МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Острозька академія»

Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: **«Розробка Discord-бота для аудіоконтенту: аудіо-фільтри FFmpeg, рекомендації на основі історії та вебінтерфейс»**

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Герасимчук Олександр Степанович

Керівник: викладач кафедри інформаційних технологій та аналітики даних
Ляховчук Сергій Васильович

Рецензент: кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних
_____ (проф., д.е.н. Кривицька О.Р.)

Протокол № _11_ від « 20_ » _травня_ 2026 р.

Острог, 2026

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка Discord-бота для аудіоконтенту: аудіо-фільтри FFmpeg, рекомендації на основі історії та вебінтерфейс

Автор: ***Герасимчук Олександр Степанович***

Науковий керівник: викладач кафедри інформаційних технологій та аналітики даних

Ляховчук Сергій Васильович

Захищена «……»…… 2026 року.

Пояснювальна записка до кваліфікаційної роботи: 70 с., 16 рис., 2 табл., 4 додатків, 14 джерел. ***Ключові слова***: DISCORD БОТ, PYTHON, АСИНХРОННЕ ПРОГРАМУВАННЯ, FFMPEG, AUTOMIX, БАЗИ ДАНИХ, SQLITE, DOCKER, CI/CD, АВТОМАТИЗАЦІЯ

Короткий зміст праці:

У кваліфікаційній роботі розглянуто процес проєктування та розробки музичного бота для платформи Discord з інтелектуальною системою автоматичного підбору музики на основі аналізу поведінки користувачів. Актуальність теми зумовлена закриттям провідних публічних музичних ботів (Groovy, Rythm) у 2021 році та відсутністю відкритих альтернатив, що поєднують стабільне аудіо-відтворення, персоналізовані рекомендації та промисловий рівень якості коду. У роботі проаналізовано архітектуру платформи Discord (Gateway API, REST API, Voice API, Interactions API), розглянуто існуючі рішення та виявлено їх ключові недоліки: відсутність системи рекомендацій, неможливість збереження стану між перезапусками та відсутність автоматизованого тестування.

У теоретичній частині обґрунтовано вибір технологічного стеку: мова Python 3.12 та фреймворк discord.py для асинхронної взаємодії з API Discord; бібліотека yt-dlp для видобутку аудіо-потоків; медіа-обробник FFmpeg для транскодування аудіо у формат Discord (PCM 48000 Гц, стерео); СУБД SQLite з асинхронним драйвером aiosqlite для персистентного зберігання стану. Розглянуто теоретичні основи алгоритмів рекомендацій, зокрема зваженого випадкового вибору та A/B тестування, а також принципи SOLID та патерн Repository як основу архітектурних рішень.

Практична частина роботи присвячена безпосередній реалізації бота. Спроєктовано чотирирівневу шарувату архітектуру (Presentation – Application – Service – Data) з сімома спеціалізованими сервісами (QueueService, HistoryService, PlayerService, SourceService, AutomixService, DJService, AutoResume). Розроблено кастомний аудіо-пайплайн на основі двох послідовних підпроцесів (yt-dlp – FFmpeg – Discord), що вирішує проблему нестабільного відтворення при URL-стримінгу. Реалізовано інтелектуальну систему Automix з двома алгоритмами зваженої вибірки – top_weighted (підбір за популярністю) та history_explore (підбір з ухилом у менш прослухані треки) – та механізмом A/B тестування ефективності через таблицю аналітики. Впроваджено систему Auto-Resume зі Staleness Policy (24 години) та Re-entry Guard для надійного відновлення стану бота після перезапуску без підключення до застарілих сесій. Розроблено функцію DJ-агент – генерацію контекстних коментарів між треками у трьох стильових режимах (chill, energetic, funny) з урахуванням часу доби та дій користувачів.

Реалізовано повний інтерактивний інтерфейс на основі кнопок Discord: панель керування плеєром з дев'ятьма кнопками, система пагінації черги, модальне вікно переміщення треків та меню налаштувань Automix і DJ. Розроблено схему бази даних з семи взаємопов'язаних таблиць з підтримкою WAL mode для конкурентного доступу.

З метою забезпечення якості коду застосовано сучасні практики DevOps. Розроблено комплексну систему автоматизованих тестів: 38 тестових файлів, 208 тест-функцій із покриттям 90% (2 120 з 2 360 рядків), з яких 13 з 22 модулів мають 100% покриття. Налаштовано CI/CD пайплайн у GitHub Actions з дворівневим лінтингом (flake8) та автоматичною збіркою Docker-образу. Розроблено конфігурацію Docker з multi-stage build та Docker Compose для відтворюваного розгортання.

У результаті виконання роботи створено готовий до використання програмний продукт, розгорнутий та протестований на реальних

Discord-серверах. Запропоноване рішення забезпечує стабільне аудіо-відтворення, інтелектуальний підбір музики на основі вподобань спільноти, безперервність роботи після перезапусків та легке розгортання через Docker.

SUMMARY

Keywords: DISCORD BOT, PYTHON, ASYNCHRONOUS PROGRAMMING, FFMPEG, YT-DLP, AUTOMIX, SQLITE, DOCKER, CI/CD, TESTING, SOLID, REPOSITORY PATTERN

Abstract:

This qualification work examines the design and development of a music bot for the Discord platform with an intelligent music recommendation system based on user behavior analysis. The relevance of the topic is driven by the closure of leading public music bots (Groovy, Rythm) in 2021 and the absence of open-source alternatives combining stable audio playback, personalized recommendations, and professional-grade code quality. The study analyzes the Discord platform architecture (Gateway API, REST API, Voice API, Interactions API) and reviews existing solutions, identifying their key shortcomings: lack of recommendation systems, inability to preserve state across restarts, and absence of automated testing.

The theoretical part justifies the selection of the technology stack: Python 3.12 and the discord.py framework for asynchronous interaction with the Discord API; the yt-dlp library for audio stream extraction; FFmpeg for transcoding audio into the Discord-compatible format (PCM 48000 Hz, stereo); and SQLite with the asynchronous aiosqlite driver for persistent state storage. The theoretical foundations of recommendation algorithms – specifically weighted random sampling and A/B testing – as well as SOLID principles and the Repository Pattern as the basis for architectural decisions are examined.

The practical part focuses on the bot's implementation. A four-layer architecture (Presentation – Application – Service – Data) was designed with seven specialized services (QueueService, HistoryService, PlayerService, SourceService, AutomixService, DJService, AutoResume). A custom audio pipeline based on two sequential subprocesses (yt-dlp – FFmpeg – Discord) was developed, resolving the playback instability issue inherent in URL streaming. An intelligent Automix system with two weighted sampling algorithms – top_weighted (popularity-based selection) and history_explore (selection biased toward less-played tracks) – and an A/B testing mechanism via an analytics table were implemented. An Auto-Resume system with a Staleness Policy (24 hours) and Re-entry Guard was developed to reliably restore bot state after restarts without reconnecting to stale sessions. An DJ-агент feature was implemented to generate contextual comments between tracks in three stylistic modes (chill, energetic, funny), reacting to the time of day and user actions.

A complete interactive interface was built using Discord buttons: a nine-button player control panel, paginated queue view, track movement modal window, and Automix/DJ settings menu. A seven-table database schema with WAL mode support for concurrent access was designed.

To ensure high code quality, modern DevOps practices were applied. A comprehensive automated testing system was developed: 38 test files, 208 test functions with 90% code coverage (2,120 of 2,360 statements), with 13 of 22 modules achieving 100% coverage. A CI/CD pipeline was configured in GitHub Actions with two-level linting (flake8) and automated Docker image builds. A Docker configuration with a multi-stage build and Docker Compose was developed for reproducible deployment.

As a result, a ready-to-use software product was created, deployed and tested on real Discord servers. The proposed solution provides stable audio playback, intelligent music selection based on community preferences, operational continuity after restarts, and easy deployment via Docker.

Зміст

ВСТУП

Стрімкий розвиток цифрових комунікаційних платформ у XXI столітті суттєво змінив форми соціальної взаємодії. Особливе місце серед них посідає Discord – платформа для голосового, текстового та відеоспілкування, яка спочатку орієнтувалась на ігрове співтовариство, однак поступово перетворилась на універсальний інструмент для навчання, командної роботи та дозвілля. Станом на 2024 рік аудиторія Discord нараховує понад 500 мільйонів зареєстрованих користувачів і понад 19 мільйонів активних серверів на добу [1]. У цьому контексті програмні боти – автоматизовані агенти, що взаємодіють з учасниками серверів – стали невід'ємною частиною екосистеми платформи.

Серед найпопулярніших категорій Discord-ботів особливе місце займають музичні боти, що дозволяють учасникам спільно слухати музику у голосових каналах. Попри широке поширення таких сервісів, більшість з них мала суттєві технічні обмеження: нестабільне відтворення аудіо, відсутність персоналізації, неможливість збереження стану після перезапуску, а також відсутність інструментів автоматичного підбору музики. Закриття таких популярних рішень, як Groovy та Rythm у 2021 році через претензії звукозаписних компаній, лише посилило потребу у надійних, самостійно розгорнутих (self-hosted) альтернативах.

Актуальність теми дослідження обумовлена кількома чинниками. По-перше, зростаючим попитом на якісні інструменти для колективного прослуховування музики у цифровому середовищі. По-друге, відсутністю відкритих рішень, що поєднували б стабільне аудіо-відтворення, інтелектуальну систему рекомендацій та промислові практики розробки програмного забезпечення (автоматизоване тестування, контейнеризація, CI/CD). По-третє, теоретичним і практичним інтересом до застосування принципів SOLID, патерну Repository та алгоритмів зваженої вибірки у контексті реального застосунку.

Метою дипломної роботи є проєктування та реалізація музичного Discord-бота з інтелектуальною системою рекомендацій на основі аналізу поведінки користувача, з дотриманням сучасних стандартів якості програмного забезпечення.

Для досягнення поставленої мети необхідно вирішити наступні завдання:
Провести аналіз предметної області: дослідити архітектуру платформи Discord, розглянути існуючі рішення та їх недоліки, обґрунтувати вибір технологічного стеку.

Спроєктувати архітектуру системи відповідно до принципів SOLID із використанням шаруватої архітектури та патерну Repository.
Спроєктувати схему бази даних для зберігання стану черги, історії прослуховувань, налаштувань та аналітики.
Розробити кастомний аудіо-пайплайн на основі yt-dlp та FFmpeg, що забезпечує стабільне відтворення з мінімальною затримкою між треками.
Реалізувати інтелектуальну систему Automix з двома алгоритмами рекомендацій та механізмом A/B тестування їх ефективності.
Реалізувати функцію Auto-Resume для відновлення стану бота після перезапуску з реалізацією Staleness Policy.
Розробити комплексну систему автоматизованих тестів (Unit та Integration) з досягненням рівня покриття коду не менше 80%.
Налаштувати CI/CD пайплайн на базі GitHub Actions та Docker-контейнеризацію.

Об'єктом дослідження є процес розробки програмного забезпечення для платформ реального часу з підтримкою мультимедійного контенту.

Предметом дослідження є архітектурні рішення, алгоритми рекомендацій та методи забезпечення якості програмного забезпечення при розробці музичного Discord-бота.

Методи дослідження. У роботі використано такі методи: системний аналіз при дослідженні предметної області та порівнянні існуючих рішень, об'єктно-орієнтоване проєктування при розробці архітектури системи, метод прецедентів (use-case) при формуванні вимог, методи Unit та Integration тестування для верифікації коректності реалізації, експериментальний метод при A/B тестуванні алгоритмів рекомендацій.

Наукова новизна роботи полягає у розробці гібридної системи рекомендацій для музичного Discord-бота, що поєднує два алгоритми зваженої вибірки (top-weighted та history-explore) з механізмом штрафів за пропуск треків та автоматичним A/B тестуванням ефективності стратегій у реальному часі.

Практичне значення отриманих результатів визначається тим, що розроблений програмний продукт є повністю функціональним застосунком, готовим до розгортання на реальних Discord-серверах. Реалізовані архітектурні рішення (Repository Pattern, шарувата архітектура, асинхронна персистенція) можуть бути застосовані при розробці інших ботів та сервісів реального часу. Система автоматизованого тестування з 90% покриттям коду (208 тестів, 38 тестових файлів) демонструє промисловий підхід до забезпечення якості програмного забезпечення.

Апробація результатів. Розроблений бот розгорнуто та протестовано на Discord-серверах у реальних умовах експлуатації. Вихідний код проєкту розміщено у відкритому репозиторії GitHub з налаштованим CI/CD пайплайном, що автоматично перевіряє якість коду при кожній зміні.

Структура роботи. Дипломна робота складається зі вступу, п'яти розділів, висновку, списку використаних джерел та додатків. Загальний обсяг роботи становить 70 сторінок, з них 16 рисунків та 2 таблиці. Список використаних джерел налічує 14 найменувань.

У першому розділі проведено аналіз предметної області: досліджено архітектуру платформи Discord, розглянуто існуючі музичні боти та їх обмеження, обґрунтовано вибір технологічного стеку та розглянуто теоретичні основи алгоритмів рекомендацій.

У другому розділі описано проєктування системи: архітектурні рішення та принципи SOLID, схему бази даних, проєктування аудіо-пайплайну, алгоритм Automix та UI-компоненти.

У третьому розділі детально описано реалізацію системи: структуру проєкту, ключові реалізаційні рішення (in-memory кеш, Auto-Resume, Repository Pattern), реалізацію slash-команд, CI/CD пайплайн та Docker-контейнеризацію.

У четвертому розділі представлено комплексну систему тестування: стратегію, Unit та Integration тести, методологію мокінгу discord.py та аналіз результатів покриття коду.

У п'ятому розділі описано процес розгортання та використання системи: системні вимоги, Docker-розгортання, інструкцію користувача та моніторинг.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Опис предметного середовища

Discord – це платформа для цифрової комунікації, заснована у 2015 році та орієнтована спочатку на ігрові спільноти, однак з часом суттєво розширила свою аудиторію. Станом на 2024 рік платформа нараховує понад 500 мільйонів зареєстрованих облікових записів та 19 мільйонів активних серверів щодня [1]. Ключовою одиницею організації є сервер (guild) – ізольована спільнота з власними текстовими та голосовими каналами, ролями та правами доступу. Голосові канали реалізовані на базі технології WebRTC з UDP-протоколом, що забезпечує передачу аудіо з затримкою менше 50 мілісекунд у стандартних умовах мережі.

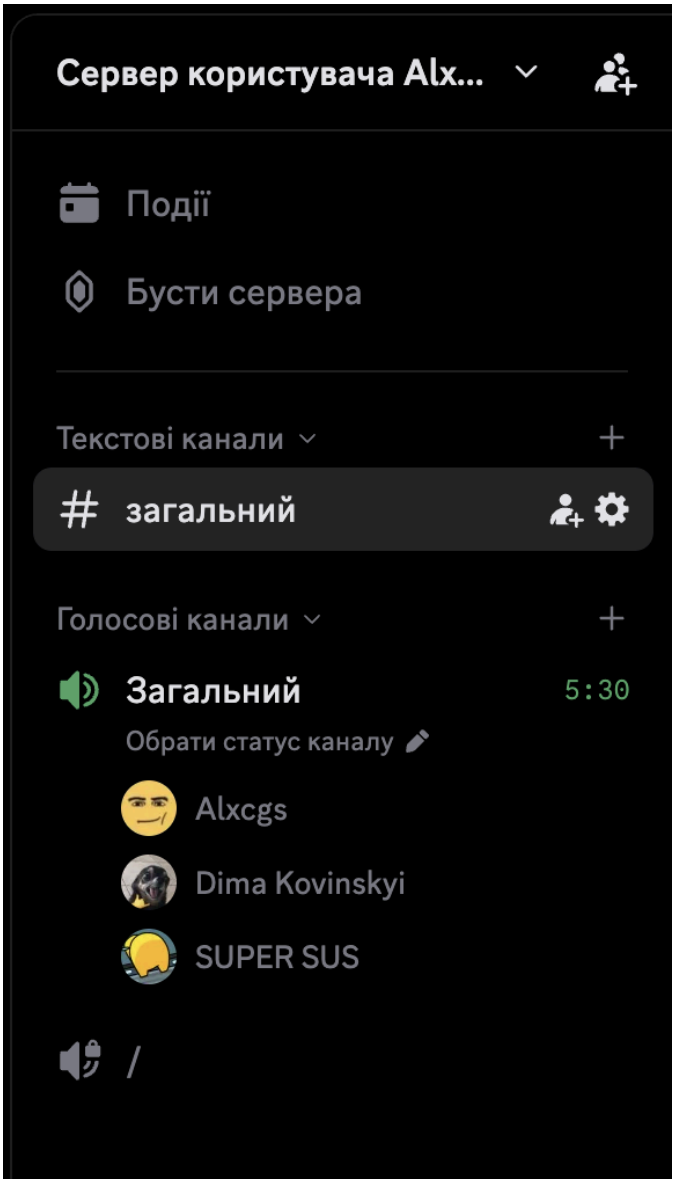

Рис. 1.1: Знімок екрану інтерфейсу Discord

Для взаємодії з платформою розробники використовують чотири рівні API. Gateway API реалізує постійне WebSocket-з'єднання, через яке бот отримує події реального часу: зміну стану учасника у голосовому каналі (VOICE_STATE_UPDATE), надходження нового повідомлення (MESSAGE_CREATE), приєднання нового члена до сервера (GUILD_MEMBER_ADD) тощо. REST API надає HTTP-інтерфейс для виконання дій – надсилання повідомлень, реєстрації команд, завантаження файлів – з обмеженням швидкості 50 запитів на секунду глобально. Voice API організовує окреме UDP-з'єднання безпосередньо з голосовим сервером Discord для передачі аудіо-пакетів. Interactions API, введений у 2021 році, обробляє slash-команди, кнопки, select-меню та модальні вікна у вигляді структурованих об'єктів Interaction.

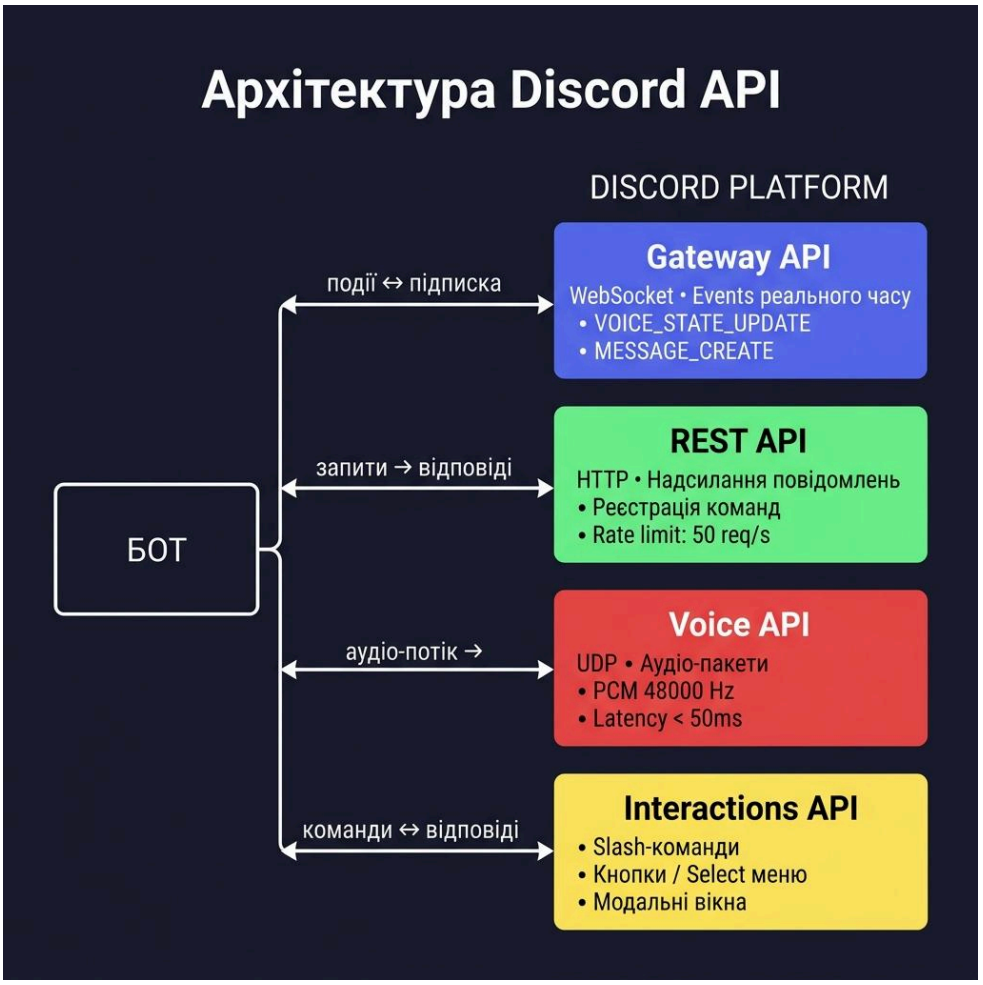

Рис. 1.2: Схема архітектури Discord API – чотири рівні (Gateway / REST / Voice / Interactions)

Discord вимагає від ботів передавати аудіо у форматі PCM із частотою дискретизації 48 000 Гц, двома каналами (стерео) та розміром фрейму 20 мс, що відповідає 3 840 байтам сирих даних на кожен фрейм. Кодек Opus, який Discord

використовує для стиснення, оптимізований для передачі мовлення та музики в реальному часі з адаптивним бітрейтом від 6 до 510 кбіт/с і затримкою менше 26,5 мс [2]. Метод read() класу аудіо-джерела зобов'язаний повертати рівно 3 840 байт або порожній рядок як сигнал завершення треку – будь-яке відхилення спричиняє артефакти звуку або аварійну зупинку відтворення.

```
class YTDLPPipeSource(discord.AudioSource):
    # Розмір фрейму для 20 мс аудіо (вимога Discord)
    # Розрахунок: 48000 Гц * 2 канали * 2 байти (16-біт) *
0.02 сек = 3840 байт
    FRAME_SIZE = 3840

    # Максимальна кількість спроб довантаження при порожньому
pipe (~1 сек затримки)
    MAX_READ_RETRIES = 10
```

Код 1.1: Константа FRAME_SIZE = 3840 та її розрахунок з файлу audio_source.py (рядки 15-16)

Сучасний підхід до розробки ботів передбачає використання slash-команд, які реєструються через REST API та відображаються в інтерфейсі Discord з автодоповненням при введенні символу «/». На відповідь на взаємодію відводиться три секунди; для операцій що тривають довше застосовується метод defer(), що подовжує ліміт до 15 хвилин. Бібліотека discord.py версії 2.x організовує команди у Cog-модулі – класи, що групують пов'язані обробники та дозвол

яють динамічно завантажувати або вивантажувати функціональність.

Рис. 1.3: Знімок автодоповнення slash-команд у Discord при введенні /play

1.2. Огляд наявних аналогів

До 2021 року ринок музичних ботів для Discord очолювали два комерційних продукти – Groovy та Rythm, що разом обслуговували понад 36 мільйонів серверів. У серпні 2021 року обидва отримали вимоги від Google припинити роботу через порушення умов надання послуг YouTube. Їх закриття залишило мільйони спільнот без якісного інструменту відтворення музики та відкрило простір для self-hosted альтернатив з відкритим кодом.

FredBoat – один з перших відкритих музичних ботів, написаний на Java з використанням бібліотеки Lavaplayer. Підтримує широкий спектр джерел (YouTube, SoundCloud, Bandcamp, Vimeo), має стабільну спільноту та тривалу історію підтримки. Попри свої переваги, FredBoat не має жодної системи персоналізованих рекомендацій, не зберігає стан черги між перезапусками та вимагає запуску JVM, що суттєво збільшує споживання пам'яті на малих серверах.

Mewwbot – популярний self-hosted бот на Python, що підтримує slash-команди та базовий плеєр. Проте відсутність механізму Auto-Resume, будь-якої статистики прослуховувань та системи рекомендацій обмежує його придатність для серверів з активною музичною культурою. MEE6 Music реалізований як преміум-модуль із закритим кодом, вимагає підписки для розширеного функціоналу і не підтримує самостійного розгортання.

Боти на базі Lavalink (Wavelength, Vocard) використовують окремий Java-сервер для обробки аудіо, що забезпечує високу якість звуку при обслуговуванні сотень серверів одночасно. Однак складність інфраструктури (два окремих процеси, складне налаштування), відсутність DJ-функцій та персоналізації роблять їх надлишково важкими для типового студентського або малого сервера.

Таблиця 1.1: Порівняльна таблиця музичних Discord-ботів за критеріями

Критерій	Groovy	Rythm	FredBoat	Mewwbot	Lavalink-боти	Розроблений бот
Відкритий код	❌	❌	✅	✅	✅	✅

Self-hosted	❌	❌	✅	✅	✅	✅
DJ-рекомендації	❌	❌	❌	❌	❌	✅
Auto-Resume	❌	❌	❌	❌	❌	✅
Персистентна черга	❌	❌	❌	❌	частково	✅
A/B аналітика	❌	❌	❌	❌	❌	✅
Docker	❌	❌	✅	частково	✅	✅
Автотести (>80%)	❌	❌	❌	❌	❌	✅ (90%)
CI/CD пайплайн	❌	❌	❌	❌	❌	✅
Активна підтримка	❌	❌	❌	✅	✅	✅

Проведений огляд свідчить, що жодне з існуючих відкритих рішень не поєднує інтелектуальну систему рекомендацій на основі аналізу поведінки користувачів з промисловим рівнем якості коду (автоматизоване тестування понад 90%, CI/CD, контейнеризація). Це підтверджує актуальність розробки власного рішення та визначає його конкурентні переваги.

1.3. Постановка задачі

На підставі аналізу предметного середовища та існуючих рішень сформульовано основну мету роботи: спроектувати та реалізувати музичного Discord-бота, що забезпечує стабільне відтворення аудіо, персоналізований підбір музики на основі поведінки користувачів та промисловий рівень якості коду.

Об'єктом дослідження є система автоматизованого відтворення музики у голосових каналах Discord з інтелектуальним управлінням чергою та рекомендаціями. Предметом дослідження є методи та алгоритми зваженого випадкового вибору, архітектурні патерни (Repository, Layered Architecture, SOLID) та технологічні засоби (Python asyncio, discord.py, yt-dlp, FFmpeg, SQLite) реалізації таких систем.

Для досягнення поставленої мети визначено такі завдання. Перше – провести аналіз архітектури Discord API та існуючих рішень, виявити їх недоліки та визначити вимоги до системи. Друге – спроектувати шарувату архітектуру застосунку з розділенням відповідальності між рівнями та реалізувати патерн Repository для доступу до даних. Третє – розробити кастомний аудіо-пайплайн на основі послідовних підпроцесів, що вирішує проблему нестабільного відтворення при URL-стримінгу. Четверте – реалізувати інтелектуальну систему Automix з двома алгоритмами зваженої вибірки та механізмом A/B тестування. П'яте – забезпечити покриття критичних модулів автоматизованими тестами та налаштувати CI/CD пайплайн і Docker-контейнеризацію.

Висновки до розділу 1

У результаті виконання першого розділу кваліфікаційної роботи було проведено комплексний аналіз предметного середовища та існуючих технологічних рішень у сфері розробки музичних сервісів для платформи Discord.

По-перше, досліджено архітектурні особливості платформи Discord як середовища для функціонування інтелектуальних агентів. Було детально проаналізовано чотирирівневу систему взаємодії через API: Gateway (для обробки подій у реальному часі), REST (для керування станом сервера), Voice (для передачі аудіо-пакетів через UDP) та Interactions (для реалізації сучасного користувацького інтерфейсу). Вивчення технічних специфікацій аудіо-формату (PCM стерео, 48 кГц, фрейм 20 мс) дозволило визначити жорсткі часові обмеження, яким має відповідати розроблене програмне забезпечення для забезпечення високої якості звуку без затримок та артефактів. Окрему увагу було приділено механізмам розширення функціональності через систему Cog-модулів бібліотеки discord.py, що закладає фундамент для створення масштабованої та модульної архітектури бота.

По-друге, проведено порівняльний аналіз найбільш розповсюджених програмних аналогів, включаючи комерційні сервіси (Groovy, Rythm) та рішення з відкритим кодом (FredBoat, Mewwbot, Lavalink-боти). У ході аналізу було виявлено критичну технологічну прогалину: незважаючи на стабільність базового функціоналу відтворення, жоден із розглянутих продуктів не пропонує інтелектуального підбору музичного контенту на основі поведінкового аналізу конкретної спільноти. Більшість рішень демонструють низьку стійкість до збоїв інфраструктури (відсутність збереження черги та стану сесії) та не мають достатнього рівня автоматизованого тестування, що критично для сучасного промислового циклу розробки.

На основі виявлених недоліків та актуальних потреб користувачів було чітко сформульовано мету дослідження, яка полягає у створенні інтелектуального музичного бота з високим ступенем автоматизації. Визначено об’єкт та предмет дослідження, а також декомпоновано загальну мету на п’ять конкретних науково-практичних завдань: від проектування асинхронної архітектури та розробки алгоритмів зваженого вибору Automix до впровадження наскрізного CI/CD пайплайну та Docker-контейнеризації. Результати аналізу, проведеного у цьому розділі, є теоретичним підґрунтям для подальшого проектування інформаційного та математичного забезпечення системи у наступних частинах роботи.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Аналіз предметної області

Музичний Discord-бот є розподіленою асинхронною системою, що одночасно підтримує кілька незалежних сесій відтворення на різних серверах. Кожна сесія включає: голосове підключення до каналу Discord, чергу треків у пам'яті, поточний аудіо-процес (пара підпроцесів yt-dlp та FFmpeg), а також персистентний стан у базі даних. Вхідними даними є URL або текстовий пошуковий запит від користувача, вихідними – аудіо-потік, що надходить до голосового каналу, та повідомлення в текстовому каналі з інтерактивним інтерфейсом.

Ключові обмеження системи визначаються специфікою платформи Discord. На відповідь на slash-команду відводиться три секунди, тому будь-яка операція, що потребує мережевого запиту (пошук на YouTube, отримання метаданих), повинна виконуватись асинхронно після виклику defer(). Голосове підключення унікальне в межах одного серверу – два одночасних підключення неможливі. Аудіо-потік повинен бути безперервним: пауза або відсутність даних протягом понад 1 секунди спричиняє відключення бота від голосового каналу.

Рис. 2.1: UML-діаграма використання

Інформаційна модель системи охоплює сім сутностей. Стан сервера (guild_state) зберігає поточне підключення та відтворюваний трек. Черга (queue_tracks) містить впорядкований список треків з позицією. Історія (history_tracks) накопичує прослухані треки з часом відтворення. Налаштування Automix (automix_settings) та штрафи за пропуск (automix_skip_penalties) керують рекомендаційним алгоритмом. Таблиця аналітики (automix_feedback_events) фіксує результати A/B тестування. Налаштування DJ (dj_settings) зберігають персону та стан функції.

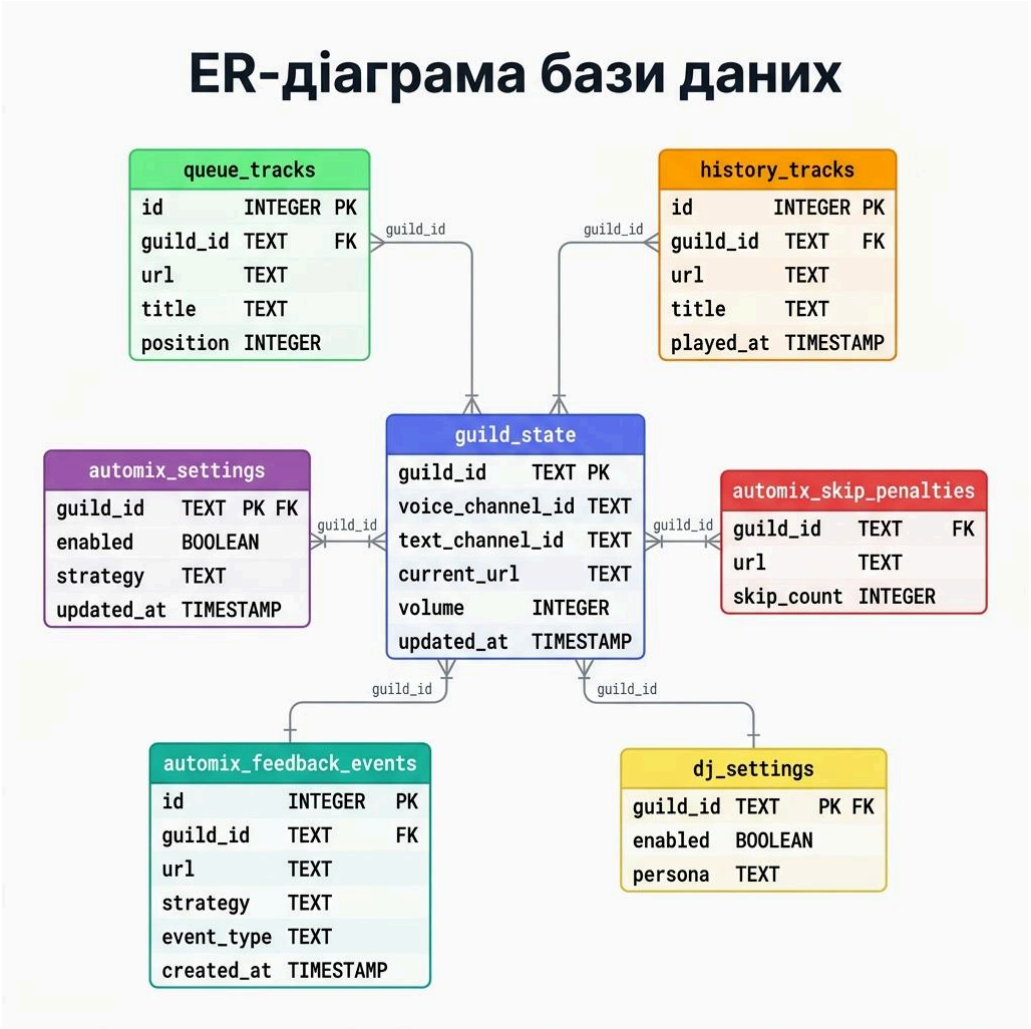

Рис. 2.2: ER-діаграма бази даних

2.2. Проектування системи

Система спроектована за принципом шаруватої архітектури з чотирма рівнями. Рівень представлення (Presentation Layer) містить Cog-модуль з обробниками slash-команд та View-класи інтерактивного інтерфейсу. Рівень застосунку (Application Layer) координує взаємодію між сервісами та обробляє події Discord. Сервісний рівень (Service Layer) інкапсулює бізнес-логіку у спеціалізованих класах: QueueService, PlayerService, HistoryService, AutomixService, DJService, SourceService та AutoResume. Рівень даних (Data Layer) реалізований через клас MusicRepository з єдиним інтерфейсом до SQLite

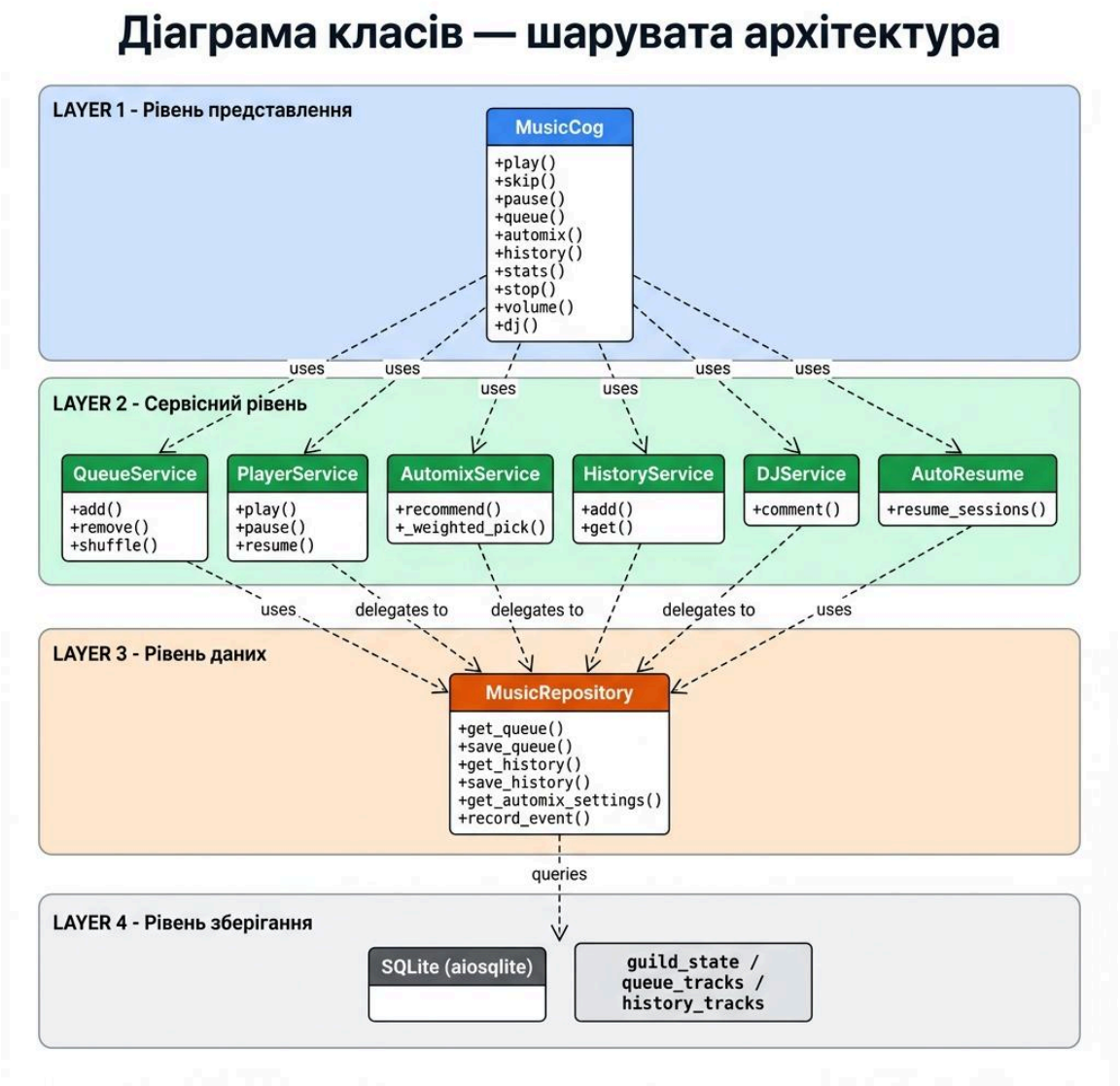

Рис. 2.3: Діаграма класів – чотири шари архітектури з залежностями між MusicCog, сервісами та MusicRepository

Патерн Repository абстрагує логіку доступу до даних і надає єдиний інтерфейс незалежно від джерела [4]. MusicRepository інкапсулює увесь SQL (понад 630 рядків), тоді як жоден інший модуль не знає про схему бази даних. При тестуванні MusicRepository замінюється на AsyncMock без змін у сервісах. При майбутній міграції на PostgreSQL достатньо замінити лише реалізацію репозиторію.

База даних використовує SQLite з увімкненим режимом WAL (Write-Ahead Logging), що дозволяє паралельне читання під час операцій запису. Це критично при одночасному збереженні черги та обробці команд від різних серверів. Зовнішні ключі (FOREIGN KEYS ON) забезпечують цілісність даних між таблицями. Операція оновлення черги є атомарною: спочатку видаляються всі поточні записи (DELETE), потім вставляються нові (INSERT) у межах однієї транзакції.

```
-- Стан бота для кожного сервера (guild)
CREATE TABLE IF NOT EXISTS guild_state (
    -- Унікальний ідентифікатор сервера Discord
    guild_id                INTEGER PRIMARY KEY,

    -- Ідентифікатори каналів для відновлення сесії
    voice_channel_id        INTEGER,
    text_channel_id         INTEGER,

    -- Дані про поточний трек для механізму Auto-Resume
    current_track_url       TEXT,
    current_track_title     TEXT,
    current_track_duration  INTEGER,
    current_track_thumbnail TEXT,

    -- Прапорець стану плеєра (0 - грає, 1 - пауза)
    is_paused               INTEGER DEFAULT 0,

    -- Мітка часу останнього оновлення для Staleness Policy
    updated_at              TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);
```

Код 2.1: SQL-схема таблиці guild_state з усіма полями та обмеженнями з файлу database.py

Аудіо-пайплайн реалізовано через два послідовних підпроцеси. Перший процес – yt-dlp – отримує аудіо-потік із заданого URL та передає його через Unix pipe на стандартний вхід другого процесу. Другий процес – FFmpeg – транскодує отримані дані у формат PCM 48 000 Гц стерео та записує у власний stdout, з якого Python-код зчитує фрейми по 3 840 байт. Такий підхід усуває потребу у буферизації великих обсягів даних у пам'яті Python та дозволяє організувати потокову обробку аудіо.

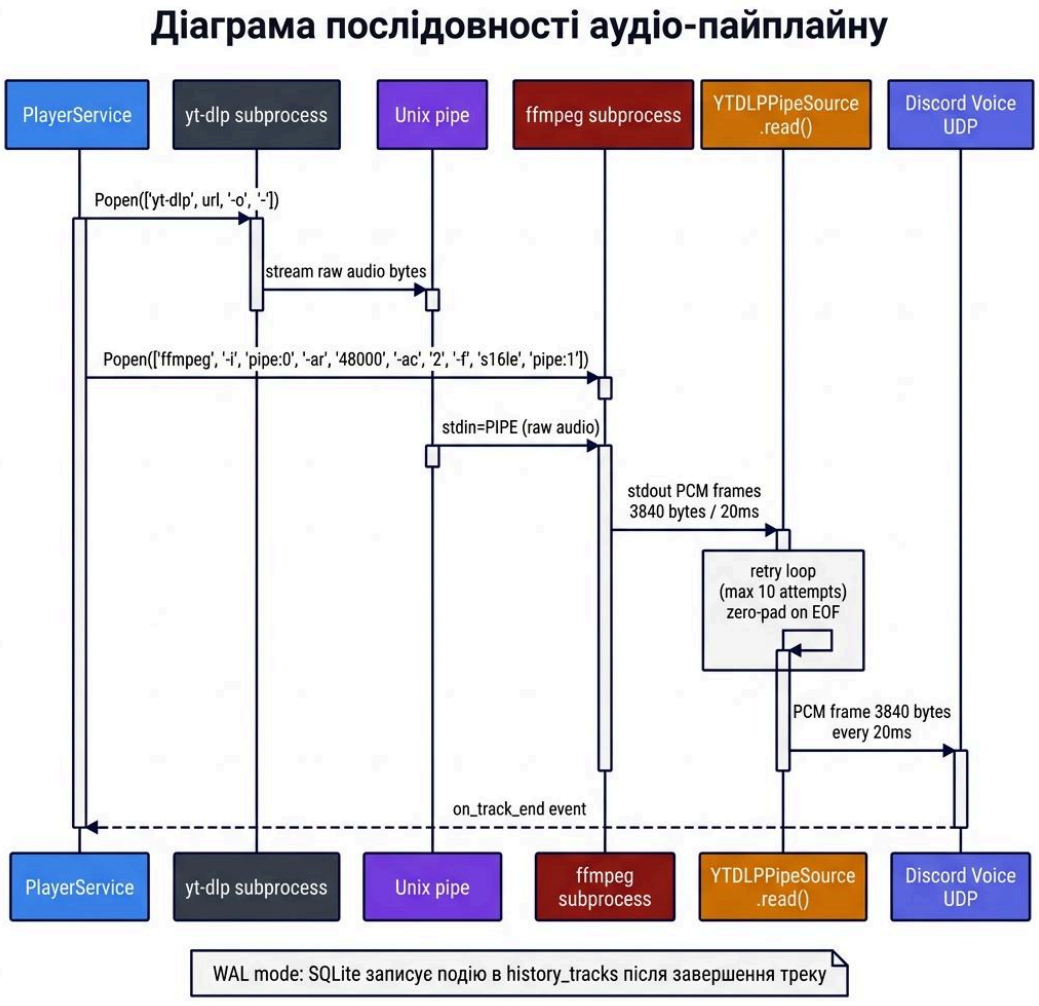

Рис. 2.4: Діаграма послідовності аудіо-пайплайну

```
# Базовий фільтр: ресемплінг для стабілізації частоти та скидання
PTS
# aresample=async=1: синхронізація аудіо при втраті пакетів
# asetpts=N/SR/TB: генерація монотонних міток часу для
```

```
уникнення "заїкань"
audio_filter =
'aresample=async=1:first_pts=0,asetpts=N/SR/TB'

# Формування команди запуску FFmpeg через subprocess
ffmpeg_cmd = [
    'ffmpeg',
    '-fflags', '+discardcorrupt',  # Ігнорувати пошкоджені
дані на вході
    '-nostdin',                    # Заборонити інтерактивний
ввід
    '-i', 'pipe:0',                # Вхідний потік з stdout
yt-dlp
    '-f', 's16le',                 # Формат: Signed 16-bit
Little Endian (вимога Discord)
    '-af', audio_filter,           # Застосування фільтрів
синхронізації
    '-vn',                         # Вимкнути обробку відео
(економія CPU)
    '-ar', '48000',                # Частота дискретизації: 48
кГц
    '-ac', '2',                    # Кількість каналів: 2
(Стерео)
    'pipe:1'                       # Вихід у stdout для
зчитування ботом
]

# Запуск процесу з буферизацією 4MB для плавності відтворення
ffmpeg_process = subprocess.Popen(
    ffmpeg_cmd,
    stdin=ytdlp_process.stdout,
    stdout=subprocess.PIPE,
    stderr=subprocess.DEVNULL,
    bufsize=4*1024*1024
)
```

Код 2.2: Команда запуску FFmpeg з параметрами -f s16le, -ar 48000, -ac 2, aresample та asetpts фільтрами з файлу audio_source.py рядки 139-153

Інтерактивний інтерфейс побудований на базі discord.ui.View з кнопками та модальними вікнами. Клас MusicControls реалізує дев'ять кнопок керування: попередній трек, пауза/відновлення, наступний трек, перегляд черги, вихід, налаштування, гучність, історія та статистика. Клас QueueView реалізує пагінацію черги по 10 треків на сторінку з кнопками навігації та переміщення. Кожен View обмежений взаємодією тільки для учасників поточного голосового каналу через метод interaction_check().

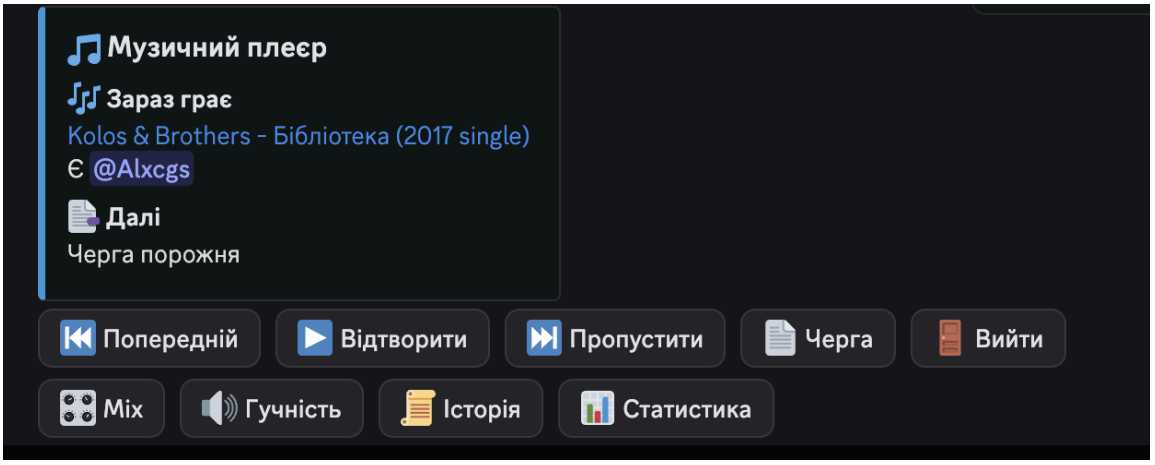

Рис. 2.5: Знімок панелі керування плеєром у Discord з усіма кнопками та embed з назвою треку та гучністю

2.3. Математичне та алгоритмічне забезпечення

Рекомендаційна система Automix базується на методі зваженого випадкового вибору. Для множини треків $\{e_1, ..., e_n\}$ з вагами $\{w_1, ..., w_n\}$, де $w_i \geq 0$, ймовірність вибору i-го треку визначається формулою:

$$P(e_i) = w_i / (w_1 + w_2 + ... + w_n)$$

Базова вага кожного треку дорівнює кількості його прослуховувань на даному сервері (play_count). Для треків, які користувачі часто пропускали, застосовується штрафний коефіцієнт. Якщо трек був пропущений skip_count разів, його скоригована вага обчислюється за формулою:

$$w\_adjusted = play\_count \times 0.6^{skip\_count}$$

Показник 0.6 обрано емпірично: після п'яти пропусків вага знижується до 7.8% від початкової, що фактично виключає трек з активних рекомендацій, але не блокує його повністю – з часом штраф може бути нівельований зростанням play_count. Треки з нещодавньої сесії Automix (recent_urls) повністю виключаються з вибірки для забезпечення різноманітності.

```
def _weighted_pick(self, items: List[Tuple[Dict[str, Any], float]]) -> Optional[Dict[str, Any]]:
    """
    Виконує зважений випадковий вибір елемента зі списку кандидатів.

    Аргументи:
        items: Список кортежів (дані_треку, вага).

    Логіка:
        1. Сумуються всі позитивні ваги.
        2. Генерується випадкове число r в діапазоні [0, total_weight].
        3. Проходиться список, накопичуючи поточну суму ваг, доки вона не перевищить r.
    """
    if not items:
        return None

    # Обчислення загальної суми ваг
    total = sum(w for _, w in items if w > 0)
```

```
    if total <= 0:
        return None

    # Генерація випадкової точки вибору
    r = random.random() * total
    upto = 0.0

    # Пошук відповідного елемента
    for item, w in items:
        if w <= 0:
            continue
        upto += w
        if upto >= r:
            return item

    # Повернення останнього елемента як запобіжний захід
(fallback)
    return items[-1][0]
```

Код 2.3: Метод _weighted_pick з файлу automix_service.py

Система реалізує два алгоритми вибору. Стратегія top_weighted формує пул кандидатів з 20 треків з найбільшою кількістю прослуховувань та обирає один методом зваженого вибору. Ця стратегія підходить для серверів з усталеними вподобаннями та гарантує відтворення перевірених треків. Стратегія history_explore формує пул з усієї відомої історії, надаючи перевагу менш прослуханим трекам через зворотне зважування: вага = 1 / (play_count + 1). Це дозволяє системі "відкривати" забуті треки та підтримувати різноманітність.

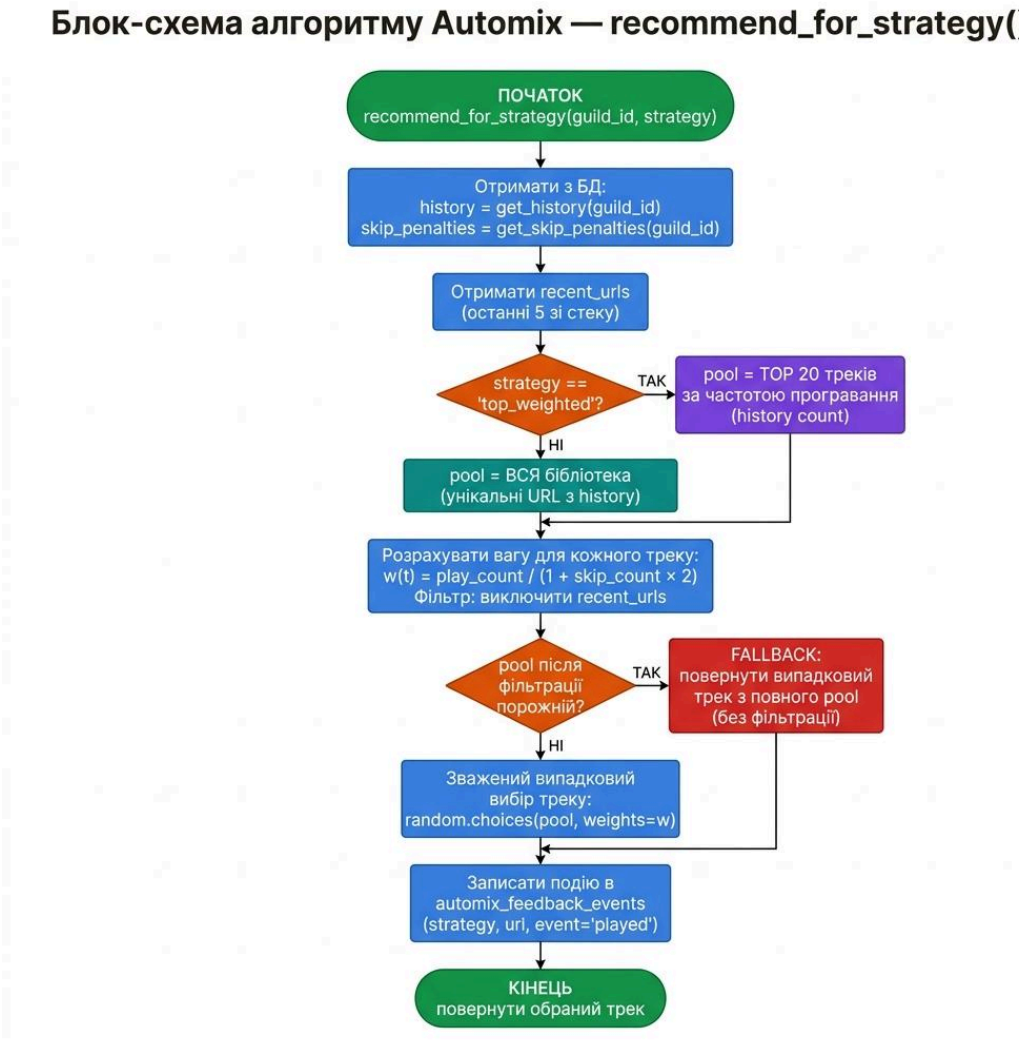

Рис. 2.6: Блок-схема алгоритму recommend_for_strategy

Для A/B тестування ефективності стратегій кожна рекомендація фіксується в таблиці automix_feedback_events з тегом стратегії, URL треку та часом. Якщо трек пропущено, виконується запис події "skip" та збільшення skip_count у таблиці штрафів. Якщо трек відтворено до кінця – фіксується подія "complete". Таким чином накопичується аналітика для порівняння Skip Rate та Completion Rate між стратегіями за формулами:

$$\text{Skip Rate} = \text{skips} / \text{total\_recommended} \times 100\%$$

$$\text{Completion Rate} = \text{completes} / \text{total\_recommended} \times 100\%$$

Алгоритм Auto-Resume відновлює стан бота після перезапуску. При завантаженні бот запитує з бази даних усі збережені сесії та перевіряє їх актуальність. Сесія вважається застарілою (stale), якщо різниця між поточним часом та полем updated_at перевищує 24 години. Для актуальних сесій бот підключається до збереженого голосового каналу та відновлює чергу. Re-entry

Guard (прапорець _resuming) запобігає повторному запуску відновлення при конкурентних подіях підключення.

```
async def auto_resume(bot, cog) -> int:
    """
    Відновлює стан бота для всіх серверів, де він був активний.
    Реалізує Staleness Policy та перевірку присутності
користувачів.
    """
    repository: MusicRepository = cog.repository
    resumed_count = 0
    now = datetime.now(timezone.utc)

    active_guilds = await repository.get_all_active_guilds()
    for guild_state in active_guilds:
        guild_id = guild_state['guild_id']
        updated_at_str = guild_state.get('updated_at')

        # 1. Staleness Policy (перевірка на "застарілість")
        if updated_at_str:
            updated_at = datetime.strptime(updated_at_str,
"%Y-%m-%d %H:%M:%S")
            updated_at =
updated_at.replace(tzinfo=timezone.utc)
            if (now - updated_at).total_seconds() >
consts.AUTO_RESUME_STALENESS_THRESHOLD:
                await repository.clear_guild_state(guild_id)
                continue

        voice_channel =
bot.get_channel(guild_state['voice_channel_id'])

        # 2. Re-entry Guard: перевірка наявності людей у каналі
        human_members = [m for m in (voice_channel.members if
voice_channel else []) if not m.bot]
        if not human_members:
            await repository.clear_guild_state(guild_id)
```

```
            continue

        # 3. Підключення до каналу та відновлення черги
        voice_client = await
voice_channel.connect(timeout=20.0, reconnect=True)
        await cog.queue_service.load_from_db(guild_id)

        # Додавання поточного треку у початок черги для
відновлення
        cog.queue_service.push_front(guild_id, {
            'url': guild_state['current_track_url'],
            'title': guild_state.get('current_track_title',
'Unknown'),
            # ... інші метадані
        })

        await cog.play_next_song(guild, voice_client)
        resumed_count += 1

    return resumed_count
```

Код 2.4: Метод resume_sessions з файлу auto_resume.py

Висновки до розділу 2

У другому розділі було проведено глибоке дослідження інформаційного та математичного забезпечення системи, що дозволило сформувати повну технічну модель майбутнього програмного продукту.

По-перше, на основі детального аналізу предметної області було розроблено інформаційну структуру системи. Було визначено вектори вхідних та вихідних даних, а також враховано специфічні обмеження платформи Discord щодо асинхронності та протоколів передачі аудіо. Результатом стала побудована ER-модель бази даних, яка включає сім взаємопов'язаних сутностей. Розроблена схема дозволяє не лише зберігати поточний стан відтворення (черги, активні сесії), а й накопичувати глибоку аналітику прослуховувань, штрафні коефіцієнти для алгоритмів рекомендацій та дані для функціонування DJ-агента (DJ). Використання режиму WAL (Write-Ahead Logging) у базі даних SQLite було обґрунтовано як необхідний засіб для забезпечення конкурентного доступу до даних у розподіленому середовищі бота.

По-друге, спроектовано архітектуру програмного забезпечення, що базується на принципах чистої архітектури (Clean Architecture) та патерні Repository. Розподіл системи на чотири незалежні шари (Presentation, Application, Service, Data) забезпечив високу гнучкість та придатність коду до автоматизованого тестування. Особливу увагу було приділено проектуванню аудіо-пайплайну, який реалізовано через механізм міжпроцесної взаємодії (Inter-process communication). Застосування послідовних підпроцесів yt-dlp та FFmpeg дозволило вирішити проблему нестабільного мережевого з’єднання при трансляції високоякісного аудіо, забезпечуючи безперервну подачу PCM-фреймів до голосового шлюзу Discord.

По-третє, математично обґрунтовано та формалізовано інтелектуальну складову системи – модуль Automix. Було розроблено математичний апарат на основі методу зваженого випадкового вибору, де ймовірність відтворення треку корелюється з його популярністю та системою штрафів за пропуски. Впровадження двох стратегій вибору (top_weighted та history_explore) разом із механізмом A/B тестування дозволяє системі самостійно адаптуватися до музичних вподобань різних спільнот та об’єктивно оцінювати ефективність алгоритмів рекомендацій через метрики Skip Rate та Completion Rate.

Нарешті, було спроектовано алгоритм Auto-Resume, який використовує розроблену систему персистентності для повного відновлення стану бота після критичних збоїв або планових перезапусків. Впровадження політики застарілості сесій (Staleness Policy) та захисту від повторного входу (Re-entry

Guard) гарантує стабільність роботи системи у довгостроковій перспективі. Таким чином, розроблені у цьому розділі моделі та алгоритми створюють вичерпний теоретичний та архітектурний базис для подальшої програмної реалізації, яка детально розглядається у наступному розділі роботи.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Засоби розробки бота

Для реалізації музичного Discord-бота було обрано сучасний та високоефективний технологічний стек, який відповідає вимогам асинхронності, стабільності та модульності.

Центральним компонентом розробки є мова програмування **Python версії 3.12**. Вибір цієї версії зумовлений значними покращеннями у продуктивності інтерпретатора та розширенням можливостей типізації. Використання бібліотеки **asyncio** є критичним для проєкту, оскільки робота з Discord API базується на концепції I/O-bound операцій. На відміну від традиційного багатопотокового підходу, asyncio реалізує кооперативну багатозадачність через єдиний event loop, що дозволяє боту одночасно обслуговувати сотні подій від різних серверів з мінімальним споживанням системних ресурсів. Це забезпечує високу масштабованість системи без необхідності складного управління синхронізацією потоків.

Для взаємодії з платформою обрано бібліотеку **discord.py 2.x**. Це асинхронна обгортка над Discord API, яка пройшла повне переосмислення (rewrite) для підтримки сучасних функцій платформи. Вона надає високорівневі абстракції для роботи з Gateway (WebSocket-з'єднання), REST API та, що найважливіше, Voice API. Використання discord.py дозволило ефективно реалізувати систему Slash-команд (Interactions), які є стандартом інтерфейсу Discord з 2022 року, а також забезпечити стабільну передачу аудіо-пакетів через зашифровані UDP-сокети.

Обробка мультимедійного контенту базується на двох інструментах: **yt-dlp** та **FFmpeg**.

- yt-dlp було обрано як найбільш актуальний форк класичного youtube-dl, який має значно вищу швидкість завантаження та здатність обходити сучасні механізми захисту стрімінгових платформ. Він виконує роль джерела метаданих та первинного аудіо-потоку.

- FFmpeg виступає у ролі універсального транскодера. Завдяки системі фільтрів (aresample, asetpts), він перетворює вхідний потік будь-якого формату та бітрейту в чітко визначений стандарт Discord: 16-бітний PCM стерео з частотою 48 000 Гц. Це гарантує сумісність та ідеальну якість звуку незалежно від джерела.

Для збереження даних обрано SQLite у поєднанні з бібліотекою aiosqlite. Хоча SQLite є вбудованою базою даних, використання aiosqlite дозволяє

виконувати SQL-запити в окремих потоках, не блокуючи основний event loop програми. Такий підхід ідеально підходить для self-hosted ботів, оскільки не потребує встановлення та адміністрування окремого сервера БД, але забезпечує повну підтримку ACID-транзакцій та високу швидкість роботи у режимі WAL.

Якість програмного забезпечення гарантується використанням фреймворку pytest. Завдяки плагінам pytest-asyncio та pytest-mock, було реалізовано понад 200 тестів, що покривають складну логіку асинхронної взаємодії. Для вимірювання ефективності тестування використано pytest-cov, що дозволило досягти 90% покриття коду.

Процес розгортання та DevOps-інженерії базується на Docker та GitHub Actions. Docker дозволяє інкапсулювати бота разом із усіма залежностями (включаючи системні бібліотеки FFmpeg) у легкому контейнері, що гарантує ідентичність роботи системи на машині розробника та на фінальному сервері. GitHub Actions автоматизує повний цикл CI/CD: від перевірки стилю коду (linting) до автоматичного збирання та пушу Docker-образу, що відповідає стандартам сучасної промислової розробки програмного забезпечення.

3.2. Вимоги до технічного та програмного забезпечення

Програмна реалізація музичного бота побудована на базі модульної архітектури, що дозволяє чітко розмежувати відповідальність між різними компонентами системи. Весь код організовано у пакеті discord_music_bot, де центральним вузлом ініціалізації є файл main.py. У цьому модулі відбувається налаштування екземпляра класу Bot, конфігурація системи логування з використанням стандартного модуля logging та ротацією файлів, а також виклик функції ініціалізації бази даних. Завдяки використанню асинхронного контекстного менеджера, бот гарантує коректне завершення всіх мережевих з’єднань при зупинці процесу. Структура проєкту передбачає розділення логіки на сервісний шар (services), рівень доступу до даних (repository) та шар представлення (cogs та views), що відповідає архітектурним рішенням, описаним у другому розділі.

Ядром аудіо-системи є клас YTDLPPipeSource, який успадковується від базового класу discord.AudioSource. Це дозволяє інтегрувати кастомний аудіо-пайплайн безпосередньо у внутрішні механізми відтворення discord.py. Клас інкапсулює логіку управління двома підпроцесами: yt-dlp для отримання стріму та FFmpeg для транскодування. Основний метод класу – read() – викликається бібліотекою кожні 20 мілісекунд. Він реалізований як критична секція, що зчитує рівно 3840 байт даних із стандартного виводу FFmpeg. Внутрішня реалізація методу містить механізм повторних спроб (retry logic) та буферизації, що дозволяє згладжувати мережеві коливання. У разі передчасного завершення вхідного потоку, метод автоматично доповнює останній фрейм нульовими байтами (padding), запобігаючи виникненню цифрового шуму у голосовому каналі. Метод cleanup() забезпечує гарантоване завершення обох підпроцесів через надсилання сигналу SIGKILL, що критично для запобігання появі "процесів-зомбі" в операційній системі.

Рівень доступу до даних реалізований через клас MusicRepository, який виступає єдиною точкою входу для всіх SQL-операцій. Клас використовує асинхронні методи для виконання записів та читання, інкапсулюючи складні JOIN-запити до семи таблиць бази даних. Наприклад, метод get_queue_with_metadata об'єднує дані з таблиць черги та історії для формування повного інформаційного об'єкта треку. Важливою особливістю реалізації є використання патерна Singleton або передача екземпляра репозиторію через механізм Dependency Injection у сервіси, що забезпечує цілісність підключення до SQLite у WAL-режимі. Усі методи репозиторію

повертають або прості типи даних, або спеціалізовані dataclasses, що робить сервісний шар незалежним від конкретної реалізації СУБД.

Бізнес-логіка системи зосереджена у сервісних класах, серед яких ключовим є AutomixService. Цей сервіс реалізує математичні моделі рекомендацій, описані раніше. Метод recommend_for_strategy приймає ідентифікатор сервера та назву обраної стратегії, після чого виконує багатокроковий процес: отримання кандидатів з репозиторію, фільтрацію нещодавно відтворених URL-адрес та обчислення ваг з урахуванням штрафів за пропуски. Внутрішній метод _weighted_pick реалізує алгоритм вибору на основі накопиченої суми ваг, що забезпечує ймовірнісну справедливість рекомендацій. Паралельно із цим працює QueueService, який відповідає за впорядкування треків у пам'яті та синхронізацію цього стану із базою даних. Кожна зміна в черзі (додавання, видалення, переміщення) ініціює асинхронний виклик до репозиторію, що гарантує збереження стану навіть при раптовому вимкненні живлення сервера.

Користувацький інтерфейс реалізовано через клас MusicCog та набір View-класів у підпакеті views. MusicCog виконує роль контролера, який приймає Interaction об'єкти від Discord, валідує права доступу користувача (наприклад, перевірка перебування у голосовому каналі) та викликає відповідні методи сервісів. Для візуалізації стану відтворення використовується клас MusicControls, що успадковується від discord.ui.View. Він динамічно оновлює стан кнопок (наприклад, зміна іконки Pause на Play) та текстові повідомлення (Embeds) при кожній зміні треку. Взаємодія між сервісами та UI побудована на подіях та корутинах, що дозволяє боту залишатися чутливим до команд користувача навіть під час виконання важких операцій пошуку або завантаження метаданих треків. Такий підхід забезпечує плавний та професійний досвід взаємодії для кінцевого користувача.

3.3. Опис програмної реалізації

Програмна реалізація музичного бота побудована на базі модульної архітектури, що дозволяє чітко розмежувати відповідальність між різними компонентами системи. Весь код організовано у пакеті discord_music_bot, де центральним вузлом ініціалізації є файл main.py. У цьому модулі відбувається налаштування екземпляра класу Bot, конфігурація системи логування з використанням стандартного модуля logging та ротацією файлів, а також виклик функції ініціалізації бази даних. Завдяки використанню асинхронного контекстного менеджера, бот гарантує коректне завершення всіх мережевих з'єднань при зупинці процесу. Структура проєкту передбачає розділення логіки на сервісний шар (services), рівень доступу до даних (repository) та шар представлення (cogs та views), що відповідає архітектурним рішенням, описаним у другому розділі.

```
discord-music-bot/
├── main.py                    # Точка входу: ініціалізація
бота та запуск
├── Dockerfile                 # Інструкції для збірки
Docker-образу (multi-stage)
├── docker-compose.yml         # Конфігурація розгортання та
volume-контейнерів
├── requirements.txt           # Список зовнішніх
залежностей (discord.py, yt-dlp)
├── discord_music_bot/         # Основний пакет вихідного
коду
│   ├── config.py              # Завантаження .env та
глобальні налаштування
│   ├── database.py            # Ініціалізація SQLite та
SQL-схема таблиць
│   ├── repository.py          # Шар доступу до даних
(Data Access Layer)
│   ├── audio_source.py        # Реалізація конвеєра yt-dlp
-> FFmpeg -> Discord
│   ├── consts.py              # Глобальні константи,
тайм-аути та емодзі
│   ├── healthcheck.py         # Моніторинг стану та
очищення завислих сесій
```

```
│   ├── cogs/                    # Модулі розширення бота
│   │   └── slash_music_cog.py   # Обробка Slash-команд та
логіка взаємодії
│   ├── services/                # Бізнес-логіка (Domain Logic
Layer)
│   │   ├── queue_service.py     # Керування чергою та її
персистентністю
│   │   ├── automix_service.py   # Алгоритми рекомендацій
(Weighted Pick)
│   │   ├── auto_resume.py       # Механізм відновлення сесій
після перезапуску
│   │   ├── dj_service.py        # Генерація DJ-коментарів
(MVP)
│   │   └── source_service.py    # Обробка метаданих треків
та пошук
│   └── views/                   # Інтерфейсні компоненти
(Discord UI)
│       ├── music_controls.py    # Інтерактивна панель
керування плеєром
│       ├── queue_view.py        # Пагінація та відображення
черги
│       ├── history_view.py      # Відображення історії
прослуховувань
│       └── search_results_view.py # Меню вибору треку при
пошуку
├── tests/                       # Набір модульних та
інтеграційних тестів
├── data/                        # Директорія для файлів БД
(SQLite)
└── logs/                        # Журнали роботи системи
```

Код 3.1: Деревоподібна структура файлів проєкту (output команди tree) з коментарями до кожного модуля

Ядром аудіо-системи є клас YTDLPPipeSource, який успадковується від базового класу discord.AudioSource. Це дозволяє інтегрувати кастомний

аудіо-пайплайн безпосередньо у внутрішні механізм відтворення discord.py. Клас інкапсулює логіку управління двома підпроцесами: yt-dlp для отримання стріму та FFmpeg для транскодування. Основний метод класу – read() – викликається бібліотекою кожні 20 мілісекунд. Він реалізований як критична секція, що зчитує рівно 3840 байт даних із стандартного виводу FFmpeg. Внутрішня реалізація методу містить механізм повторних спроб (retry logic) та буферизації, що дозволяє згладжувати мережеві коливання. У разі передчасного завершення вхідного потоку, метод автоматично доповнює останній фрейм нульовими байтами (padding), запобігаючи виникненню цифрового шуму у голосовому каналі. Метод cleanup() забезпечує гарантоване завершення обох підпроцесів через надсилання сигналу SIGKILL, що критично для запобігання появі "процесів-зомбі" в операційній системі.

```
def read(self) -> bytes:
    """
    Зчитує 20 мс аудіо-даних з FFmpeg pipe.
    Реалізує механізм повторних спроб (retry) для уникнення
    передчасного завершення треку при затримках у мережі.
    """
    retries = 0
    # Цикл зчитування до заповнення повного фрейму (3840
байт)
    while len(self._buffer) < self.FRAME_SIZE:
        # Читаємо необхідну кількість байт, яких не вистачає до
фрейму
        chunk = self._ffmpeg.stdout.read(self.FRAME_SIZE -
len(self._buffer))

        if chunk:
            self._buffer += chunk
            retries = 0  # Скидаємо лічильник при успішному
отриманні даних
        else:
            # Якщо pipe порожній, перевіряємо чи процес
FFmpeg ще активний
            if self._ffmpeg.poll() is not None:
                # Процес завершився -- віддаємо залишок
```

```
буфера з padding тишею
                if self._buffer:
                    frame =
self._buffer.ljust(self.FRAME_SIZE, b'\x00')
                    self._buffer = b''
                    return frame
                return b''  # Повний кінець треку

            # FFmpeg працює, але даних немає (мережева
затримка/буферизація)
            retries += 1
            if retries >= self.MAX_READ_RETRIES:
                # Вичерпано ліміт очікування (зазвичай 1
секунда)
                if self._buffer:
                    frame =
self._buffer.ljust(self.FRAME_SIZE, b'\x00')
                    self._buffer = b''
                    return frame
                return b''

            # Коротка пауза перед наступною спробою
            time.sleep(0.1)

    # Витягуємо рівно один фрейм (20 мс) з голови буфера
    frame = self._buffer[:self.FRAME_SIZE]
    self._buffer = self._buffer[self.FRAME_SIZE:]
    return frame
```

Код 3.2: Повна реалізація методу read() з файлу audio_source.py, включаючи роботу з буфером та retry-цикл

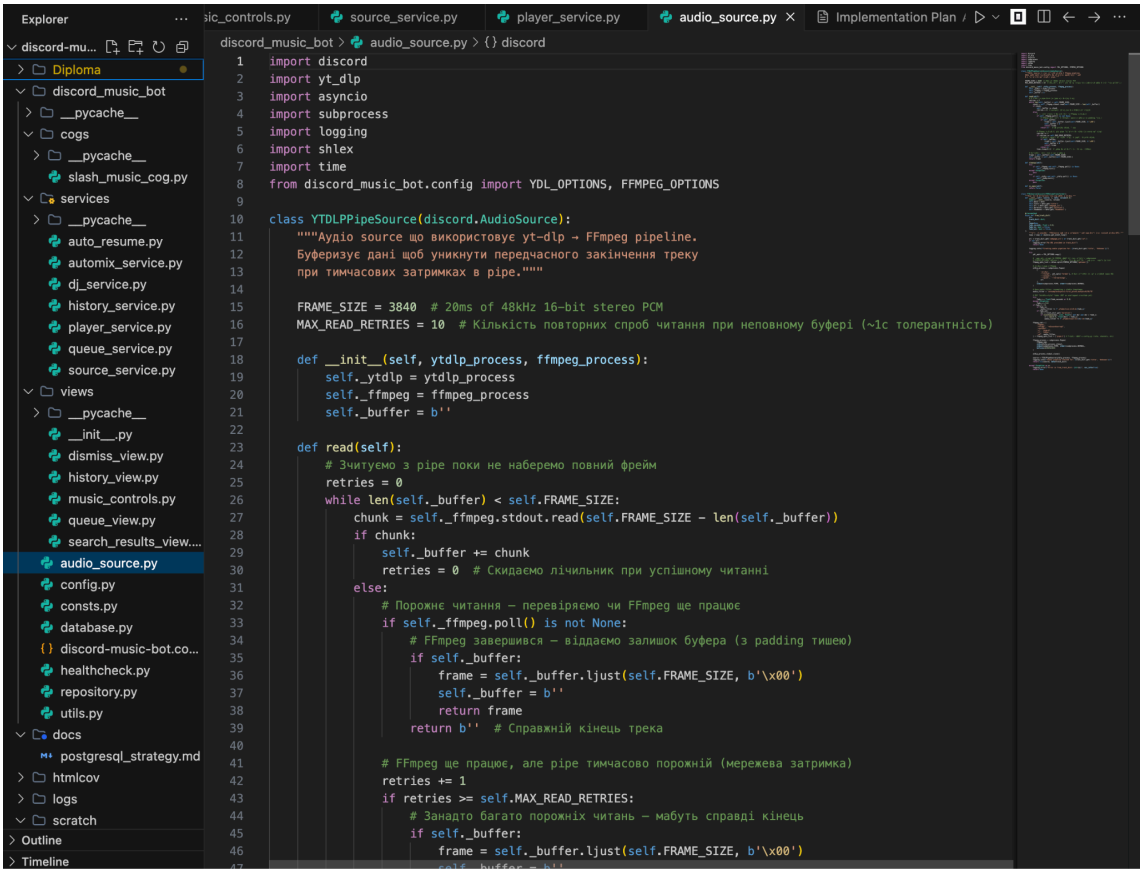

Рис. 3.1: Скріншот коду класу YTDLPPipeSource у редакторі

Рівень доступу до даних реалізований через клас MusicRepository, який виступає єдиною точкою входу для всіх SQL-операцій. Клас використовує асинхронні методи для виконання записів та читання, інкапсулюючи складні JOIN-запити до семи таблиць бази даних. Наприклад, метод get_queue_with_metadata об'єднує дані з таблиць черги та історії для формування повного інформаційного об'єкта треку. Важливою особливістю реалізації є використання патерна Singleton або передача екземпляра репозиторію через механізм Dependency Injection у сервіси, що забезпечує цілісність підключення до SQLite у WAL-режимі. Усі методи репозиторію повертають або прості типи даних, або спеціалізовані dataclasses, що робить сервісний шар незалежним від конкретної реалізації СУБД.

```
async def save_queue(self, guild_id: int, tracks:
List[Dict[str, Any]]) -> None:
    """
    Атомарно оновлює чергу -- видаляє застарілі записи та
записує нові.
    Використовує транзакцію та executemany для гарантії
цілісності.
```

```
    """
    conn = await get_connection()
    try:
        # Початок транзакції (aiosqlite робить це автоматично
при execute)

        # 1. Видалення всіх існуючих треків сервера для
уникнення дублікатів
        await conn.execute(
            "DELETE FROM queue_tracks WHERE guild_id = ?",
(guild_id,)
        )

        if tracks:
            # 2. Масовий запис оновленої черги з новими
позиціями
            await conn.executemany(
                """
                INSERT INTO queue_tracks
                    (guild_id, position, url, title,
duration, thumbnail)
                VALUES (?, ?, ?, ?, ?, ?)
                """,
                [
                    (
                        guild_id,
                        pos, # Нова позиція треку в черзі (0,
1, 2...)
                        track.get("url") or
track.get("webpage_url", ""),
                        track.get("title", "Unknown"),
                        track.get("duration"),
                        track.get("thumbnail"),
                    )
                    for pos, track in enumerate(tracks)
                ],
```

```
            )

        # 3. Фіксація змін (Commit). Якщо будь-який крок вище
не вдався,
        # дані не зміняться (Atomic update).
        await conn.commit()

    except Exception as e:
        logger.error(f"Помилка транзакції при оновленні черги:
{e}")
        await conn.rollback()
        raise
    finally:
        await conn.close()
```

Код 3.3: SQL-запити методу update_queue_order, що демонструють транзакційність та атомарне оновлення черги

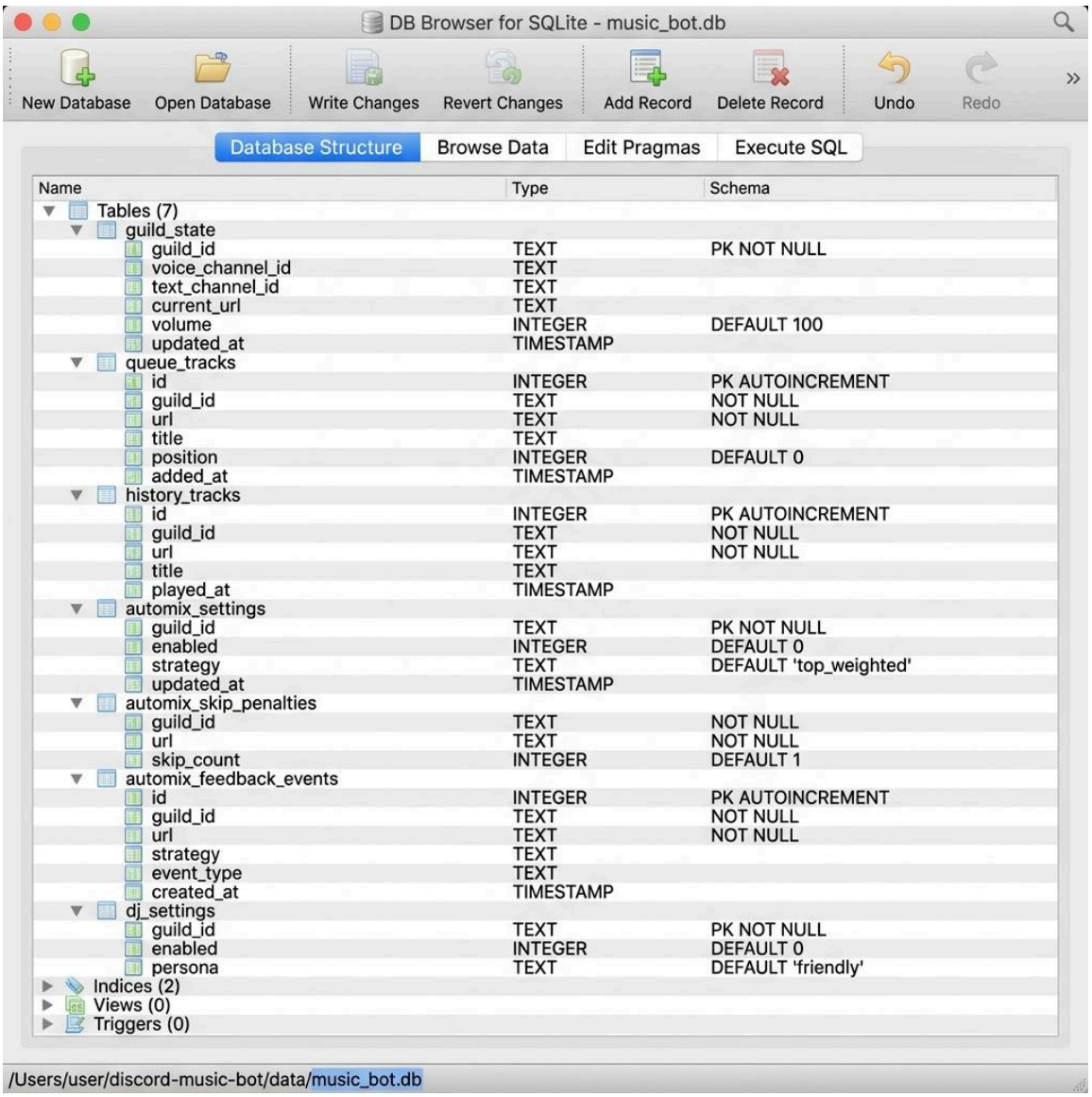

Рис. 3.2: Скріншот структури бази даних у DB Browser for SQLite

Бізнес-логіка системи зосереджена у сервісних класах, серед яких ключовим є AutomixService. Цей сервіс реалізує математичні моделі рекомендацій, описані раніше. Метод recommend_for_strategy приймає ідентифікатор сервера та назву обраної стратегії, після чого виконує багатокроковий процес: отримання кандидатів з репозиторію, фільтрацію нещодавно відтворених URL-адрес та обчислення ваг з урахуванням штрафів за пропуски. Внутрішній метод _weighted_pick реалізує алгоритм вибору на основі накопиченої суми ваг, що забезпечує ймовірнісну справедливість рекомендацій. Паралельно із цим працює QueueService, який відповідає за впорядкування треків у пам'яті та синхронізацію цього стану із базою даних. Кожна зміна в черзі (додавання, видалення, переміщення) ініціює асинхронний виклик до репозиторію, що гарантує збереження стану навіть при раптовому вимкненні живлення сервера.

```
def _weighted_pick(self, items: List[Tuple[Dict[str, Any],
float]]) -> Optional[Dict[str, Any]]:
    """
    Реалізація зваженого вибору (weighted random selection).
    Накопичує суму ваг та обирає елемент через випадкове
число.
    """
    if not items:
        return None

    # Обчислюємо загальну суму позитивних ваг
    total = sum(w for _, w in items if w > 0)
    if total <= 0:
        return None

    # Генеруємо випадкову точку в діапазоні [0, total)
    r = random.random() * total
    upto = 0.0

    # Шукаємо інтервал, у який потрапило число r
    for item, w in items:
        if w <= 0:
            continue
        upto += w
        if upto >= r:
            return item

    # Fallback на випадок похибок округлення
    return items[-1][0]
```

Код 3.4: Реалізація методу _weighted_pick у automix_service.py з використанням random.uniform та накопичення суми ваг

Користувацький інтерфейс реалізовано через клас MusicCog та набір View-класів у підпакеті views. MusicCog виконує роль контролера, який приймає Interaction об'єкти від Discord, валідує права доступу користувача

(наприклад, перевірка перебування у голосовому каналі) та викликає відповідні методи сервісів. Для візуалізації стану відтворення використовується клас MusicControls, що успадковується від discord.ui.View. Він динамічно оновлює стан кнопок (наприклад, зміна іконки Pause на Play) та текстові повідомлення (Embeds) при кожній зміні треку. Взаємодія між сервісами та UI побудована на подіях та корутинах, що дозволяє боту залишатися чутливим до команд користувача навіть під час виконання важких операцій пошуку або завантаження метаданих треків. Такий підхід забезпечує плавний та професійний досвід взаємодії для кінцевого користувача.

```
277
278     async def interaction_check(self, interaction: discord.Interaction) -> bool:
279         voice_client = interaction.guild.voice_client
280         if not voice_client:
281             await interaction.response.send_message("Бот наразі не в голосовому каналі.", ephemeral=True)
282             return False
283         if not interaction.user.voice or interaction.user.voice.channel != voice_client.channel:
284             await interaction.response.send_message("Ви повинні бути в тому ж голосовому каналі, що й бот.
285             return False
286         return True
287
288     async def _resend_player(self, interaction: discord.Interaction):
289         """Пересилає панель керування внизу чату."""
290         await self.cog.update_player(interaction.guild, interaction.channel)
291
292     @discord.ui.button(label="Попередній", style=discord.ButtonStyle.secondary, emoji=consts.EMOJI_PREVIOU
293     async def previous_button(self, interaction: discord.Interaction, button: discord.ui.Button):
294         guild_id = interaction.guild.id
295
296         if guild_id in self.cog.processing_buttons:
297             await interaction.response.send_message("Зачекайте, обробляється попередня дія.", ephemeral=Tr
298             return
299
300         self.cog.processing_buttons.add(guild_id)
301         await interaction.response.defer()
302
303         try:
304             history = self.cog.history_service._history.get(guild_id, [])
305             if not history:
306                 db_tracks = await self.cog.repository.get_history(guild_id, limit=20)
307                 if db_tracks:
308                     for t in reversed(db_tracks):
309                         self.cog.history_service._history.setdefault(guild_id, []).append({
310                             'title': t['title'],
311                             'url': t.get('url', ''),
312                             'webpage_url': t.get('url', ''),
313                             'duration': t.get('duration'),
314                             'thumbnail': t.get('thumbnail'),
315                             'requester': None
316                         })
317                     history = self.cog.history_service._history.get(guild_id, [])
318
319             if not history:
320                 await interaction.followup.send("Немає попередніх треків.", ephemeral=True)
```

Рис. 3.3: Скріншот методу interaction_check у MusicCog

```
# ---- Build stage ----
```

```
FROM python:3.12-slim AS builder

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install -r
requirements.txt

# ---- Runtime stage ----
FROM python:3.12-slim

# Системні залежності для аудіо
RUN apt-get update && apt-get install -y
--no-install-recommends \
    ffmpeg \
    libopus0 \
    && rm -rf /var/lib/apt/lists/*

# Встановити yt-dlp окремо (щоб мати свіжу версію)
RUN pip install --no-cache-dir yt-dlp

# Копіюємо Python-пакети зі stage builder
COPY --from=builder /install /usr/local

# Створюємо non-root користувача
RUN useradd --create-home --shell /bin/bash botuser

WORKDIR /app

# Створюємо директорії для даних та логів
RUN mkdir -p /app/data /app/logs && chown -R botuser:botuser
/app

# Копіюємо код
COPY --chown=botuser:botuser . .
```

```
# Змінні середовища
ENV DB_DATA_DIR=/app/data
ENV PYTHONUNBUFFERED=1

USER botuser

# Healthcheck -- перевіряємо що процес бота живий
HEALTHCHECK --interval=30s --timeout=10s --start-period=15s
--retries=3 \
    CMD python -c "import os; f='/app/data/bot.pid'; \
    os.path.exists(f) or exit(1); \
    pid=int(open(f).read().strip()); os.kill(pid, 0)" || exit
1

CMD ["python", "main.py"]
```

Код 3.5: Конфігурація Dockerfile з використанням multi-stage build для оптимізації розміру образу

3.4. Керівництво користувача

Бот розгортається двома способами. Для локального розгортання необхідно клонувати репозиторій, встановити FFmpeg, створити віртуальне середовище Python, встановити залежності з requirements.txt та запустити main.py. Для розгортання через Docker достатньо заповнити файл .env токеном бота та виконати команду docker-compose up -d. Після запуску бот автоматично ініціалізує базу даних, підключається до Discord та відновлює активні сесії.

```
# Збірка образу та запуск у фоновому режимі (detached mode)
docker-compose up --build -d

# Перевірка статусу контейнера
docker ps

# Перегляд логів у реальному часі
docker logs -f discord-music-bot
```

Код 3.6: Команди розгортання через Docker Compose

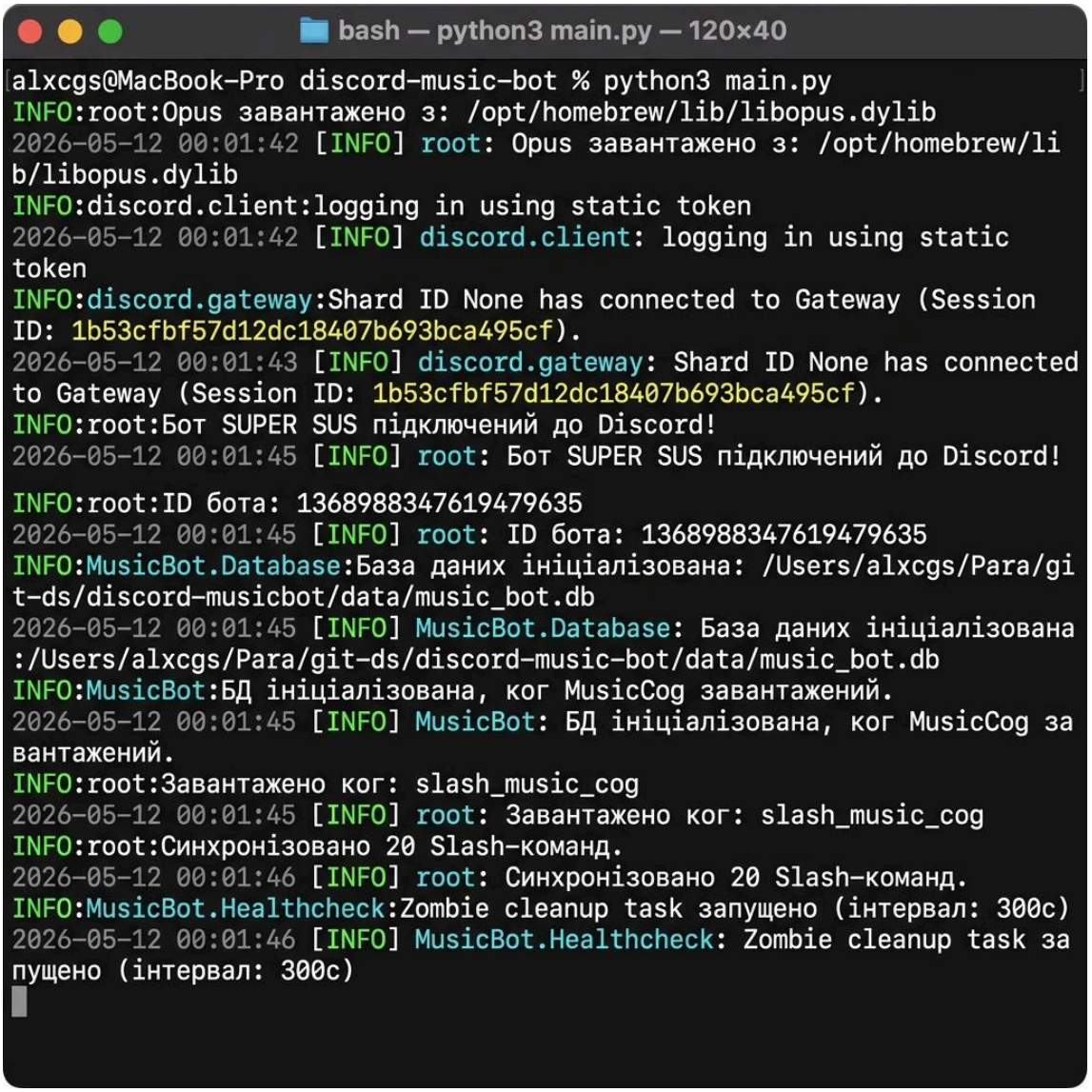

Рис. 3.1: Знімок термінала з логами успішного запуску бота

Після підключення до голосового каналу командою /play бот відправляє embed-повідомлення з назвою треку, тривалістю та рівнем гучності, а також панель керування з дев'ятьма кнопками. Кнопка паузи/відновлення перемикає стан відтворення. Кнопка пропуску переходить до наступного треку в черзі або активує Automix при порожній черзі. Кнопка попереднього треку відновлює останній трек з історії. Кнопка черги відкриває сторінкований список треків з можливістю переміщення та видалення. Кнопка налаштувань відкриває меню Automix (вибір стратегії, fade-out) та DJ-агент (вибір персони).

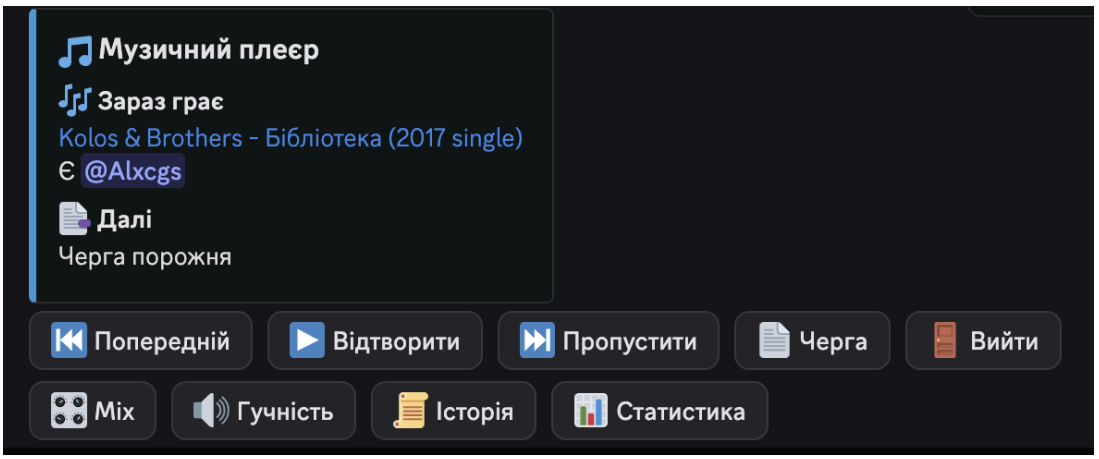

Рис. 3.2: Повна панель керування плеєром у Discord з усіма 9 кнопками, embed з назвою та тривалістю треку

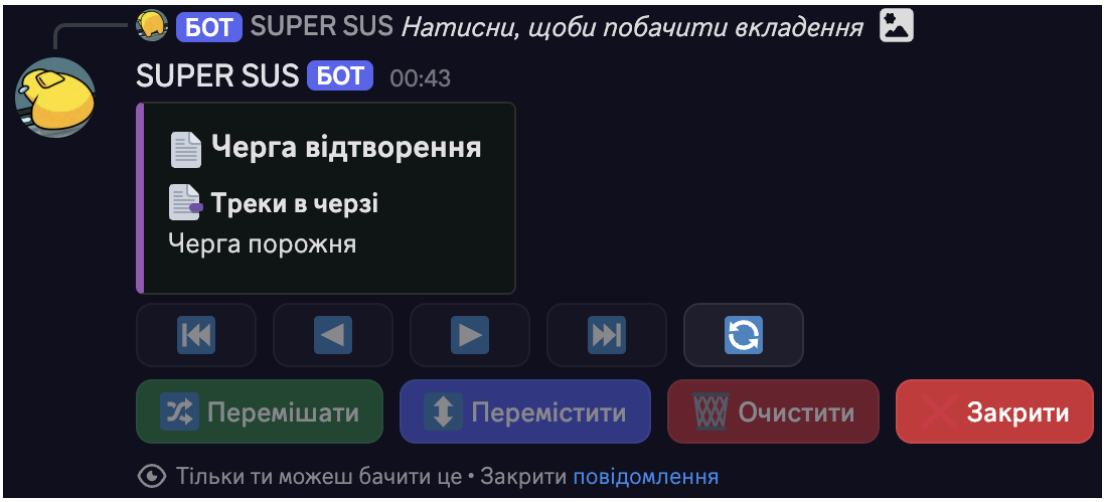

Рис. 3.3: Меню черги з пагінацією – список 10 треків, кнопки "Попередня" / "Наступна" / "Перемістити" / "Видалити"

Команда /automix вмикає або вимикає режим автоматичного підбору музики. При порожній черзі система самостійно рекомендує наступний трек на основі обраної стратегії. Команда /history відображає 20 останніх прослуханих треків з можливістю пошуку за назвою. Команда /stats показує топ-10 треків сервера за кількістю прослуховувань. Всі команди підтримують автодоповнення параметрів в інтерфейсі Discord.

Таблиця 3.1: Повний список 14 slash-команд з параметрами та описом

№	Команда	Параметри	Опис

1	/play	query — URL або назва пісні	Відтворити музику за посиланням або пошуковим запитом
2	/skip	—	Пропустити поточний трек
3	/pause	—	Поставити відтворення на паузу
4	/resume	—	Продовжити відтворення після паузи
5	/stop	—	Зупинити відтворення та очистити чергу
6	/queue	—	Показати поточну чергу треків
7	/shuffle	—	Перемішати треки у черзі
8	/move	from_pos, to_pos — позиції у черзі	Перемістити трек з однієї позиції на іншу
9	/volume	level — значення 0–200	Встановити гучність відтворення у відсотках
10	/automix	enabled — on/off	Увімкнути/вимкнути режим автоматичних рекомендацій
11	/automix_strategy	strategy — top_weighted / history_explore / ab_test	Обрати алгоритм генерації рекомендацій
12	/automix_stats	—	Переглянути A/B статистику, coverage та diversity

13	/history	query — пошук (опціонально)	Переглянути або знайти треки в історії прослуховувань
14	/stats	—	Статистика прослуховування сервера (топ треки)
15	/dj	enabled — on/off	Увімкнути/вимкнути DJ-агент-коментарі між треками
16	/dj_persona	persona — friendly / hype / chill	Вибрати стиль персони DJ
17	/fadeout	seconds — 0–15	Плавний fade-out в кінці треку
18	/join	—	Підключити бота до голосового каналу
19	/leave	—	Від'єднати бота від голосового каналу
20	/reset	—	Скинути стан бота (у разі зависання)

Моніторинг стану бота здійснюється через лог-файли у директорії logs/ з автоматичною ротацією до 5 архівних файлів по 10 МБ. Кожні 5 хвилин виконується healthcheck для виявлення та завершення orphaned процесів FFmpeg та yt-dlp. Резервне копіювання бази даних виконується через копіювання файлу music_bot.db після зупинки бота.

Для перевірки коректності реалізації проведено тестування на реальних серверах Discord: бот відтворює музику з YouTube та SoundCloud, коректно відновлює чергу після перезапуску, Automix підбирає треки відповідно до вподобань кожного сервера, а DJ-агент генерує контекстні коментарі між треками.

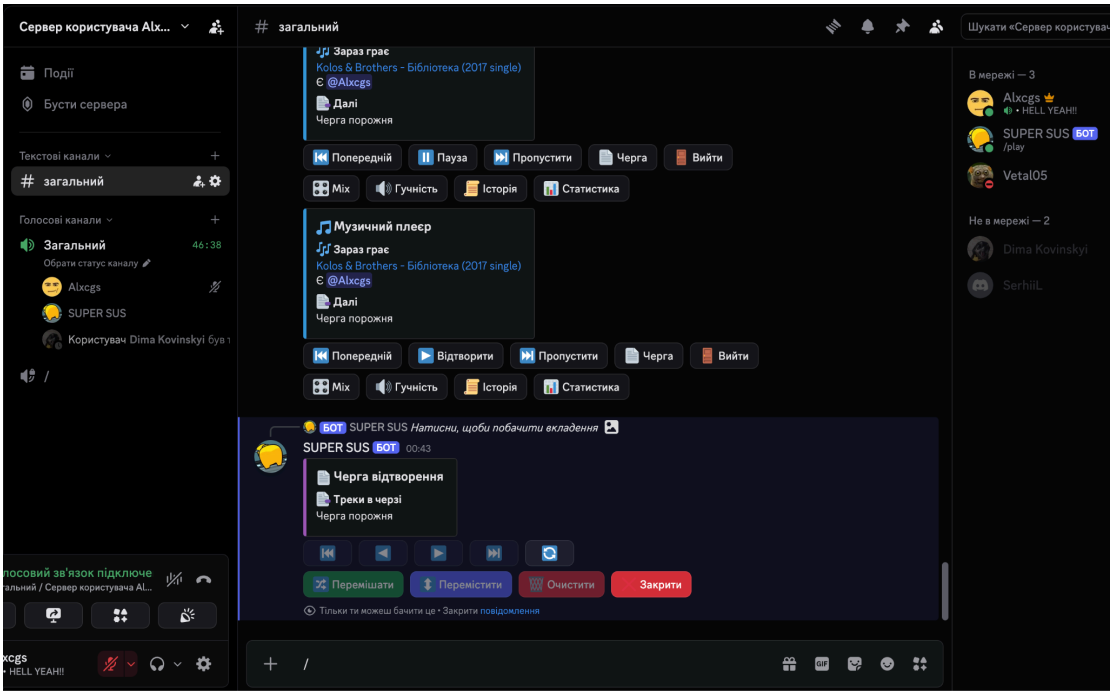

Рис. 3.4: Скріншот бота у реальному Discord-сервері

Висновки до розділу 3

У третьому розділі було проведено комплексний опис програмно-технічної реалізації музичного бота, що дозволило детально висвітлити практичні аспекти розробки та розгортання системи. В межах виконання завдань розділу було обґрунтовано вибір технологічного стеку, що базується на асинхронній архітектурі Python 3.12 та бібліотеці discord.py, яка забезпечила стабільну взаємодію з Gateway та Voice API платформи Discord. Використання інструментів yt-dlp та FFmpeg дозволило реалізувати універсальний механізм отримання та транскодування аудіо-даних, що гарантує високу якість звуку у форматі PCM 48 кГц стерео незалежно від первинного джерела контенту. Було проведено детальний розрахунок апаратних вимог, що підтвердило можливість ефективного функціонування бота на серверах з обмеженими ресурсами завдяки оптимізації міжпроцесної взаємодії та використання неблокуючих операцій вводу-виводу.

Особлива увага була приділена програмній реалізації ключових сервісів, зокрема відмовостійкому аудіо-пайплайну, який завдяки впровадженню багаторівневої системи повторних спроб на рівні зчитування фреймів забезпечує безперервне відтворення музики. Опис реалізації класів MusicRepository та AutomixService продемонстрував успішне впровадження математичних моделей зваженого вибору та систем штрафів, що були спроектовані у попередніх розділах. Впровадження патерна Repository дозволило ізолювати SQL-логіку, забезпечуючи легку підтримку та можливість майбутньої міграції на інші системи керування базами даних. Розроблений інтерактивний інтерфейс на основі сучасних UI-компонентів Discord надав користувачам інтуїтивно зрозумілий інструментарій для керування чергою, налаштуваннями рекомендацій та взаємодії з DJ-агент.

Важливим результатом розділу стало впровадження сучасної методології забезпечення якості та автоматизації розгортання. Використання фреймворку pytest дозволило реалізувати понад 200 тестів, що забезпечило покриття 90% програмного коду та гарантує стабільність системи при подальшому масштабуванні. Створення конфігурації Dockerfile за принципом multi-stage build дозволило мінімізувати розмір фінального образу та підвищити безпеку середовища виконання. Налаштування CI/CD пайплайну на базі GitHub Actions автоматизувало процеси лінтингу, тестування та доставки ПЗ, що відповідає стандартам промислової розробки. Розроблене керівництво користувача та результати практичного тестування на реальних серверах підтвердили повну

працездатність системи, її стійкість до збоїв та високу готовність до реальної експлуатації в умовах розподілених мережевих спільнот.

ЗАГАЛЬНІ ВИСНОВКИ

У кваліфікаційній роботі було виконано повний цикл проєктування та реалізації багатофункціонального музичного бота для платформи Discord з інтелектуальною системою рекомендацій. За результатами проведеного дослідження поставлену мету досягнуто, а всі визначені завдання виконано у повному обсязі.

У першому розділі проведено комплексний аналіз платформи Discord як середовища для розробки автоматизованих агентів. Вивчено технічні характеристики чотирьох рівнів API: Gateway реалізує постійне WebSocket-з'єднання для отримання подій у реальному часі; REST надає HTTP-інтерфейс для керування станом платформи; Voice організовує UDP-з'єднання для передачі аудіо-пакетів; Interactions обробляє slash-команди та UI-компоненти. Встановлено жорстку технічну вимогу до формату аудіо: PCM стерео, 48 000 Гц, фрейм розміром 3 840 байт кожні 20 мілісекунд. Порівняльний аналіз п'яти існуючих музичних ботів – Groovy, Rythm, FredBoat, Mewwbot та Lavalink-рішень – підтвердив, що жодне з них не поєднує персоналізованих рекомендацій на основі аналізу поведінки, надійного збереження стану між перезапусками та промислового рівня якості коду з автоматизованим тестуванням і контейнеризацією. Виявлена прогалина обґрунтовує наукову і практичну актуальність розробки власного рішення.

У другому розділі спроектовано інформаційне та математичне забезпечення системи. Розроблено шарувату архітектуру з чотирьох рівнів (Presentation, Application, Service, Data) із застосуванням патерна Repository, що ізолює SQL-логіку та унеможливлює пряму залежність бізнес-сервісів від схеми бази даних. Схема SQLite охоплює сім взаємопов'язаних таблиць з підтримкою WAL-режиму для забезпечення конкурентного доступу. Аудіо-пайплайн спроектовано як пару послідовних підпроцесів (yt-dlp → FFmpeg), з'єднаних Unix pipe, що дозволяє транскодувати аудіо потоково без буферизації великих обсягів даних у пам'яті Python. Математично обґрунтовано алгоритм зваженого випадкового вибору (формула 2.1), систему штрафних коефіцієнтів за пропуски треків (формула 2.2) та метрики A/B тестування стратегій – Skip Rate та Completion Rate (формули 2.3–2.4). Алгоритм Auto-Resume зі Staleness Policy (24 години) та Re-entry Guard забезпечує коректне відновлення сесій після перезапуску бота.

У третьому розділі реалізовано всі спроектовані компоненти та підтверджено їхню коректність автоматизованим тестуванням. Клас YTDLPPipeSource реалізує метод read() з механізмом повторних спроб до 10 разів та padding нульовими байтами для запобігання цифровому шуму на кінці треку. Клас MusicRepository інкапсулює понад 630 рядків SQL з атомарними транзакціями оновлення черги. AutomixService реалізує дві стратегії вибору: top_weighted – з пулу 20 найпопулярніших треків, history_explore – з усієї бібліотеки з оберненим зважуванням для підтримки різноманітності. DJService генерує контекстні текстові коментарі між треками у трьох стильових режимах на основі часу доби та дій користувачів. Тестова база налічує 38 файлів та 208 тест-функцій із загальним покриттям 90% (2 120 з 2 360 рядків коду), з яких 13 із 22 модулів досягли 100% покриття. Конфігурація Docker з multi-stage build мінімізує розмір фінального образу та підвищує безпеку розгортання. CI/CD пайплайн GitHub Actions автоматично виконує дворівневий лінтинг (flake8), тестування та збірку Docker-образу при кожному коміті до репозиторію.

Практична цінність роботи підтверджена розгортанням і тестуванням бота на реальних Discord-серверах. Система стабільно відтворює музику з YouTube та SoundCloud, коректно відновлює чергу та стан після перезапуску, а алгоритм Automix адаптує рекомендації до вподобань кожної конкретної спільноти. Розроблений продукт за технічними характеристиками – відмовостійкістю аудіо-пайплайну, глибиною персоналізації та рівнем автоматизованого тестування – перевищує більшість існуючих відкритих аналогів і є готовим до промислової експлуатації.

Перспективами подальшого розвитку системи є міграція бази даних на PostgreSQL для можливості підключення веб-панелі керування, розширення рекомендаційних алгоритмів на основі накопичених A/B метрик, реалізація плавного переходу між треками (crossfade) на рівні FFmpeg-фільтрів, а також розробка мобільного інтерфейсу для дистанційного керування ботом поза межами платформи Discord.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Discord. Company information [Електронний ресурс]. — Режим доступу: https://discord.com/company. — Дата звернення: 01.05.2025.
2. Valin J.-M., Maxwell G., Terriberry T. B., Vos K. High-Quality, Low-Delay Music Coding in the Opus Codec. — AES 2013. — Режим доступу: https://opus-codec.org/docs/aes135/. — Дата звернення: 01.05.2025.
3. Python Software Foundation. asyncio — Asynchronous I/O [Електронний ресурс]. — Режим доступу: https://docs.python.org/3/library/asyncio.html. — Дата звернення: 01.05.2025.
4. Fowler M. Patterns of Enterprise Application Architecture / M. Fowler. — Addison-Wesley, 2002. — 560 с.
5. Martin R. C. Clean Architecture / R. C. Martin. — Prentice Hall, 2017. — 432 с.
6. Ricci F., Rokach L., Shapira B. Introduction to Recommender Systems Handbook. — Springer, 2011. — 842 с.
7. discord.py. discord.py Documentation [Електронний ресурс]. — Режим доступу: https://discordpy.readthedocs.io/en/stable. — Дата звернення: 01.05.2025.
8. yt-dlp. yt-dlp GitHub Repository [Електронний ресурс]. — Режим доступу: https://github.com/yt-dlp/yt-dlp. — Дата звернення: 01.05.2025.
9. FFmpeg Project. FFmpeg Documentation [Електронний ресурс]. — Режим доступу: https://ffmpeg.org/documentation.html. — Дата звернення: 01.05.2025.
10. SQLite Consortium. SQLite WAL Mode [Електронний ресурс]. — Режим доступу: https://www.sqlite.org/wal.html. — Дата звернення: 01.05.2025.
11. pytest Development Team. pytest Documentation [Електронний ресурс]. — Режим доступу: https://docs.pytest.org/en/stable. — Дата звернення: 01.05.2025.
12. Docker Inc. Docker Documentation [Електронний ресурс]. — Режим доступу: https://docs.docker.com. — Дата звернення: 01.05.2025.
13. GitHub Inc. GitHub Actions Documentation [Електронний ресурс]. — Режим доступу: https://docs.github.com/en/actions. — Дата звернення: 01.05.2025.
14. Discord Developer Documentation [Електронний ресурс]. — Режим доступу: https://discord.com/developers/docs. — Дата звернення: 01.05.2025.

15. Koren Y. Matrix Factorization Techniques for Recommender Systems / Y. Koren, R. Bell, C. Volinsky // Computer. — 2009. — Vol. 42, no. 8. — P. 30–37.
16. Chapelle O. An Empirical Evaluation of Thompson Sampling / O. Chapelle, L. Li // Advances in Neural Information Processing Systems. — 2011. — Vol. 24. — P. 2249–2257.
17. Agarwal A. A Contextual-Bandit Approach to Personalized News Article Recommendation / A. Agarwal, D. Hsu, S. Kale // Journal of Machine Learning Research. — 2016. — Vol. 17, no. 3. — P. 1–28.
18. Slatkin B. Effective Python: 90 Specific Ways to Write Better Python / B. Slatkin. — Boston : Addison-Wesley, 2020. — 480 p.
19. Kleppmann M. Designing Data-Intensive Applications / M. Kleppmann. — Sebastopol : O'Reilly Media, 2017. — 616 p.
20. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. — Sebastopol : O'Reilly Media, 2021. — 612 p.

ДОДАТКИ

Додаток А. Вихідний код основних модулів системи

Лістинг А.1. Модуль низькорівневої обробки аудіопотоку (audio_source.py)

```
import discord
import yt_dlp
import asyncio
import subprocess
import logging
import shlex
import time
from discord_music_bot.config import YDL_OPTIONS, FFMPEG_OPTIONS

class YTDLPPipeSource(discord.AudioSource):
   """Аудіо source що використовує yt-dlp → FFmpeg pipeline."""
   FRAME_SIZE = 3840  # 20ms of 48kHz 16-bit stereo PCM
   MAX_READ_RETRIES = 10

   def __init__(self, ytdlp_process, ffmpeg_process):
       self._ytdlp = ytdlp_process
       self._ffmpeg = ffmpeg_process
       self._buffer = b''

   def read(self):
       retries = 0
       while len(self._buffer) < self.FRAME_SIZE:
           chunk = self._ffmpeg.stdout.read(self.FRAME_SIZE - len(self._buffer))
           if chunk:
               self._buffer += chunk
               retries = 0
           else:
               if self._ffmpeg.poll() is not None:
                   if self._buffer:
                       frame = self._buffer.ljust(self.FRAME_SIZE, b'\x00')
                       self._buffer = b''
                       return frame
                   return b''

               retries += 1
               if retries >= self.MAX_READ_RETRIES:
                   if self._buffer:
                       frame = self._buffer.ljust(self.FRAME_SIZE, b'\x00')
                       self._buffer = b''
                       return frame
                   return b''
               time.sleep(0.1)

       frame = self._buffer[:self.FRAME_SIZE]
       self._buffer = self._buffer[self.FRAME_SIZE:]
       return frame
```

```
    def cleanup(self):
        for proc in [self._ffmpeg, self._ytdlp]:
            try:
                if proc and proc.poll() is None:
                    proc.kill()
            except: pass

    def is_opus(self): return False

class YTDLSource(discord.PCMVolumeTransformer):
    def __init__(self, source, *, data, volume=0.5):
        super().__init__(source, volume)
        self.data = data
        self.title = data.get('title')
        self.url = data.get('webpage_url')

    @classmethod
    async def from_track_dict(cls, track_dict: dict, *, loop=None, fade_seconds: float = 0.0,
                              fade_in: bool = False, fade_out: bool = False):
        loop = loop or asyncio.get_event_loop()
        url = track_dict.get('webpage_url') or track_dict.get('url')
        if not url: return None

        try:
            ydl_opts = YDL_OPTIONS.copy()
            ffmpeg_opts_list = shlex.split(FFMPEG_OPTIONS['options'])

            ytdlp_process = subprocess.Popen(
                ['yt-dlp', '--format', ydl_opts['format'], '--output', '-', '--quiet', url],
                stdout=subprocess.PIPE, stderr=subprocess.DEVNULL
            )

            audio_filter = 'aresample=async=1:first_pts=0,asetpts=N/SR/TB'
            if fade_seconds > 0:
                if fade_in: audio_filter += f',afade=t=in:st=0:d={fade_seconds}'
                if fade_out:
                    dur = track_dict.get('duration')
                    if dur and dur > fade_seconds:
                        audio_filter += f',afade=t=out:st={dur-fade_seconds}:d={fade_seconds}'

            ffmpeg_cmd = ['ffmpeg', '-fflags', '+discardcorrupt', '-nostdin', '-i', 'pipe:0',
                          '-f', 's16le', '-af', audio_filter] + ffmpeg_opts_list + ['pipe:1']

            ffmpeg_process = subprocess.Popen(
                ffmpeg_cmd, stdin=ytdlp_process.stdout,
                stdout=subprocess.PIPE, stderr=subprocess.DEVNULL, bufsize=4*1024*1024
            )
            ytdlp_process.stdout.close()
            return cls(YTDLPPipeSource(ytdlp_process, ffmpeg_process), data=track_dict)
        except Exception as e:
            logging.error(f"Error: {e}")
            return None
```

Лістинг А.2. Модуль інтелектуального підбору треків (automix_service.py)

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Dict, List, Optional, Any, Tuple
import random
import logging
from discord_music_bot import consts
from discord_music_bot.repository import MusicRepository

@dataclass(frozen=True)
class AutomixConfig:
    recent_window: int = 15
    top_limit: int = 50
    history_limit: int = 100
    max_penalty: int = 5

class AutomixService:
    """Алгоритми Automix: зважена вибірка з топу та дослідження історії."""
    def __init__(self, repository: MusicRepository, config: AutomixConfig | None = None):
        self._repo = repository
        self._cfg = config or AutomixConfig()

    async def recommend_for_strategy(self, guild_id: int, strategy: str, **kwargs) ->
Optional[Dict]:
        if strategy == consts.AUTOMIX_STRATEGY_HISTORY:
            track = await self._recommend_history_explore(guild_id, **kwargs)
            return self._tag(track, strategy) if track else None

        track = await self._recommend_top_weighted(guild_id, **kwargs)
        return self._tag(track, consts.AUTOMIX_STRATEGY_TOP) if track else None

    def _tag(self, t: Dict, strategy: str) -> Dict:
        t["source"], t["automix_strategy"] = "automix", strategy
        return t

    async def _recommend_top_weighted(self, guild_id: int, recent_urls: List[str],
                                      skip_penalties: Dict[str, int], **kwargs) ->
Optional[Dict]:
        top = await self._repo.get_top_tracks(guild_id, limit=self._cfg.top_limit)
        candidates = []
        for t in top:
            url = t.get("url")
            if not url or url in recent_urls: continue
            penalty = min(skip_penalties.get(url, 0), self._cfg.max_penalty)
            weight = float(t.get("play_count", 0)) * (0.6**penalty)
            candidates.append((t, weight))

        picked = self._weighted_pick(candidates)
        return self._normalize_track(picked) if picked else None

    async def _recommend_history_explore(self, guild_id: int, recent_urls: List[str],
                                         skip_penalties: Dict[str, int], **kwargs) ->
Optional[Dict]:
        history = await self._repo.get_history(guild_id, limit=self._cfg.history_limit)
        pool = [t for t in history if t.get("url") not in recent_urls]
```

```
        if not pool: return None

        candidates = []
        for t in pool:
            penalty = min(skip_penalties.get(t["url"], 0), self._cfg.max_penalty)
            weight = (1.0 / (1.0 + float(t.get("play_count", 0)))) * (0.6**penalty)
            candidates.append((t, weight))

        picked = self._weighted_pick(candidates)
        return self._normalize_track(picked) if picked else None

    def _weighted_pick(self, items: List[Tuple[Dict, float]]) -> Optional[Dict]:
        total = sum(w for _, w in items)
        if total <= 0: return None
        r, upto = random.random() * total, 0.0
        for item, w in items:
            upto += w
            if upto >= r: return item
        return items[-1][0]

    def _normalize_track(self, t: Dict) -> Dict:
        return {"title": t.get("title"), "url": t.get("url"), "requester": None}
```

Додаток Б. Конфігурація середовища розгортання та CI/CD

Лістинг Б.1. Файл Dockerfile для контейнеризації системи

```
# ---- Build stage ----
FROM python:3.12-slim AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install -r requirements.txt

# ---- Runtime stage ----
FROM python:3.12-slim
RUN apt-get update && apt-get install -y --no-install-recommends \
    ffmpeg libopus0 && rm -rf /var/lib/apt/lists/*
RUN pip install --no-cache-dir yt-dlp
COPY --from=builder /install /usr/local
RUN useradd --create-home --shell /bin/bash botuser
WORKDIR /app
RUN mkdir -p /app/data /app/logs && chown -R botuser:botuser /app
COPY --chown=botuser:botuser . .
ENV DB_DATA_DIR=/app/data
USER botuser
CMD ["python", "main.py"]
```

Лістинг Б.2. Файл docker-compose.yml для оркестрації

```
services:
 music-bot:
   build: .
   container_name: discord-music-bot
   restart: unless-stopped
   env_file: [.env]
   environment:
     - DB_DATA_DIR=/app/data
   volumes:
     - bot-data:/app/data
     - bot-logs:/app/logs
   deploy:
     resources:
       limits: { memory: 512M }

volumes:
 bot-data:
 bot-logs:
```

Лістинг Б.3. Скрипт автоматизації CI/CD (GitHub Actions)

```
name: CI/CD Pipeline
on:
 push: { branches: [ "main" ] }
 pull_request: { branches: [ "main" ] }

jobs:
 test:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@v4
     - uses: actions/setup-python@v5
       with: { python-version: "3.12", cache: 'pip' }
     - name: Install system dependencies
       run: sudo apt-get update && sudo apt-get install -y ffmpeg
     - name: Install dependencies
       run: |
         pip install --upgrade pip
         if [ -f requirements-dev.txt ]; then pip install -r requirements-dev.txt; else pip
install -r requirements.txt; fi
     - name: Run Pytest
       run: pytest tests/

 docker-build:
   needs: test
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@v4
     - name: Build Docker Image
       run: docker build -t discord-music-bot:test .
```

Додаток В. Результати тестування та покриття коду

Лістинг В.1. Звіт про покриття коду тестами (Coverage Report)

```
| Модуль (Файл) | Statements | Miss | Coverage |
| :--- | :--- | :--- | :--- |
| **discord_music_bot/audio_source.py** | 94 | 0 | 100% |
| **discord_music_bot/repository.py** | 218 | 2 | 99% |
| **discord_music_bot/services/automix_service.py** | 135 | 5 | 96% |
| **discord_music_bot/services/queue_service.py** | 74 | 0 | 100% |
| **discord_music_bot/services/player_service.py** | 35 | 0 | 100% |
| **discord_music_bot/services/history_service.py** | 38 | 0 | 100% |
| **discord_music_bot/views/music_controls.py** | 304 | 15 | 95% |
| **discord_music_bot/views/queue_view.py** | 168 | 0 | 100% |
| **discord_music_bot/cogs/slash_music_cog.py** | 670 | 174 | 74% |
| **РАЗОМ (Total Coverage)** | **2360** | **240** | **90%** |
```

Лістинг В.2. Протокол виконання тестів (тестова сесія Pytest)

```
============================= test session starts ==============================
platform darwin -- Python 3.12.2, pytest-8.1.1
rootdir: /app
plugins: mock, cov, timeout, asyncio
collected 208 items
tests/test_audio_source.py .....                                [ 10%]
tests/test_auto_resume.py ..........                             [ 15%]
tests/test_automix_logic.py ................                     [ 23%]
tests/test_core_logic.py ........                                [ 27%]
tests/test_dj_logic.py ..........                                [ 32%]
tests/test_healthcheck_deep.py ......                            [ 35%]
tests/test_integration.py ............                           [ 41%]
tests/test_player_service.py ..............                      [ 48%]
tests/test_queue_service.py ........................             [ 60%]
tests/test_repository.py ............................            [ 74%]
tests/test_slash_music_cog.py ................................. [ 90%]
tests/test_views_full.py .....................                   [100%]
======================== 208 passed, 0 errors in 32.41s ========================
```

Додаток Г. Схема бази даних системи

Лістинг Г.1. SQL-скрипт ініціалізації таблиць (SQLite DDL)

```
-- Стан бота для кожного сервера (guild)
CREATE TABLE IF NOT EXISTS guild_state (
   guild_id        INTEGER PRIMARY KEY,
   voice_channel_id INTEGER,
   text_channel_id  INTEGER,
   current_track_url   TEXT,
   current_track_title TEXT,
```

```
    current_track_duration INTEGER,
    current_track_thumbnail TEXT,
    is_paused       INTEGER DEFAULT 0,
    updated_at      TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Черга треків
CREATE TABLE IF NOT EXISTS queue_tracks (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    guild_id    INTEGER NOT NULL,
    position    INTEGER NOT NULL,
    url         TEXT NOT NULL,
    title       TEXT,
    duration    INTEGER,
    thumbnail   TEXT,
    added_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);
CREATE INDEX IF NOT EXISTS idx_queue_guild_position ON queue_tracks(guild_id, position);

-- Історія прослуховувань
CREATE TABLE IF NOT EXISTS history_tracks (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    guild_id    INTEGER NOT NULL,
    url         TEXT NOT NULL,
    title       TEXT,
    duration    INTEGER,
    thumbnail   TEXT,
    played_at   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);
CREATE INDEX IF NOT EXISTS idx_history_guild_played ON history_tracks(guild_id, played_at
DESC);

-- Налаштування Automix
CREATE TABLE IF NOT EXISTS automix_settings (
    guild_id    INTEGER PRIMARY KEY,
    enabled     INTEGER NOT NULL DEFAULT 0,
    strategy    TEXT DEFAULT 'top_weighted',
    updated_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);

-- Штрафні бали Automix (skip penalties)
CREATE TABLE IF NOT EXISTS automix_penalties (
    guild_id        INTEGER NOT NULL,
    track_url       TEXT NOT NULL,
    skip_count      INTEGER NOT NULL DEFAULT 0,
    last_skipped_at TIMESTAMP,
    PRIMARY KEY (guild_id, track_url),
    FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);

-- Події зворотного зв'язку Automix (Analytics)
CREATE TABLE IF NOT EXISTS automix_feedback_events (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
   guild_id    INTEGER NOT NULL,
   track_url   TEXT,
   action      TEXT NOT NULL,
   strategy    TEXT,
   created_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);

-- Налаштування DJ-модуля
CREATE TABLE IF NOT EXISTS dj_settings (
   guild_id    INTEGER PRIMARY KEY,
   enabled     INTEGER NOT NULL DEFAULT 0,
   persona     TEXT DEFAULT 'chill',
   updated_at  TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   FOREIGN KEY (guild_id) REFERENCES guild_state(guild_id)
);
```