**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА**
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

«До захисту допущено»
Завідувач кафедри
В. М. Терещенко _______________
(підпис)

«__»_____________ 20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

РОЗРОБКА СИСТЕМИ РОЗМІТКИ АУДІО ФАЙЛІВ

Виконала студентка 4 курсу
Мирченко Аліна Георгіївна ____________
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Деревянченко Олександр Валерійович ____________
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент ____________
(підпис)

Київ – 2021

РЕФЕРАТ

Обсяг роботи 50 сторінок, 4 ілюстрації, 19 джерел посилань, 1 додаток.

ХМАРНІ ТЕХНОЛОГІЇ, ОБРОБКА АУДІО-ФАЙЛІВ, МАШИННЕ НАВЧАННЯ, РОЗПІЗНАВАННЯ МОВИ, ОБРОБКА ПРИРОДНІХ ТЕКСТІВ, ТЕГУВАННЯ АУДІО-ФАЙЛІВ, РОЗРОБКА СЕРВЕРУ, ТЕХНІЧНЕ ЗАВДАННЯ ДО ПРОДУКТУ.

Об’єктом роботи є процес аналізу аудіо- та відео-доріжок за допомогою програмного засобу «Застосунок для тегування аудіо файлів». Предметом роботи є програмний засіб для тегування аудіо- та відео-доріжок.

Метою роботи є створення програмного засобу для тегування аудіо- та відео-доріжок.

Методи розроблення: методи машинного навчання та інтелекту для розпізнавання мовлення та природніх текстів, навчання моделей машинного інтелекту, розробка програмного продукту на основі хмарних технологій. Інструменти розроблення: крос-платформовий безкоштовний редактор коду Visual Studio Code 1.49.1, мова програмування TypeScript з застосуванням платформи Node.js.

Результати роботи: виконано загальний огляд електронних засобів аналізу живого мовлення та формування тегів на основі природної мови, проаналізовано переваги та недоліки використання електронних засобів у процесі аналізу мовлення, розроблено програмний продукт «Застосунок для тегування аудіо файлів», який дозволяє завантажувати та аналізувати аудіо- та відео-файли.

За методами розробки та інструментальними засобами робота виконувалася сумісно з роботами з розв’язання задач навчання моделей машинного інтелекту.

Програмний продукт «Застосунок для тегування аудіо файлів» може застосовуватися в особистих цілях задля розпізнавання контенту аудіо- та відео-файлів та в подальшому для спрощення організації файлів та їх пошуку.

ЗМІСТ

ЗМІСТ 4
ВСТУП 5
РОЗДІЛ 1 ТЕОРЕТИЧНІ ЗАСАДИ ТЕГУВАННЯ АУДІО- ТА ВІДЕО-ФАЙЛІВ 8
1.1 Сутність, методи та форми тегування метаданих 8
1.2 Метадані у аудіо-файлах 10
1.3 Розпізнавання мови у аудіо-файлах 12
1.4 Методи обробки природномовних текстів 14
РОЗДІЛ 2 МОЖЛИВОСТІ ХМАРНИХ СЕРВІСІВ ДЛЯ РОЗПІЗНАВАННЯ ДАНИХ 17
2.1 Сервіси Amazon для розпізнавання мови 17
2.2 IBM Watson Natural Language Understanding 19
РОЗДІЛ 3 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РЕАЛІЗАЦІЇ 22
3.1 Visual Studio Code 22
3.2 TypeScript 23
3.3 Node.js 24
3.3 Docker 25
РОЗДІЛ 4 ОПИС ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 28
4.1 Архітектура серверної частини системи 28
4.2 Завантаження та первинний аналіз файлу 29
4.3 Розпізнавання мови та генерація транскрипції 32
4.4 Генерація тегів 35
4.5 Приклад роботи програми 37
ВИСНОВКИ 41
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 43
ДОДАТОК А Лістинг коду для роботи з чергами 46

ВСТУП

Оцінка сучасного стану об’єкту розробки. Із зростанням інформатизації у сучасному світі, люди все більше стикаються із проблемою роботи з даними. Наразі хмарні сховища здобувають все більшу популярність, а організація файлів в них є нагальною проблемою. Зростаючий попит на зберігання та аналіз персональних файлів створюють необхідність розробки покращених інструментів для обробки даних та отримання з них інформації, що в подальшому може використовуватись комп’ютерами в більш складних алгоритмах задля оптимізації пошуку, надання релевантного контенту та фільтрації файлів.

Зростання тенденції до споживання людьми аудіо- та відео-контенту спонукає до пошуку можливостей для роботи з даними такого типу та застосування отриманих даних для надання кардинально нових продуктів, що полегшать доступ до даних для користувачів та дозволять з легкістю ними маніпулювати.

Низка програмних продуктів надає можливість аналізувати дані та обробляти отриману інформацію, але на ринку немає таких продуктів, які б комбінували в собі подібний функціонал. Однак, є деякі сервіси, що здатні транскрибувати аудіо- та відео-файли без можливості їх тегування.

Актуальність роботи та підстави для її виконання. Наразі у світі спостерігається тенденція до використання великої кількості сервісів, що дозволяють користувачам завантажувати та продивлятись або прослуховувати відео- та аудіо-контент. Дані сервіси можуть мати розважальний характер, використовуватись для навчання (зокрема, для дистанційного навчання у період пандемії), роботи або ж у власних цілях для зберігання персональних файлів. Таким чином, обробка саме аудіо- та відео-контенту, зокрема його вмісту є неймовірно актуальною темою.

Доцільно аналізувати не лише метадані файлів, а й безпосередньо їх вміст. Таким чином можна обробити транскрипцію аудіо та протегувати її. Утворені теги можна застосовувати для визначення тематики файлу, що в подальшому можна використати для швидкого пошуку за вмістом.

Мета й завдання роботи. Метою дипломної роботи є створення програмного засобу для аналізу та тегування аудіо-доріжок. Для досягнення цієї мети поставлено такі завдання:

- Дослідити існуючі засоби аналізу та тегування аудіо-доріжок.
- Дослідити застосування різних способів обробки аудіо-файлів, генерації транскрипції та аналізу природної мови.
- Розробити технічне завдання до продукту.
- Розробити програмний застосунок.

Об’єкт, методи й засоби розроблення. Об’єктом розроблення програмного засобу «Застосунок для тегування аудіо файлів» є процес обробки та аналізу вмісту аудіо-файлів за допомогою програмного засобу.

Під час розробки програмного продукту було використано ітераційну модель життєвого циклу. Проект було розбито на логічні етапи, кожен з яких виконувався у певний термін та мав готові результати, що можна представити кінцевому користувачеві. На кожній ітерації було проаналізовано вимоги, розроблено проектування, виконано реалізацію функціоналу та протестовано його. Таким чином це дозволило уточнювати вимоги до функціоналу на кожному кроці ітерації та вдосконалювати програмний продукт в тісній взаємодії із кінцевими користувачами, валідуючи коректність роботи та зручність у використанні.

В якості інструменту створення програмного засобу було обрано Visual Studio Code 1.49.1 – крос-платформовий безкоштовний редактор коду.

Програмний застосунок було написано мовою TypeScript та з використанням платформи Node.js. TypeScript є зворотньо сумісним з JavaScript, який в свою чергу є динамічною, об'єктно-орієнтованою прототипною мовою програмування.

Фактично, після компіляції програму на TypeScript можна використовувати спільно з серверною платформою Node.js. [1]

Можливі сфери застосування. Програмний продукт може застосовуватися у системах організації файлів задля спрощення роботи з аудіо- та відео-файлами, зокрема для навігації по файлах, їх пошуку, сортування та фільтрації. Також даний інструмент може бути корисним для надання користувачам релевантної інформації на основі вмісту файлів за належного використання метаданих, сформованих у результаті роботи програми.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ ТЕГУВАННЯ АУДІО- ТА ВІДЕО-ФАЙЛІВ

1.1 Сутність, методи та форми тегування метаданих

Метадані мають різні цілі. Вони допомагають користувачам знаходити відповідну інформацію та ресурси. Вони також допомагають організувати електронні ресурси, забезпечити цифрову ідентифікацію, а також архівувати та зберігати ресурси. Метадані дозволяють користувачам отримувати доступ до ресурсів, дозволяючи знаходити ресурси за відповідними критеріями, ідентифікуючи ресурси, об'єднуючи подібні ресурси, виділяючи різнорідні ресурси та надаючи інформацію про місцезнаходження.

Метадані можуть зберігатися як внутрішньо, в тому ж файлі або структурі, що й дані (це також називається вбудованими метаданими), так і зовні, в окремому файлі чи полі з описаних даних. У сховищі даних зазвичай зберігаються метадані, відірвані від даних, але можуть бути розроблені для підтримки підходів до вбудованих метаданих. Кожен варіант має переваги та недоліки.

Внутрішня пам’ять означає, що метадані завжди переміщуються як частина даних, які вони описують; таким чином, метадані завжди доступні з даними та ними можна керувати локально. Цей метод створює надмірність (виключаючи нормалізацію) і не дозволяє керувати всіма метаданими системи в одному місці. Це, можливо, підвищує узгодженість, оскільки метадані легко змінюються щоразу, коли змінюються дані.

Зовнішнє сховище дозволяє розміщувати метадані для всього вмісту, наприклад, у базі даних, для більш ефективного пошуку та управління. Надлишку можна уникнути, нормалізувавши організацію метаданих. У цьому підході метадані можуть бути об’єднані із вмістом при передачі інформації, наприклад у потоковому носії; або на нього можна посилатися (наприклад, як веб-посилання) із переданого вмісту. З іншого боку, поділ метаданих із вмістом даних, особливо в автономних файлах, які посилаються на свої вихідні метадані в іншому місці,

збільшує можливості для невідповідності між ними, оскільки зміни обох можуть не відображатися в іншому.

Метадані можуть зберігатися як у зручному для читання, так і в двійковому вигляді. Зберігання метаданих у зручному для читання форматі, такому як XML, може бути корисним, оскільки користувачі можуть розуміти та редагувати їх без спеціальних інструментів. Однак текстові формати рідко оптимізуються під ємність зберігання, час комунікації або швидкість обробки. Формат двійкових метаданих забезпечує ефективність у всіх цих аспектах, але вимагає спеціального програмного забезпечення для перетворення двійкової інформації в зручний для читання вміст [2].

В інформаційних системах тег – це ключове слово або термін, присвоєний частині інформації (наприклад, Інтернет-закладці, цифровому зображенню, запису бази даних або комп'ютерному файлу). Цей вид метаданих допомагає описати елемент і дозволяє його знову знайти за допомогою перегляду або пошуку. Зазвичай теги вибираються неформально, залежно від системи, хоча їх також можна вибрати з контрольованого словника.

Люди використовують теги для класифікації, позначення права власності, позначення обмежень та для означення онлайн-ідентичності. Теги можуть мати форму слів, зображень або інших ідентифікаційних знаків. Аналогічним прикладом міток у фізичному світі є позначення музейних предметів. Люди використовували текстові ключові слова для класифікації інформації та об’єктів задовго до комп’ютерів. Комп’ютерні алгоритми пошуку зробили використання таких ключових слів швидким способом вивчення записів.

Позначення тегів набуло популярності завдяки зростанню соціальних закладок, обміну зображеннями та веб-сайтів у соціальних мережах. Ці веб-сайти дозволяють користувачам створювати та керувати мітками (або "тегами"), які класифікують вміст за допомогою простих ключових слів. Веб-сайти, що містять теги, часто відображають колекції тегів як хмари тегів, як це роблять деякі настільні

програми. На веб-сайтах, які об'єднують теги всіх користувачів, теги окремого користувача можуть бути корисними як для них, так і для більшої спільноти користувачів веб-сайту.

У типовій системі тегування немає явної інформації про значення або семантику кожного тегу, і користувач може застосовувати нові теги до елемента так само просто, як застосовувати старі теги. Ієрархічні класифікаційні системи можуть повільно змінюватися і коренятся в культурі та епосі, яка їх створила; на відміну від цього, гнучкість тегування дозволяє користувачам класифікувати свої колекції тегів у спосіб, який їм здається корисним, але персоналізоване різноманіття термінів може створювати проблеми під час пошуку та перегляду [3].

1.2 Метадані у аудіо-файлах

Музичні метадані – це ідентифікаційні дані, вбудовані в музичний файл, що складаються з сотень текстових мета-тегів, які приєднуються через контейнери метаданих (ID3v1 та ID3v2 на сьогоднішній день є найпоширенішими), де детально описується все, від імені виконавця та дати виходу пісні до пов'язаних жанрів та авторів пісень.

ID3 – це контейнер метаданих, який найчастіше використовується разом із форматом аудіофайлів MP3. Це дозволяє зберігати інформацію, таку як заголовок, виконавець, альбом, номер доріжки та іншу інформацію про файл, у самому файлі.

Існує дві не пов'язані між собою версії ID3: ID3v1 та ID3v2. ID3v1 приймає форму 128-байтового сегмента в кінці MP3-файлу, що містить фіксований набір полів даних. ID3v1.1 – це невелика модифікація, яка додає поле "номер доріжки" за рахунок невеликого скорочення поля "коментар". ID3v2 структурно сильно відрізняється від ID3v1, що складається з розширюваного набору "кадрів", розташованих на початку файлу, кожен з ідентифікатором кадру (три- або чотирибайтовий рядок) та однієї частини даних. 83 типи кадрів задекларовані в

специфікації ID3v2.4, і програми також можуть визначати свої власні типи. Існують стандартні рамки, що містять обкладинки, BPM, авторські права та ліцензії, тексти пісень та довільний текст та дані URL, а також інші речі. Було задокументовано три версії ID3v2, кожна з яких розширила визначення кадру.

ID3 є фактичним стандартом для метаданих у файлах MP3. Хоч ID3 розроблявся для MP3, розробникам немає перешкод вбудовувати його також і в інші формати [4].

Що стосується звукових файлів, окрім технічної інформації, такої як тип файлу, частота дискретизації та формат, описові метадані також містять цінну інформацію щодо вмісту файлу, таку як детальний опис та ключові слова. Обидва з них є важливими для отримання успішних результатів під час пошуку в бібліотеці певного звуку.

Зазвичай імена файлів є відносно короткими, тому вони не дають достатньо детальної інформації, щоб ми могли отримати ефективні результати пошуку. Перевага метаданих полягає в тому, що вони можуть містити всю детальну інформацію про характеристики файлу, внутрішнє наповнення та багато іншого.

Можливість ефективно звузити пошук до детального рівня має вирішальне значення для збору елементів, що використовуються в звуковому дизайні, завдяки можливості ідентифікувати звуки, що відповідають конкретним об’єктам, як приклад – можливість визначити звукові якості, періоди часу та локацію, наприклад місто чи країну.

У випадку аудіофайлів, метадані вбудовуються у файл, а не зберігаються як зовнішній файл, як можна бачити у файлах .xml для файлів зображень, тому ними дуже легко керувати. Залежно від формату файлу, можна автоматично вставляти у них метадані. У разі власних записів або некомерційних бібліотек описові метадані можуть бути відсутні частково або повністю, і необхідно вводити інформацію вручну.

Для запису місцезнаходження можна використовувати метадані, що включають країну, місто, дату та час, годину пік, відстань, технічні деталі. Поля ключових слів можуть бути використані для асоціювання звуків з категоріями, тому можна отримати широкий результату пошуку за їх допомогою [5].

1.3 Розпізнавання мови у аудіо-файлах

Завданням розпізнавання мови є відновлення після звукового сигналу слів природної мови, відтворенням яких є цей звуковий сигнал. Цю проблему зазвичай можна вирішити, встановивши в словнику слово стандарт, а потім порівнявши звуковий сигнал із цими зразками. Звуковий сигнал видає себе цілим вектором значень звукового тиску, виміряних в рівновіддалені моменти. Слід зазначити, що потужність простору звукового сигналу дуже велика, тому, щоб вирішити проблему розпізнавання, сигнал спочатку поділяється на вікна однакової довжини рівномірно. Вікно перетворюється з часової області в частотну, так що близькість вікна до відносно простої метрики відповідає близькості "вушної" сигнальної частини. Після цього вирішується проблема пошуку відповідності між вікном звукового сигналу та стандартним вікном словникового слова. Складність останнього завдання полягає в тому, що в різних висловлюваннях одного і того ж слова різні частини звукового сигналу мають різну ступінь стиснення або розтягування (не пропорційні) [6]. Метод динамічного програмування використовується для вирішення проблеми пошуку відповідності між вікнами сигналів.

Створення комп'ютерних систем для розпізнавання мови пов'язане з багатьма об'єктивними труднощами, які накладають багато обмежень на такі системи штучного інтелекту. Максимально допустима здатність комп’ютерів розпізнавати мови в основному пов’язана з тим, що люди, яких можна розглядати як стандарти, можуть розпізнавати осмислену мову, на що комп’ютери наразі не здатні.

Комп’ютери не можуть використовувати синтаксичні та семантичні зв’язки слів речень для виправлення помилок розпізнавання та двозначності з необхідною надійністю. Натомість сучасні системи використовують статистичні моделі для визначення взаємозв’язку між послідовними трійками слів речень. У більшості випадків людина буде використовувати іншу незвукову інформацію [7].

Як правило, для встановлення стандарту голосу використовують статистичні методи, які передбачають, що акустичні параметри фонем присвоюються відповідно до звичайних правил. Насправді картина набагато складніша, і це призводить до того, що точні моделі звукових та словесних стандартів повинні містити багато опорних елементів. Крім того, через те, що всі відомі алгоритми розпізнавання мови покладаються на дикторів, картина ускладнюється. Після налаштування на диктора, система розпізнавання забезпечить задовільні результати розпізнавання для цього типу голосу, але результати будуть значно гіршими для інших голосів. З іншого боку, надійність розпізнавання мовлення людини не залежить від типу мовлення оратора [8].

Розробляючи комп’ютерні системи розпізнавання мовлення, фахівці стикаються з проблемою приглушення стаціонарних та нестаціонарних перешкод. Існуючі різні системи голосового управління та комп’ютерного диктування тексту не використовують алгоритми шумозаглушення у своїй роботі. Цей факт пов’язаний з тим, що комп’ютерні мовні системи найчастіше використовуються в будинках чи офісах, де рівень зовнішніх перешкод не високий. Однак шумозаглушення також використовується в системах голосового управління для такого рівня техніки, як авіація. Враховуючи, що відсутність шумозаглушення в мовних системах комп’ютера вплине на відсоток помилок розпізнавання (наприклад, якщо користувач голосно видихає, система завжди розпізнає це як одне зі слів у словнику і навіть намагається врахувати контекст), можна очікувати, що найближчим часом у майбутньому алгоритми та обладнання для зменшення шуму будуть встановлені на персональні комп’ютери.

1.4 Методи обробки природномовних текстів

Обробка природної мови або Natural language processing (NLP) – це напрямок науки, що виник в зв'язку з розвитком комп'ютерних технологій, особливо штучного інтелекту, метою якого є вивчення та розуміння штучним інтелектом і іншими програмами природної мови людини, для більш ефективної взаємодії штучного інтелекту і людини. Вона є міждисциплінарним напрямком, що знаходиться на стику лінгвістики, комп'ютерних наук та штучного інтелекту. Будь-яка мова, що використовується людиною, вважається природною мовою. При цьому мова приймає різну структуру, такі як письмова мова, усне мовлення, навіть жести.

Більшість сфер діяльності людини характеризується своєю особливою термінологічною, стилістичною і т.п. специфікою, так званим професійним сленгом. Можливість обліку такої специфіки дозволяє поліпшити взаємодію програм штучного інтелекту і фахівців певної сфери. Дана проблема вирішується навчанням штучного інтелекту за допомогою спеціально підібраної бази даних.

Для створення такої бази даних використовуються різні програмні забезпечення, такі як NLTK (Natural language Toolkit), GATE (General Architecture for Text Engineering), Apache OpenNLP, тощо. Текстові корпуси в основному являють собою тексти на природній мові без структурування та обробки. NLP виконують за допомогою закономірностей, які виявлені в лінгвістичних властивостях текстів. При цьому лінгвістичний аналіз тексту визначає важливі особливості тексту.

Мовні процесори виконують обробку великої кількості слів, багато з яких є багатозначними. Мову можна представити у вигляді набору символів. Об'єднані символи застосовуються для трансляції та передачі інформації. Обробка природної мови складається з чотирьох етапів. У реальності ці кроки рідко є послідовними і

окремими етапами. Іноді деякі етапи відсутні або можуть об'єднуватися, а також можуть вводитися додаткові етапи обробки:

- Токенізація та сегментація – розділення тексту на лексеми (слова) та речення. Відбувається перехід від символів до речень і слів.
- Морфологічний аналіз – аналіз форм слів та пошук їх міток. Відбувається перехід до основ слів.
- Синатксичний аналіз – аналіз взаємозв'язку між структурою та словами.
- Семантичний та прагматичний аналіз – аналіз семантичних компонентів тексту.

Тегування частин мови – одне з основних завдань при обробці природної мови. Позначення тегом означає присвоєння кожному слову речення одного з можливих лексичних категорій відповідно до контексту, в якому вживається це слово. Наприклад, це слово може бути іменником, допоміжним дієсловом або перехідним дієсловом. Категорія, присвоєна слову, визначатиме структуру речення в якому він з'являється і, отже, його значення. Насправді тегування є необхідним кроком для синтаксичного аналізу, для систем пошуку інформації, для розпізнавання мови тощо. Більше того, позначення є складною проблемою, оскільки багато слів належать до більш ніж одного лексичного класу.

Через важливість і складність цього завдання багато роботи проводиться для отримання автоматичних міток. Автоматичні розмітки, створюються як правило на основі прихованих моделей Маркова та спираються на статистичну інформацію для встановлення ймовірності кожного сценарію. Статистичні дані витягуються з попередньо позначених текстів, що називаються корпусом. Ці стохастичні мітки не вимагають знань правил мови, а також не намагаються їх вивести, і, отже, вони можуть бути застосовані до текстів будь-якою мовою за умови, що вони були раніше навчені на корпусі цією мовою.

Контекст, у якому з’являється слово, допомагає визначити найбільш відповідний йому тег, і ця ідея є основою для більшості тегів. Але може статися

так, що попереднє слово було неоднозначним, тому може бути багато залежностей, які повинні вирішуватися одночасно [9].

РОЗДІЛ 2
МОЖЛИВОСТІ ХМАРНИХ СЕРВІСІВ ДЛЯ РОЗПІЗНАВАННЯ ДАНИХ

2.1 Сервіси Amazon для розпізнавання мови

Amazon Web Services або AWS є дочірньою компанією Amazon.com, яка забезпечує платформу хмарних обчислень для оренди приватним особам, компаніям та урядам за підпискою. Існує також безкоштовна передплата, якою можна скористатися протягом перших 12 місяців. Ця технологія дозволяє абонентам мати повний кластер віртуальних комп’ютерів, який завжди доступний через Інтернет. Віртуальна машина AWS має більшість атрибутів реального комп'ютера, включаючи апаратні пристрої (процесор, відеокарта, локальна та оперативна пам'ять, жорсткий диск або твердотільний диск); додаткова операційна система; мережа; і попередньо встановлені програми, такі як веб-сервер, база даних , CRM тощо.

Технологія AWS базується на кластерах серверів (фермах) по всьому світу. Плата за використання базується на використанні апаратного забезпечення, операційної системи, програмного забезпечення, функції мережі, обраного користувачем, а також вимог щодо доступності, надмірності, безпеки та інших параметрів. Відповідно до потреб користувача та сплачених коштів, він може зарезервувати віртуальний комп’ютер (VM), кластер віртуальних комп’ютерів (кластер VM), фізичний (реальний) комп’ютер (сервер), виділений йому, або навіть фізичний комп'ютер (кластер серверів). Amazon прагне керувати та оновлювати програмне та апаратне забезпечення, щоб відповідати необхідним стандартам безпеки [10].

AWS Transcribe – це рішення для перетворення мови в текст, яке надає Amazon Web Services, яке, як відомо, є дуже швидким і має високу точність. AWS Transcribe під капотом використовує процес глибокого навчання, що називається ASR (автоматичне розпізнавання мови) для перетворення аудіо в текст швидко і

точніше. Він також має окрему службу всередині назви Amazon Transcribe Medical, яка також може використовуватися для додатків медичної документації [11].

Для транскрипції звукового файлу Amazon Transcribe використовує три операції:

- StartTranscriptionJob – запускає пакетне завдання для транскрипції мови в аудіофайлі в текст.
- ListTranscriptionJobs – повертає список запущених завдань транскрипції. Також можна вказати статус завдань, які потрібно повернути. Наприклад, можна отримати список всіх завдань, що очікують на розгляд, або список виконаних завдань.
- GetTranscriptionJob – повертає результат завдання транскрипції. Відповідь містить посилання на файл JSON, що містить результати.

Щоб транскрибувати потокове аудіо в текст, Amazon Transcribe пропонує одну операцію:

- StartStreamTranscription – запускає двонаправлений потік HTTP / 2, де аудіо передається на Amazon Transcribe та результати транскрипції передаються у програму.

Об’єкт TranscriptionJob описує асинхронну задачу транскрипції, створену за допомогою запиту StartTranscriptionJob. В залежності від стану, об’єкт має одне значення статусу з переліку:

- QUEUED – задача надійшла в чергу;
- IN_PROGRESS – задача оброблюється сервісом;
- FAILED – задача завершилась з помилкою;
- COMPLETED – задача завершилась успішно.

Оскільки час транскрибування кожного окремого файлу може варіюватися, AWS Transcribe не надсилає вихідні дані в тому самому запиті як відповідь. А отже для того, щоб отримати результат необхідно або постійно звертатися до сервісу,

щоб перевірити, чи виконано завдання транскрипції, або потрібно мати якісь тригери подій для ідентифікації статусу завдання.

Сервіс Amazon SNS дозволяє налаштувати та керувати сповіщеннями про зміну статусу нашої задачі. Amazon SNS (Simple Notification Service) – це керована послуга, яка забезпечує доставку повідомлень від виробників (producers) споживачам (consumers). Виробники асинхронно спілкуються зі споживачами, надсилаючи повідомлення на тему, яка є логічною точкою доступу та каналом зв'язку [12].

Сервіс Transcribe видає вихідний документ, що містить транскрибований текст та метаінформацію про кожне транскрибоване слово:

- Час початку запису слова;
- Кінцевий час запису слова;
- Значення довіри до слова.

2.2 IBM Watson Natural Language Understanding

IBM Cloud - це набір хмарних обчислювальних послуг для підприємств, що надаються інформаційними технологіями IBM. [1] Він поєднує платформу як послугу (PaaS) та інфраструктуру як послугу (IaaS). Платформа є масштабованою і підтримує невеликі групи розробників та організацій, а також великі підприємства. Він знаходиться в центрах обробки даних по всьому світу [13].

Одним з продуктів, що розміщені в IBM Cloud є IBM Watson. Уотсон представив 90 серверів IBM p750, кожен з яких оснащений чотирма восьмиядерними процесорами POWER7. Загальний обсяг оперативної пам'яті перевищує 15 ТБ.

Система може отримати доступ до 200 мільйонів сторінок 4 ТБ структурованої та неструктурованої інформації, включаючи повний текст Вікіпедії.

З метою продовження успішного розвитку проекту IBM Watson до 2018 року компанія випустила програмні продукти: Watson Studio (використовується для побудови моделей машинного навчання) та Watson SDK (використовується для доступу до Інтернет-послуг IBM Watson), які можна використовувати для Linux , macOS та Windows. Зокрема, оригінальний фреймворк Core ML, створений Apple для методів машинного навчання, використовується для прискорення роботи в ОС macOS [14].

IBM Watson NLP використовує токенізацію, лематизацію та загальне представлення частини мови для забезпечення таких завдань, як двозначність слів та семантичний аналіз. Інтегровані методи глибокого навчання та машинного навчання постійно вдосконалюють мовні моделі для надання високоякісної інформації. Тематика Watson NLP присвячена інформації про законодавство, ризики та дотримання норм, нафту та газ, маркетинг та навіть спорт.

З розумінням IBM Watson Natural Language розробники можуть аналізувати семантичні особливості введення тексту, включаючи категорії, поняття, емоції, сутності, ключові слова, метадані, взаємозв'язки, семантичні ролі та установки. Використовуючи іншу модель продукту Watson – Knowledge Studio, можна легко налаштувати її для конкретної сфери чи галузі.

Knowledge Studio від IBM Watson використовується для створення моделей машинного навчання, які розуміють конкретні сфери бізнесу, створюють моделі на основі правил та знаходять сутності в документах. Надаючи інструменти для анотування неструктурованої галузевої літератури та на основі цих анотацій можна створити індивідуальну модель машинного навчання, яка розуміє мовні відтінки, значення та взаємозв’язки, характерні для обраної галузі. Після тестування готову модель можна використовувати в інших хмарних сервісах та продуктах Watson [15].

Для роботи з IBM Watson NLU існують спеціальні SDK для ряду мов програмування. Існуючі SDK забезпечують роботу з Watson API. До API викликів

відносяться методи, що надають можливість роботи з аналізом тексту, керування власними моделями, керування сентимент моделями, керування моделями категорій та керування моделями класифікацій [16].

Метод аналізу тексту у відповіді повертає JSON об’єкт, що містить в собі наступні атрибути:

- Концепції – повертає концепції високого рівня у вмісті. Наприклад, стаття про глибоке навчання може повернути поняття "Штучний інтелект", хоча цей термін не згадується.
- Сутності – визначає людей, міста, організації та інші суб’єкти у тексті.
- Ключові слова – повертає важливі ключові слова із тексту.
- Категорії – повертає п’ятирівневу таксономію тексту.
- Класифікації – класифікує текст за допомогою користувацького класифікатора тексту з довільною кількістю класів.
- Емоції – виявляє гнів, огиду, страх, радість або смуток, що передаються у тексті або контексті навколо цільових фраз, зазначених у параметрі цілей.
- Метадані – повертає інформацію з документа, включаючи ім'я автора, заголовок, канали RSS/ATOM, дату публікації. Підтримує лише типи введення URL та HTML.
- Відношення – розпізнає, коли дві сутності пов’язані, і визначає тип відношення.
- Семантичні ролі – розбирає речення за формою підмета, дії та об’єкта.
- Сентимент – аналізує загальний сентимент тексту або сентимент конкретних цільових фраз.
- Синтаксис – повертає інформацію про лексеми та речення у вхідному тексті.

РОЗДІЛ 3
ІНСТРУМЕНТАЛЬНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Система була розроблена за допомогою інтегрованого середовища розробки Visual Studio Code. Додаток створено мовою TypeScript з використанням платформи з відкритим кодом Node.js. Програма запускається та виконується у контейнері Docker. Операційна система комп'ютера Ubuntu 20.04.4 LTS.

3.1 Visual Studio Code

Visual Studio Code – це безкоштовний редактор вихідного коду, створений Microsoft для Windows, Linux та macOS. Особливості включають підтримку налагодження, виділення синтаксису, інтелектуальне заповнення коду, фрагменти коду, рефакторинг коду та вбудований Git. Користувачі можуть змінювати теми, комбінації клавіш, налаштування та встановлювати розширення, щоб додати інші функції.

Visual Studio Code – це редактор вихідного коду, який можна використовувати з багатьма мовами програмування, включаючи Java, JavaScript, Go, Node.js, Python та C ++. Він заснований на фреймворці Electron, який використовується для розробки веб-додатків на Node.js, які можуть працювати на механізмі компонування Blink.

Користувачам дозволено відкривати один або кілька каталогів замість системи проектів, які потім можна зберегти в робочій області для подальшого повторного використання. Таким чином, він може діяти як редактор коду, незалежний від будь-якої мови. Він підтримує кілька мов програмування та набір функцій, що відрізняються залежно від мови. За допомогою налаштувань можна видалити непотрібні файли та папки з дерева проекту. Багато функцій Visual Studio

Code недоступні через меню або користувальницький інтерфейс, але доступ до них здійснюється через панель команд.

Код Visual Studio можна розширити за допомогою розширень, які можна отримати через центральне сховище. Сюди входять доповнення до редактора та підтримка мови. Примітною особливістю є можливість створення розширень для додавання підтримки нових мов, тем та налагоджувачів, проведення статичного аналізу коду та додавання файлів коду за допомогою протоколу Language Server.

Visual Studio Code включає декілька розширень до FTP, що дозволяє використовувати програмне забезпечення як безкоштовну альтернативу веб-розробці. Код можна синхронізувати між редактором та сервером без завантаження іншого програмного забезпечення.

Visual Studio Code дозволяє користувачам встановлювати кодову сторінку мови програмування, що використовується для зберігання активних документів, розривів рядків та активних документів. Це дозволяє використовувати його на будь-якій платформі, будь-якій мові та будь-якій даній мові програмування [17].

3.2 TypeScript

TypeScript – це мова програмування, розроблена та підтримувана корпорацією Майкрософт. Це суворий синтаксичний додаток до JavaScript, який додає до мови необов’язкову статичну типізацію. TypeScript надає можливість розробляти великі програми та перекомпілювати їх у JavaScript. Оскільки TypeScript є надмножиною JavaScript, існуючі програми JavaScript також є дійсними програмами TypeScript.

TypeScript можна використовувати для розробки програм JavaScript (таких як Node.js або Deno), які можна запускати на клієнті та сервері. Є кілька варіантів перекомпіляції. Можна використовувати за замовчуванням перевірку TypeScript Checker або компілятор Babel для перетворення TypeScript в JavaScript.

TypeScript підтримує файли визначень, які можуть містити інформацію про типи існуючих бібліотек JavaScript, так само як файли заголовків C++ можуть описувати структуру існуючих об'єктних файлів. Це дозволяє іншим програмам використовувати значення, визначені у файлі, ніби це статично типізовані сутності TypeScript. Існують сторонні файли заголовків для деяких популярних бібліотек (таких як jQuery, MongoDB та D3.js). Заголовки TypeScript також можна використовувати в базових модулях Node.js, що дозволяє розробляти програми Node.js на TypeScript [1].

3.3 Node.js

Node.js – це середовище виконання з відкритим вихідним кодом, крос-платформене, бек-енд середовище виконання JavaScript, яке працює на движку V8 і виконує код JavaScript поза веб-браузером. Node.js дозволяє розробникам використовувати JavaScript для написання інструментів командного рядка та для сценаріїв на стороні сервера - запуску сценаріїв на стороні сервера для створення динамічного вмісту веб-сторінки до того, як сторінка буде відправлена у веб-браузер користувача. Отже, Node.js представляє парадигму "JavaScript скрізь" [6], що об'єднує розробку веб-додатків навколо однієї мови програмування, а не різних мов для сценаріїв на стороні сервера та клієнта.

Node.js дозволяє створювати веб-сервери та мережеві інструменти з використанням JavaScript та колекції "модулів", які обробляють різні основні функціональні можливості. Модулі передбачені для введення-виведення у файлових системах, мережевих взаємодій (DNS, HTTP, TCP, TLS / SSL або UDP), двійкових даних (буфери), криптографічних функцій, потоків даних та інших основних функцій. Модулі Node.js використовують API, призначений для зменшення складності написання серверних додатків.

JavaScript є єдиною мовою, яку підтримує Node.js, але існує багато мов, які можна скомпілювати в JS. Як результат, можна писати програми на Node.js мовою CoffeeScript, Dart, TypeScript, ClojureScript та іншими мовами.

Node.js в основному використовується для створення мережевих додатків, таких як веб-сервери. Найбільш суттєва різниця між Node.js та PHP полягає в тому, що більшість функцій у PHP блокуються до завершення (команди виконуються лише після попередньої команди), тоді як функції Node.js не блокуються (команди навіть виконуються паралельно), і використовують функцію зворотного виклику для позначення завершення або невдачі [19].

3.3 Docker

Docker – це набір продуктів платформи як послуги (PaaS), які використовують віртуалізацію на рівні ОС для доставки програмного забезпечення в пакетах, званих контейнерами. Контейнери ізольовані один від одного і поєднують власне програмне забезпечення, бібліотеки та конфігураційні файли. Вони можуть спілкуватися між собою за чітко визначеними каналами. Оскільки всі контейнери використовують служби одного ядра операційної системи, вони використовують менше ресурсів, ніж віртуальні машини.

Послуга має як безкоштовний, так і преміум-рівень. Програмне забезпечення, яке розміщує контейнери, називається Docker Engine. Вперше воно було запущене у 2013 році та розроблене Docker, Inc.

Docker може пакувати програми та їх залежності у віртуальний контейнер, який може працювати на будь-якому комп'ютері Linux, Windows або macOS. Це дозволяє програмам працювати в різних місцях, таких як локальні, відкриті хмари та/або приватні хмари. При запуску в Linux Docker використовує функції ізоляції ресурсів ядра Linux (такі як cgroup та простір імен ядра) та функціональні файлові системи (наприклад, OverlayFS), що дозволяє контейнерам запускатись на одному

екземплярі Linux, уникаючи тим самим запуску та обслуговування віртуальних машин. витрати. машина. Docker на macOS використовує віртуальні машини Linux для запуску контейнерів.

Docker реалізує API високого рівня, щоб забезпечити легковажні контейнери, які запускають процеси ізольовано.

Програмне забезпечення Docker як послуга складається з трьох компонентів:

- Програмне забезпечення: демон Docker, який називається dockerd, є постійним процесом, який використовується для управління контейнерами Docker та обробки об'єктів контейнерів. Демон слухає запити, надіслані через API Docker Engine. Клієнт Docker з назвою docker надає інтерфейс командного рядка CLI, що дозволяє користувачам взаємодіяти з демоном Docker.
- Об’єкти: об’єкти Docker - це різні сутності, що використовуються для побудови програм у Docker. Основними категоріями об’єктів Docker є зображення, контейнери та служби.
    - Контейнер Docker - це стандартизоване середовище упаковки для запуску програм. Керування контейнером здійснюється за допомогою Docker API або CLI.
    - Docker Image - це шаблон, призначений лише для читання, для створення контейнерів. Зображення використовуються для зберігання та доставки програм.
    - Docker дозволяє масштабувати контейнер за допомогою декількох демонів Docker. Результат називається набором інтерактивних демонів, які спілкуються через Docker API.
- Реєстри: реєстр Docker - це сховище зображень Docker. Клієнти Docker підключаються до реєстрів, щоб завантажувати зображення для використання або завантажувати зображення, які вони створили. Реєстри можуть бути публічними або приватними. Два основні публічні реєстри – Docker Hub та Docker Cloud. Docker Hub – це реєстр за замовчуванням, де Docker шукає

зображення. Реєстри Docker також дозволяють створювати повідомлення на основі подій [20].

РОЗДІЛ 4
ОПИС ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Архітектура серверної частини системи

Програмний застосунок реалізовує проходження файлу по різних сервісах, кожен з яких є незалежним. Таке рішення було прийняте через те, що етап обробки може обірватися на будь-якому з етапів, але часткові результати все одно мають зберегтися. Також, час обробки на кожному з сервісів не є фіксованим та може залежати від кількості задач, що очікують на обробку. З цією метою спілкування між окремими процесами було реалізовано через з’єднання протоколу AMQP. Спілкування ж застосунку із базою даних відбувається через запити протоколу HTTP.

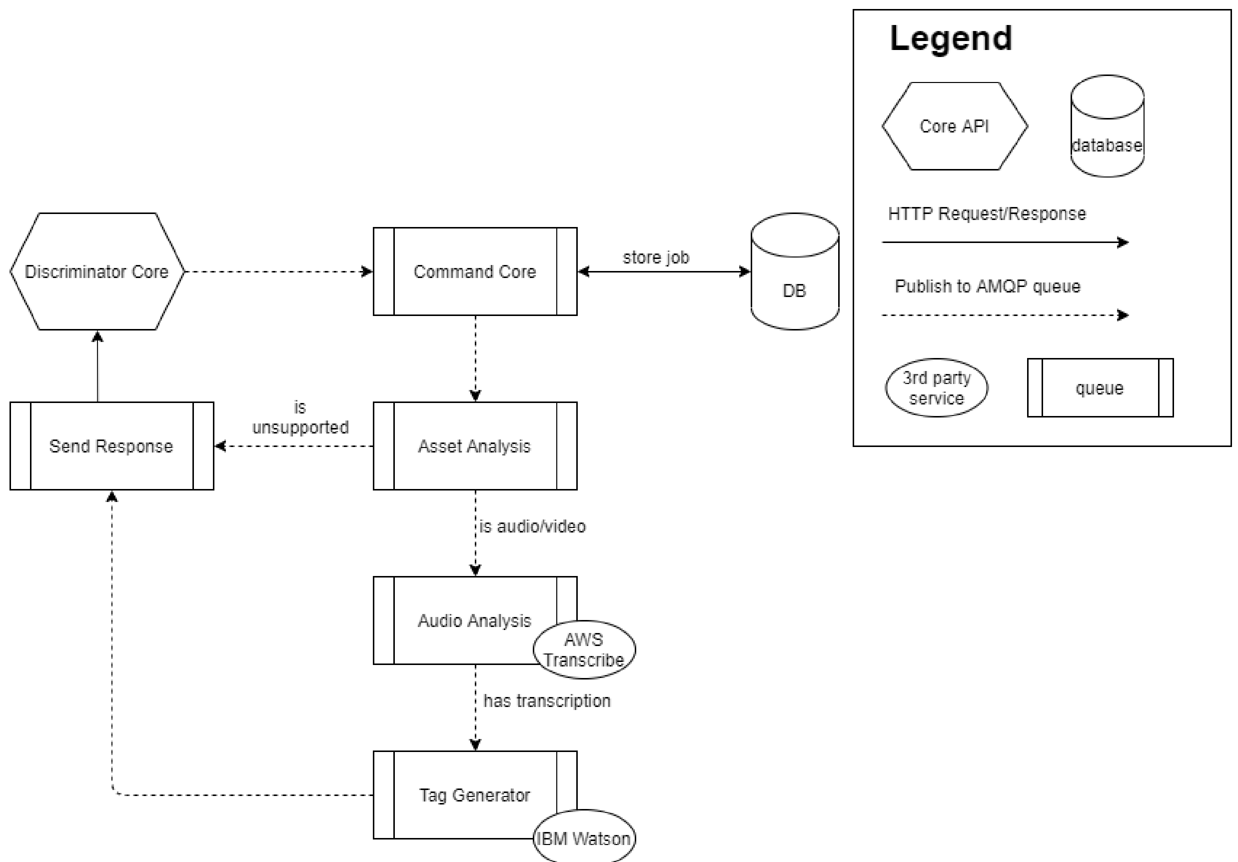

Рисунок 4.1.1. Схема роботи програми

Як видно з рис. 4.1.1, після надсилання користувачем файлу до системи, запит надсилається до головної черги. Звідти в свою чергу створюється запис до бази даних про задачу та надсилається запит до черги AssetAnalysis.

У черзі AssetAnalysis відбувається розпізнавання типу файлу. Якщо файл було розпізнано як той, що підтримується системою, то відбувається його подальша обробка. Якщо ж формат файлу не підтримується системою або сам файл не є валідним, то ми одразу надсилаємо відповідь до головного ядра програми про помилку та припиняємо обробку файлу.

Далі робота передається до черги Audio Analysis, головною задачею якої є розпізнавання транскрипції за допомогою стороннього сервісу AWS Transcribe. Якщо у вхідному файлі успішно було розпізнано мову, то робота передається на наступну чергу Tag Generator.

Черга Tag Generator має на меті безпосередньо тегування аудіо доріжки. Це відбувається за допомогою стороннього сервісу IBM Watson Natural Language Understanding. Отримані теги дописуються до відповіді та надсилаються до черги Send Response.

Черга Send Response сповіщує головне ядро програми про завершення роботи та відповідає за запис результату до бази даних. Отримані метадані дописуються до поля тієї ж задачі, що була створена на початку виконання програми.

Робота та спілкування між сервісами була реалізована за технологією контейнеризації. Було створено три контейнери: rabbit_dv2, db_dv2, dv2_core для позначення роботи з менеджером AMQP з’єднань, базою даних та головним ядром програми відповідно.

Лістинг коду для роботи з чергами подано у додатку А.

4.2 Завантаження та первинний аналіз файлу

Оскільки у подальшому файл буде надіслано до сервісу AWS Transcribe, то необхідно врахувати, які типи даних підтримуються цим сервісом. В результаті дослідження було виявлено, що AWS Transcribe підтримує лише формати MP3, MP4, WAV, FLAC. До того ж, довжина вхідного файлу не повинна перевищувати 10 хвилин.

Для того, щоб забезпечити універсальність програмного застосунку та надати користувачеві можливість завантажувати до системи будь-які файли, було прийнято рішення скористатись сервісом Transloadit, оскільки Transloadit надає NPM пакет для взаємодії з їх API та вдало інтегрується до створеної системи для усунення файлових обмежень. Перевага цього сервісу для програмного застосунку полягає в тому, що немає необхідності завантажувати ресурси та зберігати результат перетворення, оскільки він вже інтегрований з AWS S3.

Кожне завдання в Transloadit називається “ассемблі”. Для створення нової асссемблі нам потрібно надати параметри ассемблі та функцію зворотного виклику. Для створення ассемблі виконується API виклик createAssembly.

Першим аргументом методу createAssembly є об'єкт, який описує весь процес перетворення файлу. Потік розділений на етапи, на кожному кроці можуть бути використані різні модулі (роботи). У нашому випадку ми використовуємо /s3/import як джерело імпорту, /video/encode як конвертер, та /s3/store для зберігання результатів перетворення:

```
{
  waitForCompletion: false;     // enables webhook
  params: {
    notify_url: WEBHOOK_URL,    // webhook url
    steps: {
      import: {
        robot: '/s3/import',
        bucket: 'YOUR_IMPORT_BUCKET',
        bucket_region: 'YOUR_BUCKET_REGION',
        path: 'ASSET_KEY',
        key: 'YOUR_S3_KEY',
        secret: 'YOUR_S3_SECRET',
      },
      videoConvert: {
        use: 'import',
        robot: '/video/encode',
        format: 'mp4',
```

```
          result: true,
          quality: 80,
          imagemagick_stack: 'v2.0.7',
        },
        export: {
          use: 'videoConvert',
          robot: '/s3/store',
          bucket: 'public',
          bucket_region: 'YOUR_BUCKET_REGION',
          key: 'YOUR_S3_KEY',
          secret: 'YOUR_S3_SECRET',
          path: '${unique_prefix}/${file.url_name}',
          acl: 'bucket-default',
        },
      },
    },
}
```

Transloadit має підтримку фільтрації файлів на основі тривалості, розміру, типу mime та інших метаданих. Нижче наведена конфігурація для конвертації у mp3 з обмеженням тривалості для всіх об’єктів, крім “audio/mpeg”, “audio/wav”, “audio/flac”, “video/mp4”, тривалість яких менша за обмеження:

```
audioConvertFilter: {
  use: 'import',
  robot: '/file/filter',
  result: true,
  declines: [
    ['${file.mime}', '==', 'audio/mpeg'],
    ['${file.mime}', '==', 'audio/wav'],
    ['${file.mime}', '==', 'audio/flac'],
    ['${file.mime}', '==', 'audio/x-flac'],
    ['${file.mime}', '==', 'video/mp4'],
  ],
  error_on_decline: false,
},
durationFilter: {
  use: 'import',
  robot: '/file/filter',
  result: true,
  declines: [
    [
      '${file.meta.duration}',
      '<=',
      `${env.TRANSCRIBE_DURATION_LIMIT + ALLOWABLE_DURATION_DEVIATION}`,
    ],
  ],
  error_on_decline: false,
},
audioConvert: {
  use: ['audioConvertFilter', 'durationFilter'],  // using multiple inputs
  robot: '/audio/encode',
  preset: 'mp3',
  ffmpeg_stack: 'v3.3.3',
  ffmpeg: {
    ss: '0', // set starting point
    t: env.TRANSCRIBE_DURATION_LIMIT, // set duration limit
```

```
    vn: true, // disable video
    ar: 16000, // set audio sampling frequency
    ac: 1, // set number of channels
  },
},
```

Далі файл надсилається на обробку до програмного застосунку. Для коректної роботи програми та з міркувань безпеки необхідно запобігти випадкам, коли користувач надсилає невалідні файли під видом валідних (наприклад, користувач може надіслати скрипт, що може зашкодити серверу під виглядом безпечного файлу). Щоб запобігти випадкам, коли внутрішнє наповнення файлу не відповідає його розширенню, необхідно визначити так званий mime тип, що закодований у самому файлі та порівняти із типом, що був надісланий. Таким чином, ми не будемо обробляти файли, що можуть потенційно зашкодити системі або не підтримуються нами.

У результаті розпізнавання типу та внутрішнього наповнення, генерується JSON об’єкт наступного формату, який дописується до відповіді:

```
{
 "AssetAnalysis": {
    "ext": "m4a",
    "type": "audio",
    "created": "2020-01-13T17:55:50.848Z",
    "updated": "2020-01-13T17:57:00.890Z",
    "mimeType": "audio/x-m4a",
    "internalType": "audio"
  }
}
```

4.3 Розпізнавання мови та генерація транскрипції

Після завершення етапу попередньої обробки, файл надсилається у чергу AudioAnalysis до сервісу RabbitMQ. Коли приходить черга файлу на обробку, він надсилається до сервісу AWS Transcribe для генерації транскрипції. Оскільки після обробки сервісом Transloadit файл вже було завантажено на хмарне сховище S3, достатньо передати посилання на файл до задачі обробки для подальшої генерації транскрипції.

Робота із сервісом з розпізнавання мови є асинхронною та відбувається на сторонньому сервісі. Тому для отримання результатів роботи, необхідно забезпечити роботу зі сповіщеннями. AWS надає таку можливість у вигляді SNS сповіщень. Таким чином, ми можемо керувати роботою сповіщень:

```
export const handleSnsNotification = async (
  messageType: AWSSNSMessage,
  body: any,
  mqManager: QueueManager,
): Promise<void> => {
  await validateSnsNotification(body);
  switch (messageType) {
    case AWSSNSMessage.SubscriptionConfirmation:
      await handleSubscriptionConfirmation(body);
      break;
    case AWSSNSMessage.Notification:
      await handleNotification(body, mqManager);
      break;
    case AWSSNSMessage.UnsubscribeConfirmation:
      handleUnsubscribeConfirmation();
      break;
    default:
      AppLogger.warn('Unknown message type in headers');
  }
};
```

Таким чином, за успішної підписки на сповіщення, при зміні статусу нашої задачі, ми отримуємо про це сповіщення та керуємо роботою програми в залежності від відповіді:

```
export const handleNotification = async (
  body: HandleNotificationInterface,
  mqManager: QueueManager,
): Promise<void> => {
  AppLogger.info('SNS Notification received');
  const message = JSON.parse(body.Message);
  const transcriptionJob = message.detail as
AWS.TranscribeService.TranscriptionJob;
  const { TranscriptionJobStatus: jobStatus, TranscriptionJobName: jobName } =
transcriptionJob;
  if (!jobName) {
    AppLogger.error('No jobName specified');
    return;
  }

  switch (jobStatus) {
    case AWSTranscribeStatus.Completed:
      await TranscribeService.handleTranscribeCompleted(jobName, mqManager);
      break;
    case AWSTranscribeStatus.Failed:
      await TranscribeService.handleTranscribeFailed(jobName, mqManager);
      break;
    default:
      AppLogger.info(`Job ${jobName} - Unknown transcription job status update`);
  }
```

```
};
```

Також в залежності від статусу, необхідно отримати результат та дописати його до нашої бази даних й до відповіді, що в подальшому повернеться користувачу. Оскільки AWS не надсилає відповіді напряму, необхідно витягувати її з URL в відповіді у необхідному нам форматі:

```
export const handleTranscribeCompleted = async (
  jobName: string,
  mqManager: QueueManager,
): Promise<void> => {
  const job = await getJobFromDbByTranscriptionName(jobName);
  if (
    !job ||
    !job.id ||
    (job.requestJson.dv2_response.AudioAnalysis &&
      job.requestJson.dv2_response.AudioAnalysis.transcript)
  ) {
    return;
  }
  const { id } = job;
  AppLogger.info(`Job ${id} - Transcription completed. AWS Transcribe job name
${jobName}`);

  try {
    const transcriptUrl = await audioTranscriber.getTranscriptUrl(jobName);
    const transcriptResult = await axios.get(transcriptUrl).then(res => res.data);

    const transcript =
transcriptResult.results.transcripts[ZERO].transcript.trim();
    transcriptResult.results.transcripts[ZERO].transcript = transcript;

    const errorMessage = !transcript ? 'No text in audio/video' : undefined;
    const result: AudioAnalysisResult = {
      ...job.requestJson.dv2_response.AudioAnalysis,
      transcript: transcriptResult,
      error: errorMessage,
    };

    await JobService.appendStageResult(id, { stage: QueueName.AudioAnalysis,
result });

    const nextStageQueue = errorMessage ? QueueName.SendResponse :
QueueName.TagGenerator;
    await mqManager.publish(nextStageQueue, String(id));
  } catch (err) {
    const errorMessage = String(err.message || err);
    AppLogger.error(`Job ${id} - failed inside AWS SNS webhook. ${errorMessage}`);

    await JobService.updateJob(id, {
      status: JobStatus.Error,
      errorMessage,
    });
  }
};
```

Якщо ж статус був неуспішним, то ми записуємо відповідну помилку:

```
export const handleTranscribeFailed = async (jobName: string, mqManager:
QueueManager) => {
  const job = await getJobFromDbByTranscriptionName(jobName);
  if (!job || !job.id) {
    return;
  }
  const reason = await audioTranscriber.getFailureReason(jobName);
  const errorMessage = `AWS Transcribe error: ${reason}`;
  AppLogger.info(`Job ${job.id} - ${errorMessage}`);
  await JobService.updateJob(job.id, {
    status: JobStatus.Error,
    errorMessage,
  });
  await mqManager.publish(QueueName.SendResponse, String(job.id));
};
```

У результаті розпізнавання мови та генерації транскрипції формується наступний JSON об’єкт, що дописується до відповіді. JSON об’єкт має наступний формат:

```
{
  duration?: 10
  format?: 'wav' | 'mp3' | 'mov,mp4,m4a,3gp,3g2,mj2' | 'flac';
                                   // format_name from ffmpeg output
  jobName?: string;               // job name from transcribe service
  transcript?: {
    status: "FAILED" | "COMPLETED"
    jobName: string               // job name from transcribe service
    results: {
      items: TranscribeItem[];     // array of phrases with timestamps
      transcripts: [
        {
          transcript: string;      // whole transcription of an asset
        }
      ];
    };
    accountId: string;            // the account id that used for transcribing
  }
}
```

4.4 Генерація тегів

За умови успішного проходження етапу генерації транскрипції, отриманий результат попередньої задачу надсилається на чергу TagGenerator, що включає в себе роботу з сервісом IBM Watson NLU.

У наведеному нижче фрагменті коду реалізовано роботу зі стороннім сервісом IBM Watson NLU. Як видно з коду, транскрипція обробляється сервісом:

з неї генеруються категорії та ключові слова. В подальшому вони приводяться до одного формату із зазначенням ступеня довіри до результатів.

```
  public async textAnalysis(payload: WatsonRequest): Promise<WatsonResponse> {
    this.setNluSdk();
    if (!payload || !payload.inputText) {
      throw { message: 'No text for analysis' };
    }

    const result: IbmWatson = await this.nluSdk.analyze({
      text: payload.inputText,
      features: {
        categories: {
          limit: 5,
          explanation: true,
        },
        keywords: {
          limit: 5,
        },
      },
    });

    // normalize categories labels by single distinct tags, with no extra chars
    let categories: CategoriesEntity[] = [];
    if (result.categories && !!result.categories.length) {
      categories = payload.isCreateTitleRequest ? [result.categories[ZERO]] :
result.categories;
    }
    const tagsFromCategories = [...formatCategoriesByTags(categories)];

    const keywords = result.keywords || [];
    const tagsFromKeywords = keywords.map(({ text, relevance }) => ({
      name: text,
      confidence: toPercentage(relevance),
      isKeyword: true,
    }));

    const allTags = [...tagsFromKeywords, ...tagsFromCategories];

    const tags = allTags.length ? allTags : undefined;
    return {
      api_response: { ...result, tags },
    };
  }
```

Як видно з коду, до відповіді ми дописуємо як результати, що повертає сервіс IBM NLU, так і відформатовані та приведені до одного формату теги, що й будуть слугувати кінцевим результатом виконання програми. Кожна визначена категорія відділяється від списку та виступає окремим тегом, який дописується до результату із ступенем довіри.

У результаті розпізнавання тегів, маємо об’єкт JSON у наступному форматі:

```
{
```

```
    "TagList": {
      "name": "string",
      "confidence": "number"
    },
    "TagGenerator": {
        "created": "string",
        "updated": "string",
        "api_response": {
            "tags": {
                "name": "string",
                "confidence": "number"
            },
            "usage": {
                "features": "number",
                "text_units": "number"
            },
            "keywords": {
                "text": "string",
                "count": "number",
                "relevance": "number"
            },
            "language": "string",
            "categories": {
                "label": "string",
                "score": "number",
                "explanation": {
                    "relevant_text": {
                        "text": "string"
                    }
                }

            }
        }
    }
}
```

При завершенні роботи, система переходить до черги SendResponse, яка в свою чергу записує кінцевий результат до бази даних та сповіщує про це систему. В результаті, в базі даних ми маємо поле метаданих, записаних у форматі JSON, що відповідає файлу, який було завантажено користувачем.

4.5 Приклад роботи програми

Для взаємодії користувача із програмним застосунком було розроблено Telegram Bot. Після надсилання файлу до боту відбувається його обробка, а по заверешенню роботи та отриманні результатів, бот отримує про це сповіщення та надсилає результати користувачеві.

Розглянемо приклад надсилання файлу на рисунку 4.5.1.

Рисунок 4.5.1. – Фрагмент роботи боту після надсилання файлу.

Після цього файл обробляється системою та видає логи про свою роботу та її тривалість, як вказано на рисунку 4.5.2.

Після отримання кінцевих результатів бот надсилає відповідь користувачеві у чат, як показано на рисунку 4.5.3.

На прикладах наведених на рисунках 4.5.1 – 4.5.3 до системи надсилався файл аудіокниги для дітей. Як видно з результатів, загальний час обробки файлу, генерації транскрипції та тегів зайняв трохи довше за 2 хвилини. Отримані результати транскрипції та тегів не є ідеальними, але вони є задовільними для поставленої задачі.

```
Job 288457 - Acked inside SendResponse
Job 288457 - successfully sent response to CommandCore
Job 288457 - Stage results updated in db
*** Job 288457 stage duration report *** AssetAnalysis = 126.657 s AudioAnalysis = 125.407 s TagGenerator = 0.719 s TOTAL TIME IN DV2: 126.733 s ***
Job 288457 - Stage results updated in db
Job 288457 - published to SendResponse queue with priority 0 of 10
Job 288457 - Acked inside TagGenerator
Job 288457 - received by SendResponse worker
Job 288457 - Tags were obtained
Job 288457 - Tags were obtained
Job 288457 - Stage results initialized
Job 288457 - received by TagGenerator worker
Job 288457 - published to TagGenerator queue with priority 0 of 10
Job 288457 - Stage results updated in db
Job 288457 - Transcription completed. AWS Transcribe job name 288457.e84a6d75-f5a4-4942-8c07-0d8b2bc6d19f
SNS Notification received
[NAVY] [6778-1] sql_error_code = 00000 LOG: checkpoint starting: time
Job 288457 - Acked inside AudioAnalysis
Job 288457 - Stage results updated in db
Transcription job started: 288457.e84a6d75-f5a4-4942-8c07-0d8b2bc6d19f
Job 288457 - Stage results updated in db
Job 288457 - Metadata obtained. Media duration: 426.18 s; media format: mov,mp4,m4a,3gp,3g2,mj2
Job 288457 - Stage results initialized
Job 288457 - received by AudioAnalysis worker
Job 288457 - Acked inside AssetAnalysis
Job 288457 - published to AudioAnalysis queue with priority 0 of 10
Job 288457 - Stage results updated in db
Job 288457 - Type: video, Ext: mp4, Mime: video/mp4
Job 288457 - Stage results initialized
Job 288457 - received by AssetAnalysis worker
Job 288457 - Acked inside CommandCore
Job 288457 - published to AssetAnalysis queue with priority 0 of 10
```

Рисунок 4.5.2. – Фрагмент логів роботи програми.

Рисунок 4.5.3. – Фрагмент роботи боту під час надсилання результатів.

ВИСНОВКИ

Метою написання дипломної роботи було створення програмного засобу для тегування аудіо- та відео-доріжок.

У процесі виконання були виконані наступні завдання:

- поглиблено знання мови TypeScript та опанована платформа Node.js;
- поглиблено знання архітектури веб-серверів;
- опановано різні протоколи для обміну повідомленнями на сервері, зокрема AMQP та HTTP;
- опановано технології контейнеризації на основі Docker;
- досліджено методи машинного навчання для розпізнавання мови та обробки природномовних текстів;
- проаналізовано переваги та недоліки використання електронних засобів у процесі аналізу мовлення;
- досліджено та вивчено хмарні технології для роботи з галуззю машинного навчання та обробки даних;
- описано архітектуру системи для ефективної обробки даних;
- розроблено програмний застосунок, що дозволяє завантажувати та обробляти аудіо- та відео-файли користувача;
- написано клієнт у вигляді бота, що дозволяє користувачам взаємодіяти із застосунком.

Створений програмний застосунок дозволить у майбутньому аналізувати файли та формувати до них метадані на основі вмісту. Сформовані метадані можна буде використати у системах пошуку, фільтрації та сортування. Також результати, згенеровані в ході виконання програми, можуть використовуватися для формування релевантних даних на основі вмісту аудіо- та відео-контенту, тобто можуть застосовуватися як інструмент формування таргетованого вмісту та

рекламами. Дані технології є затребуваними у сучасному світі, що все далі переходить до дистанційного формату та оперує великими даними.

У системі реалізовано оптимальну архітектуру, яка дозволить великій кількості користувачів користуватися функціоналом та уникнути великої затримки в обробці файлів, забезпечуючи часткове збереження проаналізованих даних навіть при непередбачуваних помилках системи. Даний підхід гарантує цілісність та безпеку даних, що користувачі передають до системи.

Дана робота може слугувати підґрунтям для створення більш вдосконаленої системи організації даних, що дозволить користувачеві здійснювати пошук на основі вмісту файлів, що є доцільним при їх великій кількості та слабкій організованості, до того ж при недостатньо досконалих внутрішніх метаданих у певних форматах аудіо та відео. Розроблена система може бути розширена до більшої кількості підтримуваних форматів та типів даних, зокрема зображень та документів з подальшою класифікацією файлів за категоріями.

У системі реалізовано повноцінну обробку аудіо- та відео-файлів довільних форматів та надання користувачеві сформованих метаданих, що дозволяє сказати, що мета розробки програмного застосунку досягнута.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bright, Peter. Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem? [Електронний ресурс] / Bright, Peter – Режим доступу до ресурсу: https://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/.
2. Zeng, Marcia. Metadata types and functions [Електронний ресурс] / Zeng, Marcia – Режим доступу до ресурсу: https://marciazeng.slis.kent.edu/metadatabasics/types.htm.
3. Smith, Gene. Tagging: people-powered metadata for the social web [Електронний ресурс] / Smith, Gene – Режим доступу до ресурсу: https://archive.org/details/isbn_9780321529176.
4. ID3 History [Електронний ресурс] – Режим доступу до ресурсу: https://web.archive.org/web/20101224080318/http://id3.org/History.
5. The power of metadata in audio files [Електронний ресурс] – Режим доступу до ресурсу: https://forums.envato.com/t/the-power-of-metadata-in-audio-files/296742.
6. Аткина В.С. Использование программного комплекса для исследования катастрофоустойчивости информационных систем // Вестник Волгоградского государственного университета. Серия 10 Инновационная деятельность. Выпуск 5 2010-2011г. В.: Изд-во ВолГУ, 2011.
7. Оладько В.С., Ананьин Е.В., Кожевникова И.С., Датская Л.В. Анализ угроз, связанных со сканированием портов – Нефтекамский филиал БашГу, материалы XI Всероссийской студенческой научно-практической

конференции (с международным участием): «Первые шаги в науку третьего тысячелетия» - Нефтекамск: 2015 – 1031с.

8. Ананьин Е.В., Кожевникова И.С., Датская Л.В. анализ существующих методов обнаружения несанкционированного сканирования портов // Сборник статей Международной научно-практической конференции 18 октября 2015г. «ТЕНДЕНЦИИ И ПЕРСПЕКТИВЫ РАЗВИТИЯ НАУКИ XXI ВЕКА» – Ек.: РИО МНЦИИ «ОМЕГА САЙНС», 2015. – 58с.
9. Lourdes A. Metaheuristics for Natural Language Tagging [Електронний ресурс] / A. Lourdes, L. Gabriel, A. Enrique – Режим доступу до ресурсу: https://core.ac.uk/download/pdf/191309766.pdf.
10. Amazon Web Services [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/what-is-aws/.
11. Amazon Transcribe [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/transcribe/latest/dg/transcribe-dg.pdf.
12. Amazon SNS [Електронний ресурс] – Режим доступу до ресурсу: https://docs.aws.amazon.com/sns/latest/dg/welcome.html.
13. IBM Cloud [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud.
14. IBM Watson [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/watson.
15. IBM Knowledge Studio [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/cloud/watson-knowledge-studio.
16. IBM NLU Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.ibm.com/apidocs/natural-language-understanding.
17. Visual Studio Code [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/.
18. Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/docs/latest/api/.

19. Docker [Електронний ресурс] – Режим доступу до ресурсу: https://www.docker.com/.

ДОДАТОК А
Лістинг коду для роботи з чергами

```
export interface DataHandler {
  (dataHandler: SubscriptionPayload): void;
}
export interface QueueManagerConfig {
  url?: string;
  name?: string;
  prefetchCount?: number;
}

type OnMessageCallback = (message: amqp.ConsumeMessage | null) => void;

export class QueueManager {
  public static readonly MaxPriority = 10;
  private readonly url: string;
  private readonly name: string;
  private readonly prefetchCount?: number;
  private queuesList: string[] = Object.keys(QueueName);
  private connection?: Connection;
  private channel?: ConfirmChannel | null;
  private noChannelErrorMessage = (prefix: string) => `${prefix}. Channel
is not initialized`;
  private eventEmitter: EventEmitter = new EventEmitter();
  constructor(config?: QueueManagerConfig) {
    this.url = (config && config.url) || env.CLOUDAMQP_URL;
    this.name = (config && config.name) || EMPTY_STRING;
    this.prefetchCount = config && config.prefetchCount;
  }
  public async init(dataHandler?: DataHandler): Promise<QueueManager |
void> {
    try {
      if (!this.connection) {
        this.connection = await amqp.connect(this.url, {
          clientProperties: { connection_name: this.name },
        });
        this.eventEmitter = new EventEmitter();
        this.eventEmitter.once('reconnect', (err: Error) => {
          AppLogger.error(
            `Error: ${err.message || err} \n Connection to amqp was closed.
Reconnecting...`,
          );
          this.killProcess();
        });
        this.connection.on('close', () => {
          this.eventEmitter.emit('reconnect');
        });
        this.connection.on('error', (err: Error) => {
          this.eventEmitter.emit('reconnect', err);
        });
      } else {
        AppLogger.info('Connection is already initialized');
```

```
      }
      if (!this.channel) {
        this.channel = await this.connection.createConfirmChannel();
        if (this.prefetchCount) {
          await this.channel.prefetch(this.prefetchCount);
        }
      }
      await this.validateQueues();
      if (dataHandler) {
        await this.subscribeQueue(this.name as QueueName, dataHandler);
      }
      return this;
    } catch (err) {
      AppLogger.error(err);
      this.killProcess();
    }
  }
  public async subscribeQueue(
    queueName: QueueName,
    subscribeDataHandler: DataHandler,
  ): Promise<amqp.Replies.Consume> {
    if (!this.channel) {
      throw new Error();
    }
    return this.channel.consume(
      queueName,
      this.getOnMessageCallback(subscribeDataHandler, queueName),
    );
  }
  private getOnMessageCallback(
    subscribeDataHandler: DataHandler,
    queueName: QueueName,
  ): OnMessageCallback {
    return (consumeMessage: amqp.ConsumeMessage | null) => {
      if (consumeMessage === null) {
        AppLogger.error('NullMessageException');
      } else {
        const currentMessage = consumeMessage.content.toString();
        const currentPriority = Number(consumeMessage.properties.priority)
|| 0;
        const currentRetry =
Number(consumeMessage.properties.headers['retries']) || 0;
        const encryptedMessage = new
CipherService().formattedEncrypt(currentMessage);
        if (this.name === QueueName.CommandCore) {
          const requestId = JSON.parse(currentMessage).requestId;
        } else {
          const id = currentMessage;
          logJobActivity({ id, message: `received by ${this.name} worker`
});
        }
        let previousAction = '';
        subscribeDataHandler({
          message: currentMessage,
```

```
          ackAction: (jobId?: string | number) => {
            const id = jobId || currentMessage;
            if (!previousAction) {
              this.channel && this.channel.ack(consumeMessage);
              previousAction = 'ack';
            }
          },
          rejectAction: (jobId?: string | number) => {
            const id = jobId || currentMessage;
            if (!previousAction) {
              this.channel && this.channel.reject(consumeMessage, false);
              logJobActivity({
                id,
                message: `Rejected inside ${queueName}`,
                level: LogLevels.Debug,
              });
              previousAction = 'reject';
            }
          },
          publish: (queueName: QueueName, message?: string, priority?:
number) =>
            this.publish(queueName, message || currentMessage, priority ||
currentPriority),
          publishToRetry: (message?: string, priority?: number) =>
            this.publishToRetry(
              message || currentMessage,
              priority || currentPriority,
              currentRetry,
            ),
          deathReason: consumeMessage.properties.headers[
            QueueProperties.X_FIRST_DEATH_REASON
          ] as DeathReason,
          retries: currentRetry,
        });
      }
    };
  }
  public async publish(queueName: string, message: string, priority = 0):
Promise<void> {
    return new Promise((resolve, reject) => {
      if (!this.channel) {
        return;
      }
      this.channel.publish(
        ExchangeName.Discriminator,
        queueName,
        Buffer.from(message),
        {
          priority,
        }
          resolve();
        },
      );
    });
```

```
  }
  public async publishToRetry(
    message: string,
    priority: number,
    retryCount: number,
  ): Promise<void> {
    return new Promise((resolve, reject) => {
      if (!this.channel) {
        reject(new DCE010());
        return;
      }
      this.channel.publish(
        ExchangeName.Retry,
        this.name
      );
    });
  }
  public async disconnect(): Promise<void> {
    if (this.channel) {
      await this.channel.close();
      this.channel = null;
    }
  }
  public async resetQueues(): Promise<void> {
    this.connection = await amqp.connect(this.url, {
      clientProperties: { connection_name: this.name },
    });
    this.channel = await this.connection.createConfirmChannel();
    await Promise.all(
      this.queuesList.map(async q => {
        await this.channel?.deleteQueue(q);
      }),
    );
    this.killProcess();
  }
  private async validateQueues(): Promise<void> {
    if (!this.channel) {
      return;
    }
    await this.validateExchange();
    await Promise.all(
      this.queuesList.map(async q => {
        if (!this.channel) {
          return;
        }
        switch (q) {
          case QueueName.DeadLetter: {
            await this.channel.assertQueue(q, {
              arguments: {
                [QueueProperties.X_DEAD_LETTER_EXCHANGE]:
ExchangeName.Discriminator,
                [QueueProperties.X_MESSAGE_TTL]:
Number(env.RABBIT_MQ_MESSAGE_TTL),
              },
```

```
            });
            await this.channel.bindQueue(q, ExchangeName.DeadLetter, q);
            break;
          }
          case QueueName.Retry: {
            await this.channel.assertQueue(q, {
              arguments: {
                [QueueProperties.X_DEAD_LETTER_EXCHANGE]:
ExchangeName.Discriminator,
                [QueueProperties.X_MESSAGE_TTL]:
Number(env.RABBIT_MQ_MESSAGE_TTL),
              },
              maxPriority: QueueManager.MaxPriority,
            });
            await this.channel.bindQueue(q, ExchangeName.Retry, '*');
            break;
          }
          default: {
            await this.channel.assertQueue(q, {
              arguments: {
                [QueueProperties.X_MESSAGE_TTL]:
Number(env.RABBIT_MQ_MESSAGE_NOT_PROCESSED_TTL),
              },
              maxPriority: QueueManager.MaxPriority,
              deadLetterExchange: ExchangeName.DeadLetter,
            });
            await this.channel.bindQueue(q, ExchangeName.Discriminator, q);
          }
        }
      }),
    );
  }
  private async validateExchange(): Promise<void> {
    if (!this.channel) {
      AppLogger.error(this.noChannelErrorMessage('Can not validate
exchange'));
      return;
    }
    await this.channel.assertExchange(ExchangeName.DeadLetter,
ExchangeType.Fanout, {
      durable: true,
    });
    await this.channel.assertExchange(ExchangeName.Retry,
ExchangeType.Topic, {
      durable: true,
    });
    await this.channel.assertExchange(ExchangeName.Discriminator,
ExchangeType.Topic, {
      durable: true,
    });
  }
 }
```