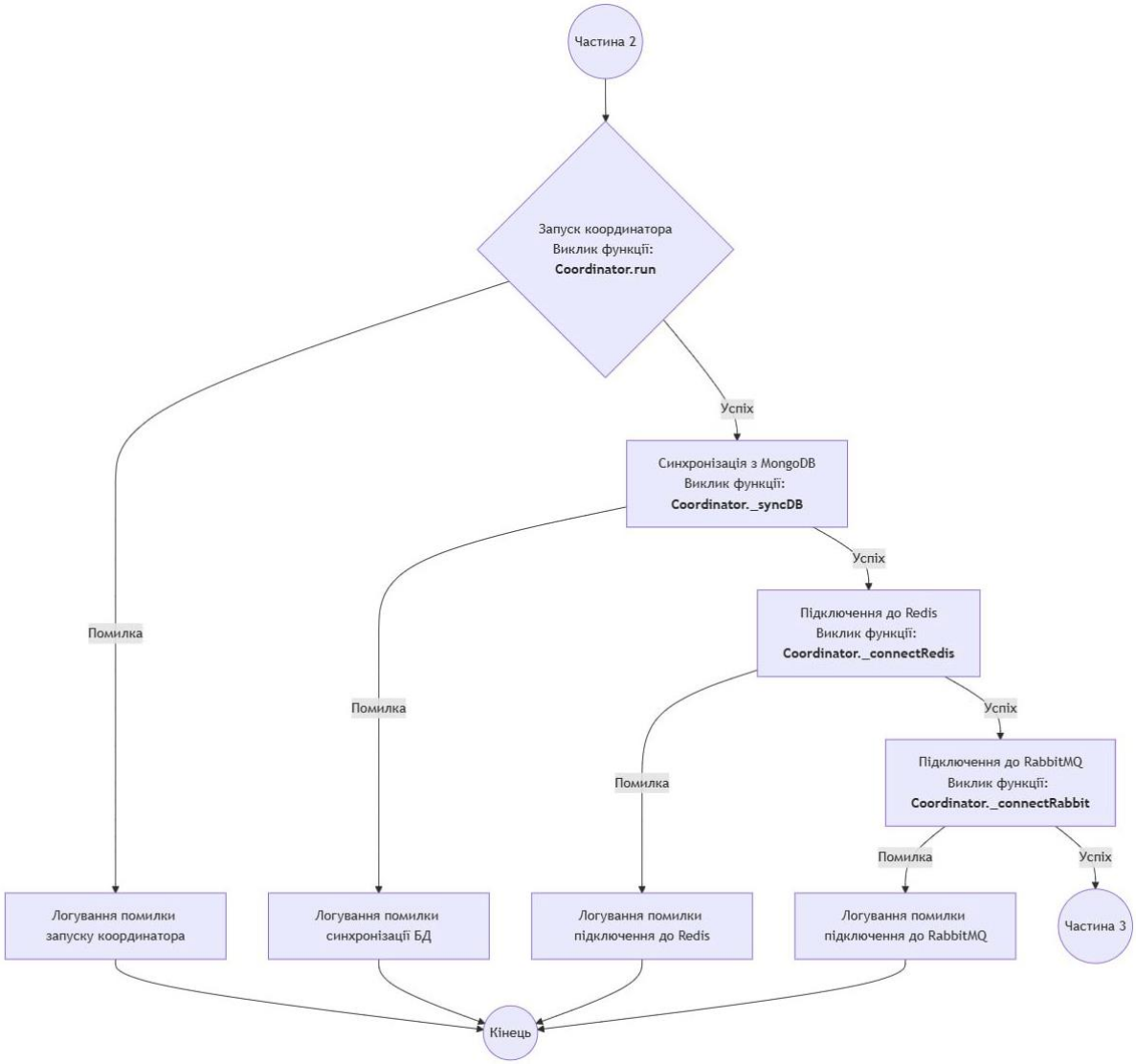


Рис. 3.14 Запуск Coordinator

Після запуску Coordinator ініціалізує WebSocket сервер для зв'язку з воркерами та починає очікування завдань. Після отримання нового завдання або завдання для скасування виконується відповідна обробка. Блок-схему зображено на рисунку 3.15.

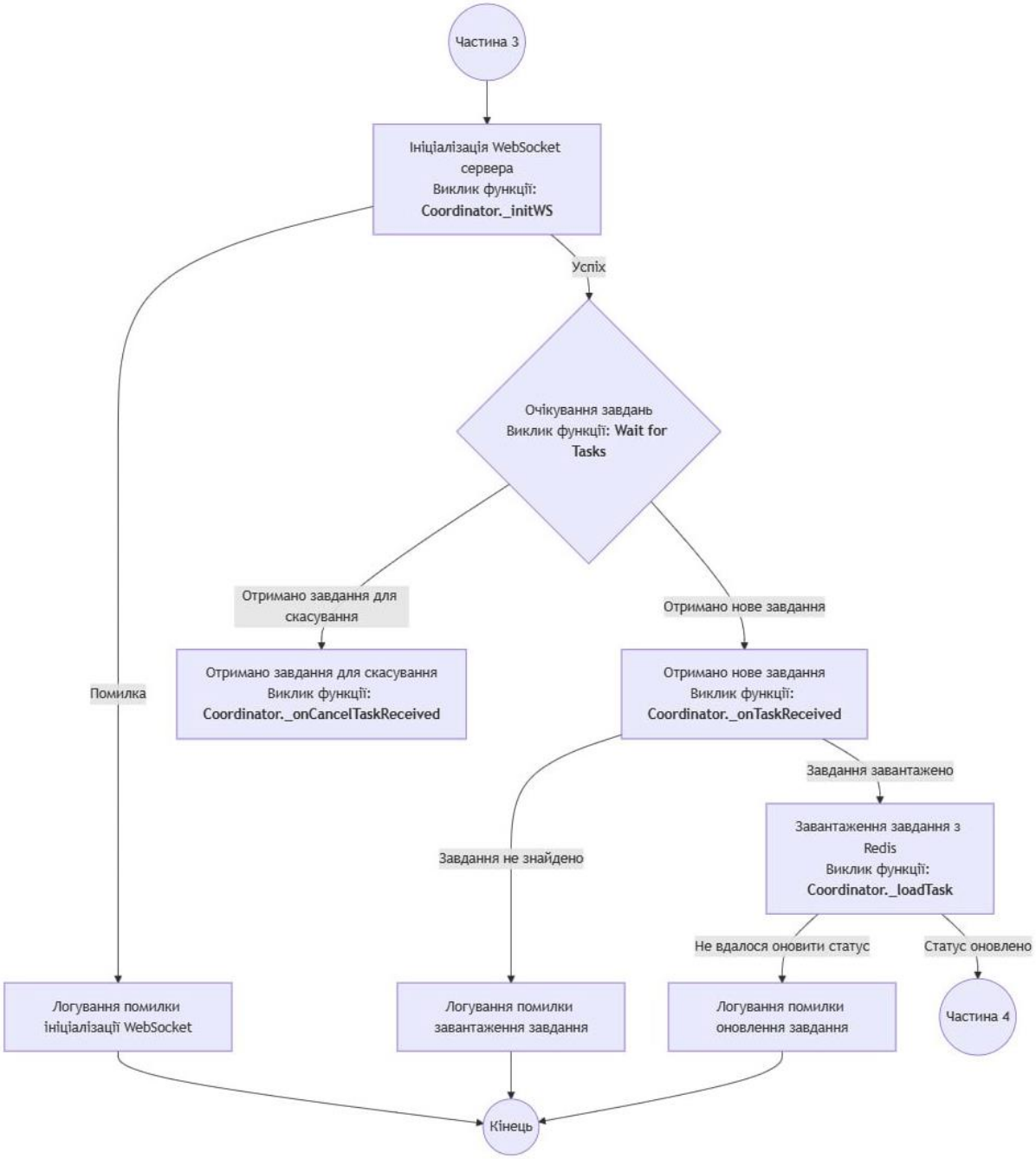


Рис. 3.15 Ініціалізація WebSocket сервера та обробка завдань у Coordinator

На наступному етапі Coordinator оновлює статус завдання і відправляє його воркеру для виконання. Якщо жоден воркер не доступний, Coordinator чекає на вільного воркера. Після завершення завдання координатор оновлює його статус. Блок-схему зображено на рисунку 3.16.

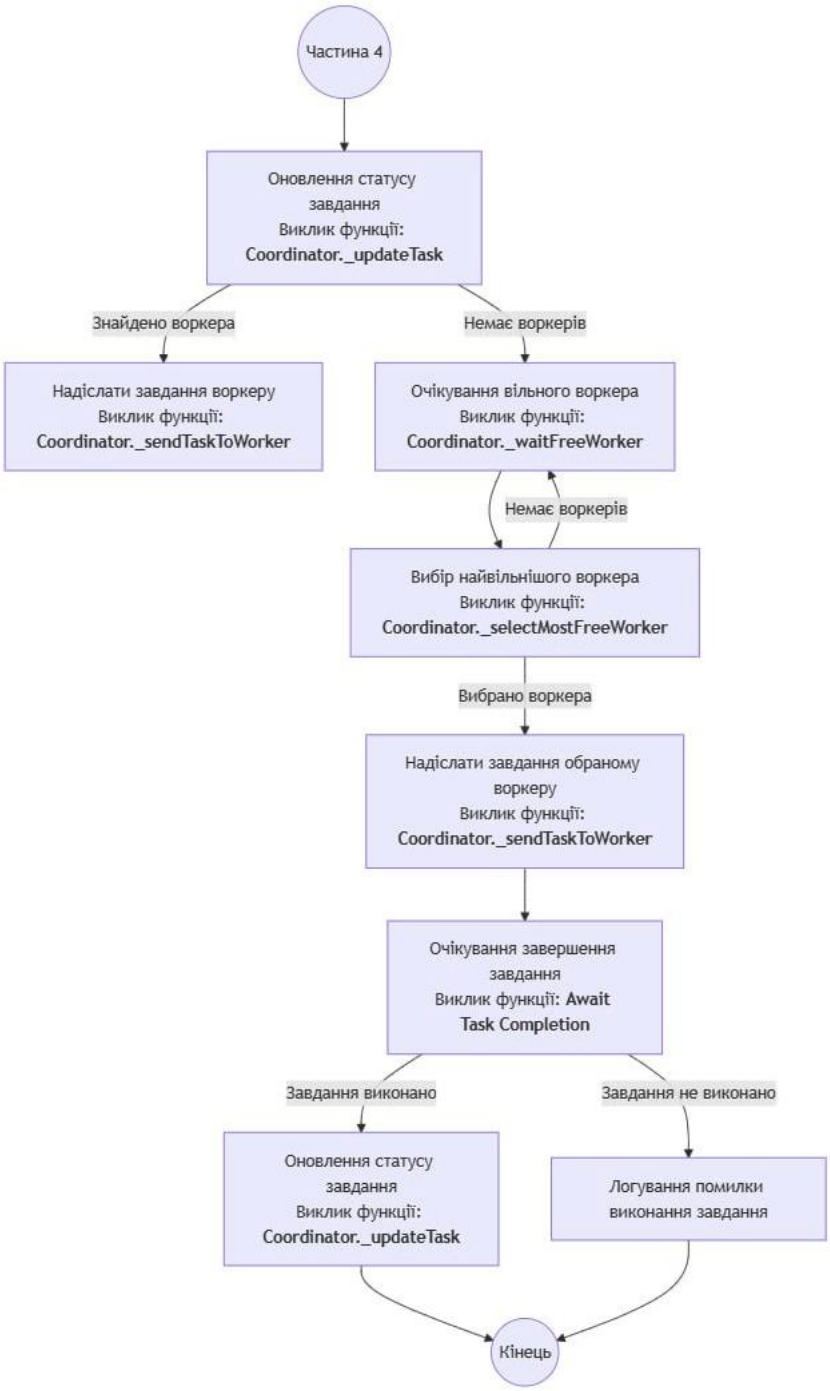


Рис. 3.16 Оновлення статусу завдання та взаємодія з воркерами

Координатор на NodeJS є критично важливим компонентом нашої системи розпізнавання мовлення. Він забезпечує ефективне розподілення завдань між воркерами, аналізуючи їхню завантаженість та використовуючи сокет-з'єднання для комунікації. Використання RabbitMQ для управління чергами завдань дозволяє здійснювати асинхронну комунікацію і гарантує надійну доставку завдань. Основні

функції координатора включають ініціалізацію додатку, запуск та синхронізацію з MongoDB, підключення до Redis і RabbitMQ, а також обробку завдань і взаємодію з воркерами.

Представлена архітектура Coordinator забезпечує високу продуктивність і масштабованість системи, що дозволяє адаптуватися до зростаючих вимог та навантажень. Завдяки ефективному розподілу завдань та аналізу завантаженості воркерів, координатор гарантує, що система працює оптимально, забезпечуючи стабільне і надійне розпізнавання мовлення.

3.2.2 Воркери на NodeJS: запуск і виконання задач, взаємодія з transcriber.py та WhisperX

Воркери (Workers) є ключовими компонентами нашої системи розпізнавання мовлення. Їх основна задача полягає у виконанні завдань, наданих координатором, з метою розпізнавання мовлення з аудіофайлів. Кожен воркер працює незалежно і паралельно, що забезпечує високу продуктивність та масштабованість системи. У цьому підпункті ми розглянемо, як запускаються і виконуються задачі воркерами на NodeJS, а також як вони взаємодіють з transcriber.pyта WhisperX.

Початкова фаза запуску воркера, що відображена у блок-схемі на рисунку 3.17, включає ініціалізацію додатку та отримання початкових даних. Це забезпечує правильне налаштування всіх необхідних компонентів і підготовку воркера до роботи.

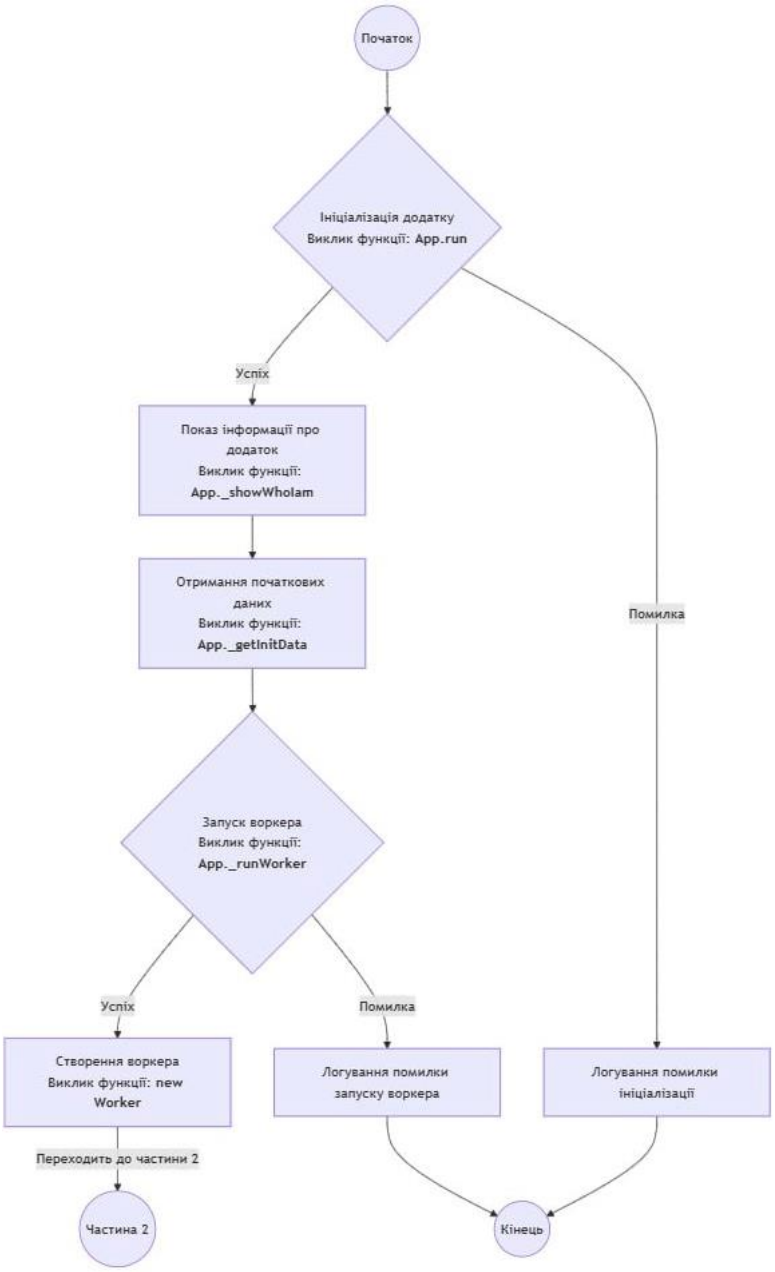


Рис. 3.17 Ініціалізація та запуск програми Worker

Після запуску Worker підключається до MongoDB, Redis та WebSocket сервера координатора. Це забезпечує необхідні з'єднання для обробки даних і комунікації з іншими компонентами системи. Зображено у блок-схемі на рисунку 3.18.

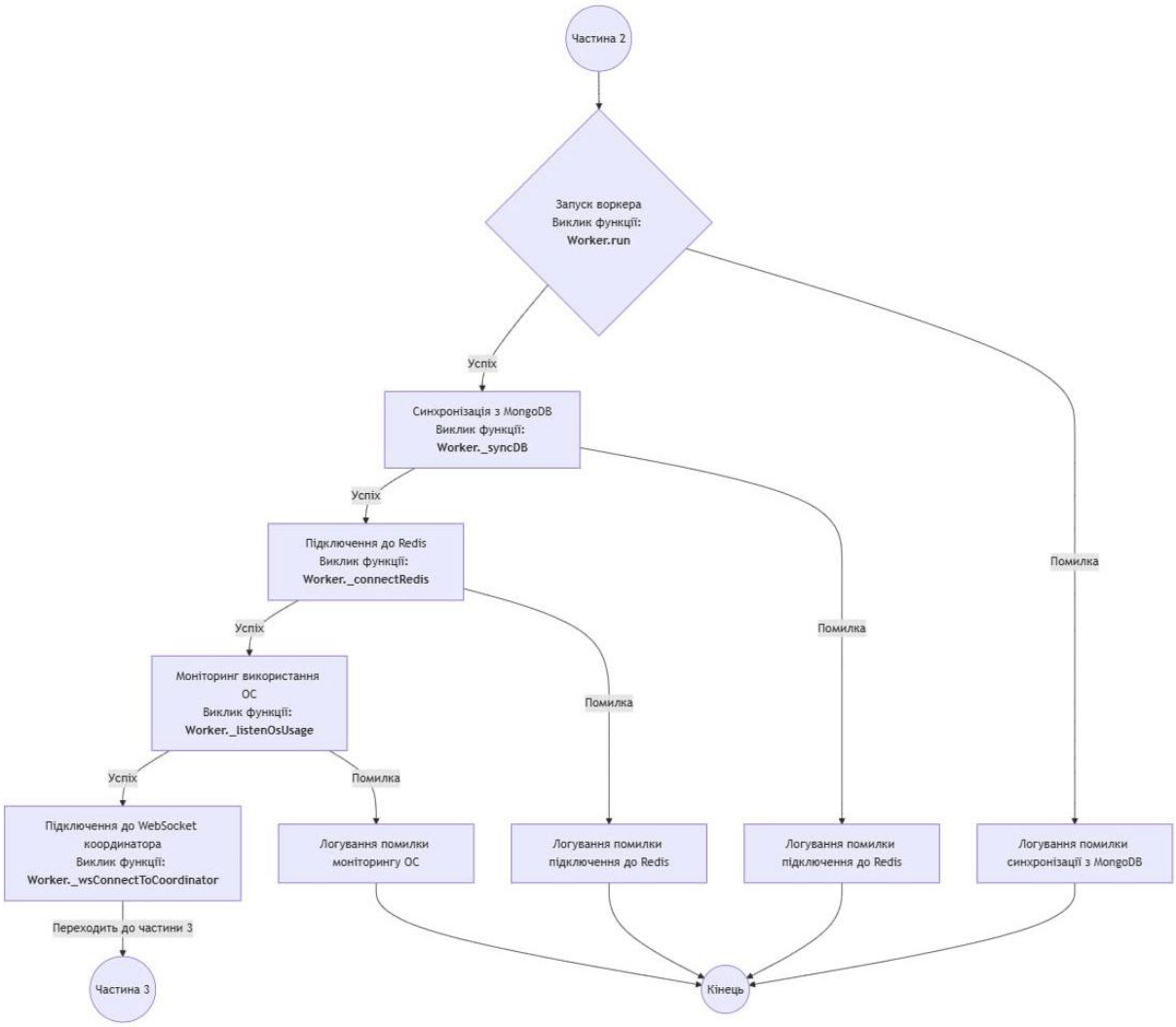


Рис. 3.18 Ініціалізація Worker з його підключенням до Coordinator

Воркери взаємодіють з координатором через WebSocket, отримуючи завдання на виконання та надсилаючи статуси про свій стан. Це забезпечує ефективний розподіл завдань і моніторинг виконання. Взаємодію Worker з Coordinator за допомогою WebSocket відображено у блок-схемі на рисунку 3.19.

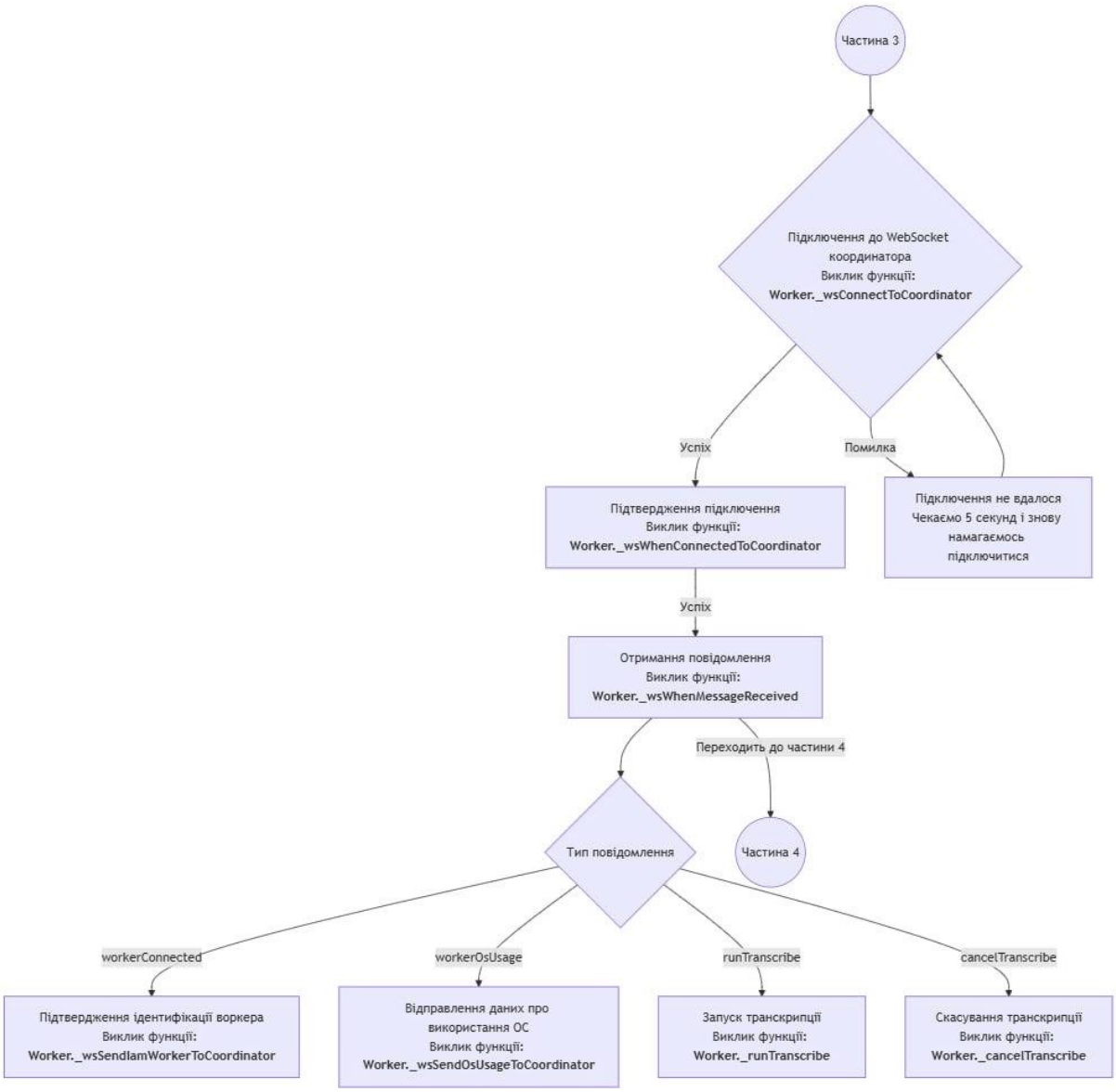


Рис. 3.19 Взаємодія Worker з Coordinator за допомогою WebSocket

Основна частина роботи воркера полягає в обробці завдань, отриманих від координатора. Воркери запускають процес розпізнавання мовлення із аудіо, використовуючи transcriber.py і модель WhisperX, та оновлюють статуси завдань у Redis і MongoDB. Це відображено у блок-схемі на рисунку 3.20.

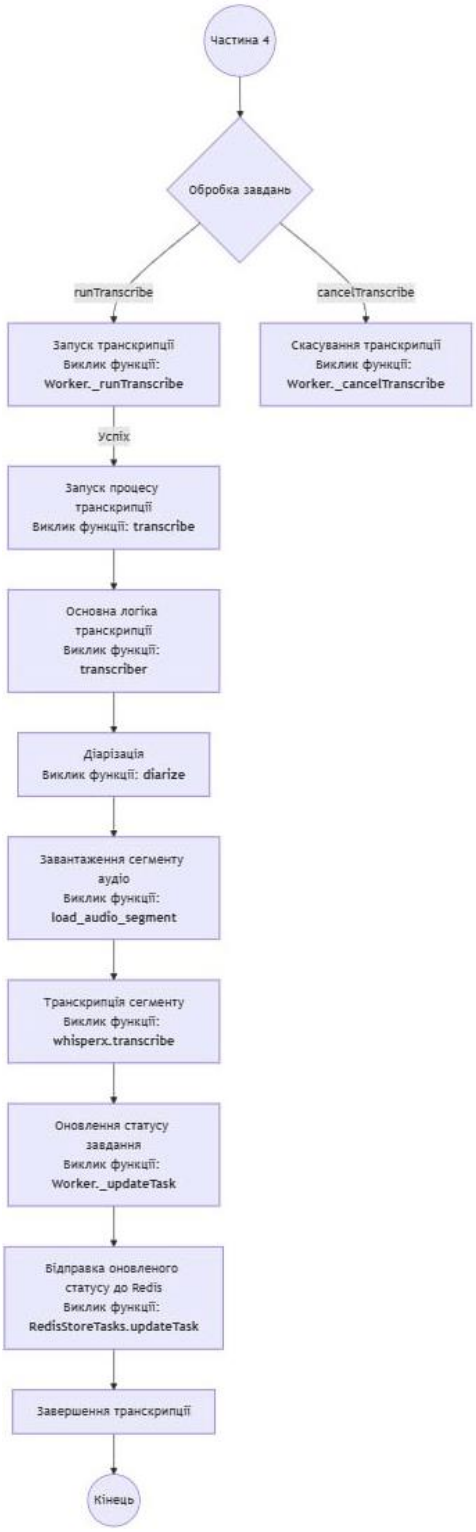


Рис. 3.20 Обробка завдань у Worker

Воркери на NodeJS є виконавчими компонентами нашої системи розпізнавання мовлення. Вони отримують завдання від координатора через

WebSocket, виконують їх за допомогою скрипту transcriber.py і моделі WhisperX, а потім оновлюють статус завдань у базі даних і Redis.

Загальна схема функціонування застосунку та взаємодія між її компонентами наведена на рисунку 3.21 у Додатку А

Використання NodeJS для реалізації воркерів забезпечує високу продуктивність і масштабованість системи, дозволяючи обробляти велику кількість завдань одночасно. Взаємодія з transcriber.py та WhisperX дозволяє досягти точного і ефективного розпізнавання мовлення, що є критично важливим для успішного виконання завдань системи

3.3 Контейнеризація сервісів

У сучасних розподілених системах контейнеризація стає ключовим компонентом для забезпечення гнучкості, масштабованості та зручності в управлінні сервісами. Контейнери дозволяють ізолювати додатки разом з усіма їхніми залежностями, забезпечуючи можливість швидкого розгортання та легкої міграції між різними середовищами. Використання контейнерів також сприяє підвищенню продуктивності та надійності системи завдяки стандартизованому підходу до розгортання.

Необхідно розглянути впровадження контейнеризації для ключових компонентів нашої системи розпізнавання мовлення, та детально розглянути процес контейнеризації для сервісів.

Для цього потрібно описати процес налаштування та розгортання бази даних MongoDB у Docker контейнерах. Детально описати налаштування брокера повідомлень RabbitMQ та Redis, а також ключових компонентів: API, Coordinator та Worker у контейнерах Docker.

Контейнеризація цих сервісів забезпечить не тільки стабільність і продуктивність системи, але і значно спростить її розгортання та обслуговування. У наступних підпунктах ми детально розглянемо кожен з цих етапів, описуючи

необхідні файли конфігурації, скрипти та команди для налаштування та запуску контейнерів.

3.3.1 Впровадження MongoDB з контейнеризацією у Docker

Контейнеризація MongoDB є важливим кроком для забезпечення гнучкості, масштабованості та легкості в розгортанні нашої системи розпізнавання мовлення. У цьому підпункті ми розглянемо процес контейнеризації MongoDB за допомогою Docker, включаючи налаштування змінних середовища, конфігурацію контейнерів у Docker Compose, а також запуск і зупинку контейнерів за допомогою командних файлів.

Для контейнеризації MongoDB була створена спеціальна структура директорії DataBase, яка зображена на рисунку 3.22, що включає всі необхідні файли та підкаталоги для налаштування та запуску контейнерів, пов’язаних з базою даних.

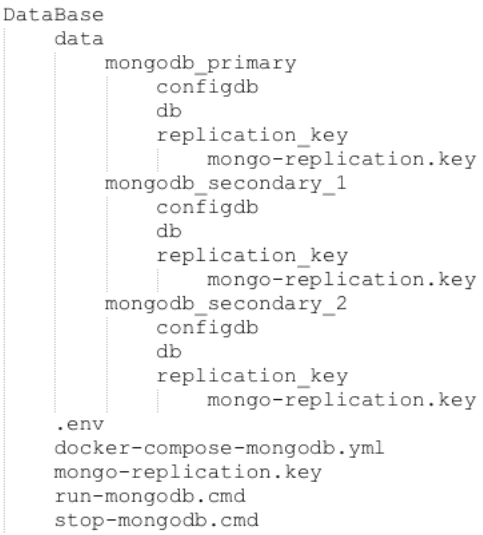


Рис. 3.22 Структура каталогів бази даних

Файл .env містить налаштування змінних середовища, які використовуються для налаштування контейнерів MongoDB. Це включає ідентифікатор реплікації, ім'я користувача, пароль, назви баз даних, імена контейнерів та порти.

Файл docker-compose-mongodb.yml містить конфігурацію для трьох контейнерів MongoDB: первинного (primary) і двох вторинних (secondary). Кожен контейнер налаштований для роботи у режимі реплікації, використовуючи ключ реплікації. Конфігурація включає такі основні параметри: образ Docker, назва контейнера, параметри командного запуску, порти, змінні середовища та томи для зберігання даних і ключів реплікації.

Приклад конфігурації контейнерів mongodb зображено на рисунку 3.23.

```
mongodb_primary:
    networks:
      - bobos-net
    image: mongo:6-jammy
    container_name: ${NAME_PRIMARY}
    restart: always
    command: ["--replSet", "${REPLICA_ID}", "--bind_ip_all", "--port", "${PORT_PRIMARY}",
"--keyFile", "/data/replication_key/mongo-replication.key"]
    ports:
      - ${PORT_PRIMARY}:${PORT_PRIMARY}
    volumes:
      - volume_data_db_primary:/data/db
      - volume_data_configdb_primary:/data/configdb
      - volume_data_key_primary:/data/replication_key
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${PASSWORD}
      MONGO_INITDB_DATABASE: ${DB}
```

Рис. 3.23 Конфігурація контейнерів MongoDB у Docker-Compose

У залежності від ролі контейнеру Primiry або Secondary змінюються лише змінні середовища у .env, що зображено на рисунку3.24.

```
REPLICA_ID=rs0
USERNAME=bo-admin
PASSWORD=Qwerty123
DB=mydb
NAME_PRIMARY=mongodb_primary
NAME_SECONDARY_1=mongodb_secondary_1
NAME_SECONDARY_2=mongodb_secondary_2
GUEST_HOST=172.24.9.21
PORT_PRIMARY=27017
PORT_SECONDARY_1=27018
PORT_SECONDARY_2=27019
TIMEOUT_INIT_REPLICA=40
```

Рис. 3.24 Змінні середовища .env для контейнерів MongoDB

Запуск контейнерів здійснюється за допомогою командного файлу «run-mongodb.cmd», який автоматично створює необхідні каталоги, перевіряє наявність

ключів реплікації та запускає контейнери. Також файл дозволяє ініціалізувати реплікаційний набір (Replica Set) під час першого запуску. Ключові команди файлу «run-mongodb.cmd» зображені на рисунку 3.25.

```
docker exec -it %NAME_PRIMARY% mongosh --username %USERNAME% --password %PASSWORD% --eval
"rs.initiate({_id: '%REPLICA_ID%', members: [{_id: 0, host: '%GUEST_HOST%:%PORT_PRIMARY%'},
{ _id: 1, host: '%GUEST_HOST%:%PORT_SECONDARY_1%'}, { _id: 2, host:
'%GUEST_HOST%:%PORT_SECONDARY_2%'}]})"

docker exec -it %NAME_PRIMARY% mongosh --username %USERNAME% --password %PASSWORD% --eval
"rs.status()"

docker exec -it %NAME_PRIMARY% mongosh --username %USERNAME% --password %PASSWORD%  --eval
"use %DB%; db.createUser({user: '%USERNAME%', pwd: '%PASSWORD%', roles: [{role: 'readWrite',
db: '%DB%'}]})"
```

Рис. 3.25 Команди для ініціалізації та перевірки стану Replica Set та створення нових користувача та бази даних у MongoDB

Зупинка контейнерів здійснюється за допомогою командного файлу «stop-mongodb.cmd», який дозволяє зупинити і видалити контейнери, а також очистити дані за потреби. Ключові команди файлу «stop-mongodb.cmd» зображено на рисунку 3.26.

```
docker stop %NAME_PRIMARY% %NAME_SECONDARY_1% %NAME_SECONDARY_2%

docker rm -f %NAME_PRIMARY% %NAME_SECONDARY_1% %NAME_SECONDARY_2%

rmdir /s %NAME_PRIMARY%
rmdir /s %NAME_SECONDARY_1%
rmdir /s %NAME_SECONDARY_2%
```

Рис. 3.26 Ключові команди файлу «stop-mongodb.cmd»

Загальна схема, яка відображає структуру DataBase зображено на рисунку 3.27.

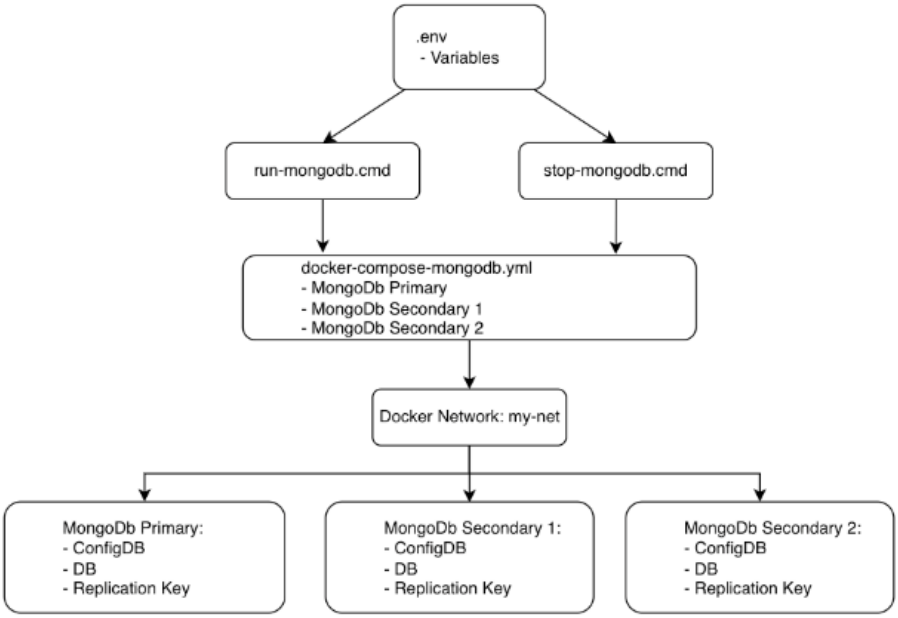


Рис. 3.27 Загальна схема DataBase

Контейнеризація MongoDB за допомогою Docker забезпечує гнучкість, масштабованість і легкість в розгортанні системи. Це дозволяє ефективно керувати базою даних, забезпечуючи високу доступність та продуктивність. Використання змінних середовища та командних файлів для запуску і зупинки контейнерів забезпечує автоматизацію процесів і спрощує підтримку системи.

3.3.2 Впровадження Redis з контейнеризацією у Docker

Контейнеризація Redis з використанням Docker дозволяє забезпечити надійний, масштабований і легко керований кеш-пам'ять для нашої системи розпізнавання мовлення. У цьому підпункті ми розглянемо процес впровадження Redis за допомогою Docker, включаючи налаштування змінних середовища, конфігурацію контейнерів у Docker Compose, а також запуск і зупинку контейнерів за допомогою командних файлів.

Для контейнеризації Redis була створена структура директорії Redis, що зображена на рисунку 3.28, яка включає всі необхідні файли та підкаталоги для налаштування та запуску контейнерів.

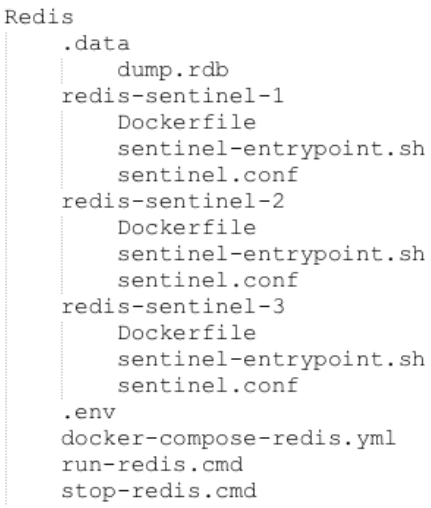


Рис. 3.28 Структура директорії Redis

Файл .env містить налаштування змінних середовища, що зображені на рисунку 3.29, які використовуються для налаштування контейнерів Redis. Це включає пароль, назву головного сервера (master), порти для сентиелів та інші конфігурації.

```
REDIS_PASSWORD=Qwerty123
REDIS_MASTER_HOST=172.24.9.21
REDIS_SENTINEL_NAME=redismaster
REDIS_SENTINEL1_PORT=26379
REDIS_SENTINEL2_PORT=26380
REDIS_SENTINEL3_PORT=26381
REDIS_SENTINEL_QUORUM=2
REDIS_SENTINEL_DOWN_AFTER=3000
REDIS_SENTINEL_FAILOVER=3000
```

Рис. 3.29 Змінні середовища .env для Redis

Redis сконфігурований у режимі Master/Slave/Sentinels, що забезпечує високу доступність та надійність даних через наступні механізми:

- Master: Це основний сервер, який приймає всі записи і виконує операції над даними. Він відповідає за обробку всіх запитів на зміну даних;
- Slave: Це вторинний сервер, який синхронізує свої дані з основним сервером (Master). Він дозволяє розподіляти навантаження на читання даних, збільшуючи продуктивність системи. У разі збою на Master, один з Slave може стати новим Master;

- Sentinels: Це процеси моніторингу, які слідкують за здоров'ям Master і Slave серверів. Вони автоматично виконують фейловер (переключення ролей) у разі збою основного сервера, обираючи новий Master з доступних Slave. Sentinels також повідомляють клієнтів про новий Master сервер після фейловера.

Кожен сентинел має: свій Dockerfile, що зображений на рисунку 3.30; вхідний скрипт і конфігураційний файл «sentinel.conf», що зображений на рисунку 3.31, які налаштовують параметри для підключення до головного сервера і управління ним.

```
FROM redis:7-alpine

ARG REDIS_MASTER_HOST
ARG REDIS_SENTINEL_PORT
ARG REDIS_SENTINEL_QUORUM
ARG REDIS_SENTINEL_DOWN_AFTER
ARG REDIS_SENTINEL_FAILOVER
ARG REDIS_SENTINEL_NAME
ARG REDIS_PASSWORD

ENV MASTER_HOST=${REDIS_MASTER_HOST}
ENV PASSWORD=${REDIS_PASSWORD}
ENV SENTINEL_NAME=${REDIS_SENTINEL_NAME}
ENV SENTINEL_PORT=${REDIS_SENTINEL_PORT}
ENV SENTINEL_QUORUM=${REDIS_SENTINEL_QUORUM}
ENV SENTINEL_DOWN_AFTER=${REDIS_SENTINEL_DOWN_AFTER}
ENV SENTINEL_FAILOVER=${REDIS_SENTINEL_FAILOVER}

RUN mkdir -p /redis

WORKDIR /redis

COPY sentinel.conf .
COPY sentinel-entrypoint.sh /usr/local/bin/

RUN chown redis:redis /redis/* && \
    chmod +x /usr/local/bin/sentinel-entrypoint.sh

EXPOSE ${REDIS_SENTINEL_PORT}


ENTRYPOINT ["sentinel-entrypoint.sh"]
```

Рис. 3.30 Dockerfile для сентінелів

```
port REDIS_SENTINEL_PORT

dir /tmp

requirepass "REDIS_PASSWORD"
sentinel resolve-hostnames yes
sentinel announce-ip REDIS_MASTER_HOST
sentinel announce-port REDIS_SENTINEL_PORT
sentinel monitor REDIS_SENTINEL_NAME REDIS_MASTER_HOST 6379 REDIS_SENTINEL_QUORUM
sentinel auth-pass REDIS_SENTINEL_NAME "REDIS_PASSWORD"
sentinel down-after-milliseconds REDIS_SENTINEL_NAME REDIS_SENTINEL_DOWN_AFTER
sentinel parallel-syncs REDIS_SENTINEL_NAME 1
sentinel failover-timeout REDIS_SENTINEL_NAME REDIS_SENTINEL_FAILOVER
```

Рис. 3.31 Конфігураційний файл сентінела sentinel.conf

Запуск контейнерів Redis відбувається за допомогою командного файлу «run-redis.cmd», команди якого відображені на рисунку 3.32.

```
@echo off
set NETWORK_NAME=bobos-net
docker network ls | findstr /r "\b%NETWORK_NAME%\b" >nul
if %ERRORLEVEL% neq 0 (
  echo Creating network %NETWORK_NAME%
  docker network create %NETWORK_NAME%
) else (
  echo Network %NETWORK_NAME% already exists
)
docker-compose -f docker-compose-redis.yml up -d
```

Рис. 3.32 Вміст командного файлу «run-redis.cmd»

Контейнеризація Redis за допомогою Docker забезпечує надійну, масштабовану і легко керовану кеш-пам'ять для нашої системи розпізнавання мовлення. Використання змінних середовища та конфігураційних файлів дозволяє автоматизувати процеси налаштування і забезпечити високу доступність та продуктивність системи. Контейнери Redis легко запускати і зупиняти за допомогою командних файлів, що спрощує підтримку і розгортання системи.

3.3.3 Впровадження RabbitMQ з контейнеризацією у Docker

Контейнеризація RabbitMQ з використанням Docker дозволяє забезпечити надійну, масштабовану і легко керовану систему обміну повідомленнями для нашої системи розпізнавання мовлення. У цьому підпункті ми розглянемо процес впровадження RabbitMQ за допомогою Docker, включаючи налаштування змінних середовища, конфігурацію контейнерів у Docker Compose, а також запуск і зупинку контейнерів за допомогою командних файлів.

Для контейнеризації RabbitMQ була створена структура директорії Broker, що зображена на рисунку 3.33, яка включає всі необхідні файли та підкаталоги для налаштування та запуску контейнерів.

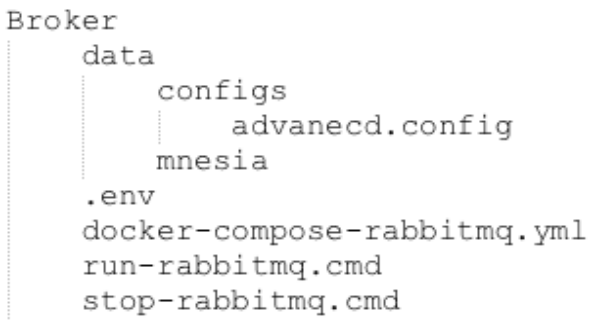


Рис. 3.33 Структура директорії Broker

Файл docker-compose-rabbitmq.yml містить конфігурацію для контейнера RabbitMQ, зображену на рисунку 3.34, з увімкненим управлінням (management). Конфігурація включає змінні середовища, порти, томи для зберігання даних і конфігураційні файли.

```
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3.13.1-management
    container_name: rabbitmq-whisperx
    hostname: rabbitmq-whisperx
    environment:
      - RABBITMQ_DEFAULT_USER=${RABBITMQ_USERNAME}
      - RABBITMQ_DEFAULT_PASS=${RABBITMQ_PASSWORD}
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - mnesia:/var/lib/rabbitmq/mnesia
      - ./data/configs/advanced.config:/etc/rabbitmq/advanced.config
    networks:
      - bobos-net
networks:
  bobos-net:
    external: true

volumes:
  mnesia:
    driver: local
    driver_opts:
      type: none
      o: bind
      device: ./data/mnesia
```

Рис. 3.34 Конфігурація Docker-Compose для RabbitMQ

Запуск контейнерів здійснюється за допомогою командного файлу «run-rabbitmq.cmd», який автоматично створює необхідну мережу Docker, каталоги та запускає контейнер RabbitMQ. Вміст файлу «run-rabbitmq.cmd» зображено на рисунку3.35.

```
@echo off

setlocal

REM Завантаження змінних середовища
for /f "usebackq tokens=1,2 delims==" %%a in (.env) do (
  set "%%a=%%b"
)

set NETWORK_NAME=bobos-net
docker network ls | findstr /r "\b%NETWORK_NAME%\b" >nul
if %ERRORLEVEL% neq 0 (
  echo Creating network %NETWORK_NAME%
  docker network create %NETWORK_NAME%
) else (
  echo Network %NETWORK_NAME% already exists
)

set "main_dir=%cd%"
cd %main_dir%

REM Створення необхідних каталогів
mkdir data
cd data
mkdir configs
mkdir mnesia

REM Створення файлу конфігурації
set "advanced_config_path=data\configs\advanced.config"
(
echo [
echo   {rabbit, [
echo     {consumer_timeout, undefined}
echo   ]}
echo ].
) > "%advanced_config_path%"

REM Запуск контейнера RabbitMQ
docker-compose -f docker-compose-rabbitmq.yml up -d

endlocal
pause
exit /b
```

Рис. 3.35 Вміст файлу «run-rabbitmq.cmd» для запуску RabbitMQ у Docker

Зупинка контейнерів здійснюється за допомогою командного файлу stop-rabbitmq.cmd, який дозволяє зупинити контейнер за допомогою команди «docker stop rabbitmq-whisperx».

Контейнеризація RabbitMQ за допомогою Docker забезпечує надійну, масштабовану і легко керовану систему обміну повідомленнями для нашої системи розпізнавання мовлення. Використання змінних середовища та конфігураційних файлів дозволяє автоматизувати процеси налаштування і забезпечити високу доступність та продуктивність системи. Контейнери RabbitMQ легко запускати і зупиняти за допомогою командних файлів, що спрощує підтримку і розгортання системи.

3.3.4 Контейнеризація API, Coordinator та Worker

Контейнеризація основних компонентів системи розпізнавання мовлення, таких як API, Coordinator та Worker, дозволяє забезпечити їхню гнучкість, масштабованість та легкість у розгортанні та підтримці. Розглянемо більш детально процес контейнеризації цих компонентів за допомогою Docker. Кожен з цих компонентів, для його контейнеризації у Docker містить три файли:

- «docker-compose.yml» – конфігурація Docker Compose для запуску контейнера;
- «Dockerfile» – інструкції створення образу Docker;
- «_in_docker.sh» – скрипт для налаштування і запуску додатка всередині контейнера.

На рисунку 3.36 продемонстровані Dockerfile для контейнерів API та Coordinator (для них він є однаковим).

```
FROM node:20
RUN apt-get update && apt-get install -y iputils-ping
WORKDIR /app
COPY . .
EXPOSE 8080
RUN chmod +x _in_docker.sh
ENTRYPOINT ["/app/_in_docker.sh"]
```

Рис. 3.36 Dockerfile для контейнерів API та Coordinator

Для ефективного розпізнавання мовлення з аудіо необхідно використовувати графічну карту (GPU). Використання GPU значно підвищує продуктивність обробки аудіо даних, оскільки графічна карта здатна паралельно виконувати велику кількість обчислень. За замовчуванням контейнери Docker не мають доступу до графічних карт. Тому для ефективного виконання процесу розпізнавання мовлення у контейнері Worker необхідно налаштувати Docker для роботи з GPU.

Для того, щоб контейнер Worker мав доступ до GPU фізичного хоста, на якому він запущений, у «docker-compose.yml» вказані параметри, що зображені на рисунку 3.37.

```
deploy:
  resources:
    reservations:
      devices:
        - driver: nvidia
          count: all
          capabilities: [gpu]
environment:
  - NVIDIA_VISIBLE_DEVICES=all
  - NVIDIA_DRIVER_CAPABILITIES=compute,utility,video
```

Рис. 3.37 Конфігурація Docker-Compose для використання GPU у контейнері Worker

На рисунку 3.38 продемонстровано загальну конфігурацію Docker-Compose для контейнерів API, Coordinator та Worker.

```
whisperx-api:
    build:
      context: .
      dockerfile: Dockerfile
    image: whisperx-api:1.0
    container_name: whisperx-api
    env_file:
      - ./.env
    volumes:
      - shared_data:/shared_data
      - .:/app
    networks:
      - my-net
    ports:
      - 8084:8080

whisperx-coordinator:
    build:
      context: .
      dockerfile: Dockerfile
    image: whisperx-coordinator:1.0
    container_name: whisperx-coordinator
    env_file:
      - ./.env
    volumes:
      - shared_data:/shared_data
      - .:/app
    networks:
      - my-net
    ports:
      - 8082:8080

whisperx-worker:
    build:
      context: .
      dockerfile: Dockerfile
    image: whisperx-worker:1.0
    container_name: whisperx-worker
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
    env_file:
      - ./.env
    volumes:
      - shared_data:/shared_data
      - .:/app
    environment:
      - NVIDIA_VISIBLE_DEVICES=all
      - NVIDIA_DRIVER_CAPABILITIES=compute,utility,video
    networks:
      - my-net
```

Рис. 3.38 Конфігурація контейнерів API, Coordinator та Worker у Docker-Compose

Також, для роботи з графічною картою для Worker необхідно щоб його Dockerfile містив відповідні інструкції, які підтримують CUDA та cuDNN, що забезпечують необхідне середовище для роботи з GPU. Для цього використовується базовий образ «nvidia/cuda» та встановлені наступні залежності Python: torch (з підтримкою CUDA), torchaudio, numpy, pydub, pyannote.audio, ctranslate2, а також встановлений модуль WhisperX. Dockerfile для Worker зображений на рисунку 3.39.

```
Dockerfile:

    FROM nvidia/cuda:11.8.0-cudnn8-devel-ubuntu22.04

    RUN apt-get update && apt-get install -y \
        curl \
        wget \
        python3 \
        python3-pip \
        ffmpeg \
        && rm -rf /var/lib/apt/lists/*

    RUN if [ ! -f /usr/bin/python ]; then ln -s /usr/bin/python3 /usr/bin/python; fi
    RUN if [ ! -f /usr/bin/pip ]; then ln -s /usr/bin/pip3 /usr/bin/pip; fi

    ENV PATH="/usr/bin:${PATH}"
    WORKDIR /app
    COPY . .

    RUN pip install --no-cache-dir --upgrade pip \
        && pip install torch==2.0.0+cu118 torchaudio==2.0.0 --extra-index-url https://download.pytorch.org/whl/cu118 \
        && pip install --no-cache-dir numpy==1.26.4 \
        && pip install --no-cache-dir pydub==0.25.1 \
        && pip install --no-cache-dir pyannote.audio ctranslate2==3.24.0


    RUN /bin/bash -c "cd /app/_to_install_whisperx && pip install ."

    RUN curl -fsSL https://deb.nodesource.com/setup_20.x | bash - \
        && apt-get install -y nodejs

    CMD ["/app/_in_docker.sh"]


_in_docker.sh:

    #!/bin/sh

    mkdir -p /shared_data
    npm install
    npm run start
```

Рис. 3.39 Dockerfile для Worker

Контейнеризація основних компонентів системи розпізнавання мовлення (API, Coordinator та Worker) за допомогою Docker забезпечує їх гнучкість, масштабованість та легкість у розгортанні та підтримці. Використання змінних середовища та конфігураційних файлів дозволяє автоматизувати процеси налаштування та забезпечити високу доступність та продуктивність системи. Контейнери цих компонентів легко запускати і зупиняти за допомогою командних файлів, що спрощує підтримку і розгортання системи.

Для ефективного використання GPU в контейнерах Worker необхідно налаштувати Docker для роботи з графічними картами, що дозволяє значно підвищити продуктивність обробки аудіо даних. Це досягається за допомогою налаштувань у Docker Compose, які вказують на використання GPU та забезпечують доступ контейнера до графічної карти хост-машини. Використання GPU дозволяє значно швидше виконувати обчислення, необхідні для розпізнавання мовлення, що робить систему більш ефективною та продуктивною.

3.4 Аналіз ефективності системи

Процес оцінки ефективності та оптимізації системи є критично важливим етапом у забезпеченні її надійності, продуктивності та відповідності функціональним вимогам. Метою цього процесу є виявлення можливих проблем у роботі системи, оцінка її продуктивності під різним навантаженням, а також підтвердження, що всі компоненти взаємодіють коректно. Це дозволяє забезпечити високу якість системи перед її впровадженням у реальну експлуатацію. Для досягнення цієї мети ми проведемо випробування нашої розробки двома способами:

1) Тест 1: Одночасно три аудіо файли на одному Worker

За допомогою даного тесту ми зможемо оцінити продуктивність системи при інтенсивному навантаженні на один Worker. Це дозволить визначити максимальну продуктивність та стійкість окремого Worker при одночасному розпізнаванні трьох аудіо файлів.

Так ми зможемо зрозуміти, як система справляється з високим навантаженням на один компонент і які обмеження продуктивності виникають при такій конфігурації. Це допоможе виявити вузькі місця в ресурсах і визначити можливості для оптимізації використання CPU, RAM та GPU.

2) Тест 2: Одночасно три аудіо файли, по одному на трьох паралельних Worker

За допомогою даного тесту ми зможемо оцінити продуктивність системи при розподілі навантаження між кількома Worker. Це дозволить визначити ефективність горизонтального масштабування і здатність системи до паралельної обробки завдань.

Провівши цей тест, ми зрозуміємо, як розподіл завдань між кількома Worker впливає на загальну продуктивність системи. Це допоможе оцінити вигоди від мікросервісної архітектури та виявити можливості для динамічного балансування навантаження. Такий підхід забезпечить більш ефективне використання ресурсів і підвищить стійкість системи до навантажень.

Використовуючи ці дві тестові стратегії, ми зможемо отримати повну картину про продуктивність і ефективність нашої системи в різних умовах навантаження, що дозволить провести необхідні оптимізації та покращити загальну роботу системи.

3.4.1 Методологія оцінки ефективності

Для забезпечення високої продуктивності та стабільності системи були застосовані наступні методи тестування:

- *Функціональне тестування:* Перевірка кожного сервісу на відповідність вимогам та коректне виконання його функцій. Це включало перевірку API, Coordinator та Worker на їх здатність правильно обробляти запити та виконувати розпізнавання мовлення.
- *Навантажувальне тестування:* Імітація високих навантажень на систему для оцінки її продуктивності при різних рівнях навантаження. Було проведено тестування з одночасним розпізнаванням трьох аудіо файлів на одному Worker та на трьох паралельних Worker для оцінки продуктивності та стійкості системи.
- *Тестування продуктивності:* Оцінка ефективності використання ресурсів (CPU, RAM, GPU) та часу обробки завдань при різних конфігураціях.

Зокрема, було проведено тести з одночасним розпізнаванням трьох аудіо файлів для визначення оптимальної конфігурації системи.

3.4.2 Результати оцінки ефективності

На основі проведених тестів були отримані наступні результати:

Тест 1: Одночасно три аудіо файли з хронометражем на одному Worker

Результати тесту наведені у таблиці 3.1.

Таблиця 3.1

Випробування одночасного розпізнавання мовлення
з трьох однакових аудіо файлів на одному Worker

Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
00:00:00	5,0	29,0	3,0	10,8	0,6	34,0
00:00:00	100,0	36,0	2,0	48,4	1,9	36,0
00:11:33	100,0	36,7	2,0	59,0	3,1	37,0
00:23:07	50,0	37,0	30,0	69,0	5,0	41,0
00:34:40	36,0	55,0	60,0	97,4	13,1	45,0
00:46:13	55,0	62,7	63,0	97,4	14,4	46,0
00:57:46	58,0	65,8	68,0	98,7	15,0	44,0
01:09:20	41,0	62,4	47,0	98,7	18,1	42,0
01:20:53	60,0	61,7	55,0	98,7	20,0	44,0
01:32:26	47,0	61,1	54,0	98,7	23,1	46,0
01:44:00	40,0	59,3	56,0	97,4	25,0	44,0
01:55:33	38,0	58,3	59,0	96,0	28,1	45,0
02:07:06	41,0	58,0	60,0	97,4	30,0	46,0
02:18:40	39,0	56,1	58,0	97,4	31,3	47,0
02:30:13	38,0	54,8	59,0	98,7	35,0	46,0
02:41:46	37,0	53,6	59,0	98,7	36,9	47,0
02:53:19	35,0	52,4	60,0	97,4	39,4	45,0
03:04:53	34,0	50,2	61,0	97,4	40,6	44,0

Цей тест показав, що при одночасному розпізнаванні трьох аудіо файлів на одному Worker спостерігається високе навантаження на CPU та GPU, що може

призводити до зниження продуктивності, що видно на діаграмі зображеній на рисунку 3.40.

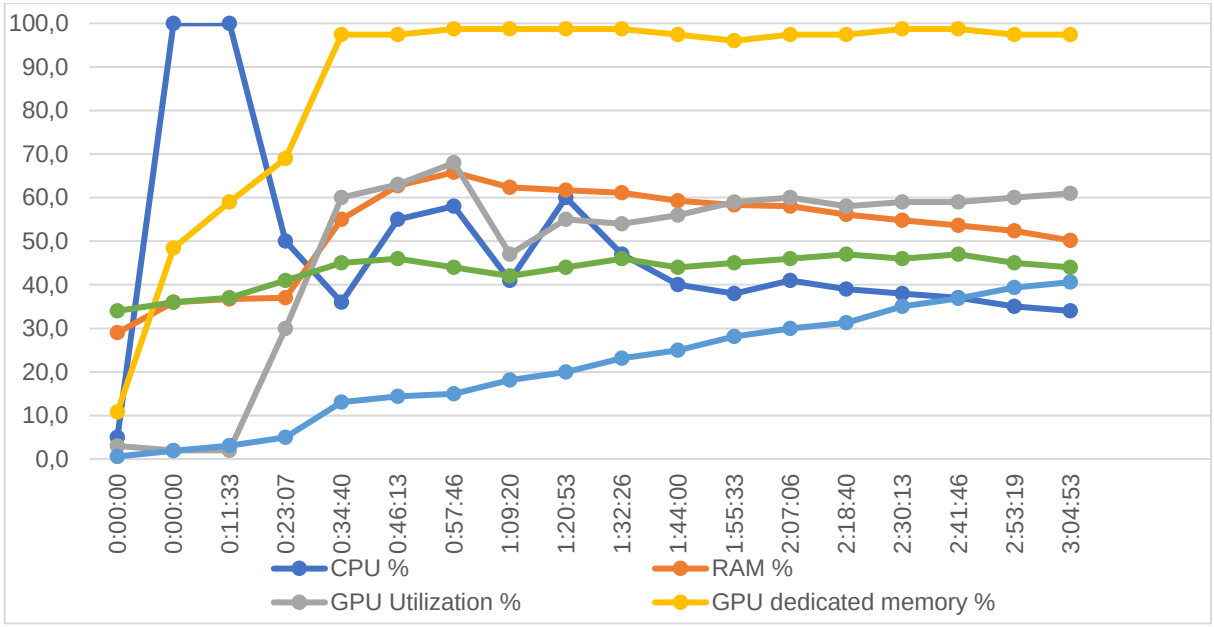


Рис. 3.40 Діаграма результатів тесту-1

Основні результати тесту:

CPU використання сягало 100% на початку та під час розпізнавання.

GPU використання варіювалося, але теж показувало значне навантаження, особливо при обробці інтенсивних задач.

Температура GPU піднімалася до 47°C, що вказує на високий рівень обчислювальної інтенсивності завдань.

Тест 2: Одночасно три аудіо файли, по одному на трьох паралельних Worker

Результати тесту наведені у таблиці 3.2.

Таблиця 3.2

Випробування одночасного розпізнавання мовлення
з трьох однакових аудіо файлів на трьох різних Workers

Worker-1						
Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
00:00:00	5,0	30,0	3,0	8,9	0,6	33,0
00:02:20	60,0	39,2	2,0	33,2	1,3	34,0

00:04:40	37,0	50,0	20,0	98,4	1,9	36,0
00:07:00	55,0	60,0	30,0	98,4	5,0	37,0
00:09:20	40,0	60,0	25,0	98,4	10,0	40,0
00:11:40	48,0	60,5	40,0	99,7	15,0	41,0
Worker-2						
Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
00:00:00	5,0	30,0	3,0	8,9	0,6	33,0
00:02:20	60,0	39,2	2,0	33,2	1,3	34,0
00:04:40	37,0	50,0	20,0	98,4	1,9	36,0
00:07:00	55,0	60,0	30,0	98,4	5,0	37,0
00:09:20	40,0	60,0	25,0	98,4	10,0	40,0
00:11:40	48,0	60,5	40,0	99,7	15,0	41,0
Worker-3						
Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
00:00:00	5,0	30,0	3,0	8,9	0,6	33,0
00:02:20	60,0	39,2	2,0	33,2	1,3	34,0
00:04:40	37,0	50,0	20,0	98,4	1,9	36,0
00:07:00	55,0	60,0	30,0	98,4	5,0	37,0
00:09:20	40,0	60,0	25,0	98,4	10,0	40,0
00:11:40	48,0	60,5	40,0	99,7	15,0	41,0

Результати у вигляді діаграми зображені на рис. 3.41

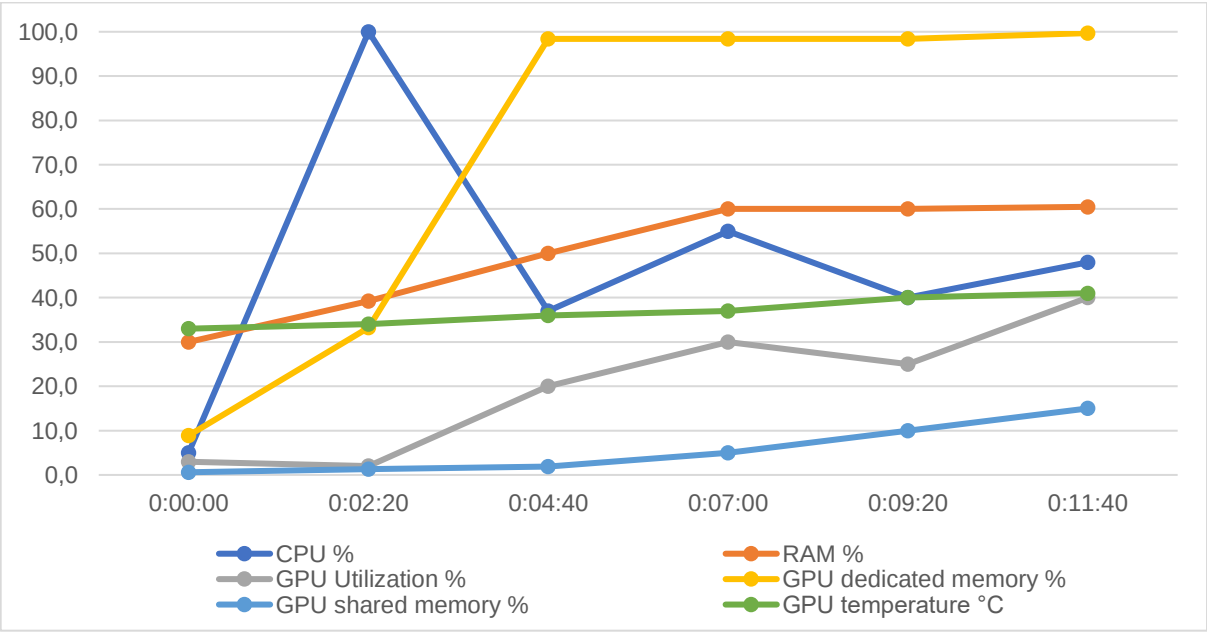


Рис. 3.41 Діаграма результатів тесту-2

Цей тест показав значне покращення продуктивності при розподілі навантаження між трьома паралельними Worker. Основні результати тесту:

Кожен Worker мав значно нижче навантаження на CPU, яке рідко перевищувало 60%.

GPU використання було більш рівномірно розподілене, що дозволило уникнути пікових навантажень.

Температура GPU залишалася стабільною і не перевищувала 41°C, що свідчить про ефективне використання ресурсів.

На основі цих результатів були проведені додаткові оптимізації:

Впровадження механізмів динамічного балансування навантаження між Worker для забезпечення рівномірного розподілу задач.

Використання додаткових Worker для подальшого зниження навантаження і підвищення загальної продуктивності системи.

Загалом, результати тестування показали, що мікросервісна архітектура значно покращує ефективність розпізнавання мовлення завдяки горизонтальному масштабуванню і ефективному використанню ресурсів. Але рекомендовано одночасно на одному Worker проводити розпізнавання мовлення у текст лише з одного аудіо файлу одночасно. Подальші оптимізації спрямовані на поліпшення стійкості системи та забезпечення її здатності адаптуватися до змінних умов навантаження.

3.5 Оцінка економічної доцільності системи

Процес створення та впровадження нової системи розпізнавання мовлення вимагає не тільки технічної оцінки її продуктивності, але й економічного аналізу для визначення доцільності інвестицій. Тому необхідно надати оцінку вартості системи на основі актуальних цін на апаратні елементи, а також порівняти цю вартість з хмарними рішеннями. Це дозволить визначити, який підхід є більш економічно вигідним та ефективним у довгостроковій перспективі.

Враховуючи, що апаратна частина, на якій випробувати Worker відповідає лише мінімальним вимогам, тому можна вартість одного Worker становить від 162996 грн. Згідно результатів проведених випробувань, для підприємства середнього і великого рівня, лише одного Worker буде недостатньо, тому будемо рахувати, що Worker буде у кількості 3 одиниці (має бути хоча б дві основних та одна резервна машини Також для підрахунку економічної доцільності варто враховувати споживання електроенергії. Так як, з урахуванням старіння та зношення апаратної частини, підприємства зазвичай проводять оновлення приблизно один раз на три роки, то будемо рахувати вартість у урахуванням споживання електроенергії на протязі трьох років.

Розробка та проведення випробувань відбувалось на апаратному рішенні, що наведене у таблиці 3.3 та таблиці 3.4.

Таблиця 3.3

Вартість апаратної збірки Worker з урахуванням споживання електроенергії

Назва	Споживання енергії, Вт/год.	Середня узагальнена ціна, грн.
Графічна карта (GPU): NVIDIA RTX 3070 8GB	220	20600
Процесор (CPU): Intel Core i7-8700 3.20 GHz	65	12400
Оперативна пам'ять (RAM): 32 GB DDR4	10	6000
Пам’ять для зберігання: 256 GB SSD	3	1600
Материнська плата	25	5000
Корпус для станції (разом з вентиляторами)	10	2000
Блок живлення (PSU) 750W	20	4000
Охолодження процесора	5	1000
Ціна за споживання енергії за 30 днів		433 грн
Ціна за споживання енергії за 1 рік		5196 грн
Ціна апаратної збірки		52600 грн
Ціна апаратної збірки за 3 одиниці		157800 грн
Сума з урахуванням споживання енергії за 1 рік		162996 грн
Сума з урахуванням споживання енергії за 3 роки		173388 грн

Таблиця 3.4

Вартість апаратної збірки для API, Coordinator, MongoDB, Redis та мережеве сховище з урахуванням споживання електроенергії

Назва	Споживання енергії, Вт/год	Середня узагальнена ціна, грн
Процесор (CPU): Intel Core i5-10400F	65	6000
Оперативна пам'ять (RAM): 16 GB DDR4	5	2800
Пам’ять для зберігання: 4 TB SSD	5	16000
Материнська плата	25	5000
Корпус для станції (разом з вентиляторами)	10	2000
Блок живлення (PSU) 550W	20	3000
Охолодження процесора	5	1000
Ціна за споживання енергії за 30 днів		163 грн
Ціна за споживання енергії за 1 рік		1956 грн
Ціна апаратної збірки за 1 одиницю		35800 грн
Сума з урахуванням споживання енергії за 1 рік		35963 грн
Сума з урахуванням споживання енергії за 3 роки		37756 грн

Згідно наведених таблиць вартість нашої збірки становить: $162\ 996$ грн $+$ $35\ 963$ грн $=$ $198\ 959$ грн на один рік або $173\ 388$ грн $+$ $37\ 756$ грн $=$ $211\ 144$ грн на період протягом 3 років.

Розглянемо можливість використання хмарних рішень для розгортання системи розпізнавання мовлення. Це дозволить отримати доступ до потужних обчислювальних ресурсів, знизити початкові інвестиційні витрати та спростити управління та обслуговування системи. Так як зазвичай провайдери хмарних систем розпізнавання мовлення виставляють рахунок не за ресурси, а за використаний час розпізнавання мовлення, то для більш справедливого економічного порівняння, будемо рахувати наступним чином. Так, як було визначено, що для підприємств необхідно мати хоча б три Worker, і за допомогою випробувань було визначено, що на одному Worker, для більш ефективного застосування, доцільно одночасно розпізнавати лише один файл. Також, згідно проведених випробувань, було визначено, що на одному Worker аудіо файл з хронометражем 1 год (3600 сек) розпізнається за 11 хв (660 сек). Тому, за

пропорцією, можемо порахувати, що на нашій збірці одним Worker за одну годину можна обробляти $\frac{3600 \text{ сек}}{660 \text{ сек}} \times 3600 \text{ аудіо секунд} = 19636 \text{ аудіо секунд}$, тобто аудіо з хронометражем 5 год 27 хв 16 сек. Так як маємо роботу одночасно трьох Worker, то за одну годину наша система потенційно може розпізнати мовлення з $3 \times 19636 = 58908$ аудіо секунд або хронометражем 16 год 21 хв 48 сек за одну годину. Більшість підприємств в Україні працюють за 8-годинним робочим графіком, і, нехай, лише одна цього часу кожного дня виділена процесам розпізнавання мовлення на один робочий день, тому що далеко не всі зможуть згенерувати таку велику кількість контенту для розпізнавання мовлення з нього, і не з усього контенту необхідно це робити. Тоді, з урахуванням наведеного, а також тим, що у 2024 році вважається 262 робочих дня, то за цей період нашою системою буде оброблено приблизно $58908 \times 1 \times 262 = 15\,433\,896$ аудіо секунд. Тоді за три роки приблизно буде $15\,433\,896 \times 3 = 185\,206\,752$ аудіо секунд. Тоді, для порівняння нашої системи з хмарними рішеннями будемо використовувати саме це число як кількість використаних секунд за певним тарифом у період за 3 роки.

У таблиці 3.5 наведена актуальна вартість за стандартну підписку хмарних систем розпізнавання мовлення у різних відомих провайдерів, таких як: OpenAI, Azure та Google.

Таблиця 3.5

Актуальна вартість за стандартну підписку хмарних систем розпізнавання мовлення у провайдерів: OpenAI, Azure та Google

Провайдер	Ціна за 1 сек, грн	Ціна за 1 год, грн	Ціна за 1 робочий рік, грн	Ціна за 3 робочих роки, грн
OpenAI	0,0042	15,00	64 822	194 466
Azure	0,0042	15,00	64 822	194 466
Google	0,0055	19,80	88 886	254 659

Наведені вище хмарні рішення за вказану ціну надають лише обчислювальні потужності за допомогою API. Ці рішення, у нашому випадку, замінюють Worker, і тоді відпадає необхідність у горизонтальному масштабуванні Worker, тоді й також

відпадає потреба у Coordinator. Тоді, для того, щоб працювало таке рішення на підприємстві, достатньо однієї машини, яку ми використовували для API, Coordinator, MongoDB, Redis та мережеве сховище (все на одній машині). Разом з використанням такого апаратного рішення, враховуючи найдешевше хмарне, вартість такого хмарного рішення буде $64\,822 \text{ грн} + 35963 \text{ грн} = 100\,785 \text{ грн}$ на один рік або $194\,466 \text{ грн} + 37756 \text{ грн} = 234\,222 \text{ грн}$ на 3 роки.

Підсумовуючи, можна зробити висновок, що наша система зможе окупитися лише за 3 роки у порівнянні з хмарними рішеннями. Але хмарні рішення більш гнучкі і потенційно зможуть обробити набагато більше аудіо часу за той самий період, що і наша система. Також, з використанням системи на основі хмарного рішення зменшується потреба та складність у обслуговуванні системи. Наша система економічно доцільна лише тоді, якщо вона обирається на довготривалий період, хоча б на три роки для не дуже великих підприємств, адже чим більше підприємств, тим більше аудіо потрібно обробляти і тим більше потрібно Worker, і тим дорожчою стає система. Також, у урахуванням старіння апаратного обладнання, використання за таким довготривалим періодом потребує оновлення апаратної частини. Тому, особливо для коротких періодів використання, для великих підприємств варто розглядати таку систему, як нашу, у поєднанні з хмарними рішеннями.

3.6 Висновки до розділу

Третій розділ даної роботи охоплює всі аспекти розробки та впровадження системи розпізнавання мовлення. Було описано реалізацію API для взаємодії з користувачами за допомогою ExpressJS, що забезпечило надійну комунікацію між користувачами і системою.

Далі розглянуто розробку Coordinator та Worker, зокрема взаємодію Координатора з чергою задач та обробку задач на Воркерах з використанням

transcriber.py та WhisperX. Це забезпечило ефективну та злагоджену роботу системи.

Контейнеризація сервісів за допомогою Docker включала впровадження MongoDB, Redis та RabbitMQ, а також контейнеризацію API, Coordinator та Worker, що додало гнучкості та зручності в управлінні системою.

Проведені тести дали змогу оцінити ефективність системи, випробовуючи її продуктивність у різних умовах навантаження. Це дозволило виявити можливості для подальшої оптимізації.

Економічна оцінка системи порівняла витрати на апаратне забезпечення та електроенергію з хмарними рішеннями від OpenAI, Azure та Google. Виявилося, що система є економічно вигідною для використання на довготривалий період (3 роки і більше) для невеликих підприємств. Водночас хмарні рішення забезпечують більшу гнучкість і знижують витрати на обслуговування, що робить їх привабливими для великих підприємств та короткострокових проектів.

Таким чином, розглянуто всі аспекти розробки, впровадження та економічної оцінки системи розпізнавання мовлення, що дозволяє приймати зважені рішення щодо її використання у реальних умовах експлуатації.

ВИСНОВКИ

У даній роботі було досліджено ефективність застосування Docker та RabbitMQ з системою розпізнавання мовлення з розробкою такої системи. Проаналізовано актуальність теми, підкреслено важливість розпізнавання мовлення у сучасному світі, а також виявлено недоліки та обмеження існуючих систем. Огляд літератури та досліджень дозволив здійснити аналіз сучасних підходів та технологій, які використовуються у цій сфері. На основі цього сформульовано задачу дослідження та визначено очікувані результати.

Теоретична підготовка до вирішення задачі включала аналіз існуючих рішень, зокрема огляд мікросервісних архітектур та використання Docker і RabbitMQ у мікросервісах. Використання Docker у нашій роботі стало критично важливим для забезпечення гнучкості, масштабованості та зручності в управлінні сервісами. Docker дозволив ізолювати кожен компонент системи в окремому контейнері, що спрощує їх розгортання та оновлення. RabbitMQ, у свою чергу, забезпечив надійну обробку та передачу повідомлень між компонентами системи, що дозволило створити злагоджену та ефективну взаємодію між ними.

Було обрано інструменти та технології, зокрема WhisperX для розпізнавання мовлення, MongoDB, Redis та інструменти для координації та виконання завдань. Проектування архітектури передбачало опис мікросервісного продукту та взаємодії між компонентами (API, Coordinator, Worker, MongoDB, Redis, RabbitMQ).

Розробка та впровадження системи охоплювали створення API для взаємодії з користувачами за допомогою ExpressJS, розробку Coordinator та Worker на NodeJS, контейнеризацію сервісів за допомогою Docker (MongoDB, Redis, RabbitMQ, API, Coordinator, Worker). Проведені тести дозволили оцінити ефективність системи та виявити можливості для подальшої оптимізації.

Економічна оцінка системи порівняла витрати на апаратне забезпечення та електроенергію з хмарними рішеннями від OpenAI, Azure та Google. Виявилося, що

система є економічно вигідною для використання на довготривалий період (3 роки і більше) для невеликих підприємств. Водночас хмарні рішення забезпечують більшу гнучкість і знижують витрати на обслуговування, що робить їх привабливими для великих підприємств та короткострокових проектів.

Таким чином, у роботі охоплено всі аспекти розробки, впровадження та економічної оцінки системи розпізнавання мовлення. Це дозволяє приймати зважені рішення щодо її використання у реальних умовах експлуатації, забезпечуючи максимальну ефективність та продуктивність.

ПЕРЕЛІК ПОСИЛАНЬ

1. Berdasco, A.; López, G.; Diaz, I.; Quesada, L.; Guerrero, L.A. User Experience Comparison of Intelligent Personal Assistants: Alexa, Google Assistant, Siri and Cortana. Proceedings 2019, 31, 51.

2 Mienye, I.D.; Swart, T.G.; Obaido, G. Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications. Information 2024, 15, 517.

3 Staudemeyer, Ralf C. and Eric Rothstein Morris. “Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks.” ArXiv abs/1909.09586 (2019): n. pag.

4 P. Xu, X. Zhu and D. A. Clifton, "Multimodal Learning With Transformers: A Survey," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 10, pp. 12113-12132, Oct. 2023, doi: 10.1109/TPAMI.2023.3275156.

5 V. M. Reddy, T. Vaishnavi and K. P. Kumar, "Speech-to-Text and Text-to-Speech Recognition Using Deep Learning," 2023 2nd International Conference on Edge Computing and Applications (ICECAA), Namakkal, India, 2023, pp. 657-666, doi: 10.1109/ICECAA58104.2023.10212222.

6 Tapia, F.; Mora, M.Á.; Fuertes, W.; Aules, H.; Flores, E.; Toulkeridis, T. From Monolithic Systems to Microservices: A Comparative Study of Performance. Appl. Sci. 2020, 10, 5797.

7 S. Kaiser, M. S. Haq, A. Ş. Tosun and T. Korkmaz, "Container Technologies for ARM Architecture: A Comprehensive Survey of the State-of-the-Art," in IEEE Access, vol. 10, pp. 84853-84881, 2022, doi: 10.1109/ACCESS.2022.3197151.

8 W. Wang, "Research on Using Docker Container Technology to Realize Rapid Deployment Environment on Virtual Machine," 2022 8th Annual International Conference on Network and Information Systems for Computers (ICNISC), Hangzhou, China, 2022, pp. 541-544, doi: 10.1109/ICNISC57059.2022.00112.

9 Mondal, S.K.; Zheng, Z.; Cheng, Y. On the Optimization of Kubernetes toward the Enhancement of Cloud Computing. Mathematics 2024, 12, 2476.

10 R. Li, J. Yin and H. Zhu, "Modeling and Analysis of RabbitMQ Using UPPAAL," 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Guangzhou, China, 2020, pp. 79-86, doi: 10.1109/TrustCom50675.2020.00024.

11 Rathore, M.; Bagui, S.S. MongoDB: Meeting the Dynamic Needs of Modern Applications. Encyclopedia 2024, 4, 1433-1453. https://doi.org/10.3390/encyclopedia4040093

12 Bain, M., et al. WhisperX: Time-Accurate Speech Transcription of Long-Form Audio. International Speech Communication Association, 2023, pp. 4489–93.

13 P. Zhang, L. Xing, N. Yang, G. Tan, Q. Liu and C. Zhang, "Redis++: A High Performance In-Memory Database Based on Segmented Memory Management and Two-Level Hash Index," 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), Melbourne, VIC, Australia, 2018, pp. 840-847, doi: 10.1109/BDCloud.2018.00125.

14 D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesel, “The Kaldi speech recognition toolkit,” in Proceedings of the IEEE 2011 Workshop on Automatic Speech Recognition and Understanding, 2011

15 Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper Deep Speech 2: End-to-End Speech Recognition in English and Mandarin arXiv:1512.02595 Tue, 8 Dec 2015

16 Murge, Lokesh & Sampangi, Sandhya. (2024). REST AS A WEB ARCHITECTURAL DESIGN 1. International Journal of Engineering Technology and Management Sciences. 2. 461-464.

17 Necula, Sabina. (2024). Exploring The Model-View-Controller (MVC) Architecture: A Broad Analysis of Market and Technological Applications. 10.20944/preprints202404.1860.v1.

18 Zhang, Y.; Wang, S.; Liu, Y.; Li, X. "A Survey on Federated Learning: Concept and Applications." IEEE Transactions on Neural Networks and Learning Systems, vol. 32, no. 5, pp. 2125-2141, May 2021, doi: 10.1109/TNNLS.2020.3019625.

19 Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D.M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; Amodei, D. "Language Models are Few-Shot Learners." arXiv:2005.14165, 2020.

20 Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. "Attention is All You Need." Advances in Neural Information Processing Systems, vol. 30, 2017.

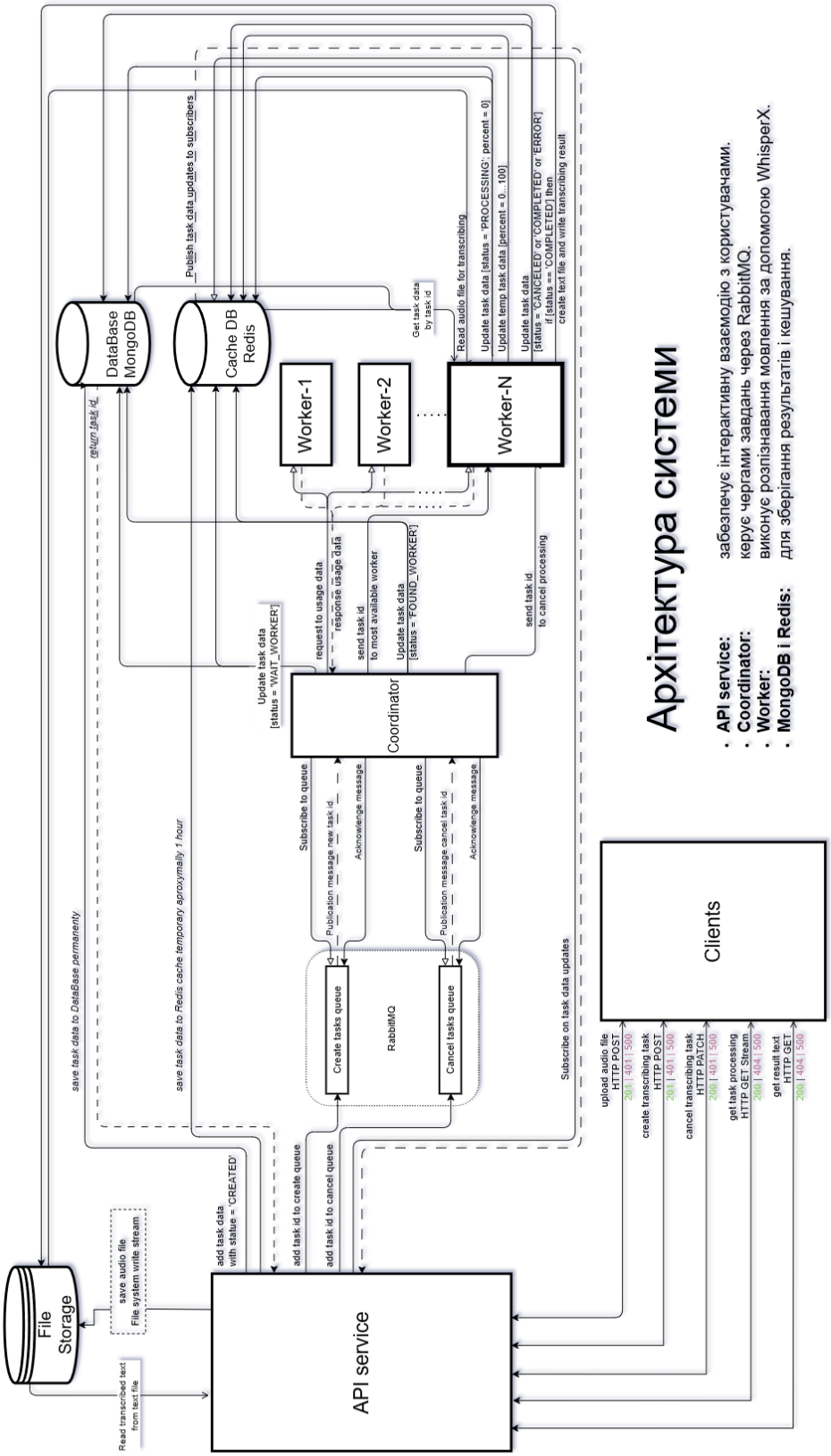
21 He, K.; Zhang, X.; Ren, S.; Sun, J. "Deep Residual Learning for Image Recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778.

22 Kingma, D.P.; Welling, M. "Auto-Encoding Variational Bayes." arXiv:1312.6114, 2013.

23 Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. "Generative Adversarial Nets." Advances in Neural Information Processing Systems, vol. 27, 2014.

ДОДАТКИ

Додаток А. Загальна схема функціонування застосунку та взаємодія між її компонентами



ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка мікросервісної архітектури для масштабованої системи розпізнавання мови із використанням Docker і RabbitMQ»

Виконав: здобувач вищої освіти гр. ІСДМ-63
Андрій БАЧКОВ

Керівник: Андрій БОНДАРЧУК
д.т.н., професор

Київ, 2024

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Актуальність розпізнавання мовлення у сучасному світі

1. Зростання аудіоконтенту
2. Економія часу
3. Підвищення доступності
4. Інтеграція в бізнес

Метою роботи є розробка та аналіз мікросервісної архітектури для масштабованої системи розпізнавання мови, яка використовує Docker для контейнеризації та RabbitMQ для забезпечення ефективного обміну повідомленнями між компонентами системи.

Об’єктом дослідження є система розпізнавання мови, яка побудована на основі мікросервісної архітектури.

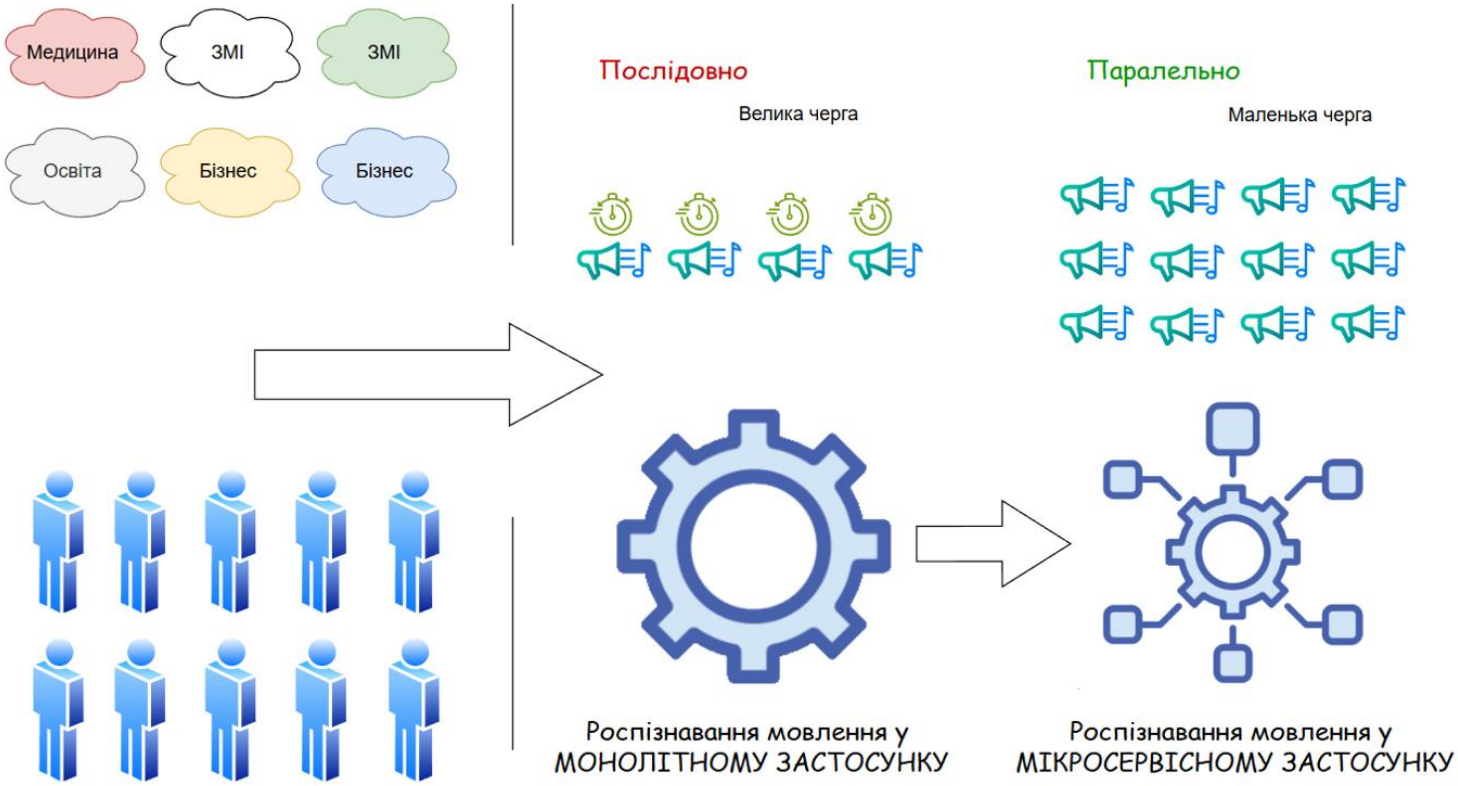
Предметом дослідження є процеси проектування, розробки та масштабування системи розпізнавання мови за допомогою Docker та RabbitMQ.

Основні завдання:

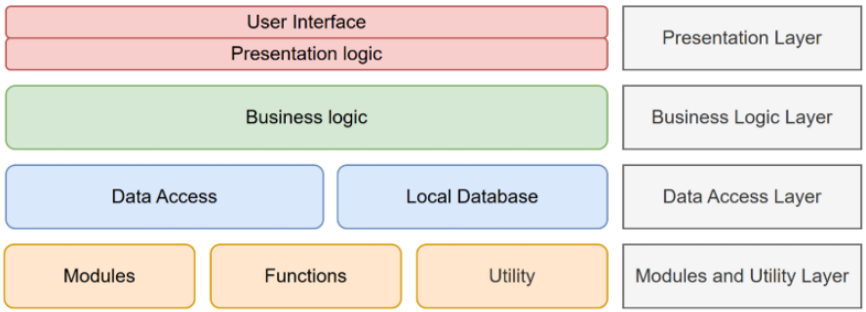
- Виявити недоліки та обмеження існуючих систем;
- Проаналізувати сучасні підходи та технології;
- Вибрати інструменти та технології для розробки системи;
- Розробити систему розпізнавання мовлення на основі мікросервісної архітектури;
- Провести оцінку ефективності та економічної доцільності системи;
- Визначити її переваги та недоліки у порівнянні з хмарними рішеннями.

Проблема ПОВІЛЬНОГО розпізнавання мовлення

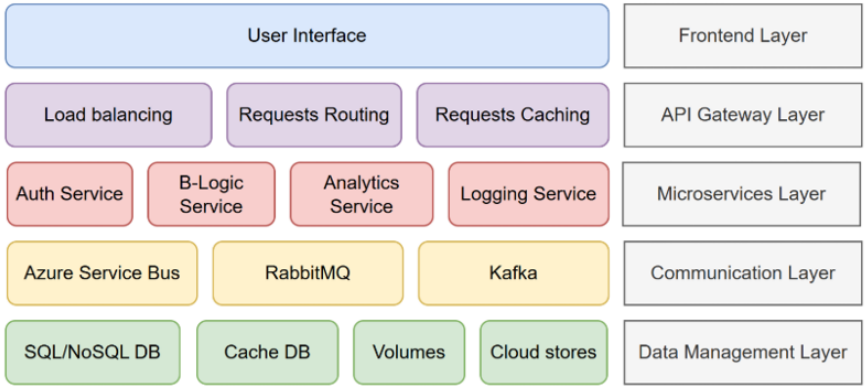
в монолітних застосунках без маштабування задач розпізнавання



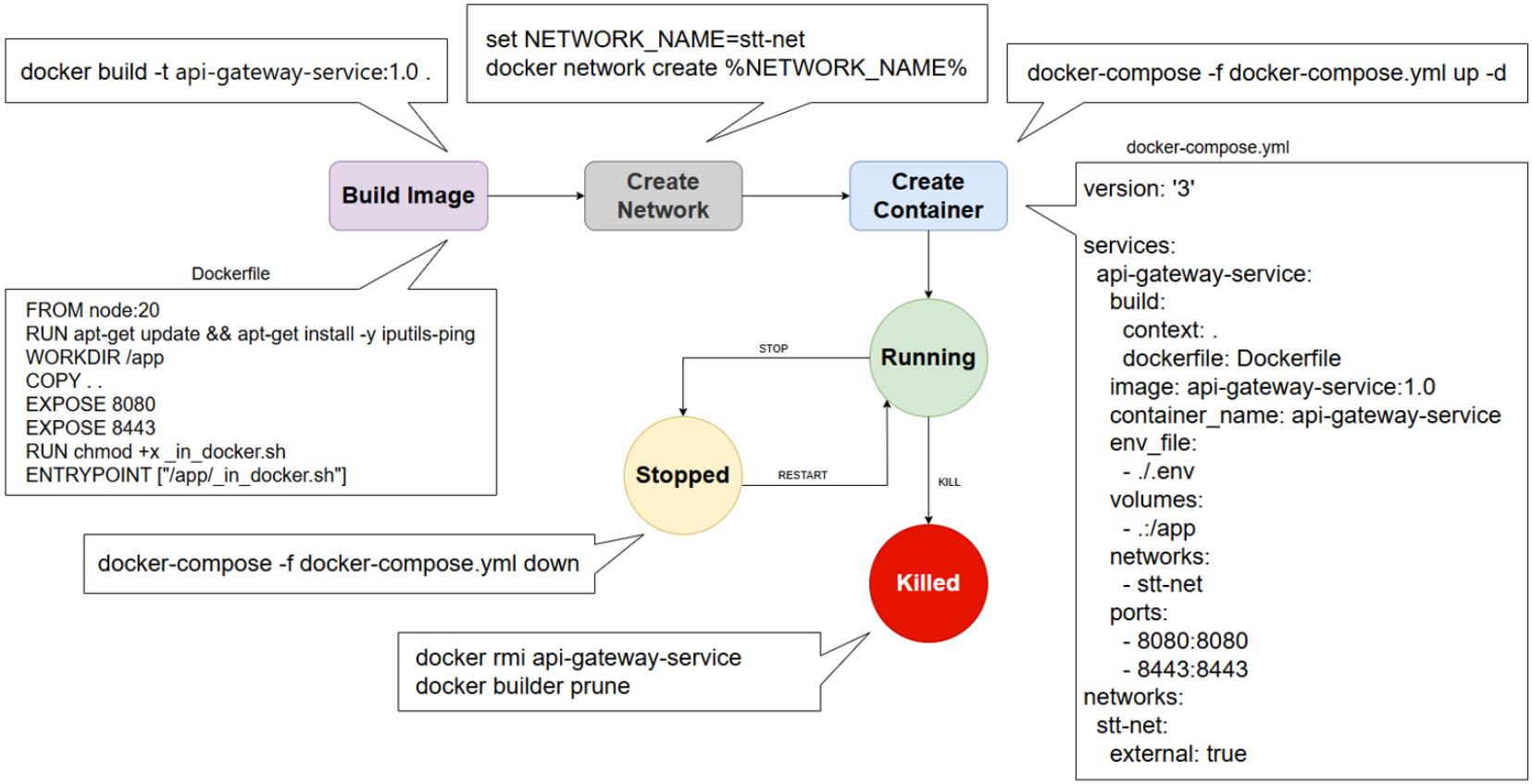
Монолітна архітектура

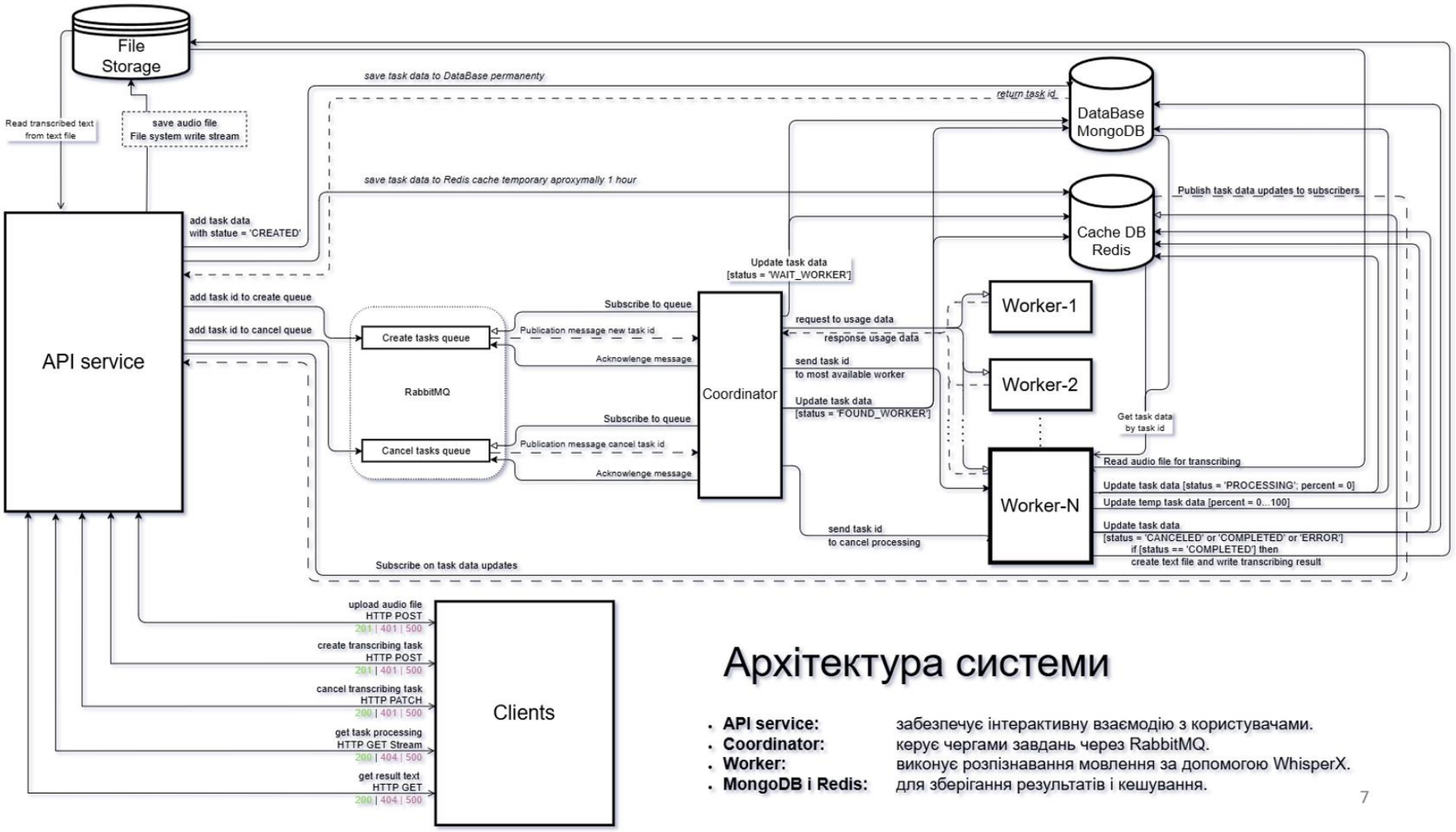


Мікросервісна архітектура



Життевий цикл Docker контейнера



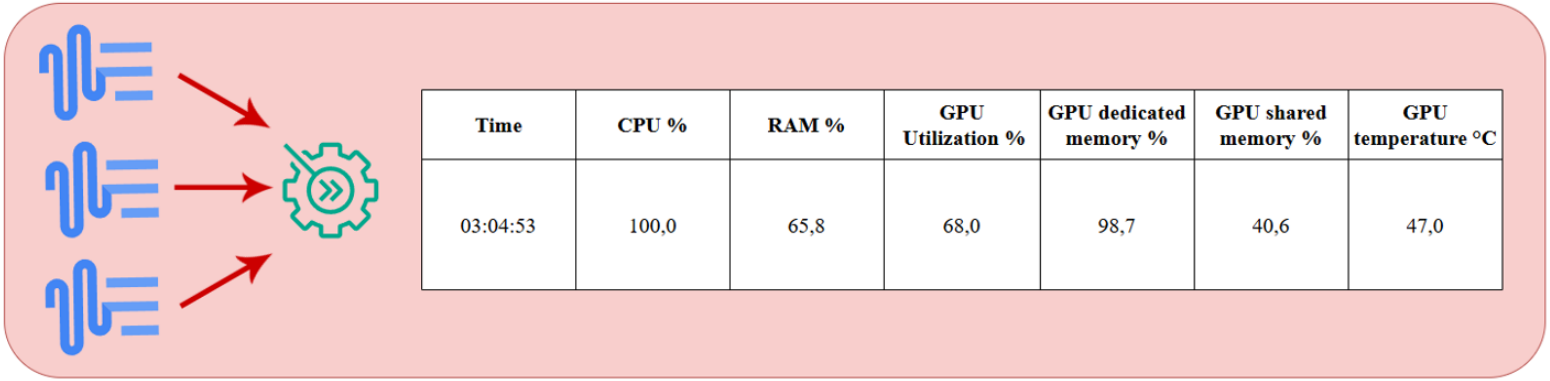


Архітектура системи

- **API service:** забезпечує інтерактивну взаємодію з користувачами.
- **Coordinator:** керує чергами завдань через RabbitMQ.
- **Worker:** виконує розпізнавання мовлення за допомогою WhisperX.
- **MongoDB і Redis:** для зберігання результатів і кешування.

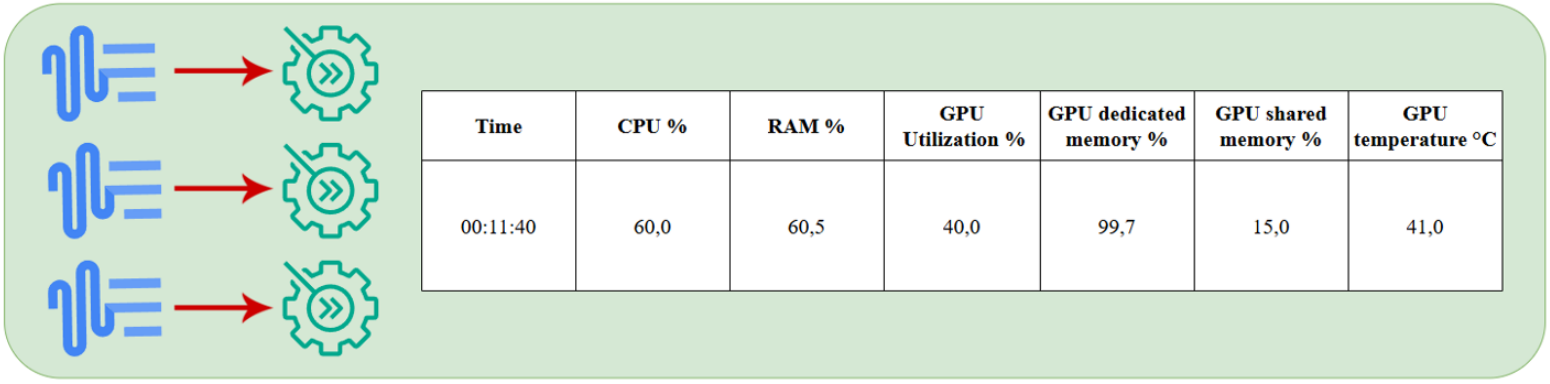
Тестування продуктивності

Тест 1: ***Одночасно*** *три аудіо файли на* ***одному*** *Worker-і*



Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
03:04:53	100,0	65,8	68,0	98,7	40,6	47,0

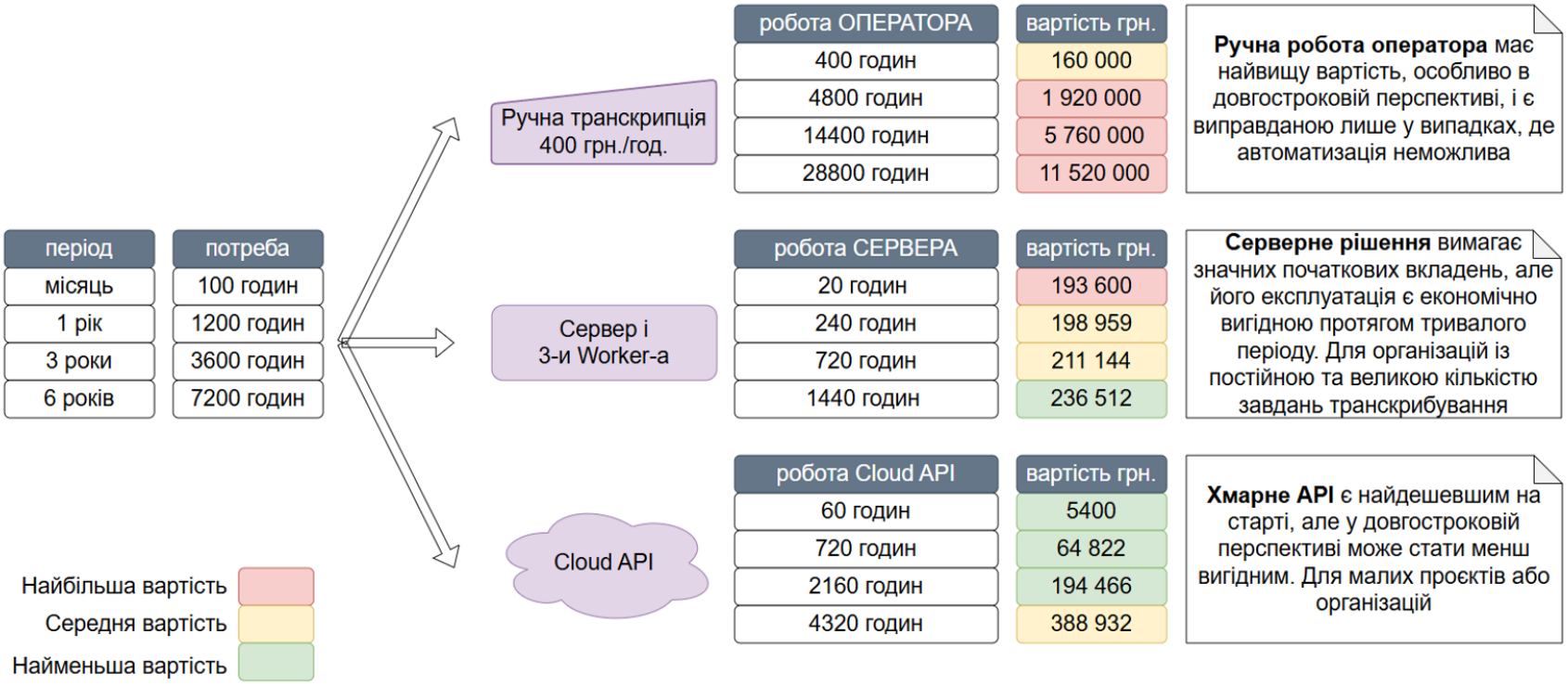
Тест 2: ***Одночасно*** *три аудіо файли на* ***трьох*** *Worker-ах*



Time	CPU %	RAM %	GPU Utilization %	GPU dedicated memory %	GPU shared memory %	GPU temperature °C
00:11:40	60,0	60,5	40,0	99,7	15,0	41,0

~ x14р. швидше!

Оцінка ефективності та економічної доцільності



Висновки

Ефективність запропонованої архітектури

Розроблена мікросервісна архітектура демонструє високу продуктивність і стабільність при виконанні задач розпізнавання мовлення. Використання контейнеризації та RabbitMQ забезпечує гнучкість у масштабуванні системи.

Вибір оптимальних технологій

Інтеграція WhisperX дозволила досягти високої точності розпізнавання мовлення, зберігаючи низькі затрати часу на обробку даних. Використання MongoDB і Redis ефективно забезпечило зберігання та кешування даних, а Node.js дозволив реалізувати зручний API.

Економічна доцільність

Система вимагає значно менших фінансових витрат у порівнянні з оплатою праці операторів чи використанням сторонніх хмарних сервісів. Програмне забезпечення має високий рівень окупності при тривалому використанні.

Практичне значення роботи

Розроблена система дозволяє автоматизувати процеси розпізнавання мовлення, що значно зменшує витрати часу та людських ресурсів. Вона може бути використана як основа для створення комерційних продуктів або дослідницьких проектів.

Дякую за увагу!