

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ ПРОГРАМНОЇ СИСТЕМИ АВТОМАТИЗОВАНОГО
КЕРУВАННЯ TELEGRAM-КОНТЕНТОМ
DEVELOPMENT OF A SOFTWARE SYSTEM FOR AUTOMATED
TELEGRAM CONTENT MANAGEMENT

Виконав: студент 4 курсу, групи КНТ-113сп

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

БЕДСЬКИХ А.В.

(ПРИЗВИЩЕ та ініціали)

Керівник ГЛАДКОВА О.М.

(ПРИЗВИЩЕ та ініціали)

Рецензент ГОЛУБ Т.В.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

БЕДСЬКИХ Артема Віталійовича
(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення програмної системи автоматизованого керування Telegram-контентом. Development of a Software System for Automated Telegram Content Management

керівник проєкту (роботи) к.т.н., доцент, ГЛАДКОВА Ольга Миколаївна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 07 ” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року
 3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГЛАДКОВА О.М., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

_____ Артем БЕДСЬКИХ
(підпис) (Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Ольга ГЛАДКОВА
(підпис) (Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 85 с., 4 табл., 23 рис., 2 дод., 15 джерел.

TELEGRAM-БОТ, PYTHON, SQLITE, АВТОМАТИЗАЦІЯ, API, ПАРСИНГ, АЛГОРИТМИ РАНЖИРУВАННЯ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ.

Об'єкт дослідження – процес інженерії програмного забезпечення публікації контенту в месенджерах.

Предмет дослідження – програмні засоби розробки Telegram-ботів та агрегації даних.

Мета роботи – підвищення рівня автоматизації та швидкості публікації контенту в Telegram-каналі.

Матеріали, методи та технічні засоби: мова програмування Python, бібліотеки асинхронного програмування (asyncio), фреймворк SQLAlchemy, СУБД SQLite, Telegram Bot API, Shikimori API.

Результати. Розроблено програмне забезпечення Telegram-бота з використанням мови Python. Реалізовано алгоритм системи оцінювання для вибору контенту, систему кешування медіафайлів та інтеграцію із зовнішнім API Shikimori.

Висновки. Виконано проєктування архітектури Telegram-бота. Розроблено ПЗ для автопостингу з урахуванням оптимізації мережевих запитів. Здійснено тестування модулів вибірки, бази даних та системи взаємодії з користувачами.

Галузь використання – розважальні спільноти, адміністратори Telegram-каналів, системи автоматизації контент-менеджменту.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 85 pages, 4 tables, 23 figures, 2 appendixes, 15 sources.

TELEGRAM BOT, PYTHON, SQLITE, AUTOMATION, API, PARSING, RANKING ALGORITHMS, SOFTWARE.

Object of study is the process of automating content publication in messengers.

Subject of study is software tools for developing Telegram bots and data aggregation.

The purpose of the work is to increase the level of automation and the speed of content publication in a Telegram channel.

Materials, methods, and tools: Python programming language, asynchronous programming libraries (asyncio), SQLAlchemy framework, SQLite DBMS, Telegram Bot API, Shikimori API.

Results. Telegram bot software was developed using the Python language. A scoring system algorithm for content selection, a media file caching system, and integration with the external Shikimori API were implemented.

Conclusions. The architecture design of the Telegram bot was completed. Auto-posting software was developed taking into account the optimization of network requests. Testing of the selection modules, database, and user interaction system was carried out.

The field of use is a entertainment communities, Telegram channel administrators, content management automation systems.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	7
Вступ.....	8
1 Аналіз предметної області.....	10
1.1 Аналіз управління контентом та агрегації даних у месенджерах.....	10
1.2 Аналіз програмного забезпечення для автоматизації Telegram-каналів.....	13
1.3 Висновки за розділом 1	20
2 Матеріали і методи.....	22
2.1 Вибір та обґрунтування мови програмування	22
2.2 Вибір системи управління базами даних та засобів взаємодії	24
2.3 Обґрунтування вибору технологій інтеграції (API)	28
2.4 Висновки за розділом 2	30
3 Опис програми.....	32
3.1 Архітектура та структура програмного забезпечення	32
3.2 Структура бази даних та моделі даних	35
3.3 Розробка алгоритму агрегації та скорингу контенту	37
3.4 Проєктування інтерфейсу взаємодії Telegram-бота	43
3.5 Висновки за розділом 3	45
4 Експлуатація, тестування та експериментальне дослідження програми	47
4.1 Призначення й умови застосування програмного забезпечення	47
4.2 Інструкція по експлуатації програмного забезпечення.....	48
4.3 Виконання програмного забезпечення	50
4.4 Тестування програмного забезпечення.....	53
4.5 Висновки за розділом 4	57
Висновки	59
Перелік джерел посилання	61
Додаток А Текст програми.....	63
Додаток Б Слайди презентації	79

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

БД – База даних;

ПЗ – Програмне забезпечення;

СУБД – Система управління базами даних;

ORM – Object-Relational Mapping;

ШІ – Штучний інтелект.

HTTP – HyperText Transfer Protocol

ВСТУП

У сучасному інформаційному просторі месенджери, зокрема Telegram, стали одними з провідних платформ для розповсюдження контенту. Проте зі зростанням обсягів інформації адміністратори тематичних каналів стикаються з проблемою ефективного управління ресурсами. Ручний пошук, перевірка релевантності та підготовка публікацій, особливо у таких насичених нішах як аніме-індустрія (де бази даних налічують десятки тисяч записів), стає критично повільним і суб'єктивним процесом.

Традиційні підходи до адміністрування не дозволяють оперативно обробляти великі масиви даних, що підкреслює необхідність впровадження інтелектуальних систем автоматизації. Розробка програмного забезпечення, здатного самостійно оцінювати контент, уникати дублювання та оптимізувати доставку медіафайлів через механізми кешування, є необхідним кроком для підвищення якості роботи медіаресурсів. Це дозволяє не лише зняти рутинне навантаження з модераторів, а й значно пришвидшити доставку актуальної інформації до кінцевого споживача, що є ключовим фактором конкурентоспроможності в цифровій екосистемі.

У дипломній кваліфікаційній роботі бакалавра розв'язується актуальне завдання розробки програмного забезпечення для Telegram-бота автопостингу, спрямоване на оптимізацію процесів контент-менеджменту.

Мета роботи – підвищення рівня автоматизації та швидкості публікації контенту в Telegram-каналі.

Для досягнення поставленої мети у кваліфікаційній роботі бакалавра необхідно розв'язати такі задачі:

- виконати аналіз предметної області та існуючих програмних засобів для автоматизації роботи в середовищі Telegram;
- здійснити проектування архітектури Telegram-бота та оптимальної структури реляційної бази даних;

- розробити та впровадити алгоритм системи оцінювання (скорингу) для інтелектуального вибору найбільш релевантного контенту;
- реалізувати механізми кешування зображень та інтеграцію із зовнішніми джерелами даних (Shikimori API) для пришвидшення генерації постів;
- виконати комплексне тестування розробленого програмного забезпечення на предмет стабільності, швидкості публікації та коректності роботи системи блокування дублікатів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз управління контентом та агрегації даних у месенджерах

Месенджери поступово вийшли за межі звичайних інструментів приватного спілкування і перетворилися на повноцінні платформи для поширення новин, розважального та тематичного контенту. Особливо помітно це в Telegram, який, за даними Statista, у січні 2025 року охоплював понад 11% світового населення, а середньомісячна аудиторія становила близько 950 млн користувачів [1]. Сам Telegram ще раніше наголошував на масштабі своєї аудиторії, повідомляючи про 200 млн активних користувачів у 2018 році, що добре показує динаміку зростання платформи [2]. Для тематичних спільнот це означає не просто наявність великої кількості користувачів, а й наявність середовища, де контент споживається швидко, регулярно і без складного доступу до додаткових сервісів. Саме тому Telegram сьогодні часто використовують не лише для листування, а й для каналів з аніме, кіно, іграми, освітніми матеріалами та іншими нішевими тематиками.

Для тематичного каналу важливими є не тільки самі публікації, а й їхня регулярність, структурованість і актуальність. У каналах з аніме-контентом користувачі очікують швидкого доступу до описів тайтлів, постерів, жанрів, рейтингів, дати виходу, кількості епізодів та інших атрибутів. Якщо канал ведеться вручну, адміністратор змушений щоразу шукати інформацію в різних джерелах, перевіряти її актуальність, форматовувати текст поста, підбирати зображення та стежити за однаковим стилем публікацій. У невеликому каналі це ще може бути прийнятним, однак із ростом обсягу контенту процес стає помітно важчим і менш ефективним.

Проблема ручного управління контентом особливо загострюється тоді, коли канал працює з великим каталогом записів. Якщо йдеться про десятки тисяч елементів – наприклад, про базу аніме або фільмів – ручний пошук і копіювання даних стають непридатними для регулярної роботи. Адміністратор витрачає час не на створення корисного контенту, а на технічні

операції: пошук назви, звірку опису, перевірку жанрів, оновлення рейтингу, виправлення помилок форматування. Зі збільшенням кількості записів зростає і ризик дублювання, пропуску важливих атрибутів або використання застарілих даних. У результаті канал може втрачати якість, а публікації – однорідність і довіру з боку аудиторії.

Окремою проблемою є форматування. Telegram як платформа має зручний інтерфейс для читання, але одночасно накладає вимоги на структуру подачі інформації. Щоб пост був візуально привабливим і легко сприймався, у ньому потрібно узгодити текст, емодзі, списки, посилання, а іноді й мультимедійні елементи. При ручному створенні кожної публікації це виконується через повторювану рутинну роботу. Якщо ж канал має десятки або сотні постів на місяць, така схема швидко стає нераціональною. Саме тому для подібних задач дедалі частіше застосовують автоматизацію, яка дозволяє відокремити підготовку контенту від його публікації та мінімізувати людський фактор.

Перехід до автоматизованих підходів у керуванні контентом базується насамперед на використанні баз даних. База даних дає змогу централізовано зберігати відомості про об'єкти предметної області – у даному випадку про аніме: назви, описи, жанри, статус, рік випуску, рейтинги, зображення, посилання на першоджерело тощо. На відміну від текстових файлів або ручного внесення в повідомлення, база даних забезпечує впорядковане зберігання, швидкий пошук і можливість оновлення без втрати цілісності даних. Для невеликих і середніх проєктів доцільним рішенням є SQLite, оскільки ця СУБД не потребує окремого сервера, легко інтегрується в застосунок і підходить для локального зберігання структурованої інформації.

Однак сама база даних не вирішує проблему актуальності відомостей. Якщо дані про аніме потрібно підтримувати в актуальному стані, їх слід брати з першоджерел або надійних зовнішніх сервісів. Тут ключову роль відіграють API. Веб-API дозволяє програмно отримувати інформацію з зовнішнього ресурсу у стандартизованому форматі без необхідності ручного копіювання.

Для тематичних ботів це особливо важливо, оскільки інформація може оновлюватися: з'являються нові серії, змінюються рейтинги, додаються описи або постери. Використання API зменшує кількість помилок, пришвидшує наповнення бази та спрощує масштабування системи.

У контексті аніме-тематики доцільним джерелом є Shikimori API, який надає доступ до даних про аніме, мангу, персонажів, рейтинги та пов'язані елементи [3]. Наявність такого API дає змогу будувати автоматизовану схему роботи: бот отримує запит, звертається до зовнішнього сервісу, витягує потрібні відомості, зберігає їх у SQLite та формує готову публікацію. Такий підхід забезпечує не лише швидкість, а й системність. Замість разового ручного внесення інформації створюється механізм, який може повторно використовуватися для багатьох записів і різних типів контенту.

Варто звернути увагу і на специфіку месенджерів як середовища для такої автоматизації. Telegram має відкритий Bot API, що робить його зручним для розробки ботів і інтеграції з зовнішніми сервісами [4]. Крім того, платформа підтримує канали як формат одностороннього поширення інформації, що особливо підходить для тематичних сторінок. Саме в каналах автоматизація дає найбільший ефект: адміністратор може налаштувати генерацію публікацій за розкладом, фільтрувати записи за жанрами чи статусом, а також формувати добірки без ручної підготовки кожного поста. Це знижує навантаження на автора каналу та підвищує стабільність оновлень.

У наукових і прикладних роботах із аналізу контенту в месенджерах часто підкреслюється, що автоматизація потрібна не лише для економії часу, а й для кращої організації інформації. Хоча у випадку аніме-бота основна задача полягає не в аналізі емоційного забарвлення, а в управлінні каталогом контенту, загальна логіка є спільною: інформація повинна зберігатися у структурованому вигляді, швидко оновлюватися та подаватися користувачу в зручному форматі. Саме база даних і API дають технічну основу для такого підходу.

Ще одна практична перевага автоматизації полягає в масштабованості. Якщо тематичний канал зростає, збільшується не лише кількість читачів, а й обсяг даних, які потрібно обробляти. Система, побудована на ручному управлінні, погано адаптується до такого зростання: що більше контенту, то більше часу потрібно на його підготовку. Натомість бот з базою даних може працювати з великими наборами записів майже однаково ефективно незалежно від їх кількості. У такій архітектурі нова публікація – це не окремий ручний процес, а результат запиту до даних і автоматичного формування шаблону поста.

Отже, аналіз підходів до управління контентом у месенджерах показує, що ручне ведення тематичних каналів швидко вичерпує свої можливості, особливо коли йдеться про великі каталоги та регулярне оновлення матеріалів. Зростання ролі Telegram як платформи для споживання контенту створює попит на системи, які здатні автоматизувати пошук, зберігання, форматування та публікацію даних. Використання SQLite як локальної бази даних і зовнішніх API, зокрема Shikimori API, дає змогу побудувати рішення, яке одночасно є простим у впровадженні, стійким до навантаження та зручним для подальшого розвитку. Саме ці підходи формують основу для створення Telegram-бота, орієнтованого на автоматизовану публікацію аніме-контенту.

1.2 Аналіз програмного забезпечення для автоматизації Telegram-каналів

Перейдемо до аналізу інструментів, які використовуються для керування Telegram-каналами. Найчастіше такі рішення орієнтовані на спрощення щоденної роботи адміністратора: підготовку публікацій, їхнє планування, базове форматування та відстеження результатів розміщення. Серед поширених сервісів можна виділити Controller Bot, FleepBot, TGADM.IN та інші подібні рішення, що працюють як допоміжні інструменти для відкладеного постингу та адміністрування каналів.

Controller Bot належить до класу сервісів, які дозволяють створювати та публікувати повідомлення у відкладеному режимі, а також зручно оформлювати текст перед відправленням [5].

У практичному використанні це означає, що адміністратор може підготувати пост заздалегідь, додати форматування, а потім призначити точний час публікації. Подібний функціонал особливо корисний для каналів, де важлива регулярність виходу контенту, але він не виходить за межі механізму доставки вже підготовленого матеріалу (рисунок 1.1).

Більшості користувачів для невеликих каналів з новинним контентом чи блогів такого функціоналу буде достатньо.

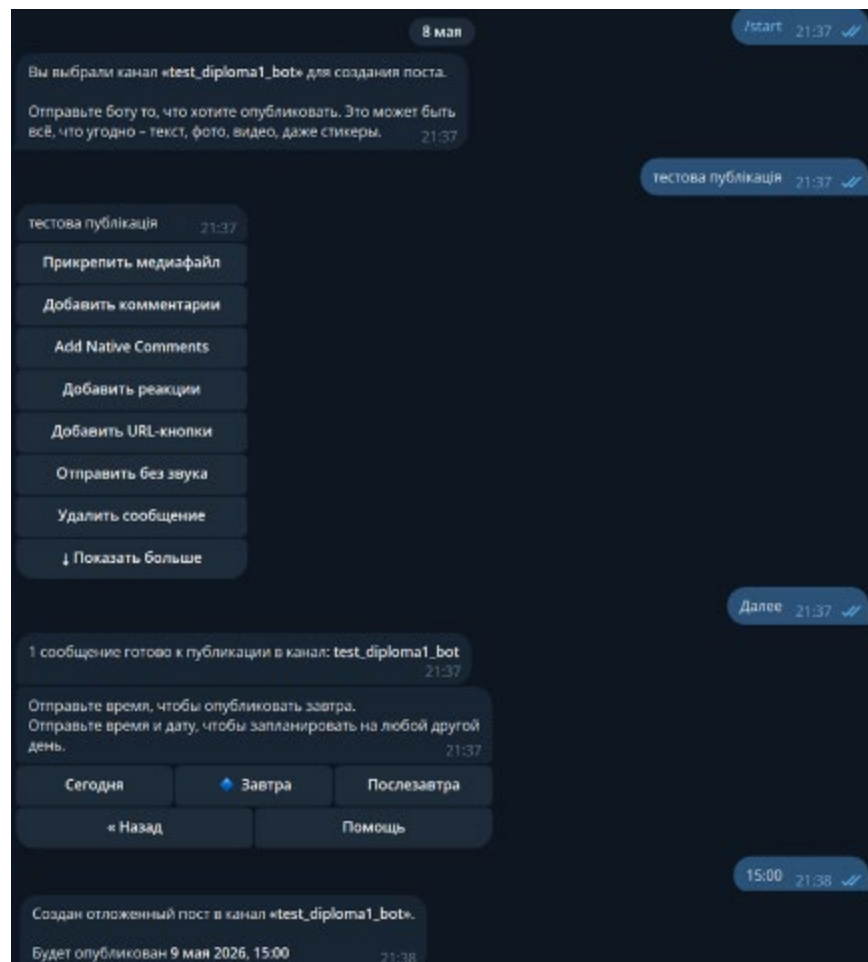


Рисунок 1.1 – Керування автопублікацією за допомогою Controller Bot

FleerBot виконує великий набір допоміжних функцій: підтримує відкладений постинг, статистику, роботу з кнопками, реакціями,

налаштування параметрів публікацій і частково інструменти для взаємодії з аудиторією.

Також у застосунок інтегровані функції ШІ, за допомогою яких є можливість генерувати текст та картинки майбутніх публікацій, за додаткову плату. Налаштувавши розписання, надсилати інформацію в канал стає ще легше, але публікуватися буде одне й те саме повідомлення (рисунк 1.2) [6].

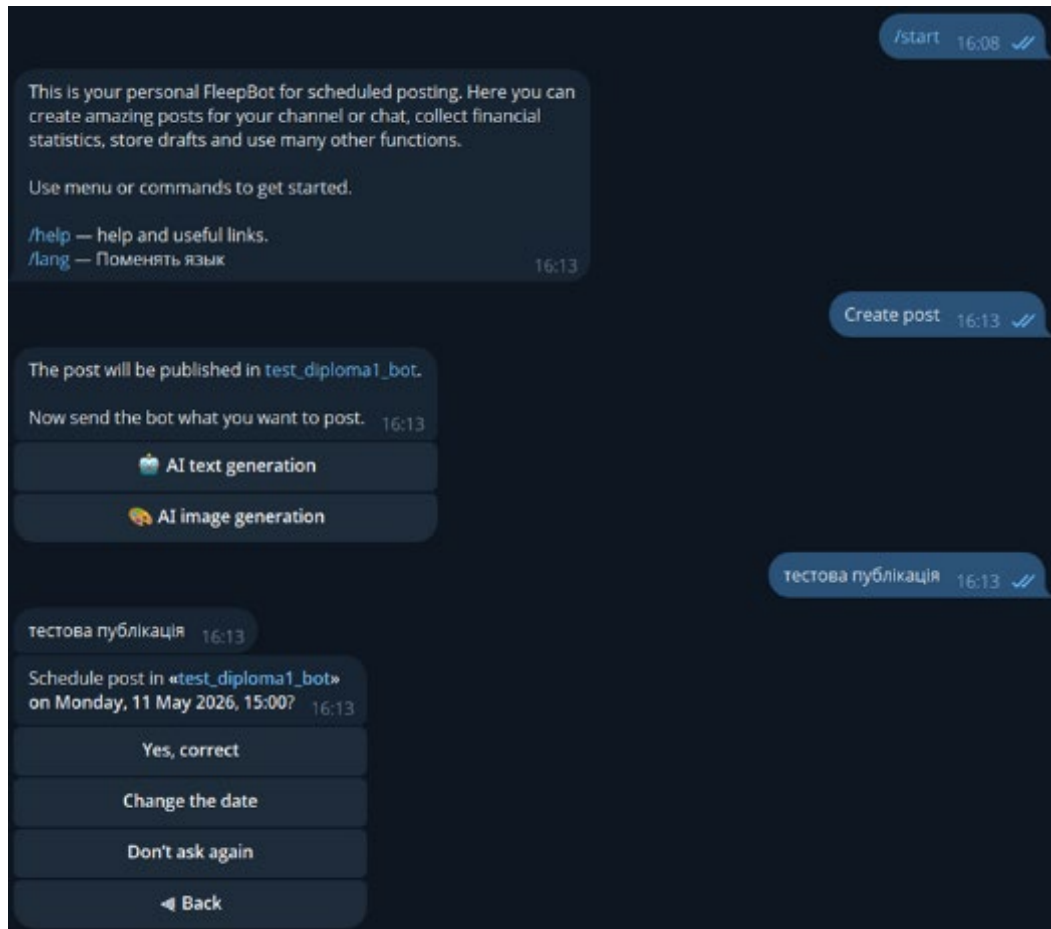


Рисунок 1.2 – Керування автопублікацією за допомогою FleepBot

Окремої уваги заслуговує сервіс TGADM.IN. Він має всі функції конкурентів, продуманий інтерфейс і пропонує генерацію постів за допомогою алгоритмів ШІ. Проте його архітектурний підхід відрізняється від потреб каталогізованих баз даних. ШІ ідеально підходить для створення креативних або новинних текстів, але має критичний недолік при роботі з точною інформацією (наприклад, аніме-тайтлами): схильність до помилок (галюцинацій) та неможливість гарантувати строгу стандартизацію даних

(рейтинг, кількість епізодів, точний набір жанрів). Крім того, подібні сервіси не формують локальної реляційної бази даних для ведення складних зв'язків та системи лімітів публікацій. Тобто TGADM.IN виступає як "розумний редактор", але не як автономний агрегатор точних даних (рисунок 1.3).

Доступ до нього можливо отримати безкоштовно на термін 7 днів [7].

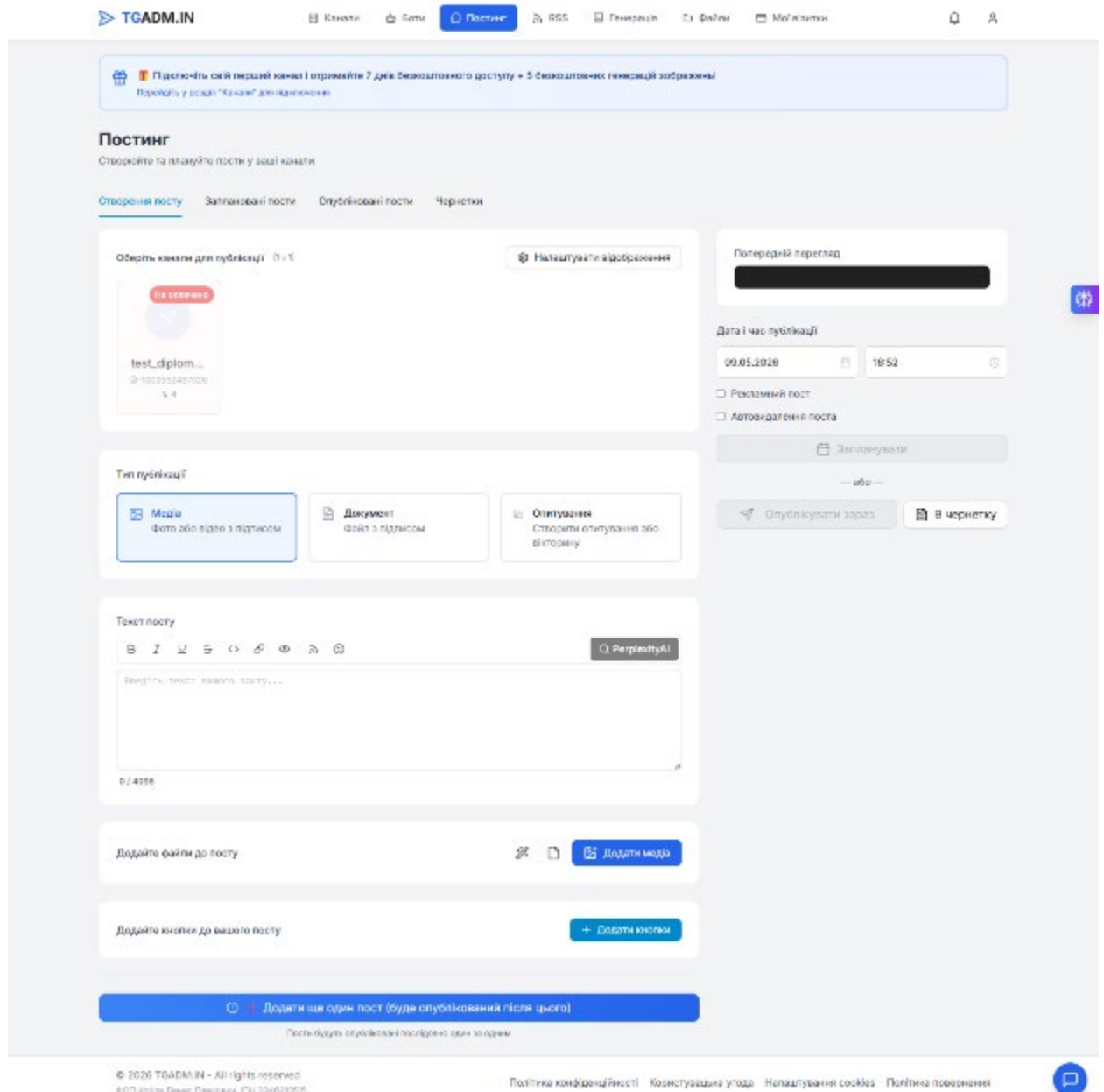


Рисунок 1.3 – Керування автопублікацією за допомогою TGADM.IN

У цілому всі ці інструменти значно полегшують рутинні дії адміністратора, однак їхня архітектурна роль залишається подібною: вони

організують розміщення матеріалів, але не формують повний цикл роботи з контентом.

Саме тут проявляється ключова межа універсальних рішень. Вони є насамперед транспортом між адміністратором і Telegram-каналом: користувач самотійно шукає інформацію, сам пише текст, сам підбирає зображення й сам передає все це до бота для подальшої публікації. Іншими словами, такі сервіси не беруть на себе агрегацію даних із першоджерел, не виконують парсинг і не мають вбудованого механізму наповнення тематичної бази. Для невеликого каналу це може бути прийнятно, але для великого тематичного проєкту така модель виявляється недостатньо ефективною.

Особливо відчутні недоліки універсальних ботів при роботі з великими каталогами, наприклад з базами аніме, що містять десятки тисяч записів. У такій предметній області контент має не лише текстову частину, а й набір структурованих параметрів: назву, опис, жанри, статус, рік випуску, кількість епізодів, постер, рейтинг, посилання на джерело тощо. Якщо ці дані збираються вручну, адміністратор щоразу витрачає час на пошук, перевірку та форматування інформації. Зі збільшенням обсягу записів рутинна робота починає переважати над власне веденням каналу, а ризик помилок, дублювання або використання застарілих відомостей зростає.

Окремо слід виділити відсутність у таких системах вбудованої бази даних предметної області. Бот може зберігати окремі параметри публікації або вести просту статистику, але він не забезпечує повноцінного структурованого каталогу, який можна фільтрувати, оновлювати та перевикористовувати в різних сценаріях. Для тематичного каналу це принципово важливо, оскільки контент не є випадковим набором повідомлень – це впорядкована сукупність записів, які мають повторно використовуватися, оновлюватися і подаватися в єдиному стилі. Без бази даних адміністратор змушений щоразу відтворювати один і той самий процес підготовки публікації практично з нуля.

Ще одна слабка сторона універсальних рішень полягає у відсутності алгоритмів інтелектуального оцінювання контенту. Для тематичних баз,

зокрема аніме-каталогів, корисним є механізм, який дозволяє визначати пріоритетність матеріалів: що публікувати раніше, які записи є актуальними, який контент краще відповідає поточній активності каналу. У наявних ботах подібні завдання не автоматизовані. Вони не виконують скоринг та не аналізують якість або релевантність записів. У результаті всі ці функції повністю покладаються на адміністратора.

Не менш важливою є проблема оптимізації трафіку та повторного використання медіафайлів. У великому каналі основне навантаження створюють не лише тексти, а й зображення: постери, прев'ю, банери, скріншоти. Якщо сервіс не має механізму глибокого кешування або локального повторного використання медіа, повторна публікація матеріалу призводить до додаткових операцій із файлами, що збільшує час обробки та навантаження на систему. Для невеликого каналу це може бути непомітно, але в умовах тисяч зображень і регулярного автопостингу така неефективність стає суттєвою. Отже, при роботі з великим архівом контенту недостатньо лише опублікувати пост – потрібно ще й раціонально організувати зберігання та повторне використання медіаданих.

З технічної точки зору також важливо, що Telegram як платформа підтримує механізми запланованих повідомлень, а Bot API забезпечує базове форматування тексту. Проте ці можливості самі по собі не вирішують задачу автоматизації контент-менеджменту. Вони лише створюють основу для реалізації більш складної системи, у якій бот не просто відправляє повідомлення, а взаємодіє з базою даних, отримує дані із зовнішнього API, відбирає записи за певними критеріями та формує готовий пост у потрібному форматі. Саме така схема є значно ближчою до реальних потреб тематичного каналу, ніж використання окремого універсального сервісу.

Порівняємо функціональні можливості існуючих універсальних рішень та запропонованого спеціалізованого програмного забезпечення, дивитись табл. 1.1.

Таблиця 1.1 – Порівняння ПЗ для автоматизації Telegram-каналів

Функціональна можливість	Controller Bot	FleepBot	TGADM.IN	Розроблюване ПЗ
Відкладений постинг та базове форматування	+	+	+	+
Генерація текстів за допомогою ШІ	–	+	+	–
Автоматична агрегація точних даних (Парсинг API)	–	–	–	+
Наявність вбудованої реляційної БД об'єктів	–	–	–	+
Алгоритм багатокритеріального оцінювання	–	–	–	+
Глибоке кешування медіафайлів	–	–	–	+

Отже, аналіз існуючого програмного забезпечення показав, що універсальні інструменти не здатні повною мірою задовольнити потреби адміністрування каналів з великими базами даних. Виходячи з цього, при розробці спеціалізованого програмного забезпечення Telegram-бота для автопостингу необхідно забезпечити такі функціональні вимоги:

- збір та первинна обробка даних із зовнішніх джерел;
- збереження та структурування контенту в локальній реляційній базі даних;
- реалізація алгоритму інтелектуального вибору контенту на основі багатокритеріального скорингу;
- впровадження механізму блокування дублювання контенту;
- оптимізація доставки медіафайлів за допомогою механізму кешування ідентифікаторів серверів Telegram;
- забезпечення можливості гнучкого налаштування інтервалів автоматичної публікації.

1.3 Висновки за розділом 1

Аналіз управління контентом та агрегації даних у месенджерах показав, що існують різні методи організації публікацій у Telegram. Методи можуть варіюватися від повністю ручного керування до автоматизованих систем із використанням баз даних та зовнішніх API. Оцінка ефективності методів змінюється в залежності від обсягів інформації, регулярності оновлень та вимог до якості контенту. Завдяки ретельному аналізу різних методів управління контентом, стало зрозумілим, що кожен підхід має свої переваги та обмеження. Наприклад, ручні методи адміністрування можуть бути ефективними для невеликих авторських блогів, тоді як автоматизовані методи (агрегація через API) є критично необхідними та більш підходящими для каналів із великими тематичними каталогами (наприклад, базами аніме).

Аналіз програмного забезпечення для автоматизації Telegram-каналів виявив, що існує різноманітність програм та сервісів, що пропонують різні функції для оптимізації роботи адміністраторів. Вони включають відкладений постинг, підтримку базового форматування, зручний інтерфейс користувача, керування кількома каналами та перегляд статистики, інтеграція ШІ. Важливо враховувати вимоги та специфічні потреби користувача при виборі програмного забезпечення для контент-менеджменту. Також було виявлено, що багато універсальних програм виступають лише "транспортном" для готових повідомлень, не забезпечуючи автономного збору точних даних чи багатокритеріального скорингу матеріалів. Важливо також звернути увагу на оптимізацію мережевого трафіку та швидкість роботи, оскільки відсутність глибокого кешування медіафайлів збільшує навантаження на систему при регулярному автопостингу.

Одним із ключових факторів успішної автоматизації є здатність програмного забезпечення ефективно працювати з великими масивами структурованих даних, забезпечуючи високу якість вибірки для кожної публікації. Під час аналізу було виявлено, що інтеграція із зовнішніми

інтерфейсами, такими як Shikimori API, є ключовою функцією для безперервного та безпомилкового наповнення бази даних. Також було виявлено, що використання реляційних систем управління базами даних (СУБД) значно полегшує збереження, фільтрацію та подальшу роботу з агрегованою інформацією.

Отже, аналіз даної предметної області показав різноманітність інструментів для адміністрування Telegram-каналів, а також важливість вибору оптимального рішення. На основі цього можна зробити висновок, що вибір або створення ПЗ має базуватися на конкретних потребах проєкту, враховуючи необхідність парсингу, кешування та унікальні алгоритми відбору записів.

За результатами проведеного аналізу зроблено висновок, що у наш час існує досить багато програмних засобів для роботи в Telegram. Проте універсальні планувальники чи сервіси з генеративним штучним інтелектом є менш гнучкими у відношенні до специфічних потреб управління структурованими каталогами. З метою задоволення цих користувацьких потреб важливо розробляти власне ПЗ Telegram-бота, яке об'єднає базу даних, парсер та систему інтелектуальної вибірки.

Сформульовано функціональні вимоги до програмного забезпечення Telegram-бота.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір та обґрунтування мови програмування

Для розробки Telegram-бота в межах даного дипломного проєкту було обрано мову програмування Python. Такий вибір пояснюється не лише популярністю мови, а й її практичною придатністю до задач, пов'язаних з автоматизацією, роботою з API та обробкою структурованих даних.

Python було створено наприкінці 1980-х років, а його розробку розпочав Гвідо ван Россум у 1989 році. Від самого початку мова формувалася як інструмент, орієнтований на читабельність коду, простоту синтаксису та зручність супроводу. Саме ця філософія стала однією з причин її широкого поширення у прикладній розробці [8]. Для невеликих і середніх програмних систем, особливо таких, що мають чітко визначену предметну область, Python є зручним тим, що дозволяє реалізувати логіку без зайвої складності та залишає код зрозумілим навіть після тривалої перерви в роботі над проєктом.

У контексті Telegram-ботів ця властивість набуває особливого значення. Бот для автоматизованої публікації контенту не є одноразовим скриптом, а виступає частиною більшої системи, де важливими є обробка запитів користувача, робота з базою даних, звернення до зовнішнього API та формування готових повідомлень для каналу. У такій архітектурі мова програмування повинна забезпечувати не лише можливість швидко написати код, а й простоту його подальшого розвитку.

Ще однією важливою перевагою Python є його екосистема бібліотек. Для цієї мови існує велика кількість рішень, які спрощують взаємодію з Telegram Bot API, виконання HTTP-запитів, роботу з JSON-відповідями, а також інтеграцію з базами даних. Це дає змогу зосередитися саме на функціональній частині проєкту, а не на реалізації низькорівневих механізмів.

Окремо слід відзначити підтримку асинхронного програмування. Для Telegram-бота це не другорядна деталь, а одна з ключових технічних вимог. Програма постійно виконує операції, пов'язані з мережею: отримує

повідомлення, надсилає відповіді, звертається до API, оновлює записи в базі даних і формує публікації. Якщо такі дії виконуються послідовно, система може затримуватися під час очікування відповіді від сервера. Модуль `asyncio`, який є частиною стандартної бібліотеки Python, дозволяє будувати асинхронну логіку та ефективно обробляти кілька мережових запитів без блокування основного потоку виконання [9].

Ще одним аргументом на користь Python є його стабільне місце у сфері бекенд-розробки. Її популярність підтверджується як професійними рейтингами, так і результатами опитувань розробників. У подібних джерелах (наприклад, Stack Overflow Developer Survey та TIOBE Index) Python стабільно входить до числа найуживаніших мов програмування, що свідчить про високий рівень довіри до неї в індустрії. Вагомим фактором також є велика спільнота розробників, яка створює документацію та готові рішення для парсингу даних та асинхронних запитів [10].

Для наочного порівняння Python доцільно зіставити з двома поширеними альтернативами – JavaScript (Node.js) та Java (таблиця 2.1). Це дозволяє оцінити не лише популярність мов, а й їхню придатність саме для задач Telegram-бота.

Python має найбільш збалансовані характеристики саме для цього проєкту. Мова забезпечує високу швидкість розробки, зручну роботу з даними, які надходять через API, і достатньо просту модель асинхронного виконання.

JavaScript також є сильним варіантом для мережових сервісів, однак у задачах, де важливі читабельність коду, структурована обробка даних і швидке створення робочого прототипу, Python часто виявляється більш практичним. Java, своєю чергою, вирізняється надійністю, але зазвичай потребує більшого обсягу коду та має вищий поріг входження.

Таблиця 2.1 – Обґрунтування вибору мови програмування

Критерій	Python	JavaScript (Node.js)	Java
Зручність та швидкість кодування	+++	++	+
Підтримка асинхронності "з коробки"	+++	+++	++
Наявність спеціалізованих бібліотек для Telegram-ботів	+++	++	+
Зручність для парсингу даних (API)	+++	+++	+
Поріг входження	+++	++	+

Отже, вибір Python як мови програмування є обґрунтованим і відповідає технічним вимогам системи. Саме ця мова найкраще поєднує простоту написання коду, розвинену екосистему бібліотек, підтримку асинхронності та зручність інтеграції з зовнішніми API.

2.2 Вибір системи управління базами даних та засобів взаємодії

Для реалізації Telegram-бота, який працює з каталогом аніме, зберігає ідентифікатори медіафайлів, а також відстежує вже опублікований контент у межах механізму cooldown, необхідна надійна реляційна база даних. У межах цього проєкту база даних виконує не допоміжну, а центральну функцію: саме в ній зберігаються структуровані відомості про записи каталогу, параметри публікацій, службові ознаки стану матеріалів та зв'язки між контентом і медіаданими. Тому вибір СУБД має враховувати не лише зручність роботи, а й простоту розгортання, стабільність та достатній рівень продуктивності для одно-процесної архітектури Telegram-бота.

У даному випадку доцільним є використання SQLite. Це легка вбудована реляційна СУБД, яка зберігає всі дані у звичайному локальному файлі та не потребує встановлення окремого сервера [11]. Така модель є особливо зручною для Telegram-бота, який працює як єдиний застосунок і не потребує складної клієнт-серверної інфраструктури. Фактично вся база даних існує як один файл, з яким програма працює напрямку через стандартні засоби доступу, що значно спрощує розгортання на сервері та зменшує кількість можливих точок відмови.

Практична цінність SQLite для даного проєкту полягає в її відповідності масштабові задачі. Бот не є розподіленою багатокористувацькою системою з паралельними транзакціями на великому навантаженні. Його основна функція – автоматизовано зберігати та опрацьовувати каталог контенту, а також швидко отримувати потрібні записи для публікації. Для таких сценаріїв SQLite є цілком достатньою, оскільки вона забезпечує повноцінну підтримку SQL, транзакційність і цілісність даних, не ускладнюючи архітектуру зайвими компонентами.

Окремою перевагою є простота розгортання. Якщо використовувати PostgreSQL або MySQL, то розробнику доведеться налаштовувати окремий сервер баз даних, управляти обліковими записами, конфігурацією підключення, правами доступу та резервним копіюванням. У випадку SQLite цього не потрібно: достатньо створити файл бази даних і підключити його до застосунку. Для невеликого проєкту така схема є не лише зручною, а й практичною, оскільки дозволяє швидше перейти від етапу проєктування до етапу реалізації та тестування.

Для взаємодії з базою даних у проєкті доцільно використовувати SQLAlchemy як ORM-рівень [12]. ORM дозволяє працювати з таблицями бази даних як з об'єктами мови Python, що суттєво спрощує код і робить його більш структурованим. Замість ручного формування великої кількості SQL-запитів розробник оперує моделями, полями та методами, що знижує ризик помилок і полегшує супровід коду. Для системи, де передбачено зберігання різних типів

даних – аніме, медіафайлів, службових станів публікацій і таймерів cooldown – це є особливо корисним.

Важливо також враховувати безпековий аспект. Використання "чистого" SQL-коду, особливо в комбінації з ручним формуванням рядків запиту, може створювати ризики SQL-ін'єкцій. Хоча для локального Telegram-бота загроза може здаватися незначною, краще мінімізувати такі ризики ще на етапі проєктування. SQLAlchemy дає змогу значною мірою уникнути цієї проблеми завдяки параметризованим запитам і більш високому рівню абстракції, що робить роботу з даними безпечнішою та прозорішою.

Ще однією важливою перевагою SQLAlchemy є підтримка асинхронного підходу. У поєднанні з Python asyncio це дозволяє будувати сучасну архітектуру, у якій запити до бази даних не блокують основний потік виконання бота. Для Telegram-бота, який одночасно може обробляти повідомлення, звертатися до API та записувати або читати дані з бази, така модель є зручною і продуктивною. Асинхронний доступ до СУБД зменшує затримки та покращує відгук системи, особливо в режимі регулярного автопостингу та перевірки стану контенту.

Окремо слід наголосити на гнучкості SQLAlchemy з точки зору подальшого розвитку проєкту. Якщо з часом виникне потреба змінити SQLite на іншу СУБД, наприклад PostgreSQL, це можна буде зробити з мінімальними змінами в логіці застосунку. Саме в цьому полягає одна з головних переваг ORM: вона відокремлює бізнес-логіку від конкретної реалізації бази даних. Для проєкту це важливо, оскільки архітектура системи повинна залишатися придатною для масштабування, навіть якщо на початковому етапі використовується максимально проста конфігурація.

Основні характеристики SQLite, PostgreSQL та MySQL з точки зору потреб даного проєкту для порівняння наведено у таблиці 2.2.

Таблиця 2.2 – Обґрунтування вибору системи управління базами даних

Критерій	SQLite	PostgreSQL	MySQL
Простота налаштування та розгортання	+++	++	++
Відсутність необхідності в окремому сервері	+++	+	+
Ефективність споживання ресурсів (ОЗП/ПЗП)	+++	++	++
Зручність для розподілених високонавантажених веб-систем	+	+++	++

Як видно, SQLite найкраще відповідає умовам даного проєкту на етапі початкової реалізації. Вона не потребує окремого сервера, майже не ускладнює розгортання і працює з мінімальним споживанням ресурсів, що є важливим для невеликого Telegram-бота. PostgreSQL, навпаки, є значно кращим вибором для розподілених високонавантажених систем, однак для поточної архітектури її можливості є надлишковими. MySQL займає проміжну позицію, але також потребує окремої інфраструктури, яка не є необхідною в межах цього проєкту.

Отже, зв'язка SQLite та SQLAlchemy є найраціональнішим вибором для архітектури Telegram-бота. SQLite забезпечує просте, надійне та ресурсоефективне зберігання даних, а SQLAlchemy – зручний, безпечний і розширюваний рівень взаємодії з таблицями бази даних. У поєднанні ці засоби створюють достатньо гнучку основу для реалізації каталогу аніме, механізму cooldown, збереження `file_id` та подальшого автопостингу без надмірного ускладнення системи.

2.3 Обґрунтування вибору технологій інтеграції (API)

Сучасні програмні системи все частіше базуються на мікросервісній взаємодії, де окремі компоненти обмінюються даними через відкриті інтерфейси програмування додатків (API). Для Telegram-бота, який автоматизує роботу з аніме-контентом, це означає, що основна логіка поєднується з функціями зовнішніх сервісів: саме API забезпечують зв'язок між застосунком і зовнішнім світом. У даному випадку ключовими є дві інтеграції: з платформою Telegram, яка відповідає за комунікацію з користувачами, та з зовнішнім джерелом даних про контент, яке формує тематичний каталог для автопостингу.

Для взаємодії з Telegram використовується офіційний Telegram Bot API, що реалізується як HTTP-інтерфейс. Цей API дозволяє відправляти повідомлення, отримувати оновлення про дії користувачів й виконувати інші операції, пов'язані з каналами та чатами. Для програмної роботи з цим інтерфейсом у проєкті застосовується спеціалізована асинхронна бібліотека-обгортка `python-telegram-bot`. Вона спрощує роботу з HTTP-запитами до Bot API у вигляді зручних об'єктів мови Python. Така організація дає змогу працювати з повідомленнями, медіа та командами на рівні високої абстракції, не вдаючись до ручного формування рядків запиту, що значно спрощує розробку й зменшує ризик помилок [13].

Однією з головних переваг Telegram Bot API є підтримка механізму `file_id`. Коли бот отримує або завантажує зображення, платформа повертає унікальний ідентифікатор цього файлу. У подальшому цей `file_id` можна використовувати для повторної відправки того ж медіафайлу без його фізичного завантаження на сервери Telegram. Для великого каталогу медіа, як у випадку з аніме-постерами, це суттєво економить мережевий трафік, прискорює обробку запитів і зменшує навантаження як на сервер із ботом, так і на сервери месенджера. З практичної точки зору це означає, що один раз

завантажені зображення можуть бути багаторазово використані для різних публікацій, а система лише зберігає відповідні `file_id` у локальній базі даних.

Для наповнення тематичного каталогу контенту обрано зовнішній REST API сервісу Shikimori. Цей інтерфейс відповідає за доступ до великої бази аніме-тайтлів, їхніх описів, жанрів, рейтингів, кількості епізодів, статусів виходу та інших метаданих. Відмінність Shikimori від багатьох інших джерел полягає в тому, що він віддає дані у чітко структурованому JSON-форматі, а також підтримує розширені описи та деталізовану статистику, які критично необхідні для роботи алгоритму скорингу та автоматизованого формування постів.

Для даного проєкту важливим є також те, що Shikimori має вбудовану підтримку локалізацій, у тому числі описів кирилицею. Це забезпечує кращу відповідність текстового контенту потребам цільової аудиторії. У результаті бот може використовувати вже підготовлені тексти, а не вдаватися до додаткового перекладу або ручного форматування, що значно спрощує роботу з десятками тисяч тайтлів. З інженерної точки зору така структура JSON-даних разом із наявністю локалізацій робить Shikimori API найефективнішим джерелом для формування каталогу.

Для порівняння методів отримання контенту можна розглянути три типові підходи: використання Shikimori API, MyAnimeList API та парсинг HTML-сторінок (табл. 2.3).

Таблиця 2.3 – Порівняння методів отримання контенту для бази даних

Метод	Зручність отримання структурованих даних (JSON)	Наявність локалізованих описів (кирилиця)	Ризик блокування зі сторони сервера	Швидкість обробки мережевого запиту
Shikimori API	+++	+++	+	+++
MyAnimeList API	+++	+	++	++
Web Scraping	+	+	-	-

Як видно з таблиці 2.3, Shikimori API виявляється найбільш раціональним варіантом для даного проєкту. Він надає дані у структурованому JSON-форматі, має повноцінну підтримку кириличних описів, мінімальний ризик блокування через використання офіційного інтерфейсу, а відповіді від сервера приходять швидко. MyAnimeList API, хоч і також пропонує зручний JSON-доступ, має мінімальну підтримку локалізацій, через що він є менш придатним для цільової аудиторії. Веб-скрейпінг, навпаки, вимагає складного аналізу HTML-структури, залежить від змін у верстці сторінки, має критично високий ризик блокування та пов'язаний з довгими відповідями сервера, що робить цей метод нестабільним.

Отже, зв'язка Telegram Bot API (через бібліотеку python-telegram-bot) та Shikimori API створює ідеальну екосистему для автоматизації контент-менеджменту. Telegram Bot API забезпечує надійну комунікацію з платформою, використовуючи file_id для оптимізації роботи з медіа, а Shikimori API постачає структурований, добре локалізований і актуальний контент. Таке поєднання інтерфейсів дозволяє розроблюваному ПЗ ефективно отримувати, зберігати та публікувати матеріал без ручного втручання.

2.4 Висновки за розділом 2

Для реалізації ПЗ було проведено аналіз та вибір оптимальних інструментів розробки. Обґрунтовано використання мови програмування Python, яка завдяки лаконічному синтаксису, багатій екосистемі спеціалізованих бібліотек та нативній підтримці асинхронності є найбільш раціональним вибором для створення ботів. Це дозволяє ефективно обробляти безліч мережових запитів без блокування основного потоку виконання програми.

Аналіз систем управління базами даних показав, що для однопроцесної архітектури Telegram-бота найкраще підходить легка реляційна СУБД SQLite. Вона не потребує розгортання окремого сервера, що значно спрощує процес

впровадження системи та мінімізує споживання апаратних ресурсів. Для безпечної та зручної взаємодії з базою даних обрано ORM SQLAlchemy, яка забезпечує захист від SQL-ін'єкцій, спрощує написання запитів та підтримує асинхронну роботу з даними.

Для комунікації з месенджером обрано офіційний Telegram Bot API (реалізований через бібліотеку `python-telegram-bot`), що дозволяє оптимізувати мережевий трафік за допомогою механізму кешування медіафайлів `file_id`. Для безперервного отримання актуального контенту обрано Shikimori REST API, який надає деталізовані та структуровані дані у форматі JSON із повноцінною підтримкою кириличних локалізацій.

Отже, обраний стек технологій (Python, SQLite, SQLAlchemy, Telegram Bot API та Shikimori API) повністю покриває функціональні вимоги до розроблюваної системи. Він створює надійну, швидкодіючу та гнучку архітектурну основу для реалізації алгоритмів парсингу, скорингу та автоматизованого управління контентом у Telegram-каналі.

3 ОПИС ПРОГРАМИ

3.1 Архітектура та структура програмного забезпечення

Розроблений Telegram-бот є не просто набором окремих команд, а цілісною програмною системою, в якій кожен компонент виконує чітко визначену роль. Такий підхід важливий саме для проєкту з автоматизованої публікації контенту, оскільки тут одночасно працюють запити користувачів, фоновий автопостинг, звернення до бази даних і взаємодія із зовнішнім API. Щоб система залишалася керованою, логіка програми була розділена на кілька рівнів, а взаємодія між ними організована через зрозумілі інтерфейси.

Ланцюжок роботи системи виглядає послідовно. Користувач взаємодіє з Telegram-клієнтом на смартфоні або комп'ютері. Далі клієнт передає повідомлення на сервери Telegram, після чого оновлення потрапляють до Telegram Bot API. Уже серверна частина бота, реалізована на Python, отримує ці оновлення, обробляє їх і формує відповідь або нову дію. Така схема дозволяє відокремити інтерфейс спілкування від внутрішньої логіки системи. Зовні користувач бачить звичайного бота в Telegram, але всередині працює багаторівнева програма з чітким розподілом відповідальності.

Для отримання оновлень використовується метод Long Polling. Це рішення виявилось зручнішим за Webhooks для даного проєкту, оскільки не потребує виділеної статичної IP-адреси, налаштування зворотного HTTPS-з'єднання та підготовки окремих SSL-сертифікатів. Так бота можна запускати майже в будь-якому середовищі без складної мережевої конфігурації. До того ж Long Polling добре підходить для системи, яка працює в режимі постійного фонового циклу й сама ініціює запит на нові оновлення. Це значно спрощує розгортання, тестування та подальший супровід [14].

Архітектурно програму побудовано за модульним принципом, близьким до чистої архітектури. Такий підхід дозволяє розділити зовнішню взаємодію, бізнес-логіку та роботу з даними. У межах цього проєкту можна виділити три основні шари: шар взаємодії з месенджером, шар бізнес-логіки

та шар даних і алгоритмів. Кожен із них виконує свою частину задачі й не бере на себе зайвих функцій. Це робить систему зрозумілішою, а головне – придатною до розширення.

Шар взаємодії з месенджером відповідає за прийом команд, повідомлень і службових оновлень від Telegram. Саме тут відбувається первинна обробка тексту, видалення зайвих тегів, приведення повідомлення до стандартного вигляду та перенаправлення запиту далі у внутрішню логіку. Цей шар не приймає рішень щодо того, який контент показати або коли його опублікувати. Його задача значно простіша: коректно прийняти дані від Telegram, очистити їх від технічних елементів і передати вглиб системи. Завдяки цьому логіка обробки повідомлень не змішується з логікою роботи бази даних чи з правилами автопостингу.

Шар бізнес-логіки виконує роль координатора всієї системи. Саме тут зосереджено основний сценарій роботи бота: обробка команд адміністратора, запуск окремих сервісних процедур, планування дій, запуск фонових асинхронних циклів автопостингу та реалізація алгоритмів ваг для вибору аніме. У цьому шарі визначається, що робити після отримання нової команди, як перевірити стан бази, який запис може бути опублікований, а який ще має перебувати в режимі очікування. Також тут алгоритм зваженого вибору використовує дані з БД для того, щоб визначити, який матеріал краще опублікувати наступним (враховуючи пріоритет тайтлу, його актуальність, наявність медіафайлу та час від попередньої публікації). Окремо реалізується цикл, який періодично перевіряє готовність контенту до відправлення. Така схема зручна тим, що користувацькі дії та автоматичні процеси не конфліктують між собою. Вони існують у межах одного застосунку, але логічно розділені.

Шар даних є нижнім рівнем системи. Саме тут відбувається взаємодія з базою даних, зберігання службової інформації та кешування медіа. База даних містить записи каталогу, ідентифікатори медіафайлів, часові обмеження повторної публікації та інші ознаки, які потрібні для роботи системи. Цей шар

не приймає рішень про вибір контенту, а лише забезпечує доступ до репозиторіїв та виступає надійним джерелом даних на запит шару бізнес-логіки. Отже такий підхід дозволяє уникати повторів і зробити автопостинг більш різноманітним (рисунок 3.1).

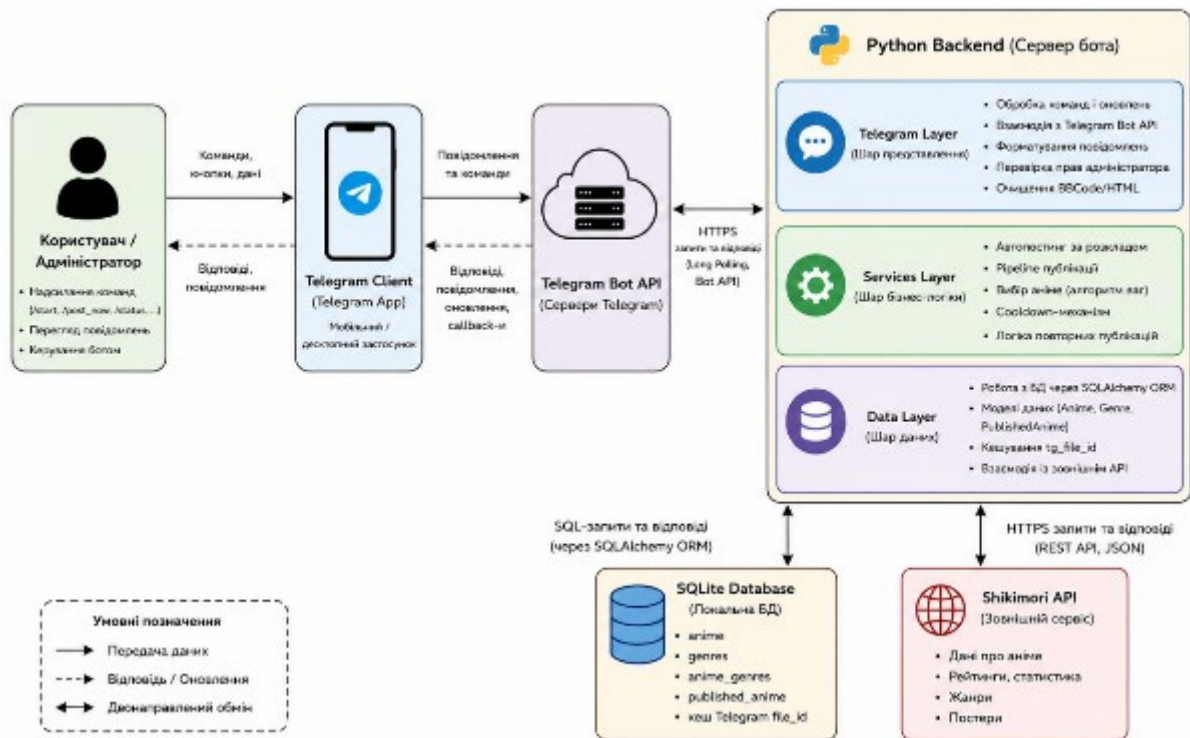


Рисунок 3.1 – Структура взаємодії компонентів ПЗ Telegram-бота

Окрему увагу в архітектурі приділено потокобезпеці. Оскільки бот одночасно працює з двома типами подій – ручними командами адміністратора та фоновим автопостингом – виникає ризик стану гонитви (race condition). Така ситуація можлива тоді, коли два процеси намагаються змінити один і той самий ресурс у майже однаковий момент часу. Щоб цього уникнути, у системі використовується файлове блокування. Воно не дозволяє одночасно виконувати взаємовиключні дії, наприклад, коли адміністратор запускає ручну публікацію в той самий момент, коли фоновий цикл уже готує власний пост.

Файлове блокування в даному випадку виконує функцію простого й надійного механізму синхронізації. Якщо один процес уже опрацьовує подію, інший чекає завершення операції. Це особливо важливо для системи, де в базі

даних можуть змінюватися статуси постів, оновлюватися таймери публікацій або змінюватися стан контенту. Без такого захисту могли б виникати дублювання, некоректні оновлення або ситуації, коли один і той самий запис публікується в канал двічі. Тому файлове блокування тут виступає критичним елементом стабільності всієї архітектури [15].

Загалом структура програмного забезпечення побудована так, щоб кожен рівень був максимально автономним, але водночас пов'язаним з іншими частинами системи через чіткі інтерфейси. Telegram-шар приймає та нормалізує події, сервісний шар координує сценарії роботи, а шар даних забезпечує збереження та відбір контенту. Така організація дає змогу підтримувати простоту коду, спрощує тестування окремих модулів і створює хорошу основу для подальшого розвитку бота. Саме тому обрана архітектура є вдалою для задачі автоматизованої публікації контенту в Telegram.

3.2 Структура бази даних та моделі даних

База даних є одним з основних компонентів розробленого Telegram-бота. Вона використовується для зберігання інформації про аніме, медіафайли, історію публікацій і службові параметри системи. Для проєкту обрано реляційну СУБД SQLite. Це рішення зручне для локального розгортання, оскільки база зберігається у звичайному файлі та не вимагає адміністрування окремого сервера. Взаємодію з даними реалізовано через ORM SQLAlchemy. Такий підхід спрощує роботу з об'єктами в коді на Python та значно зменшує ризики SQL-ін'єкцій завдяки параметризації запитів під капотом.

Центральна сутність каталогу – таблиця anime. Один рядок у ній відповідає конкретному тайтлу і містить усі необхідні для автопостингу параметри.

Ключовим ідентифікатором запису є поле shikimori_id – унікальний ID тайтлу із зовнішнього API Shikimori. Він використовується для синхронізації нашого локального каталогу з першоджерелом. Завдяки цьому бот може

регулярно оновлювати статистику записів, не прив'язуючись до можливих змін у їхніх текстових назвах.

Багатомовність та пошук реалізовані через поля `title_ru`, `title_en` та `title_original`. Поле `title_ru` зберігає локалізовану кириличну назву, отриману безпосередньо з API Shikimori, і виступає основним заголовком при формуванні поста. Англійська та оригінальна назви зберігаються здебільшого як резервні варіанти для коректного пошуку в базі.

Важливу роль в алгоритмі вибору контенту відіграють статистичні метрики: `rating` (середній бал), `members` (популярність тайтлу серед користувачів платформи) та `year` (рік виходу). Саме ці показники дозволяють системі скорингу розраховувати вагу кожного аніме, балансуючи публікації між визнаною класикою та гучними новинками.

Окреме архітектурне рішення для оптимізації продуктивності – поле `tg_file_id`. Коли бот вперше завантажує постер у Telegram, сервери месенджера повертають унікальний ідентифікатор цього медіафайлу. Цей рядок записується в базу. При наступних публікаціях скрипт відправляє лише цей короткий ідентифікатор замість самого зображення. Це радикально економить мережевий трафік і прискорює відправку повідомлень.

Для нормалізації даних жанри винесено в окрему таблицю `genres`. Зв'язок із конкретними тайтлами реалізовано як "багато-до-багатьох" (Many-to-Many) через проміжну таблицю `anime_genres`. Таке розділення усуває дублювання тексту і дозволяє алгоритму швидко фільтрувати каталог, оскільки пошук працює за цілочисельними індексами, а не по рядках.

Історія взаємодії з каналом фіксується в таблиці `published_anime`. Вона працює як журнал транзакцій. Поле `published_at` містить мітку часу і слугує базою для системи контролю повторів. Алгоритм бота перевіряє цю таблицю, щоб одне й те саме аніме не публікувалося частіше, ніж дозволено внутрішніми налаштуваннями.

Цікавим інженерним нюансом є наявність поля `episodes` у цій таблиці. Воно фіксує кількість серій тайтлу не в реальному часі, а саме як "знімок" на

момент публікації поста. Це дозволяє боту розрізняти завершені серіали та онгоїнги. Якщо при наступному парсингу виявиться, що поточна кількість епізодів в API більша за збережену в `published_anime`, алгоритм розуміє, що вийшла нова серія, і достроково знімає обмеження на повторну публікацію. Логічну структуру спроектованої бази даних ілюструє рис. 3.2.

У рамках загальної архітектури системи описані моделі відносяться до шару даних (Data Access Layer). Цей рівень є "пасивним": він лише надійно зберігає та віддає інформацію за запитом, тоді як усі рішення щодо відбору та форматування постів приймає шар бізнес-логіки.

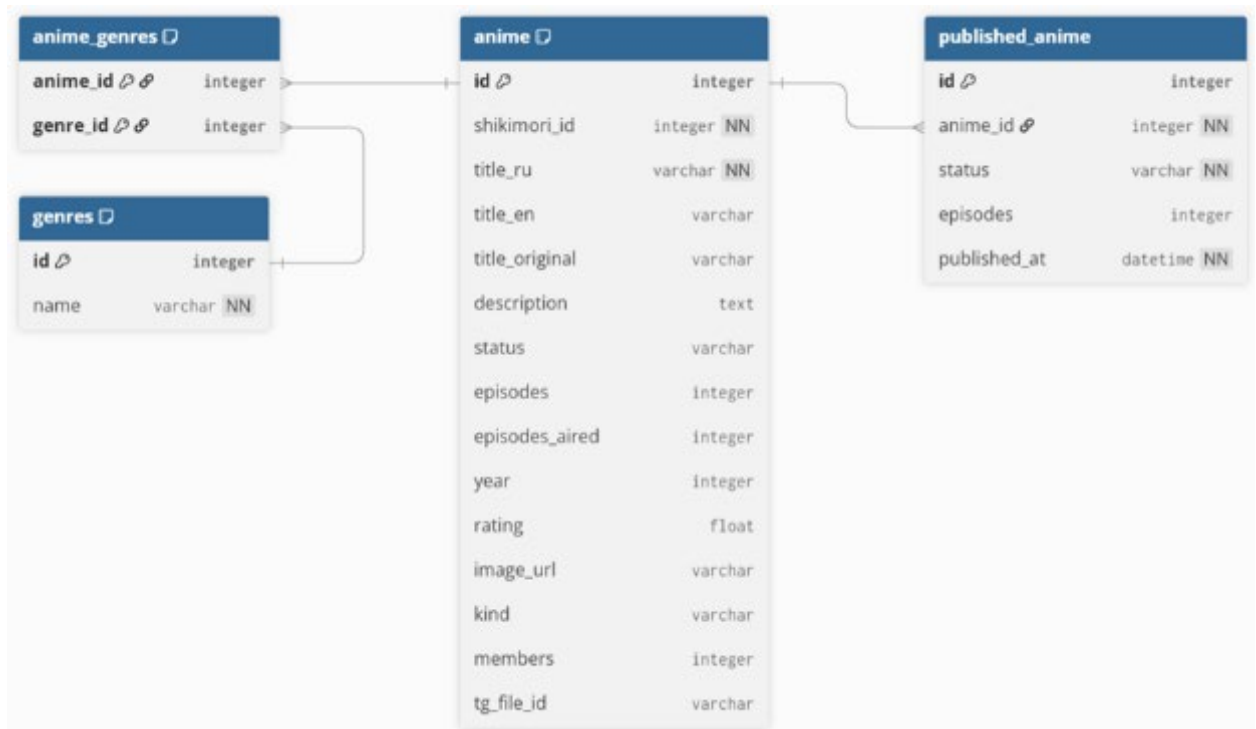


Рисунок 3.2 – ER-діаграма бази даних

3.3 Розробка алгоритму агрегації та скорингу контенту

Алгоритм вибору контенту є ключовим компонентом розробленої системи. Його головна мета – запобігти ефекту повторюваності виключно популярних хітів та уникнути жанрової одноманітності в Telegram-каналі. Замість простого випадкового вибору, система використовує багатокроковий

алгоритм зваженого скорингу, який балансує стрічку новин, що потенційно підвищує різноманітність контенту.

Процес формування пулу кандидатів та фільтрація розпочинається із запиту до бази даних для формування початкового пулу потенційних кандидатів. Щоб забезпечити баланс, вибірка формується з двох частин: топ-300 найпопулярніших тайтлів (за кількістю додавань у списки користувачів) та 200 абсолютно випадкових записів із каталогу. Такий підхід гарантує, що в ротацию потрапляють як визнані шедеври, так і маловідомі, але якісні роботи.

Далі пул проходить етап жорсткої програмної фільтрації. Алгоритм відсіює записи, які не підходять для форматування (наприклад, мають занадто короткий текстовий опис), а також нерелевантні формати випуску (музичні кліпи, рекламні промо-ролики). Критичним кроком на цьому етапі є перевірка через систему контролю повторів: тайтли, час попередньої публікації яких ще не перетнув встановлений ліміт (cooldown), безумовно виключаються з поточного пулу.

Кожному кандидату, що успішно пройшов етап фільтрації, розраховується базова математична вага (W_{base}). Вона формується з двох основних статистичних метрик бази даних та бонусів.

Основні метрики – це рейтинг (*rating*) та популярність (*members*). Важливим інженерним рішенням є використання для популярності логарифмічної функції: $\log_{10}(members + 1)$. Додавання одиниці під знаком логарифма є програмним захистом від помилки обчислення нуля ($\log_{10}(0)$) для абсолютно нових тайтлів. Загалом логарифмічне згладжування вирішує проблему нормалізації: воно не дозволяє великим популярним релізам із мільйонами глядачів математично задавити якісні, але менш відомі нішеві роботи.

Додаткові модифікатори алгоритму дозволяють йому надавати перевагу більш актуальним тайтлам:

- B_{year} (бонус за рік виходу): ступінчастий модифікатор, який стимулює публікацію свіжого контенту, новинки (поточного або минулого

року) отримують максимальний бонус (+5.0). Для тайтлів віком 3–7 років бонус поступово знижується, а для класичних робіт, старіших за 15 років, застосовується невеликий штраф (-1.0);

- $B_{episodes}$ (бонус за формат): повноцінні телесезіони (12 і більше епізодів) отримують позитивний приріст (+1.5), тоді як одиничні випуски чи короткометражки (1 епізод) злегка штрафуються (-0.5);

- B_{status} (бонус актуальності): найвищий пріоритет надається серіалам, що виходять прямо зараз (+2.0) та офіційним анонсам (+1.0), щоб підтримувати стрічку новин каналу в актуальному стані.

Базову вагу кандидата розраховують як лінійну комбінацію цих елементів з відповідними множниками значущості (w_1, w_2) за формулою (3.1):

$$W_{base} = w_1 \cdot rating + w_2 \cdot \log_{10}(members + 1) + B_{year} + B_{episodes} + B_{status}, \quad (3.1)$$

де W_{base} – базова математична вага кандидата;

w_1, w_2 – константні вагові множники значущості;

$rating$ – середній оцінковий бал тайтлу;

$members$ – кількість користувачів, які додали тайтл у свої списки;

B_{year} – ступінчастий бонус за рік виходу;

$B_{episodes}$ – бонус за кількість епізодів (формат);

B_{status} – бонус актуальності (статус виходу).

Для забезпечення різноманітності контенту розроблено окремий модуль, який накладає динамічні штрафи на базову вагу. Алгоритм аналізує історію останніх успішних публікацій у каналі. Якщо система виявляє тренд – наприклад, протягом останніх днів було опубліковано багато аніме одного жанру або статусу – алгоритм генерує понижувальні штрафні коефіцієнти ($k < 1.0$) для аналогічних тайтлів із поточного пулу.

Це м'яке обмеження: система не блокує такі тайтли повністю, а лише тимчасово знижує їхній пріоритет, підвищуючи ймовірність вибору контенту інших категорій.

Загальну формулу розрахунку фінальної ефективної ваги кандидата визначають за формулою (3.2):

$$W_{eff} = \max(W_{base} \cdot \prod_{i=1}^n k_{penalty_i}, 0.1), \quad (3.2)$$

де W_{eff} – фінальна ефективна вага кандидата;

W_{base} – базова математична вага;

$\prod_{i=1}^n k_{penalty_i}$ – добуток усіх n накладених динамічних штрафних коефіцієнтів.

Програмна функція $\max(..., 0.1)$ обмежує мінімальне значення ваги на рівні 0.1, залишаючи релізу мінімальну ненульову ймовірність бути обраним. Загальну логіку та послідовність кроків описаного алгоритму вибору контенту наведено у вигляді блок-схеми на рис. 3.3.

Після розрахунку ефективної ваги пул сортується за спаданням. Програмний модуль відбирає топ-50 найкращих кандидатів, після чого робить серед них фінальний випадковий вибір. Внесення елемента випадковості на останньому етапі гарантує, що послідовність постів неможливо передбачити навіть при знанні формули скорингу.

Описаний алгоритм вбудовано у процес автопостингу. Процес ініціюється фоновим планувальником. Для уникнення стану гонитви (race condition) з ручними командами адміністратора, скрипт захоплює атомарне файлове блокування (lockfile). Після цього алгоритм обирає тайтл, формує повідомлення, відправляє його через Telegram Bot API (використовуючи кешований tg_file_id за наявності), фіксує факт публікації в базі даних та звільняє блокування.

Таким чином, розроблений алгоритм виконує автоматизований відбір контенту, що керується математичними правилами для уникнення одноманітності стрічки.

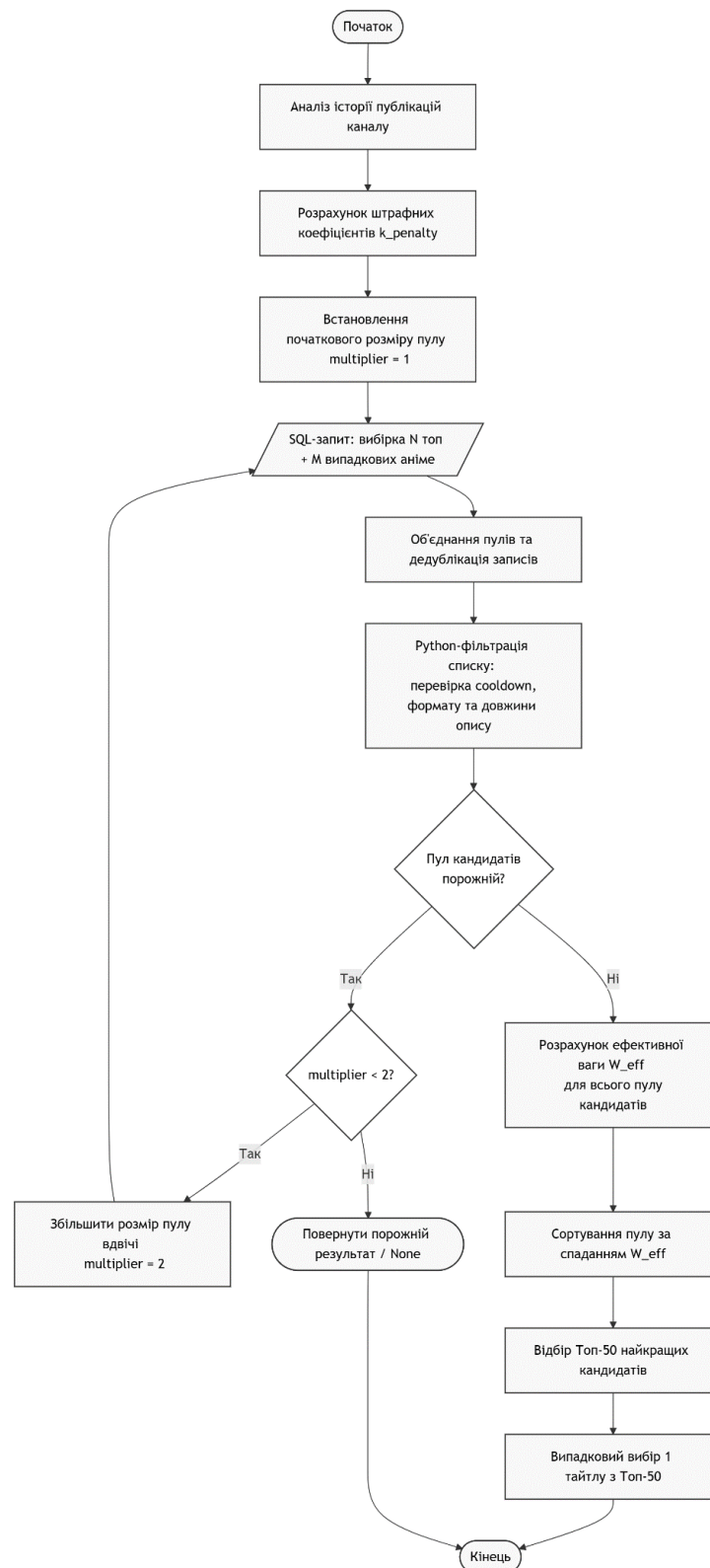


Рисунок 3.3 – Блок-схема алгоритму вибору контенту

Детальніше взаємодію програмних компонентів системи та послідовність операцій під час одного циклу публікації продемонстровано на діаграмі послідовності на рис. 3.4.

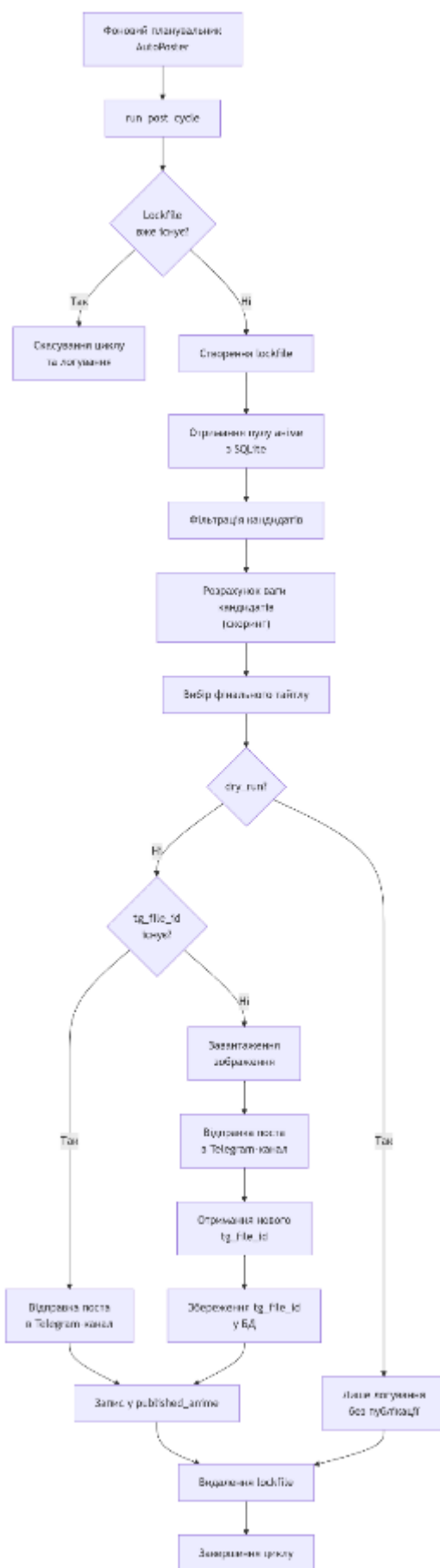


Рисунок 3.4 – Блок-схема процесу автоматичної публікації

3.4 Проектування інтерфейсу взаємодії Telegram-бота

Інтерфейс розробленої системи відрізняється від традиційних десктопних чи веб-додатків відсутністю класичного графічного вікна. Взаємодія повністю побудована на базі платформи Telegram. Архітектурно інтерфейс бота логічно розділено на дві ізольовані зони: публічну (для підписників цільового каналу) та адміністративну (для керування системою через особистий чат).

Публічна частина призначена виключно для споживання контенту кінцевими користувачами. Інтерфейс публікації спроектовано за принципом інформаційної картки, яка складається з двох блоків:

- візуальний блок: постер тайтлу, що привертає увагу аудиторії;
- інформаційний блок: текстовий підпис до медіафайлу.

Текст поста формується у вигляді plain-тексту без розмітки. Візуальна ієрархія досягається за рахунок емодзі-іконок перед кожним полем: назва, рік виходу, кількість епізодів, статус та рейтинг. Далі виводяться жанри у форматі хештегів, після чого подається очищений від BBCode-розмітки опис сюжету. Inline-клавіатури у публічних постах відсутні навмисно – щоб не перевантажувати стрічку каналу і концентрувати увагу читача на контенті.

Адміністративний інтерфейс захищений подвійною перевіркою: ідентифікатора користувача та ідентифікатора чату, що налаштовуються через безпечні змінні середовища. Він надає адміністратору повний спектр функцій для моніторингу та керування системою через набір Telegram-команд:

а) моніторинг стану (/status): виводить поточні параметри роботи системи, зокрема режим симуляції (увімкнено/вимкнено), стан автоматичного постингу, налаштований інтервал між публікаціями у секундах та індикатор активного циклу публікації в реальному часі;

б) ручний запуск публікації (/post_now): негайно ініціює один цикл публікації з поточними налаштуваннями. Якщо активний режим симуляції, публікація виконується віртуально (без надсилання в канал і без запису в базу

даних), а адміністратор отримує текстове підтвердження результату роботи алгоритму;

в) керування автопостингом (/posting_on, /posting_off): команди, що вмикають або вимикають фоновий цикл автоматичних публікацій без перезапуску базового процесу бота. Зміни застосовуються миттєво;

г) керування режимом симуляції (/dry_run_on, /dry_run_off): перемикають режим тестування (dry run). В активному стані всі команди публікації виконуються без реального надсилення повідомлення в цільовий канал, функціонал використовується для безпечного тестування логіки вибору контенту в робочому середовищі (production);

д) налаштування динамічного інтервалу (/interval <min> <max>): встановлює межі випадкового часового інтервалу між автоматичними публікаціями (наприклад, діапазон від 6 до 9 годин), зміна набуває чинності при наступному ітераційному спрацюванні циклу;

е) ручний пошук та миттєва публікація (/post_anime): найбільш інтерактивний модуль інтерфейсу, який підтримує два режими введення:

- 1) за зовнішнім ідентифікатором: публікує конкретний тайтл за його Shikimori ID миттєво, без додаткових підтверджень;

- 2) за назвою: система виконує пошук у два етапи. Спочатку виконується точний пошук за полями назв, а у разі відсутності результатів застосовується алгоритм нечіткого пошуку (fuzzy search) за допомогою бібліотеки rapidfuzz серед найпопулярніших тайтлів.

Для обробки результатів пошуку спроектовано інтерактивну видачу. При кількох знайдених результатах бот надсилає повідомлення з прикріпленою Inline-клавіатурою, де кожна кнопка відповідає окремому аніме зі списку. Оскільки результатів може бути багато, інтерфейс підтримує посторінкову навігацію (пагінацію). Процес гортання реалізовано через оновлення (редагування) існуючого повідомлення, що не засмічує історію чату.

Після натискання на кнопку обраного тайтлу бот публікує пост у цільовий канал, надсилає адміністратору підтвердження, а сесійна клавіатура автоматично видаляється. Також передбачено кнопку скасування для закриття сесії пошуку без виконання публікації.

Такий підхід до проєктування інтерфейсу забезпечує повну відсутність інтерактивних елементів у публічній стрічці каналу (мінімалізм для читача) та водночас надає адміністратору гнучкий, стійкий до помилок набір інструментів для керування системою в реальному часі.

3.5 Висновки за розділом 3

У третьому розділі було проведено проєктування програмного забезпечення для автоматизації, агрегації та публікації контенту в середовищі Telegram. За результатами проведеної роботи можна зробити наступні висновки:

- обґрунтовано та розроблено архітектуру системи на основі принципів «чистої архітектури» (Clean Architecture) – це дозволило відокремити бізнес-логіку від зовнішніх сервісів (Telegram API, Shikimori API), що забезпечує зручність підтримки, розширення та модифікації системи;
- спроектовано реляційну модель бази даних SQLite та розроблено відповідну ER-діаграму – обрана структура дозволяє надійно зберігати метадані аніме-тайтлів, відстежувати історію публікацій через механізм затримок (cooldown) та оптимізувати мережевий трафік шляхом кешування ідентифікаторів медіафайлів;
- розроблено алгоритм автоматизованого відбору контенту-впровадження математичної моделі з логарифмічною нормалізацією популярності та системою динамічних штрафних коефіцієнтів дозволило вирішити проблему жанрової одноманітності стрічки та забезпечити баланс між популярними хітами та нішевими тайтлами;

- спроектовано користувацький та адміністративний інтерфейси взаємодії, які поєднують мінімалістичний публічний вигляд публікацій для читачів та багатофункціональний адміністративний модуль, реалізація таких функцій, як нечіткий пошук (fuzzy search), посторінкова навігація (пагінація) в Inline-меню та режим симуляції (dry run), забезпечує зручність керування та тестування системи;

- логіку взаємодії компонентів системи візуалізовано за допомогою блок-схем та діаграм послідовності, що дозволяє детально простежити шлях даних від запиту до зовнішнього API до моменту фіксації успішної публікації в базі даних.

Отримані результати створюють основу для подальшої реалізації, тестування та практичного використання системи.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Призначення й умови застосування програмного забезпечення

Розроблене ПЗ призначене для автоматизації процесів пошуку, агрегації, автоматизованого відбору та публікації аніме-контенту в цільових Telegram-каналах. Програма розрахована на використання адміністраторами та контент-менеджерами для оптимізації їхньої рутинної роботи та підтримки стабільної активності аудиторії каналу.

Оскільки система реалізована у вигляді автономного серверного застосунку з монолітною архітектурою, вона є невимогливою до апаратних ресурсів і може працювати на базових віртуальних серверах (VPS). Під час роботи система автоматично формує журнали подій (logs), що використовуються для діагностики помилок та моніторингу процесу публікації.

Мінімальні апаратні вимоги для стабільної роботи системи:

- процесор із 2 ядрами з тактовою частотою від 2 ГГц;
- оперативна пам'ять рекомендовано 4 ГБ для стабільної роботи операційної системи та застосунку;
- дисковий простір від 1 ГБ вільного місця для зберігання файлу бази даних anime.db та журналів системи.

Програмні вимоги та середовище виконання:

- операційна система сімейства Linux (рекомендовано Ubuntu 20.04/22.04 для серверного розгортання) або Windows 10/11 (для локальної розробки);
- інтерпретатор: Python версії 3.11 або вище;
- база даних: вбудована СУБД SQLite3;
- мережа: стабільне інтернет-з'єднання для комунікації з Telegram Bot API та Shikimori API.

4.2 Інструкція по експлуатації програмного забезпечення

4.2.1 Звернення до програми

Для початку роботи з системою адміністратору необхідно виконати первинне налаштування, наповнення бази даних та розгортання програмного середовища.

4.2.2 Встановлення залежностей

Система використовує сторонні бібліотеки (зокрема `python-telegram-bot`, `sqlalchemy`, `rapidfuzz`). Для ізоляції залежностей проєкту рекомендується створити віртуальне середовище (`virtual environment`, `venv`) та встановити необхідні пакети за допомогою менеджера пакетів `pip`: «`pip install -r requirements.txt`»

4.2.3 Налаштування змінних середовища

Для безпечного зберігання конфіденційних даних використовується конфігураційний файл `.env`, який має бути створений у кореневій директорії проєкту. Файл повинен містити наступні обов'язкові параметри:

- `BOT_TOKEN` – унікальний токен для доступу до Telegram Bot API, отриманий через службу BotFather;
- `CHANNEL_USERNAME` – публічне ім'я (`username`) або числовий ідентифікатор цільового каналу для публікацій;
- `ADMIN_IDS` – список числових ідентифікаторів адміністраторів (через кому) для надання доступу до команд керування;
- `ALLOWED_CHAT_IDS` – список ідентифікаторів чатів, де дозволено використовувати команди для адмінів;

- DRY_RUN – логічний прапорець (true/false) для активації режиму безпечної симуляції при початковому запуску (без реального надсилання постів у канал);
- POSTING_ENABLED – початковий стан (увімкнено/вимкнено) фонових циклів автоматичних публікацій після запуску системи;
- POST_INTERVAL_MIN та POST_INTERVAL_MAX – нижня та верхня межі (у секундах) для генерації випадкового часового інтервалу між автоматичними постами;
- LOG_LEVEL – рівень деталізації системних журналів (наприклад, INFO для звичайної роботи або DEBUG для глибокої діагностики).

4.2.4 Ініціалізація та наповнення бази даних

Перед запуском самого бота необхідно підготувати базу даних. Для цього в проєкті передбачено набір службових скриптів, які запускаються послідовно:

- команда «python scripts/init_db.py» – створює реляційні таблиці в базі даних;
- команда «python scripts/full_parse.py» – виконує парсинг та початкове наповнення каталогу аніме.

4.2.5 Запуск системи

Після підготовки даних запуск головного процесу бота виконується командою: «python -m bot.telegram_bot».

Після успішного запуску бот переходить у режим очікування (long polling) для обробки адміністративних команд, а асинхронний цикл (asyncio) ініціює фоновий процес автоматичної публікації контенту через визначені проміжки часу.

4.3 Виконання програмного забезпечення

Процес виконання програми та взаємодія з розробленою системою логічно розділяється на два основні сценарії використання: робота адміністратора в закритому особистому чаті з ботом (керування) та споживання контенту кінцевими користувачами в публічному каналі (відображення).

Будь-яка взаємодія з ботом починається з перевірки прав доступу. Якщо користувач не внесений до списку адміністраторів (ADMIN_IDS), система блокує обробку запиту та видає системне повідомлення про відмову в доступі.

Для авторизованого адміністратора передбачено комплекс команд для моніторингу та керування:

- діагностика системи: при виклику команди «/status» бот надсилає звітне повідомлення, яке містить поточний стан автопостингу, статус режиму симуляції (dry run), встановлений часовий інтервал між публікаціями та індикатор активності циклу публікації в реальному часі;
- гнучке налаштування: адміністратор може «на льоту» керувати поведінкою системи за допомогою команд перемикачів, команди «/posting_on» та «/posting_off» керують фоновим циклом публікацій, команди «/dry_run_on» та «/dry_run_off» перемикають режим безпечного тестування, а команда «/interval» дозволяє задати нові межі для генерації випадкового часу очікування між постами та всі зміни супроводжуються текстовим підтвердженням від бота (рисунок 4.1);
- примусова публікація: команда «/post_now» ініціює негайний цикл відбору та публікації аніме, якщо систему переведено в режим тестування (dry run), адміністратор отримує повідомлення з назвою обраного тайтлу, але реальне відправлення в канал та запис до бази даних не виконуються, для уникнення одночасного запуску декількох циклів публікації використовується механізм файлового блокування (lockfile);

– модуль пошуку: пошук конкретного контенту ініціюється командою «/post_anime» - у разі використання текстового запиту алгоритм може знайти декілька можливих результатів, у такому випадку бот формує повідомлення з інтерактивною Inline-клавіатурою, де кожна кнопка містить назву, формат, рік виходу та ідентифікатор тайтлу (для зручної навігації великими списками інтерфейс генерує кнопки пагінації («Назад» та «Вперед»), перемикання сторінок відбувається шляхом редагування поточного повідомлення, що дозволяє уникнути засмічення історії чату (рисунк 4.2), та після вибору тайтлу повідомлення з клавіатурою автоматично видаляється (окрім текстового пошуку, підтримується прямий пошук за числовим ідентифікатором Shikimori).



Рисунок 4.1 – Робота команд перевірки статусу та зміни інтервалів

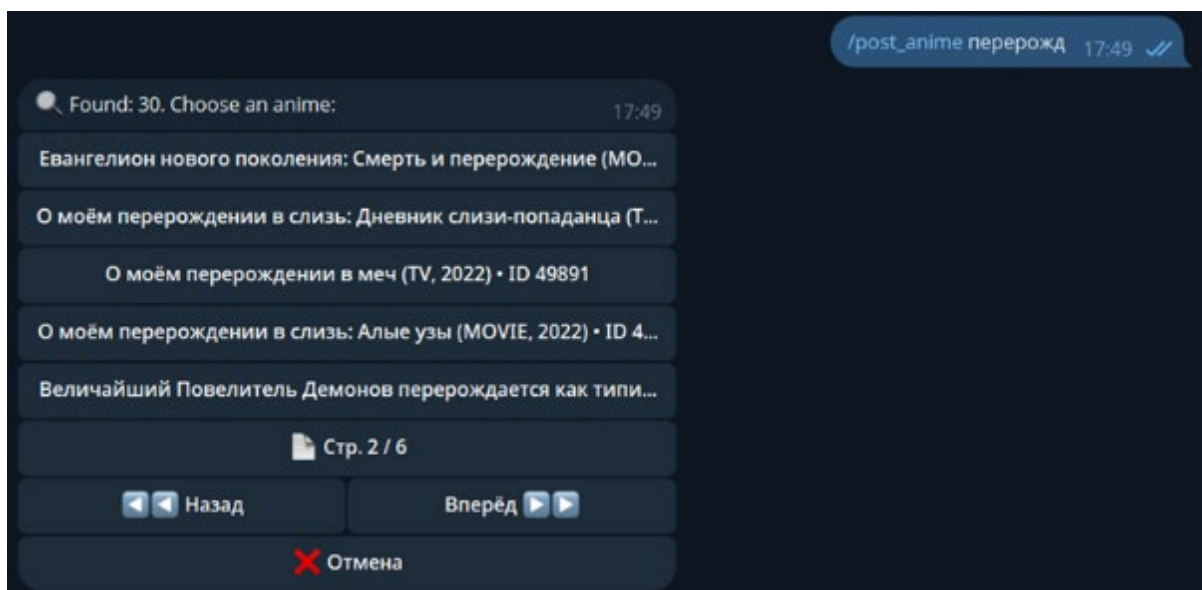


Рисунок 4.2 – Робота команди пошуку та пагінація результатів

Результатом роботи алгоритмів відбору є сформований пост, який автоматично надсилається в цільовий Telegram-канал.

З метою оптимізації швидкості роботи та економії трафіку, алгоритм відправлення медіафайлів (постерів) працює у двох режимах. При першій публікації зображення завантажується за прямим посиланням та надсилається у вигляді байтового масиву. Під час усіх наступних публікацій (якщо тайтл пройшов період затримки cooldown) система використовує внутрішній кешований ідентифікатор Telegram (tg_file_id).

Текстовий блок публікації (caption) генерується у форматі звичайного тексту без використання складної HTML-розмітки. Структурування інформації досягається за рахунок використання тематичних емодзі, що виступають візуальними маркерами для основних полів: локалізованої назви, року виходу, кількості епізодів, рейтингу та статусу.

Перед публікацією текстові дані проходять програмну обробку:

- статуси виходу локалізуються зрозумілою мовою (наприклад, статус «anons» перетворюється на «Анонс»);
- жанри перекладаються з англійської мови та автоматично конвертуються в клікабельні хештеги (наприклад, «#романтика»);
- опис сюжету очищується від залишкових системних тегів (BBCode) та обмежується до 700 символів. У разі перевищення ліміту текст акуратно обрізається по межі останнього слова.

Для спрощення сприйняття контенту публічні повідомлення не містять інтерактивних елементів або додаткових клавіатур (рисунок 4.3).

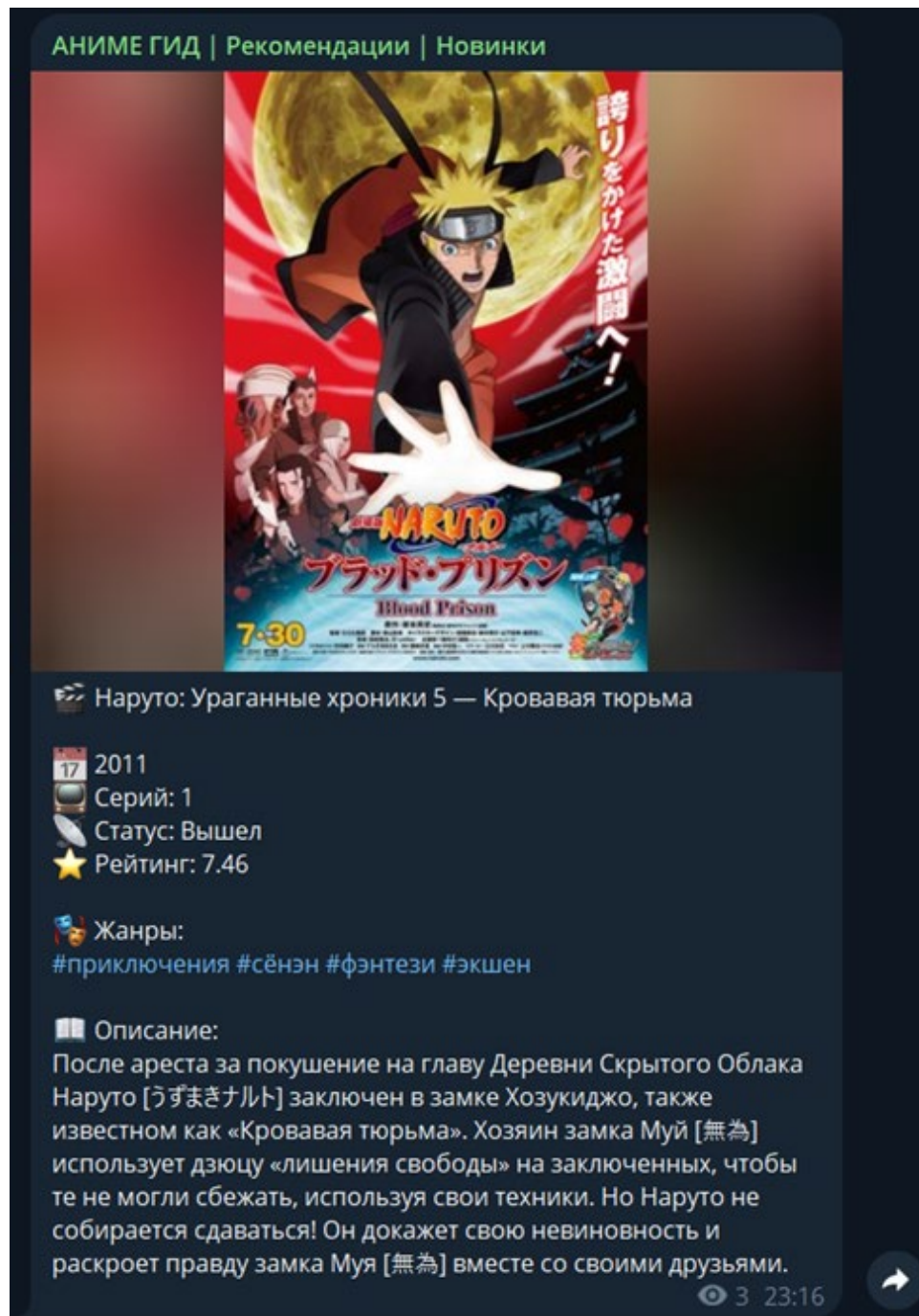


Рисунок 4.3 – Вигляд опублікованого повідомлення у Telegram-каналі

4.4 Тестування програмного забезпечення

Для забезпечення високої надійності, відмовостійкості та коректної роботи розробленої системи було проведено комплексне автоматизоване тестування. В якості основного інструментарію використано сучасний

фреймворк `pytest` (версії 8.0.0 та вище). Тестове покриття охоплює 35 сценаріїв, розподілених на 7 логічних модулів.

Процес верифікації програмного забезпечення було побудовано за такими ключовими напрямками:

а) ізоляція тестового середовища (Fixtures & Mocks) – критично важливою умовою тестування була повна незалежність від робочих даних (production) та зовнішніх сервісів, для реалізації цього підходу застосовано наступні інженерні рішення:

1) база даних в оперативній пам'яті: усі тести, що взаємодіють із рівнем даних, використовують ізольовану in-memory базу SQLite (`sqlite:///memory:`), завдяки фабриці сесій (`session_factory`) та використанню `StaticPool`, для кожного тесту створюється окреме ізольоване тестове середовище;

2) імітація зовнішніх API (Mocking): для уникнення реальних мережевих запитів під час тестування, інтеграція з Telegram API (`send_post_with_cache`) та системні блокування (lock-файли) були імітовані за допомогою модуля `unittest.mock.patch`;

б) модульне тестування бізнес-логіки (Unit Testing) – окремим етапом проведено перевірку чистих функцій та математичних алгоритмів системи без звернення до бази даних:

1) алгоритм розрахунку базової ваги: підтверджено коректність обчислень логарифмічної функції від популярності – перевірено граничні значення, де алгоритм успішно обробляє екстремально низьку популярність без виникнення математичних помилок обчислення логарифма нуля;

2) система штрафних коефіцієнтів: протестовано окремий модуль (`priority_weights`), який відповідає за нарахування штрафів на основі історії публікацій – перевірено логіку формування часових груп (`year buckets`) та динамічний розрахунок множників для пониження пріоритету одноманітного контенту;

3) форматування тексту: перевірено стійкість системи до невалідної розмітки – тести підтвердили коректне очищення тексту від залишкових тегів BBCode, безпечну обрізку довгого опису без розриву слів та правильну трансляцію жанрів у клікабельні хештеги;

4) логіка затримок (Cooldown): протестовано життєвий цикл публікацій – система успішно блокує повторні публікації завершених серіалів до спливу 90 днів, для поточних серіалів ("онгоїнгів") алгоритм не ігнорує ліміт повністю, а коректно скорочує його до мінімально допустимого інтервалу у 2 дні за умови появи нових епізодів;

в) інтеграційне тестування та перевірка граничних випадків – для перевірки взаємодії між компонентами створено комплексні інтеграційні тести:

1) комплексний вибір контенту: протестовано інтеграційну функцію `get_random_anime()`, підтверджено успішну фільтрацію низькоякісних кандидатів, перевірено повернення коректного результату та гарантовано безпечне повернення `None` у випадку абсолютно порожньої бази даних;

2) життєвий цикл публікації: перевірено механізм блокування, у разі перехоплення lock-файлу іншим процесом, система коректно перериває роботу, запобігаючи дублюванню контенту;

3) режим симуляції (`dry_run`): підтверджено, що алгоритм повністю проходить цикл відбору, але гарантовано не створює нових записів у базі даних і не надсилає запити до Telegram API;

4) механізм відкочування транзакцій (Rollback): перевірено поведінку у випадку імітованої мережевої помилки (`NetworkError`) від Telegram, система скасовує зміни в базі даних, щоб тайтл не отримав хибний статус опублікованого.

Запуск створеного тестового набору виконувався через термінал у віртуальному середовищі проєкту з параметром детального виводу результатів (-v) за допомогою команди: «`venv\Scripts\python.exe -m pytest tests\ -v`».

Результати виконання продемонстрували успішне проходження всіх 35 розроблених тестових сценаріїв. Швидкість виконання всього набору не перевищує кількох секунд. Під час тестування не було виявлено критичних помилок, аварійного завершення програми або невдалих тестових сценаріїв. Під час виконання було зафіксовано лише одне системне попередження (DeprecationWarning). Попередження пов'язане із застарілим методом `datetime.utcnow()` у тестовому коді та не стосується логіки самої системи – виробничий код вже використовує `timezone-aware` об'єкти (рисунк 4.4).

```
(venv) PS C:\projects\anime_channel> venv\Scripts\python.exe -m pytest tests\ -v
===== test session starts =====
platform win32 -- Python 3.12.10, pytest-9.0.2, pluggy-1.6.0 -- C:\projects\anime_channel\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\projects\anime_channel
configfile: pytest.ini
plugins: anyio-4.12.1
collected 35 items

tests/test_calculate_weight.py::test_calculate_weight_always_positive PASSED [ 2%]
tests/test_calculate_weight.py::test_calculate_weight_recent_anime_gets_bonus PASSED [ 5%]
tests/test_calculate_weight.py::test_calculate_weight_old_anime_gets_penalty PASSED [ 8%]
tests/test_calculate_weight.py::test_calculate_weight_status_bonus PASSED [ 11%]
tests/test_can_publish.py::test_can_publish_released_cooldown PASSED [ 14%]
tests/test_can_publish.py::test_can_publish_ongoing_new_episodes PASSED [ 17%]
tests/test_can_publish.py::test_can_publish_never_published PASSED [ 20%]
tests/test_can_publish.py::test_can_publish_naive_datetime PASSED [ 22%]
tests/test_can_publish.py::test_can_publish_ongoing_min_cooldown_not_passed PASSED [ 25%]
tests/test_can_publish.py::test_can_publish_unknown_status_uses_60_day_default PASSED [ 28%]
tests/test_formatter.py::test_clean_text_removes_bbcode_character_tag PASSED [ 31%]
tests/test_formatter.py::test_clean_text_removes_url_with_text PASSED [ 34%]
tests/test_formatter.py::test_clean_text_removes_bare_url PASSED [ 37%]
tests/test_formatter.py::test_clean_text_empty_string PASSED [ 40%]
tests/test_formatter.py::test_truncate_text_short_text_unchanged PASSED [ 42%]
tests/test_formatter.py::test_truncate_text_respects_word_boundary PASSED [ 45%]
tests/test_formatter.py::test_format_genres_known_genre_returns_hashtag PASSED [ 48%]
tests/test_formatter.py::test_format_genres_unknown_genre_skipped PASSED [ 51%]
tests/test_formatter.py::test_format_genres_empty_list PASSED [ 54%]
tests/test_get_random_anime_integration.py::test_get_random_anime_returns_result PASSED [ 57%]
tests/test_get_random_anime_integration.py::test_get_random_anime_returns_none_when_empty PASSED [ 60%]
tests/test_get_random_anime_integration.py::test_get_random_anime_excludes_low_quality PASSED [ 62%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_ok PASSED [ 65%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_no_anime PASSED [ 68%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_error_on_select_failure PASSED [ 71%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_locked PASSED [ 74%]
tests/test_priority_weights.py::test_year_bucket_none_returns_old PASSED [ 77%]
tests/test_priority_weights.py::test_year_bucket_boundaries PASSED [ 80%]
tests/test_priority_weights.py::test_get_penalty_multipliers_balanced_no_penalty PASSED [ 82%]
tests/test_priority_weights.py::test_get_penalty_multipliers_clamped_to_min PASSED [ 85%]
tests/test_priority_weights.py::test_get_penalty_multipliers_no_crash_on_empty PASSED [ 88%]
tests/test_priority_weights.py::test_get_penalty_for_anime_combines_multipliers PASSED [ 91%]
tests/test_publish_integration.py::test_publish_anime_dry_run_does_not_write_db PASSED [ 94%]
tests/test_publish_integration.py::test_publish_anime_creates_published_record PASSED [ 97%]
tests/test_publish_integration.py::test_publish_anime_rollback_on_send_error PASSED [100%]

===== warnings summary =====
tests/test_can_publish.py::test_can_publish_naive_datetime
  C:\projects\anime_channel\tests\test_can_publish.py:53: DeprecationWarning: datetime.datetime.utcnow() is
  deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetim
  s in UTC: datetime.datetime.now(datetime.UTC).
    naive_dt = datetime.utcnow() - timedelta(days=200)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== 35 passed, 1 warning in 0.34s =====
(venv) PS C:\projects\anime_channel>
```

Рисунок 4.4 – Результати виконання автоматизованих тестів у консолі

4.5 Висновки за розділом 4

У четвертому розділі було проведено практичне дослідження розробленого ПЗ, описано умови його розгортання та виконано комплексне тестування. За результатами проведеної роботи можна зробити наступні висновки.

Визначено технічні вимоги до середовища виконання. Завдяки вибору монолітної архітектури та асинхронного підходу до обробки даних, система демонструє високу ефективність і здатна стабільно функціонувати на серверних потужностях з мінімальним обсягом оперативної пам'яті. Вимогу до версії інтерпретатора Python 3.11 обґрунтовано використанням сучасних методів обробки часових зон.

Сформовано поетапну інструкцію з експлуатації, яка забезпечує швидке розгортання проєкту. Вона охоплює процес ізоляції залежностей у віртуальному середовищі, конфігурування системи через змінні оточення у файлі `.env` та правильну послідовність ініціалізації бази даних за допомогою спеціалізованих службових скриптів.

Під час аналізу виконання програми підтверджено повну працездатність обох рівнів інтерфейсу. Адміністративний модуль успішно обробляє команди керування, реалізує пошук та зручну пагінацію результатів. Публічний інтерфейс у каналі забезпечує коректне відображення контенту з автоматичною локалізацією даних та використанням кешованих ідентифікаторів медіафайлів для економії ресурсів.

Шляхом автоматизованого тестування за допомогою фреймворку `pytest` доведено високу надійність програмного забезпечення. Успішне проходження 35 тестових сценаріїв підтвердило коректність математичних розрахунків ваги, стабільність логіки затримок та безпеку транзакцій у базі даних завдяки використанню ізольованих `in-memory` сесій та механізмів відкочування транзакцій.

Результати тестування та дослідної експлуатації підтверджують, що розроблена система повністю відповідає технічному завданню, працює без критичних помилок і готова до впровадження у реальне середовище для автоматизації ведення тематичних Telegram-каналів.

ВИСНОВКИ

З метою автоматизації рутинних процесів контент-менеджменту в роботі вирішено актуальну задачу проектування та розробки програмного забезпечення для агрегації, відбору та публікації аніме-контенту в месенджері Telegram.

У результаті виконання дипломної кваліфікаційної роботи отримано наступні практичні та технічні результати.

Оцінено стан проблеми та проаналізовано існуючі рішення. Аналіз предметної області показав, що сучасні інструменти для ведення Telegram-каналів переважно пропонують або ручний відкладений постинг, або базову рандомізацію. Виявлено ключову проблему: відсутність механізмів відстеження історії публікацій призводить до жанрової одноманітності стрічки. Зроблено висновок про необхідність розробки власного програмного забезпечення з алгоритмами зваженого відбору контенту.

Сформульовано вимоги та спроектовано архітектуру системи. Обґрунтовано вибір мови програмування Python 3.11 або вище та фреймворку python-telegram-bot для забезпечення асинхронної взаємодії. Спроектовано монолітний серверний застосунок із багаторівневою архітектурою та чітким внутрішнім розділенням відповідальностей між компонентами. Для зберігання метаданих розроблено реляційну модель бази даних на основі СУБД SQLite з використанням ORM SQLAlchemy.

Розроблено алгоритм зваженого скорингу контенту. Вирішено задачу автоматичного балансування стрічки публікацій. Створено математичну модель розрахунку ефективної ваги тайтлу, яка використовує логарифмічну нормалізацію показників популярності та систему динамічних штрафних коефіцієнтів. Алгоритм забезпечує підвищений пріоритет для свіжих релізів та накладає часові обмеження на повторні публікації.

Реалізовано дворівневий інтерфейс взаємодії. Спроектовано інтерфейс адміністратора з підтримкою системи команд, гнучкого налаштування

інтервалів та режиму безпечного тестування. Для оптимізації пошуку впроваджено алгоритми нечіткого збігу та посторінкову навігацію в Inline-меню. Публічний інтерфейс оптимізовано шляхом кешування медіафайлів та автоматичної локалізації текстових описів.

Проведено дослідну експлуатацію та тестування системи. За допомогою фреймворку `pytest` виконано автоматизоване модульне та інтеграційне тестування розробленого програмного забезпечення (успішно пройдено 35 тестових сценаріїв). Підтверджено стійкість алгоритмів до граничних значень, надійність ізоляції транзакцій бази даних (з використанням `rollback`-механізмів) та загальну стабільність фонових процесу.

Сформульовано рекомендації щодо використання та перспективи розвитку. Розроблене програмне забезпечення рекомендується до практичного використання контент-менеджерами та адміністраторами тематичних спільнот для автоматизації наповнення Telegram-каналів. Перспективи подальших досліджень у цій галузі включають: розширення системи шляхом інтеграції з API інших соціальних мереж, інтеграцію великих мовних моделей (LLM) для автоматизованої генерації та унікалізації текстових описів, а також проведення рефакторингу коду для оптимізації роботи з часовими зонами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Telegram global reach 2025. *Statista*. URL: <https://www.statista.com/statistics/1343525/telegram-reach-worldwide/> (date of access:15.05.2026).
2. 200,000,000 Monthly Active Users. *Telegram Messenger*. URL: <https://telegram.org/blog/200-million> (date of access:15.05.2026).
3. Shikimori API Documentation. *PublicAPI*. URL: <https://publicapi.dev/shikimori-api> (date of access:15.05.2026).
4. Telegram Bot API. *Telegram*. URL: <https://core.telegram.org/bots/api> (date of access:15.05.2026).
5. Telegram Controller Bot. *TutKit*. URL: <https://www.tutkit.com/en/text-tutorials/3753-instructions-for-setting-up-the-telegram-controller-bot> (date of access:15.05.2026).
6. FleepBot – Telegram Channel Management. URL: <https://fleep.bot/en> (date of access:15.05.2026).
7. TGADM.IN – Система автоматизації публікацій. URL: <https://tgadm.in/> (дата звернення:15.05.2026).
8. History of Python. URL: <https://ir.lib.vntu.edu.ua/bitstream/handle/123456789/10471/461.pdf> (date of access:15.05.2026).
9. Asyncio. *Python Software Foundation*. URL: <https://docs.python.org/3/library/asyncio.html> (date of access:15.05.2026).
10. Python overtakes JavaScript across multiple programming language rankings in 2026. *ByteIota*. URL: <https://byteiota.com/python-overtakes-javascript-2026-developer-surveys/> (date of access:15.05.2026).
11. SQLite Overview. *SQLite*. URL: <https://www.sqlite.org/about.html> (date of access:15.05.2026).
12. SQLAlchemy Documentation. *SQLAlchemy*. URL: <https://docs.sqlalchemy.org/> (date of access:15.05.2026).

13. Python-telegram-bot Documentation. URL: <https://python-telegram-bot.org/> (date of access:15.05.2026).
14. Deployment Types: Long Polling vs Webhooks. *grammy*. URL: <https://grammy.dev/guide/deployment-types> (date of access:15.05.2026).
15. Stop Double-Running Your Python Scripts: A Simple, Zero-Dependency Fix. *Medium*. URL: <https://medium.com/codex/stop-double-running-your-python-scripts-a-simple-zero-dependency-fix-9114ef93c9a9> (date of access:15.05.2026).

ДОДАТОК А
Текст програми

A.1 Текст файла telegram_bot.py

```

import logging
import sys
import asyncio
import os
import time
import random
import traceback

from telegram import InlineKeyboardButton, InlineKeyboardMarkup, Update
from telegram.error import BadRequest, NetworkError
from telegram.ext import (
    ApplicationBuilder,
    CallbackQueryHandler,
    CommandHandler,
    ContextTypes,
)

from app.logging_config import setup_logging
from app.settings import (
    ADMIN_IDS,
    ALLOWED_CHAT_IDS,
    BOT_TOKEN,
)
import app.settings as settings
from app.lockfile import LockAlreadyHeldError, file_lock
from services.post_cycle import run_post_cycle
from services.auto_poster import auto_poster_loop

logger = logging.getLogger(__name__)

# Holds a reference to the auto-poster background task so it can be cancelled on
shutdown.
_auto_poster_task_holder: list[asyncio.Task | None] = [None]

PAGE_SIZE = 5 # Anime entries shown per page in search results

# -----
# Helpers
# -----

def _is_admin(update: Update) -> bool:
    user = update.effective_user

```

```

chat = update.effective_chat
user_id = user.id if user else None
chat_id = chat.id if chat else None
if ADMIN_IDS and (user_id not in ADMIN_IDS):
    logger.warning("Access denied: user_id=%s not in ADMIN_IDS", user_id)
    return False
if ALLOWED_CHAT_IDS and (chat_id not in ALLOWED_CHAT_IDS):
    logger.warning("Access denied: chat_id=%s not in
ALLOWED_CHAT_IDS", chat_id)
    return False
return True

```

```

def _build_search_keyboard(results: list, page: int, session_id: str) ->
InlineKeyboardMarkup:

```

```

"""

```

Build an inline keyboard for one page of search results.

```

results: [(anime_id, title_ru, title_en, year, kind, shikimori_id), ...]
session_id: unique session key embedded in every callback_data string
"""

```

```

max_page = max(0, (len(results) - 1) // PAGE_SIZE)
page = max(0, min(page, max_page))
start = page * PAGE_SIZE
end = start + PAGE_SIZE

```

```

buttons = []
for anime_id, title_ru, title_en, year, kind, shikimori_id in results[start:end]:
    title = title_ru or title_en or "Без названия"
    meta_parts = []
    if kind:
        meta_parts.append(kind.upper())
    if year:
        meta_parts.append(str(year))
    meta = f"({', '.join(meta_parts)})" if meta_parts else ""
    label = f"{title} {meta} • ID {shikimori_id}"
    buttons.append([InlineKeyboardButton(label,
callback_data=f"pick:{anime_id}:{session_id}")])

```

Page indicator button (non-interactive).

```

buttons.append([InlineKeyboardButton(
    f"📄 Стр. {page + 1} / {max_page + 1}",
    callback_data=f"noop:{session_id}",
)])

```

```

nav = []
if page > 0:
    nav.append(InlineKeyboardButton("◀◀ Назад", callback_data=f"page: {page - 1}: {session_id}"))
    if end < len(results):
        nav.append(InlineKeyboardButton("Вперёд ▶▶",
callback_data=f"page: {page + 1}: {session_id}"))
    if nav:
        buttons.append(nav)
    buttons.append([InlineKeyboardButton("✕ Отмена",
callback_data=f"cancel: {session_id}"))]
    return InlineKeyboardMarkup(buttons)

```

```

def _publish_anime_by_db_id(anime_id: int, dry_run: bool) -> str:
    """Fetch anime by internal DB id and publish it. Returns a status string."""
    from db.database import SessionLocal
    from db.models import Anime
    from services.publisher import publish_anime
    session = SessionLocal()
    try:
        anime = session.query(Anime).filter(Anime.id == anime_id).first()
        if not anime:
            return "Аниме не найдено в базе данных."
        title = anime.title_ru or anime.title_en or "Без названия"
        publish_anime(anime, dry_run=dry_run)
        prefix = "[DRY RUN] " if dry_run else ""
        return f"{prefix}Опубликовано: {title}"
    finally:
        session.close()

```

`_SEARCH_TTL = 15 * 60 # Search sessions expire after 15 minutes`

```

def _cleanup_stale_searches(bot_data: dict) -> None:
    """Remove expired search sessions (older than _SEARCH_TTL seconds)."""
    searches = bot_data.get("anime_searches", {})
    now = time.time()
    stale = [k for k, v in searches.items() if now - v.get("created_at", 0) >
_SEARCH_TTL]
    for k in stale:
        del searches[k]

```

```

# -----
# Command handlers
# -----

async def post_now(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return

    dry_run = settings.get_runtime_dry_run()
    await update.effective_chat.send_message(f"Starting post cycle
(dry_run={dry_run})...")
    try:
        loop = asyncio.get_running_loop()
        result = await loop.run_in_executor(None, lambda:
run_post_cycle(dry_run=dry_run))

        if isinstance(result, dict) and result.get("status") == "error":
            await update.effective_chat.send_message(
                "Anime selection failed (see logs/bot.log). Post skipped."
            )
            return

        if dry_run:
            title = result.get("title") if isinstance(result, dict) else None
            lines = ["Post cycle started (dry_run=True)"]
            if title:
                lines.append(f"Would publish: {title}")
            lines.append("DRY RUN: message NOT sent, DB NOT updated")
            await update.effective_chat.send_message("\n".join(lines))

            await update.effective_chat.send_message("Post cycle complete.")
    except Exception as e:
        logger.exception("Error in /post_now: %s", e)
        await update.effective_chat.send_message("Command failed. Check logs for
details.")

async def status(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return

```

```

dry_run = settings.get_runtime_dry_run()
posting_enabled = settings.get_posting_enabled()
min_s, max_s = settings.get_post_interval()
is_posting_now = settings.get_is_posting_now()
text = (
    "Bot is running.\n"
    f"DRY_RUN: {'ON' if dry_run else 'OFF'}\n"
    f"AUTO posting: {'ON' if posting_enabled else 'OFF'}\n"
    f"Interval: {min_s}..{max_s} sec\n"
    f"Posting now: {'YES' if is_posting_now else 'NO'}"
)
await update.effective_chat.send_message(text)

```

```

async def dry_run_on(update: Update, context: ContextTypes.DEFAULT_TYPE)
-> None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return
    settings.set_runtime_dry_run(True)
    logger.info("DRY_RUN enabled via Telegram")
    await update.effective_chat.send_message("DRY RUN: ON")

```

```

async def dry_run_off(update: Update, context: ContextTypes.DEFAULT_TYPE)
-> None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return
    settings.set_runtime_dry_run(False)
    logger.info("DRY_RUN disabled via Telegram")
    await update.effective_chat.send_message("DRY RUN: OFF")

```

```

async def posting_on(update: Update, context: ContextTypes.DEFAULT_TYPE) -
> None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return
    settings.set_posting_enabled(True)
    logger.info("AUTO posting enabled via Telegram")
    await update.effective_chat.send_message("AUTO posting: ON")

```

```

async def posting_off(update: Update, context: ContextTypes.DEFAULT_TYPE) -
> None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return
    settings.set_posting_enabled(False)
    logger.info("AUTO posting disabled via Telegram")
    await update.effective_chat.send_message("AUTO posting: OFF")

async def set_interval(update: Update, context: ContextTypes.DEFAULT_TYPE)
-> None:
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return
    try:
        if not context.args or len(context.args) != 2:
            await update.effective_chat.send_message("Usage: /interval <min_sec>
<max_sec>")
            return
        min_s = int(context.args[0])
        max_s = int(context.args[1])
        settings.set_post_interval(min_s, max_s)
        logger.info("Interval changed via Telegram: %s..%s", min_s, max_s)
        await update.effective_chat.send_message(f"Interval set: {min_s}..{max_s}
sec")
    except Exception:
        logger.exception("Error setting interval")
        await update.effective_chat.send_message("Error: could not set interval.")

async def post_anime(update: Update, context: ContextTypes.DEFAULT_TYPE) -
> None:
    """
    /post_anime <shikimori_id|title>

    Numeric input: direct lookup by shikimori_id, publishes immediately.
    Text input: AND-search across title_ru / title_en:
    0 results → "Nothing found" message
    1 result → publish immediately
    >1 results → inline keyboard with pagination
    """
    if not _is_admin(update):
        await update.effective_chat.send_message("Access denied.")
        return

```

```

if not context.args:
    await update.effective_chat.send_message(
        "Usage:\n"
        " /post_anime 5114      — by shikimori_id\n"
        " /post_anime Naruto    — by title (title_ru / title_en)"
    )
    return

query = " ".join(context.args).strip()
if len(query) > 100:
    await update.effective_chat.send_message("Query too long. Maximum 100
characters.")
    return
dry_run = settings.get_runtime_dry_run()
loop = asyncio.get_running_loop()

# — Lookup by shikimori_id

```

```

if query.lstrip("-").isdigit():
    def _find_by_id() -> str:
        from db.database import SessionLocal
        from db.models import Anime
        session = SessionLocal()
        try:
            row = session.query(Anime.id).filter(
                Anime.shikimori_id == int(query)
            ).first()
            if not row:
                return f"Anime not found: shikimori_id={query}"
            return _publish_anime_by_db_id(row.id, dry_run)
        finally:
            session.close()

    await update.effective_chat.send_message(f"Looking up
shikimori_id={query}...")
    try:
        result = await loop.run_in_executor(None, _find_by_id)
        await update.effective_chat.send_message(result)
    except Exception as e:
        logger.exception("Error in /post_anime by id: %s", e)
        await update.effective_chat.send_message("Publish failed. Check logs for
details.")
    return

```

— AND-search by title

```
def _search_by_title() -> list:
    from db.database import SessionLocal
    from db.models import Anime
    from sqlalchemy import func
    session = SessionLocal()
    try:
        words = query.lower().split()
        q = session.query(
            Anime.id, Anime.title_ru, Anime.title_en,
            Anime.year, Anime.kind, Anime.shikimori_id,
        )
        for word in words:
            pattern = f"%{word}%"
            q = q.filter(
                func.lower(Anime.title_ru).like(pattern)
                | func.lower(Anime.title_en).like(pattern)
            )
        rows = q.order_by(Anime.members.desc(),
            Anime.year.desc()).limit(50).all()
        return [(r.id, r.title_ru, r.title_en, r.year, r.kind, r.shikimori_id) for r in
            rows]
    finally:
        session.close()

def _search_fuzzy() -> list:
    """Fuzzy search via rapidfuzz. Searches top-3000 by popularity with
    normalised strings."""
    from rapidfuzz import fuzz, process
    from db.database import SessionLocal
    from db.models import Anime
    session = SessionLocal()
    try:
        rows = session.query(
            Anime.id, Anime.title_ru, Anime.title_en,
            Anime.year, Anime.kind, Anime.shikimori_id,
        ).order_by(Anime.members.desc()).limit(3000).all()

        keys: list[str] = []
        data: list[tuple] = []
        for r in rows:
            combined = f"{r.title_ru or ''} {r.title_en or ''}".strip().lower()
```

```

        if combined:
            keys.append(combined)
            data.append((r.id, r.title_ru, r.title_en, r.year, r.kind, r.shikimori_id))

        matches = process.extract(
            query.lower().strip(), keys,
            scorer=fuzz.WRatio, limit=50, score_cutoff=65,
        )
        # m[2] is the index into data (stable even when multiple titles share the
        same string)
        return [data[m[2]] for m in matches]
    finally:
        session.close()

    try:
        results = await loop.run_in_executor(None, _search_by_title)
    except Exception as e:
        logger.exception("Error searching anime: %s", e)
        await update.effective_chat.send_message("Search failed. Check logs for
        details.")
        return

    # Fuzzy fallback: only triggered when AND-search returns nothing
    # and at least one query word is longer than 2 characters.
    fuzzy_used = False
    if not results and any(len(w) >= 3 for w in query.strip().split()):
        try:
            results = await loop.run_in_executor(None, _search_fuzzy)
            if results:
                fuzzy_used = True
                logger.info("Fuzzy search for %r: found %d results", query, len(results))
        except Exception as e:
            logger.exception("Error in fuzzy search: %s", e)

    if not results:
        await update.effective_chat.send_message(f"Nothing found for: {query!r}")
        return

    # Single result — publish immediately without showing buttons.
    if len(results) == 1:
        try:
            result = await loop.run_in_executor(
                None, lambda: _publish_anime_by_db_id(results[0][0], dry_run)
            )

```

```

        await update.effective_chat.send_message(result)
    except Exception as e:
        logger.exception("Error publishing single result: %s", e)
        await update.effective_chat.send_message("Publish failed. Check logs for
details.")
    return

```

Multiple results — generate a session_id and send the keyboard in one message.

```

total = len(results)
suffix = " (first 50 shown)" if total == 50 else ""
header = (
    f"🔍 Fuzzy search for {query!r}: {total} {suffix}"
    if fuzzy_used else
    f"🔍 Found: {total} {suffix}"
)
session_id = f"{int(time.time())}{random.randint(1000, 9999)}"
_cleanup_stale_searches(context.bot_data)
context.bot_data.setdefault("anime_searches", {})
context.bot_data["anime_searches"][session_id] = {
    "results": results,
    "page": 0,
    "created_at": time.time(),
}
keyboard = _build_search_keyboard(results, 0, session_id)
sent = await update.effective_chat.send_message(
    f"{header}. Choose an anime:",
    reply_markup=keyboard,
)
context.bot_data["anime_searches"][session_id]["message_id"] =
sent.message_id

```

```

# -----
# Callback handlers (inline buttons for /post_anime)
# -----

```

```

async def _cb_pick(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:

```

```

    """User picks an anime → publish → send result → delete the keyboard
message."""

```

```

    query = update.callback_query

```

```

    if not _is_admin(update):

```

```

        await query.answer("Access denied.", show_alert=True)

```

```

    return
if not query.message:
    await query.answer()
    return
try:
    parts = query.data.split(":") # pick: {anime_id}:{session_id}
    anime_id = int(parts[1])
    session_id = parts[2]
except (IndexError, ValueError):
    await query.answer("Invalid button data", show_alert=True)
    return
await query.answer() # must be called exactly once on the success path
_cleanup_stale_searches(context.bot_data)
search_data = context.bot_data.get("anime_searches", {}).get(session_id)
if search_data and search_data.get("message_id") != query.message.message_id:
    return
dry_run = settings.get_runtime_dry_run()
chat_id = query.message.chat_id
try:
    loop = asyncio.get_running_loop()
    result_text = await loop.run_in_executor(
        None, lambda: _publish_anime_by_db_id(anime_id, dry_run)
    )
except Exception as e:
    logger.exception("Error publishing via pick button: %s", e)
    result_text = "Publish failed. Check logs for details."
context.bot_data.get("anime_searches", {}).pop(session_id, None)
await context.bot.send_message(chat_id, result_text) # send result first
try:
    await query.message.delete() # then remove the keyboard
except Exception:
    pass

```

```

async def _cb_page(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:

```

```

    """Pagination — updates the keyboard in-place without sending a new
message."""

```

```

    query = update.callback_query
    await query.answer()
    if not _is_admin(update):
        return
    if not query.message:
        return
    try:

```

```

    parts    = query.data.split(":") # page:{page}:{session_id}
    page     = int(parts[1])
    session_id = parts[2]
except (IndexError, ValueError):
    await query.answer("Invalid button data", show_alert=True)
    return
_cleanup_stale_searches(context.bot_data)
search_data = context.bot_data.get("anime_searches", {}).get(session_id)
if not search_data:
    try:
        await query.message.delete()
    except Exception:
        pass
    return
if search_data.get("message_id") != query.message.message_id:
    return
results = search_data["results"]
max_page = max(0, (len(results) - 1) // PAGE_SIZE)
page     = max(0, min(page, max_page))
search_data["page"] = page
keyboard = _build_search_keyboard(results, page, session_id)
try:
    await query.edit_message_reply_markup(reply_markup=keyboard)
except BadRequest:
    pass # "Message is not modified" — harmless, ignore

async def _cb_cancel(update: Update, context: ContextTypes.DEFAULT_TYPE) -
> None:
    """Cancel — deletes the keyboard message, sends nothing."""
    query = update.callback_query
    await query.answer()
    if not query.message:
        return
    try:
        parts    = query.data.split(":") # cancel:{session_id}
        session_id = parts[1]
    except IndexError:
        return
    _cleanup_stale_searches(context.bot_data)
    search_data = context.bot_data.get("anime_searches", {}).get(session_id)
    if search_data and search_data.get("message_id") != query.message.message_id:
        return
    context.bot_data.get("anime_searches", {}).pop(session_id, None)
    try:

```

```

        await query.message.delete()
    except Exception:
        pass

```

```

async def _cb_noop(update: Update, context: ContextTypes.DEFAULT_TYPE) ->
None:
    """No-op button (page indicator label)."""
    await update.callback_query.answer()

```

```

# -----
# Entry point
# -----

```

```

def main() -> None:
    setup_logging()

    if not BOT_TOKEN:
        logger.error("BOT_TOKEN is not set in environment or settings")
        sys.exit(1)

    if not ADMIN_IDS:
        logger.warning(
            "⚠ ADMIN_IDS is not set — ALL Telegram users have access to bot "
            "management commands (/post_now, /posting_on, etc.). "
            "Set ADMIN_IDS in .env to restrict access."
        )

    try:
        lock_path = os.path.join(
            os.path.dirname(os.path.dirname(__file__)), "logs", "telegram_bot.lock"
        )

        with file_lock(lock_path):
            async def _post_init(application) -> None:
                _auto_poster_task_holder[0] = asyncio.create_task(auto_poster_loop())

            async def _post_shutdown(application) -> None:
                task = _auto_poster_task_holder[0]
                if task and not task.done():
                    task.cancel()
                try:
                    await task
                except asyncio.CancelledError:

```

```

        pass
        _auto_poster_task_holder[0] = None

    async def _on_error(update: object, context:
ContextTypes.DEFAULT_TYPE) -> None:
        err = getattr(context, "error", None)
        if err is None:
            return
        if isinstance(err, NetworkError):
            logger.warning(
                "Telegram: network/timeout error, will retry (get_updates): %s",
                err,
            )
            return
        logger.error("Telegram handler error: %s", err)
        logger.error(
            "Telegram error traceback:\n%s",
            "".join(traceback.format_exception(err)),
        )

    application = (
        ApplicationBuilder()
        .token(BOT_TOKEN)
        .post_init(_post_init)
        .post_shutdown(_post_shutdown)
        .build()
    )

    application.add_error_handler(_on_error)

    # Command handlers
    application.add_handler(CommandHandler("post_now", post_now))
    application.add_handler(CommandHandler("post_anime", post_anime))
    application.add_handler(CommandHandler("status", status))
    application.add_handler(CommandHandler("dry_run_on", dry_run_on))
    application.add_handler(CommandHandler("dry_run_off", dry_run_off))
    application.add_handler(CommandHandler("posting_on", posting_on))
    application.add_handler(CommandHandler("posting_off", posting_off))
    application.add_handler(CommandHandler("interval", set_interval))

    # Callback handlers for /post_anime inline buttons
    application.add_handler(CallbackQueryHandler(_cb_pick,
pattern=r"^pick:\d+:[\w\~]+)$"))
    application.add_handler(CallbackQueryHandler(_cb_page,
pattern=r"^page:\d+:[\w\~]+)$"))

```

```

        application.add_handler(CallbackQueryHandler(_cb_cancel,
pattern=r"^cancel:[\w\~]+")
        application.add_handler(CallbackQueryHandler(_cb_noop,
pattern=r"^noop:"))

        logger.info("Telegram bot started")
        application.run_polling()
    except LockAlreadyHeldError:
        logger.warning("Telegram bot already running (lock=%s)", lock_path)
        return
    except Exception:
        logger.exception("Fatal error in Telegram bot")
        sys.exit(1)

if __name__ == "__main__":
    main()

```

ДОДАТОК Б
Слайди презентації



Рисунок Б.1 – Слайд 1

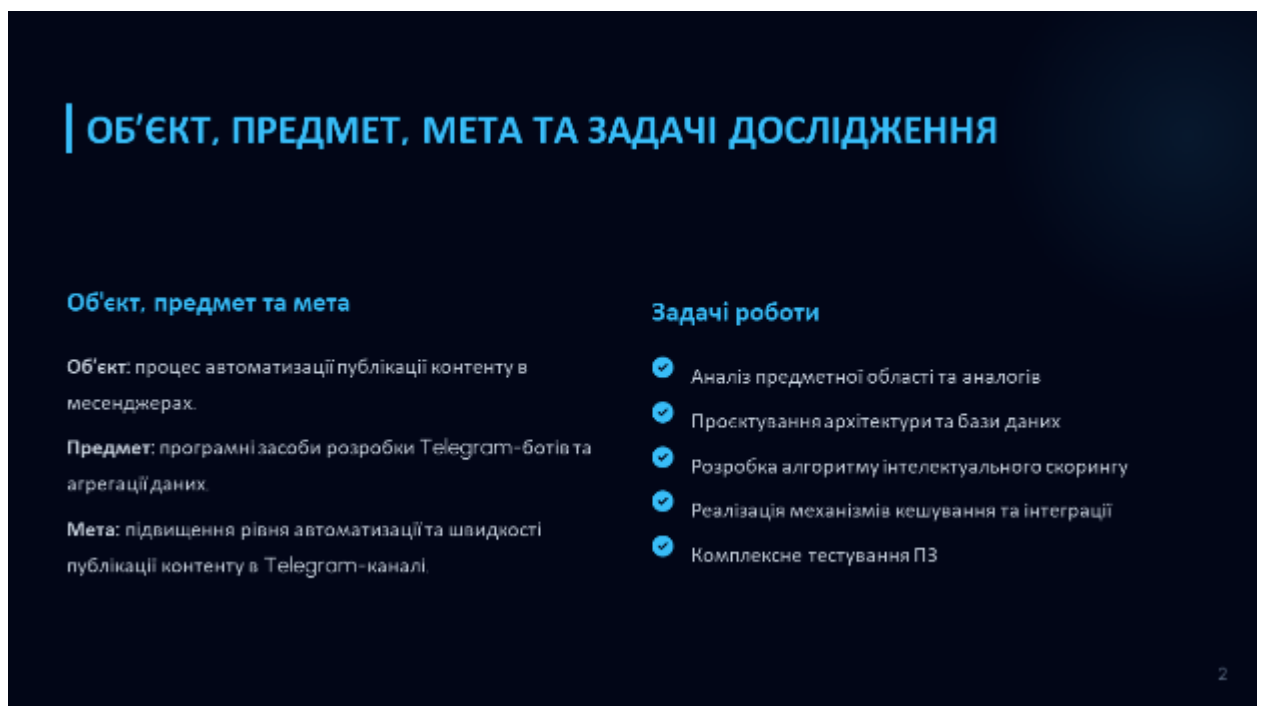


Рисунок Б.2 – Слайд 2

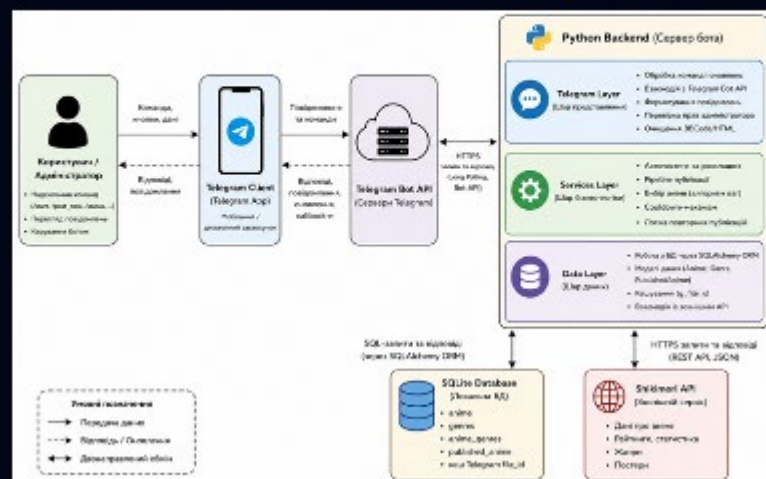
АНАЛІЗ ІСНУЮЧИХ ЗАСОБІВ АВТОМАТИЗАЦІЇ

Можливість	Controller Bot	FleepBot	TGADM.IN	Розроблене ПЗ
Відкладений постинг та базове форматування	+	+	+	+
Генерація текстів за допомогою ШІ	-	+	+	-
Автоматична агрегація точних даних через API	-	-	-	+
Наявність вбудованої реляційної БД	-	-	-	+
Алгоритм оцінювання ваг	-	-	-	+
Глибоке кешування медіафайлів	-	-	-	+

3

Рисунок Б.3 – Слайд 3

АРХІТЕКТУРА ТА ВЗАЄМОДІЯ КОМПОНЕНТІВ



4

Рисунок Б.4 – Слайд 4

```

    erDiagram
        anime_genres ||--o{ anime : "has"
        anime ||--o{ published_anime : "has"
        anime_genres ||--o{ anime : "has"

        anime_genres {
            integer anime_id PK
            integer genre_id FK
        }

        anime {
            integer id PK
            integer shikimori_id
            varchar title_ru NN
            varchar title_en
            varchar title_original
            text description
            varchar status
            integer episodes
            integer episodes_aired
            integer year
            float rating
            varchar image_url
            varchar kind
            integer members
            varchar tg_file_id
        }

        published_anime {
            integer id PK
            integer anime_id FK NN
            varchar status NN
            integer episodes
            datetime published_at NN
        }
  
```

```

graph LR
    Start([Начало]) --> Input[ввести количество слоев нейронной сети]
    Input --> Check1{Проверка на наличие самоколебаний}
    Check1 --> Output1[Вывести количество слоев нейронной сети]
    Output1 --> End1([Конец])
    Check1 -- да --> End1
    Check1 -- нет --> Increase1[Увеличение количества слоев нейронной сети]
    Increase1 --> Check1
  
```

Рисунок Б.6 – Слайд 6

МАТЕМАТИЧНА МОДЕЛЬ СКОРИНГУ КОНТЕНТУ

$$W_{eff} = \max \left(W_{base} \cdot \prod_{i=1}^n k_{penalty_i}, 0.1 \right)$$

Рисунок Б.8 – Слайд 8

ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

Критерій	Python	JavaScript (Node.js)	Java
Швидкість розробки	Висока (+++)	Середня (++)	Низька (+)
Асинхронність	asyncio (+++)	Event Loop (+++)	Потоки (++)
Бібліотеки для ботів	PTB, Aiogram (+++)	Telegraf (++)	TelegramBots (++)
Робота з API/JSON	httpx (+++)	axios (+++)	HttpClient (++)
Поріг входження	Низький (+++)	Середній (++)	Високий (+)

Рисунок Б.9 – Слайд 9

АДМІНІСТРАТИВНИЙ ТА ПУБЛІЧНИЙ ІНТЕРФЕЙС

Bot is running.
CPU_USAGE: 0%
AUTO_REPLY: ON
Internal sec: 2160.3240 sec.
Posting now: NO

Internal sec: 2160.3240 sec.

Found 30. Choose an article

Розкриття нового повідомлення: Смерть і переживання (MO...)

О моєму переживанні в світі: Дітями світу поважаю (T...)

О моєму переживанні в світі: (TV, 2022) - ID 48991

О моєму переживанні в світі: Альє улі (MOME, 2022) - ID 4...

Велика книга! Повітряний Демонів переживаннях як титан...

Стр. 2 / 6

Назад Вперед

Отримати

АНИМЕ ГАЙД | Рекомендації | Новини



Naruto: The Last Movie — Класична картина

2011
Серія: 1
Оцінка: 7.46
Рейтинг: 7.46

Жанр:
Пригодницький, Акційний, Фантастичний

Опис:
Після арешту за пошуки на шляху Деревця Сирого Об'єкта Наруто [NARUTO] закінчені в даний час. Наруто, який...
...історію свого життя, який він використовує, щоб...
...своїм життям. Наруто [NARUTO] використовує свої сили, щоб...
...розкрити правду про свого друга Мун [MUN] і його друзів.

© 2023

Рисунок Б.10 – Слайд 10

ТЕСТУВАННЯ СИСТЕМИ

```

* (env) PS C:\projects\anime_channels\src\scripts\python.exe -m pytest tests/ -v
platform win32 -- Python 3.12.0, pytest-8.0.2, pluggy-1.0.0 -- C:\projects\anime_channels\src\scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\projects\anime_channels
outfiles: pytest-0
plugins: anyio-4.12.1
collected 55 items

tests/test_calculate_weight.py::test_calculate_weight_always_positive PASSED [ 18%]
tests/test_calculate_weight.py::test_calculate_weight_recent_anime_gets_bonus PASSED [ 36%]
tests/test_calculate_weight.py::test_calculate_weight_old_anime_gets_penalty PASSED [ 54%]
tests/test_calculate_weight.py::test_calculate_weight_status_bonus PASSED [ 72%]
tests/test_can_publish.py::test_can_publish_released_anime PASSED [ 90%]
tests/test_can_publish.py::test_can_publish_ongoing_new_episodes PASSED [ 90%]
tests/test_can_publish.py::test_can_publish_already_published_anime PASSED [ 90%]
tests/test_can_publish.py::test_can_publish_unknown_status_anime_is_default PASSED [ 90%]
tests/test_formatter.py::test_clean_test_removes_blank_character_tag PASSED [ 90%]
tests/test_formatter.py::test_clean_test_removes_whitespace PASSED [ 90%]
tests/test_formatter.py::test_clean_test_removes_bare_url PASSED [ 90%]
tests/test_formatter.py::test_clean_test_escapes_string PASSED [ 90%]
tests/test_formatter.py::test_encode_test_short_test_unchanged PASSED [ 90%]
tests/test_formatter.py::test_encode_test_respects_word_boundary PASSED [ 90%]
tests/test_formatter.py::test_format_gives_known_gens_return_testing PASSED [ 90%]
tests/test_formatter.py::test_format_gives_unknown_gens_skipped PASSED [ 90%]
tests/test_formatter.py::test_format_gives_empty_list PASSED [ 90%]

tests/test_get_random_anime_integration.py::test_get_random_anime_returns_result PASSED [ 90%]
tests/test_get_random_anime_integration.py::test_get_random_anime_returns_none_when_empty PASSED [ 90%]
tests/test_get_random_anime_integration.py::test_get_random_anime_excludes_low_quality PASSED [ 90%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_ok PASSED [ 90%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_no_anime PASSED [ 90%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_error_on_select_failure PASSED [ 90%]
tests/test_post_cycle_integration.py::test_run_post_cycle_returns_locked PASSED [ 90%]
tests/test_priority_weights.py::test_year_bucket_returns_old PASSED [ 90%]
tests/test_priority_weights.py::test_year_bucket_returns_new PASSED [ 90%]
tests/test_priority_weights.py::test_get_penalty_multipliers_balanced_to_penalty PASSED [ 90%]
tests/test_priority_weights.py::test_get_penalty_multipliers_clamped_to_min PASSED [ 90%]
tests/test_priority_weights.py::test_get_penalty_multipliers_increment_every_season PASSED [ 90%]
tests/test_priority_weights.py::test_get_penalty_for_anime_calculates_multipliers PASSED [ 90%]
tests/test_public_integration.py::test_publish_anime_dry_run_does_not_write_db PASSED [ 90%]
tests/test_public_integration.py::test_publish_anime_creates_published_record PASSED [ 90%]
tests/test_public_integration.py::test_publish_anime_rollback_on_post_error PASSED [ 90%]

----- Settings Summary -----
tests/test_can_publish.py::test_can_publish_anime_datetime
  C:\projects\anime_channels\tests\test_can_publish.py:35: DeprecationWarning: datetime.datetime.utcnow() is
  deprecated and scheduled for removal in a future version. Use timezone-aware objects to represent datetimes
  in UTC: datetime.datetime.now(datetime.UTC).
    raise dt = datetime.utcnow() + timedelta(days=30)
  -- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
  0 passed, 1 warning in 0.45s
  
```

11

Рисунок Б.11 – Слайд 11

ВИСНОВКИ

- ✓ Спроєктовано та реалізовано автономну систему автоматизованого керування Telegram-контентом.
- ✓ Впроваджено математичний алгоритм скорингу, що забезпечує жанрову різноманітність та пріоритетизацію актуальних релізів.
- ✓ Реалізовано систему кешування медіафайлів (file_id), що дозволило значно скоротити мережевий трафік.
- ✓ Забезпечено надійність ПЗ через покриття тестами ключових компонентів (35 тестів, 100% успіху).
- ✓ Практична цінність полягає у повній автоматизації рутинної роботи адміністратора тематичних каналів.

12

Рисунок Б.12 – Слайд 12