**Міністерство освіти і науки України**
**Тернопільський національний технічний університет імені Івана Пулюя**

**Комп'ютерно-інформаційних систем і програмної інженерії**
(повна назва факультету )
**Програмної інженерії**
(повна назва кафедри)

# КВАЛІФІКАЦІЙНА РОБОТА

**на здобуття освітнього ступеня**

**бакалавр**
(назва освітнього ступеня)

**на тему: Розробка програмного забезпечення інтерактивного чат-бота для організації кіберспортивних матчів**

Виконав: студент IV курсу, групи СП-42
спеціальності 121 – Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) Микитюк Н.В (прізвище та ініціали)

Керівник (підпис) Бревус В.М. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю.М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М.Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2026

Міністерство освіти і науки України
**Тернопільський національний технічний університет імені Івана Пулюя**

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ проф. Петрик М.Р.
(підпис) (прізвище та ініціали)
« 6 » квітня 2026 р.

# ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Микитюк Назар Вікторович

1. Тема роботи Розробка програмного заберзпечення інтерактивного чат-бота для організації кіберспортивних матчів

Керівник роботи Бревус В.М. к.т.н
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом по університету від « 06 » квітня 2026 року №

2. Термін подання студентом роботи 22.06.2026

3. Вихідні дані до роботи технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз предметної області, огляд існуючих рішень та вибір технологій.
2. Проєктування архітектури та алгоритмів чат-бота.
3. Програмна реалізація та тестування чат-бота.
4. Безпека життєдіяльності, основи охорони праці.
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема роботи. 2. Актуальність, мета, задачі дослідження
3. Існуючі технології реалізації подібних систем.
4. Функціональні та нефункціональні вимоги .5. Загальна архітектура системи.
6. Варіанти використання. 7. Компоненти програми для налаштування параметрів системи.
8. Програмні засоби та технології. 9. Інтерфейси реалізації застосунку.
10. Тестування. 11. Висновки по роботі. 12. Слайди презентації.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Безпека життєдіяльності, основи охорони праці | | | |
| Нормоконтроль | *Стоянов Ю.М. к.т.н., доц. каф ПІ* | | |

7. Дата видачі завдання 6 квітня 2026 р.

**КАЛЕНДАРНИЙ ПЛАН**

| № з/п | Назва етапів кваліфікаційної роботи | Термін виконання етапів роботи | Примітка |
|---|---|---|---|
| 1. | *Розробка технічного завдання* | *6.04 – 12.04* | |
| 2. | *Робота над першим розділом «Аналіз предметної області, огляд існуючих рішень та вибір технологій»* | *13.04 – 26.04* | *Виконано* |
| 3. | *Робота над другим розділом «Проектування архітектури та алгоритмів чат-бота»* | *27.04 – 03.05* | *Виконано* |
| 4. | *Робота над третім розділом «Програмна реалізація та тестування чат-бота»* | *04.05 – 17.05* | *Виконано* |
| 5. | *Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»* | *18.05 – 24.05* | *Виконано* |
| 6. | *Оформлення пояснювальної записки і графічного матеріалу* | *25.05 – 7.06* | *Виконано* |
| 7. | *Перевірка на академічний плагіат, перевірка керівником та консультантами* | *8.06 – 14.06* | *Виконано* |
| 8. | *Попередній захист кваліфікаційної роботи бакалавра* | *15.06 – 21.06* | |
| 9. | *Захист кваліфікаційної роботи бакалавра* | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Студент _____ Микитюк Н.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бревус В.М.
(підпис) (прізвище та ініціали)

# АНОТАЦІЯ

Розробка програмного забезпечення інтерактивного чат-бота для організації кіберспортивних матчів // Кваліфікаційна робота освітнього рівня «Бакалавр» // Микитюк Назар Вікторович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП – 42 // Тернопіль, 2026 // С. – 72, рис. – 31, табл. – 0, кресл. – 0, додат. – 2, бібліогр. – 21.

Ключові слова: розробка чат-бота, кіберспорт, Discord API, асинхронне програмування, Python, PostgreSQL, рейтингова модель TrueSkill.

Першочерговою метою цієї кваліфікаційної роботи є дослідження процесів аналізу, проєктування, розробки та тестування інтерактивного чат-бота для організації та автоматизації кіберспортивних матчів.

В першому розділу приділено увагу дослідженню предметної області та аналізу популярних додатків до неї. Розглянуто вибір технологій розробки та проведено аналіз вимог.

Другий розділ присвячено опису проєктування архітектури чат-бота, моделюванню бази даних та математичному обґрунтуванню алгоритмів підбору і рейтингу гравців.

Третій розділ зосереджено на програмній реалізації модулів чат-бота та автоматизованому тестуванні його функціоналу.

Четвертий розділ присвячено питанням охорони праці та безпеки в надзвичайних ситуаціях.

# ABSTRACT

Development of interactive chatbot software for organizing esports matches // Qualification work of the "Bachelor" educational level // Mykytiuk Nazar Viktorovych // Ternopil National Technical University named after Ivan Puluj, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, Group SP – 42 // Ternopil, 2026 // P. – 72, figs. – 31, tables – 0, draw. – 0, append. – 2, bibliogr. – 21.

Keywords: chatbot development, esports, Discord API, asynchronous programming, Python, PostgreSQL, TrueSkill rating model.

The primary goal of this qualification work is to study the processes of analysis, design, development, and testing of an interactive chatbot for organizing and automating esports matches.

The first chapter focuses on the study of the subject area and the analysis of popular analogous platforms. The choice of development technologies is considered and the analysis of requirements is carried out.

The second chapter is devoted to the description of the chatbot architecture design, database modeling, and mathematical justification of matchmaking and rating algorithms.

The third chapter focuses on the software implementation of chatbot modules and automated testing of its functionality.

The fourth chapter is dedicated to the issues of occupational safety and emergency management.

# ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

Backend – серверна частина програмної системи, яка відповідає за обробку команд, бізнес-логіку, роботу з базою даних та алгоритмами.

Discord – комунікаційна платформа, яка використовується для створення серверів, каналів, текстового та голосового спілкування користувачів.

Discord API – програмний інтерфейс Discord, який дозволяє створювати ботів, обробляти команди, повідомлення та події серверу.

Discord-бот – програмний застосунок, який працює в середовищі Discord і автоматично виконує визначені дії у відповідь на команди користувачів або події.

Docker – платформа контейнеризації, що використовується для ізольованого запуску програмного забезпечення та його залежностей.

Docker Compose – інструмент для опису та запуску кількох пов’язаних контейнерів, наприклад контейнера бота та контейнера бази даних.

Embed – структуроване повідомлення Discord, яке дозволяє відображати інформацію у вигляді оформленого блоку з полями, заголовком і описом.

Git – система контролю версій, яка використовується для збереження історії змін у програмному коді.

In-house матч – закритий внутрішній матч, який організовується всередині окремої ігрової спільноти або Discord-серверу.

JSON – JavaScript Object Notation, текстовий формат обміну та збереження структурованих даних.

MMR – Matchmaking Rating, числовий показник сили гравця, який використовується для балансування команд.

ORM – Object-Relational Mapping, підхід до роботи з реляційною базою даних через об’єктні моделі мови програмування.

PostgreSQL – об’єктно-реляційна система керування базами даних, яка використовується для збереження даних чат-бота.

Python – мова програмування високого рівня, яка використовується для реалізації програмної логіки чат-бота.

Ready-check – етап підтвердження готовності гравців до початку матчу після формування складів команд.

Slash-команда – команда Discord, яка вводиться через символ “/” і має вбудовані підказки параметрів для користувача.

SQL – Structured Query Language, мова структурованих запитів для роботи з реляційними базами даних.

SQLAlchemy – бібліотека Python для роботи з базами даних, зокрема через ORM-моделі.

TrueSkill – рейтингова модель для оцінювання навичок гравців у командних іграх, яка враховує оцінку сили гравця та невизначеність цієї оцінки.

Unit-тест – автоматизований тест, який перевіряє окремий компонент або функцію програмного забезпечення.

БД – база даних, організоване сховище інформації, що використовується програмною системою.

Кіберспорт – форма змагальної діяльності, у якій учасники змагаються у відеоіграх.

Матчмейкінг – процес автоматичного підбору гравців і формування команд для майбутнього матчу.

ПЗ – програмне забезпечення.

Рольова черга – черга гравців, у якій кожен учасник реєструється відповідно до вибраної ігрової ролі.

СУБД – система управління базами даних.

Чат-бот – програмний агент, який взаємодіє з користувачами через текстові команди та повідомлення.

**ЗМІСТ**

# ВСТУП

Кіберспорт сьогодні є масовим явищем сучасної цифрової культури, а навколо популярних командних дисциплін формуються локальні ігрові спільноти, учасники яких регулярно проводять закриті внутрішні (in-house) матчі. Найпоширенішим середовищем для координації таких спільнот є платформа Discord, проте організація матчів у ній здебільшого виконується вручну.

Ручний збір гравців, розподіл їх за ролями та формування команд є тривалим і суб'єктивним процесом, що нерідко призводить до дисбалансу складів і зниження якості ігрового процесу. Тому актуальною є розробка програмного засобу, який автоматизує рольові черги, формування збалансованих команд та ведення внутрішнього рейтингу безпосередньо в середовищі Discord.

Метою кваліфікаційної роботи є підвищення зручності та об'єктивності організації in-house матчів у локальних кіберспортивних спільнотах шляхом розробки інтерактивного Discord-чат-бота, що автоматизує збір гравців, формування збалансованих команд і ведення рейтингу учасників.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

1) проаналізувати предметну область організації in-house матчів та наявні технічні рішення;

2) сформувати функціональні та нефункціональні вимоги до програмного забезпечення;

3) спроєктувати архітектуру системи та структуру бази даних;

4) розробити алгоритм формування й балансування команд на основі рейтингової моделі TrueSkill;

5) реалізувати основні модулі чат-бота та взаємодію з користувачами через Slash-команди;

6) виконати тестування розробленого програмного забезпечення.

Об'єкт дослідження – процес організації та проведення кіберспортивних in-house матчів у локальних ігрових спільнотах.

Предмет дослідження – методи, алгоритми та програмні засоби автоматизації збору гравців, балансування команд і ведення рейтингу на платформі Discord.

Методи дослідження – методи об'єктно-орієнтованого та модульного проєктування програмного забезпечення; рейтингова модель TrueSkill і методи

теорії ймовірностей для оцінювання сили гравців; методи асинхронного програмування; методи ручного та автоматизованого тестування.

Практичне значення одержаних результатів полягає в тому, що розроблений чат-бот може бути розгорнутий на Discord-сервері ігрової спільноти та використаний для автоматизації організації in-house матчів, забезпечуючи справедливий баланс команд і прозору рейтингову систему. Систему побудовано за модульним принципом, що дає змогу розширювати її новими ігровими режимами та додатковою статистикою.

Апробація результатів роботи. Працездатність розробленого програмного забезпечення підтверджено ручним і автоматизованим тестуванням основних сценаріїв роботи чат-бота: формування рольових черг, балансування команд та оновлення рейтингу після завершення матчу

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ВИБІР ТЕХНОЛОГІЙ

У цьому розділі розглянуто предметну область організації кіберспортивних in-house матчів, визначено основні особливості такого формату та потреби локальних ігрових спільнот. Також проведено огляд існуючих технічних рішень, які можуть використовуватися для організації кіберспортивних матчів, та визначено їх переваги й обмеження.

Окрему увагу приділено вибору технологій для реалізації інтерактивного Discord-чат-бота. На основі аналізу предметної області та аналогів сформовано функціональні й нефункціональні вимоги до програмного забезпечення.

## 1.1 Особливості та правила організації in-house матчів у кіберспорті

Кіберспорт зародився з локальних турнірів, де комп’ютерні ентузіасти об’єднувалися у спільноти для спільних змагань. Сьогодні фундаментом кіберспортивної екосистеми залишаються ігрові платформи та онлайн-спільноти. Гравці у таких спільнотах часто грають in-house матчі – закриті внутрішні тренувальні ігри, де змагаються учасники одного серверу, наприклад у Discord.

Правила проведення таких матчів засновані на змагальних механіках популярних командних дисциплін, зокрема League of Legends у форматі 5 на 5 гравців. В in-house іграх ключову роль відіграє процес збору учасників та балансування сил. Для цього може використовуватися система рольових черг, де гравці реєструються на конкретні ігрові позиції – одну або декілька з п’яти ролей у League of Legends. Щойно розпочинається гра, система автоматично видаляє залучених гравців з усіх інших черг, запобігаючи конфліктам.

Для математичного розрахунку сили гравців та їх автоматичного балансування у команди використовується система рейтингу, заснована на алгоритмі Microsoft TrueSkill. На відміну від класичної системи Elo, TrueSkill

краще підходить для командних ігор, оскільки враховує не лише результат матчу, а й невизначеність оцінки навички гравця. Кожен гравець отримує незалежний рейтинг для кожної з ролей, який є спільним у межах усього серверу. Для кожного гравця система зберігає параметри TrueSkill: mu, що відображає оцінку навички, та sigma, що відображає невизначеність цієї оцінки. Для нових учасників використовуються початкові значення рейтингу, після чого вони змінюються залежно від результатів зіграних in-house матчів. Як тільки в черзі набирається достатня кількість гравців для формування двох повноцінних команд по 5 ролей, алгоритм автоматично розподіляє учасників так, щоб сили обох команд були максимально рівними [9].

Проводить та контролює такі матчі адміністратор спільноти за допомогою спеціального набору команд. Адміністратор має можливість керувати чергами, примусово видаляти конкретних гравців, повністю очищати чергу в каналі, скасовувати матчі або фіксувати перемогу певної команди без необхідності додаткової валідації від учасників. Це мінімізує вплив людського фактора і робить процес проведення in-house ігор швидким, контрольованим та прозорим.

## 1.2 Аналіз існуючих технічних рішень для організації кіберспортивних матчів

Для організації кіберспортивних матчів сьогодні використовуються різні технічні рішення: платформи матчмейкінгу, сервіси для проведення змагань, рейтингові системи та спеціалізовані інструменти для ігрових спільнот. Вони дозволяють координувати гравців, створювати матчі, вести статистику та формувати змагальне середовище. Проте більшість таких рішень орієнтована на публічні матчі або масштабні кіберспортивні події, а не на невеликі закриті in-house спільноти.

У межах роботи розглянуто два популярні рішення, які частково вирішують задачу організації кіберспортивних матчів: FACEIT та InhouseQueue.

### 1.2.1 FACEIT

FACEIT (рис. 1.1) є однією з найвідоміших кіберспортивних платформ для змагального матчмейкінгу. Сервіс дозволяє гравцям брати участь у матчах, підвищувати рейтинг, створювати команди та змагатися з іншими учасниками в межах підтримуваних ігрових дисциплін.

Перевагою FACEIT є готова інфраструктура для змагального середовища. Платформа має власну рейтингову систему, статистику гравців, механізми підбору суперників та засоби організації матчів. Це робить її зручною для гравців, які хочуть брати участь у публічних рейтингових іграх або змаганнях із великою кількістю учасників.

Однак для невеликих in-house спільнот FACEIT не завжди є оптимальним рішенням. Використання платформи вимагає переходу за межі Discord, окремої реєстрації користувачів та адаптації процесу гри до правил зовнішнього сервісу. Крім того, така платформа не забезпечує повної гнучкості для локального Discord-серверу, де адміністратор може хотіти самостійно керувати чергами, ролями, балансуванням команд та внутрішнім рейтингом учасників.

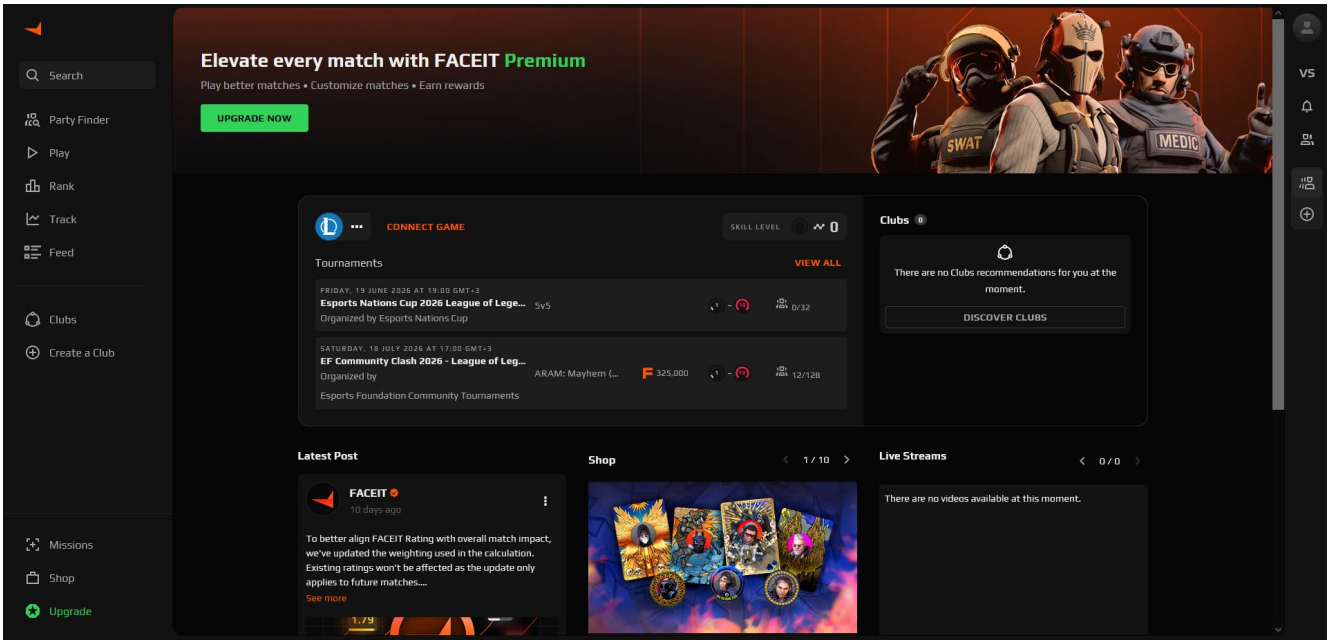



Рисунок 1.1 – Головна сторінка сайту FACEIT

### 1.2.2 InHouseQueue

InHouseQueue – це Discord-бот, призначений для організації внутрішніх custom-ігор на сервері. Він дозволяє створювати черги гравців, запускати матчі, формувати команди та вести статистику в межах Discord-спільноти.

Перевагою InHouseQueue є те, що він орієнтований саме на in-house формат. Бот підтримує різні командні ігри, зокрема League of Legends, Valorant, Overwatch та Dota 2. Для League of Legends передбачено формат 5 на 5 з ролями Top, Jungle, Mid, ADC та Support (рис. 1.2). Також бот підтримує налаштування розміру команд, створення каналів для матчів, ведення історії ігор, таблиці лідерів, ready-up етап та адміністративні команди.

InHouseQueue має широкий набір додаткових функцій: систему сезонів, captain queue, MMR decay, блокування гравців, вибір мап, журнал дій адміністраторів, кастомні таблиці лідерів і webhook-інтеграції. Це робить його потужним готовим рішенням для спільнот, які хочуть швидко організувати in-house матчі без розробки власного програмного забезпечення.

Недоліком такого рішення є обмежена гнучкість для специфічних вимог окремого проєкту. Готовий бот працює за наперед визначеною логікою, тому його складно адаптувати під власну структуру бази даних, власний алгоритм балансування, окремі правила рейтингу або специфічну модель роботи з ролями. Крім того, використання стороннього бота означає залежність від зовнішнього сервісу, його оновлень, обмежень і доступності.

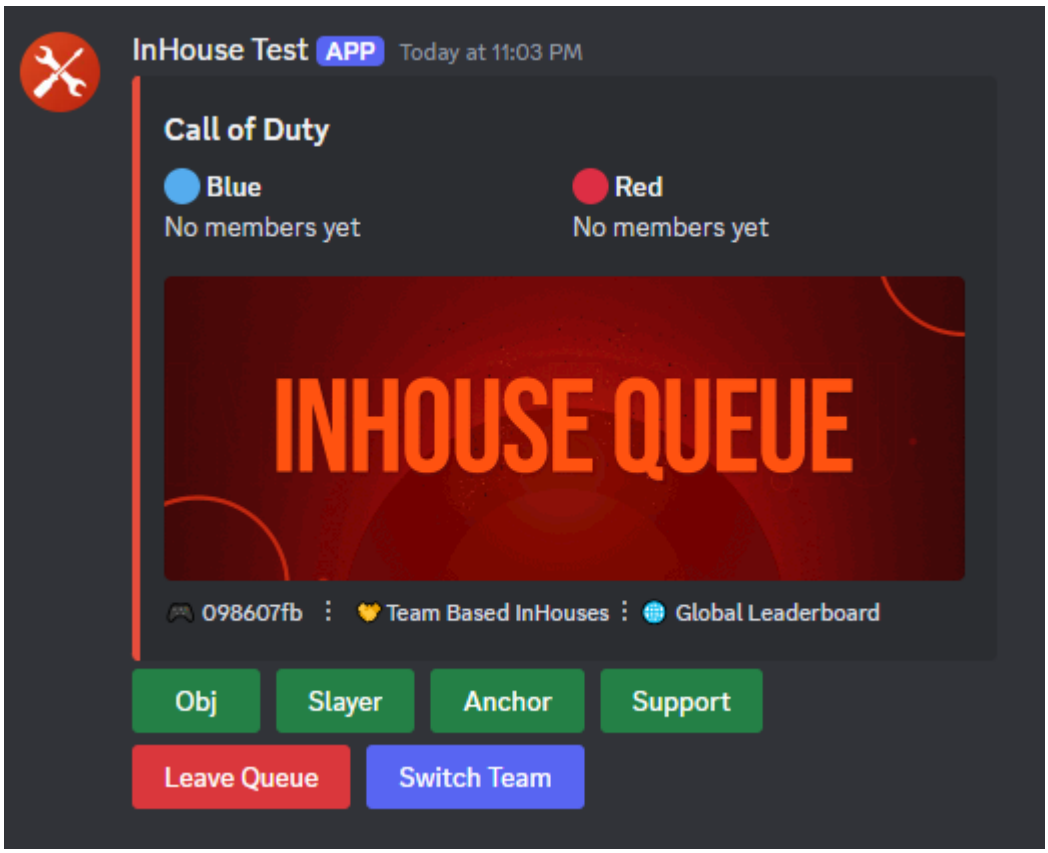



Рисунок 1.2 – Приклад меню лоббі InHouseQueue

Проведений аналіз показує, що InHouseQueue є найближчим аналогом розроблюваного програмного забезпечення, оскільки також реалізований у вигляді Discord-бота для організації in-house матчів. Він має широкий набір готових функцій, але не завжди дозволяє повністю адаптувати логіку роботи під конкретні вимоги окремої спільноти [11].

FACEIT, у свою чергу, є потужною платформою для змагального матчмейкінгу, однак вона орієнтована переважно на публічні ігри та широку аудиторію. Для невеликих Discord-спільнот така система може бути надлишковою.

Тому доцільним є створення власного інтерактивного чат-бота, який працюватиме безпосередньо в межах Discord-серверу та автоматизуватиме основні процеси організації in-house матчів: запис гравців у рольові черги, формування збалансованих команд, керування матчем, збереження історії ігор та оновлення рейтингу учасників.

## 1.3 Вибір технологій розробки

Для реалізації інтерактивного чат-бота для організації кіберспортивних матчів було обрано стек технологій, який забезпечує асинхронну обробку подій, стабільну роботу з базою даних, зручну інтеграцію з Discord та можливість швидкого розгортання системи. Основними технологіями проєкту є Python, бібліотека discord.py, база даних PostgreSQL, ORM SQLAlchemy та інструменти контейнеризації Docker і Docker Compose.

### 1.3.1 Python

Python є однією з найпопулярніших мов програмування для розробки серверних застосунків, автоматизованих систем і чат-ботів. Вона має простий синтаксис, велику кількість бібліотек та підтримує асинхронне програмування, що є важливим для обробки подій у реальному часі [2].

У межах даного проєкту Python використовується як основна мова розробки програмної логіки бота. З її допомогою реалізовано обробку команд користувачів, роботу з чергами, формування матчів, взаємодію з базою даних та оновлення рейтингу гравців.

Важливою перевагою Python є підтримка асинхронної моделі виконання через механізми async/await. Це дозволяє ефективно обробляти декілька подій одночасно, не блокуючи роботу бота під час звернення до Discord API або бази даних. Такий підхід добре підходить для чат-ботів, які повинні швидко реагувати на дії багатьох користувачів [7].

### 1.3.2 Discord.py та Slash-команди

Для інтеграції з платформою Discord використовується бібліотека discord.py. Вона надає засоби для створення Discord-ботів, обробки повідомлень, команд, подій сервера та взаємодії з користувачами [8].

У проєкті discord.py використовується для реалізації основної взаємодії між користувачем і системою. За допомогою цієї бібліотеки бот може приймати команди, додавати гравців у черги, створювати повідомлення про матчі, обробляти адміністративні дії та виводити результати гри.

Команди бота є основним способом взаємодії користувача з програмною системою. Через них гравець може приєднатися до черги, вийти з неї, переглянути історію матчів або зміну рейтингу. Адміністратор, у свою чергу, може очищати чергу, скасовувати матчі та фіксувати результат гри. Такий формат взаємодії є зручним для Discord-спільнот, оскільки не потребує окремого веб-інтерфейсу.

### 1.3.3 PostgreSQL та SQLAlchemy

Для збереження даних у проєкті використовується реляційна база даних PostgreSQL. Вона підходить для зберігання структурованої інформації про користувачів, черги, матчі, рейтинги, ролі гравців та історію ігор [12].

PostgreSQL забезпечує надійне збереження даних і дозволяє ефективно працювати зі зв’язаними таблицями. Це важливо для даного проєкту, оскільки система повинна зберігати не лише окремі профілі гравців, а й історію матчів, результати команд, рейтингові зміни та статистику за ролями.

Для взаємодії з базою даних використовується SQLAlchemy. Ця бібліотека дозволяє описувати структуру таблиць у вигляді моделей Python та виконувати операції з базою даних без написання великої кількості SQL-запитів вручну. У проєкті використовується асинхронний підхід до роботи з базою даних, що дозволяє не блокувати виконання бота під час зчитування або запису інформації [3].

### 1.3.4 Docker та Docker Compose

Для розгортання програмного забезпечення використовується Docker. Цей інструмент дозволяє запускати застосунок у контейнері з усіма необхідними залежностями, що спрощує налаштування середовища та зменшує ризик помилок під час запуску на іншому комп’ютері або сервері [5].

Docker Compose використовується для одночасного запуску кількох компонентів системи, зокрема самого Discord-бота та бази даних PostgreSQL. Це дозволяє описати конфігурацію всього проєкту в одному файлі та швидко запустити систему однією командою [6].

Використання контейнеризації робить проєкт зручним для тестування, перенесення та подальшого розгортання. Такий підхід відповідає сучасним

практикам розробки серверних застосунків і дозволяє підтримувати стабільну роботу програмного забезпечення в різних середовищах.

### 1.4 Визначення та аналіз вимог

Для ефективної розробки інтерактивного чат-бота для організації кіберспортивних матчів необхідно встановити чіткі функціональні та нефункціональні вимоги. Вони формуються на основі аналізу предметної області, потреб локальних in-house спільнот та особливостей роботи Discord-ботів [1].

Функціональні вимоги визначають основні можливості програмного забезпечення, а нефункціональні – технічні характеристики системи, зокрема швидкодію, стабільність, зручність та можливість подальшого розширення.

Функціональні вимоги:

1. Управління чергами гравців:
– можливість додавання гравця до черги через Slash-команду;
– можливість виходу з черги;
– підтримка рольових черг для позицій Top, Jungle, Mid, ADC та Support;
– автоматичне видалення гравця з інших черг після створення матчу.
2. Формування in-house матчу:
– автоматична перевірка достатньої кількості гравців у черзі;
– формування двох команд у форматі 5 на 5;
– балансування команд з урахуванням рейтингу гравців;
– створення повідомлення з інформацією про склад команд.
3. Рейтингова система:
– збереження параметрів TrueSkill для кожного гравця та ролі;
– оновлення внутрішнього рейтингу після завершення матчу;
– можливість перегляду статистики гравця.

4. Адміністративне керування:
– очищення черги адміністратором;
– примусове видалення гравця з черги;
– скасування активного матчу;
– фіксація перемоги однієї з команд.
5. Збереження даних:
– збереження інформації про гравців у базі даних PostgreSQL;
– збереження історії матчів;
– збереження результатів і рейтингових змін;
– можливість відновлення даних після перезапуску системи.

Нефункціональні вимоги:

1. Продуктивність:
– бот повинен швидко реагувати на команди користувачів;
– система має коректно обробляти одночасні дії кількох гравців;
– звернення до бази даних не повинні блокувати роботу бота.
2. Зручність використання:
– взаємодія з ботом повинна здійснюватися через зрозумілі Slash-команди;
– повідомлення бота мають бути короткими та інформативними;
– користувач повинен легко розуміти стан черги, матчу та власного рейтингу.
3. Надійність:
– бот повинен стабільно працювати під час активного використання;
– система має коректно обробляти помилки Discord API та бази даних;
– дані про матчі та рейтинги не повинні втрачатися після перезапуску.
4. Масштабованість:
– архітектура повинна дозволяти додавання нових ігрових режимів;
– система має підтримувати розширення списку команд;
– база даних повинна дозволяти зберігати більшу кількість матчів і користувачів.

5. Розгортання:

– програмне забезпечення повинно запускатися у Docker-контейнері;
– база даних PostgreSQL має запускатися разом із ботом через Docker Compose;
– налаштування системи повинні зберігатися в конфігураційних змінних.

Визначення функціональних і нефункціональних вимог є важливим етапом розробки чат-бота для організації кіберспортивних матчів. Функціональні вимоги описують основні можливості системи: роботу з чергами, формування матчів, ведення рейтингу, адміністрування та збереження даних. Нефункціональні вимоги забезпечують стабільність, швидкодію, зручність використання та можливість подальшого розвитку проєкту.

Основна проблема, яку вирішує дана дипломна робота, полягає в автоматизації організації in-house матчів у межах локальної Discord-спільноти. Розроблений чат-бот дозволить зменшити кількість ручних дій з боку адміністраторів, пришвидшити процес збору гравців, забезпечити баланс команд та вести прозору рейтингову систему для учасників.

# 2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА АЛГОРИТМІВ ЧАТ-БОТА

У цьому розділі розглянуто процес проєктування інтерактивного чат-бота для організації кіберспортивних матчів. Основну увагу приділено визначенню архітектури системи, структурі бази даних, алгоритму формування та балансування команд, а також організації програмних модулів.

Проєктування є важливим етапом розробки програмного забезпечення, оскільки дозволяє визначити склад основних компонентів системи, їх взаємозв’язки та принципи обміну даними. На основі цього надалі виконується програмна реалізація чат-бота та перевірка його функціональності.

## 2.1 Архітектурне проєктування системи

Для розгортання системи використовується Docker та Docker Compose. Контейнеризація дозволяє запускати Discord-бота разом із базою даних у підготовленому середовищі, що спрощує встановлення, тестування та подальше перенесення системи на сервер. Проєктування архітектури є важливим етапом розробки інтерактивного чат-бота для організації кіберспортивних матчів. На цьому етапі визначається загальна структура програмного забезпечення, основні компоненти системи, принципи їх взаємодії та сценарії використання.

Розроблювана система реалізується у вигляді Discord-бота, який працює всередині серверу ігрової спільноти. Discord використовується як основний користувацький інтерфейс, через який гравці та адміністратори взаємодіють із системою за допомогою Slash-команд. Такий підхід дозволяє не створювати окремий веб-застосунок, а виконувати всі основні дії безпосередньо в середовищі, яке вже використовується гравцями для комунікації.

Загальна архітектура системи складається з чотирьох основних частин: користувачів Discord, Discord-серверу, backend-частини чат-бота та бази даних PostgreSQL. Backend-частина реалізована мовою Python із використанням бібліотеки discord.py. Вона приймає команди від користувачів, виконує основну

логіку роботи системи та взаємодіє з базою даних. Спрощену архітектуру системи наведено на (рис 2.1)

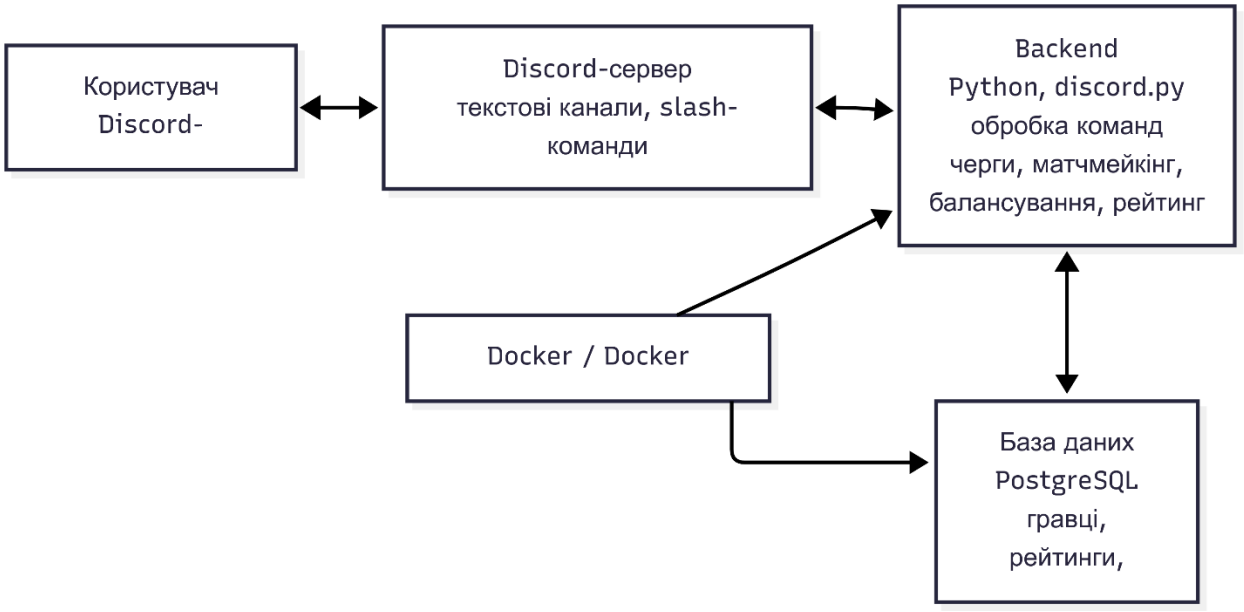



Рисунок 2.1 – Архітектура інтерактивного чат-бота для організації кіберспортивних матчів

Основна логіка чат-бота охоплює обробку команд, керування чергами, формування матчів, балансування команд і оновлення рейтингу. Гравці можуть приєднуватися до черги за вибраною роллю, виходити з неї, підтверджувати готовність до гри та переглядати власну статистику. Адміністратор має розширені можливості для керування процесом: очищення черги, видалення гравця, скасування матчу та фіксацію переможця.

Після набору достатньої кількості учасників система переходить до формування in-house матчу. Для League of Legends підтримується формат 5 на 5 з ролями Top, Jungle, Mid, ADC та Support. Алгоритм підбирає гравців так, щоб у кожній команді був повний набір ролей, а різниця між очікуваною силою команд була мінімальною.

Для балансування використовується рейтингова модель TrueSkill. Для кожного гравця зберігаються параметри trueskill_mu та trueskill_sigma, які характеризують оцінку навички та рівень невизначеності цієї оцінки. На основі цих значень система розраховує очікувану ймовірність перемоги команд, а після

завершення гри оновлює рейтингові показники учасників відповідно до результату матчу.

Для збереження даних використовується база даних PostgreSQL. У ній зберігається інформація про гравців, рейтинги за ролями, активні черги, створені матчі, учасників матчів, Discord-канали та конфігурацію серверу. Взаємодія з базою даних реалізована за допомогою SQLAlchemy ORM, що дозволяє описувати таблиці у вигляді Python-моделей і працювати з ними на рівні об’єктів.

Розгортання backend-частини та бази даних здійснюється за допомогою Docker і Docker Compose. Це спрощує запуск системи, забезпечує однакове середовище виконання та полегшує перенесення програмного забезпечення на сервер.

Окремо під час проєктування визначено основні ролі користувачів системи. Головними акторами є гравець та адміністратор. Гравець взаємодіє з ботом для участі у матчах, а адміністратор контролює процес організації гри та може втручатися у нестандартних ситуаціях. Основні варіанти використання системи наведено на (рис. 2.2)

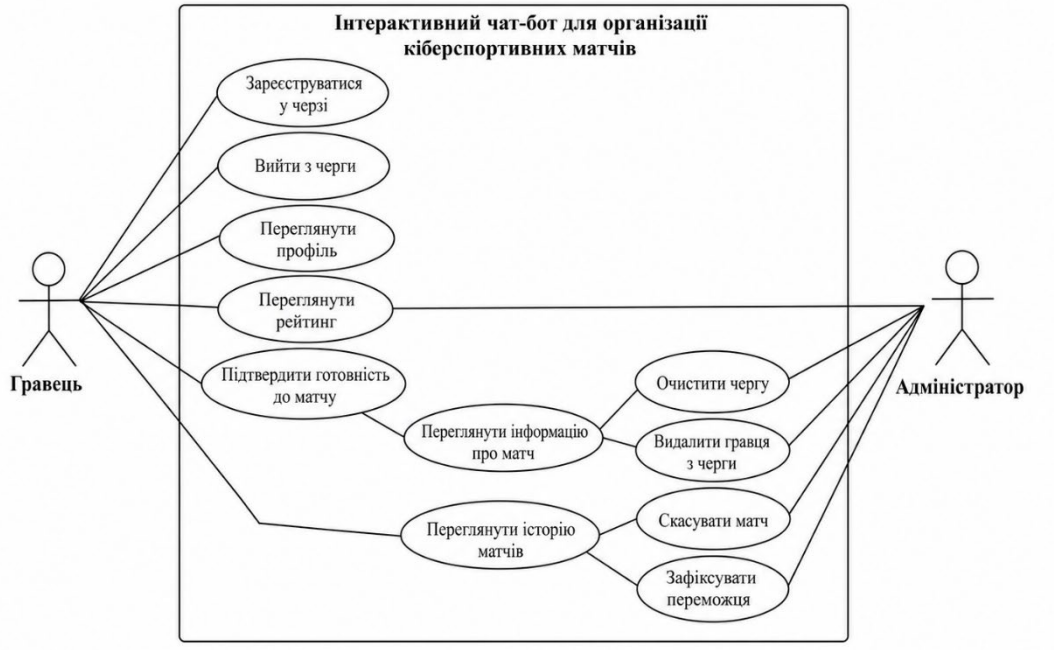



Рисунок 2.2 – Діаграма варіантів використання чат-бота

Діаграма варіантів використання дозволяє показати, які функції системи доступні різним типам користувачів. Це спрощує подальше проєктування логіки бота, оскільки кожна дія користувача може бути пов’язана з окремою командою або сценарієм роботи системи.

Отже, обрана архітектура дозволяє створити гнучке програмне забезпечення для автоматизації in-house матчів. Discord використовується як інтерфейс взаємодії, backend-частина реалізує основну логіку роботи, PostgreSQL забезпечує збереження даних, а Docker і Docker Compose спрощують розгортання системи.

### 2.2 Проєктування бази даних

Для збереження даних інтерактивного чат-бота використовується реляційна база даних PostgreSQL. Вона забезпечує надійне збереження структурованої інформації про користувачів, канали Discord, черги, матчі, рейтинги, ролі гравців та історію ігор.

База даних є важливою частиною системи, оскільки чат-бот повинен зберігати не лише поточний стан черг, а й інформацію про гравців, їхні рейтинги за ролями та результати проведених in-house матчів. Це дозволяє відновлювати інформацію після перезапуску бота, вести статистику та оновлювати рейтинг гравців після кожної завершеної гри.

Основними сутностями бази даних є гравці, рейтинги гравців, черги, матчі, учасники матчів, інформація про канали та конфігурація серверу. Такий поділ дозволяє уникнути дублювання даних і забезпечує зручну роботу з інформацією.

Сутність гравця зберігає основну інформацію про учасника Discord-серверу. До таких даних належать ідентифікатор користувача Discord, ідентифікатор серверу, ім’я гравця та назва команди. Оскільки один користувач може брати участь у матчах на різних серверах, профіль гравця визначається парою значень: Discord ID та ID серверу.

Окрема сутність рейтингу використовується для збереження показників гравця за кожною ігровою роллю. Для League of Legends підтримуються ролі Top,

Jungle, Mid, ADC та Support. Для кожної ролі зберігаються параметри TrueSkill: `trueskill_mu`, що відображає оцінку навички, та `trueskill_sigma`, що відображає невизначеність цієї оцінки. На основі цих параметрів формується внутрішній MMR, який використовується для відображення сили гравця.

Сутність черги зберігає інформацію про гравців, які очікують на створення матчу. До неї входять ідентифікатор каналу, роль гравця, ідентифікатор користувача, ідентифікатор серверу, час входу в чергу, інформація про ready-check та можливого duo-партнера. Наявність часу входу в чергу дозволяє враховувати порядок очікування під час формування матчу.

Сутність матчу зберігає інформацію про створену in-house гру. До неї належать ідентифікатор матчу, дата створення, ідентифікатор серверу, очікувана ймовірність перемоги синьої команди та команда-переможець. Значення очікуваної ймовірності перемоги використовується для оцінки якості балансування команд перед початком гри.

Сутність учасника матчу є зв’язуючою між гравцями та матчами. Вона визначає, який гравець брав участь у конкретній грі, за яку сторону він грав, яку роль займав та які рейтингові показники мав на момент створення матчу. Це дозволяє зберегти склад команд і надалі аналізувати історію участі кожного гравця.

Додатково у базі даних зберігається інформація про канали Discord, які використовуються ботом. Для кожного каналу фіксується його ідентифікатор, ідентифікатор серверу та тип каналу, наприклад канал черги або канал рейтингу. Також передбачено таблицю конфігурації серверу, де налаштування зберігаються у JSON-форматі.

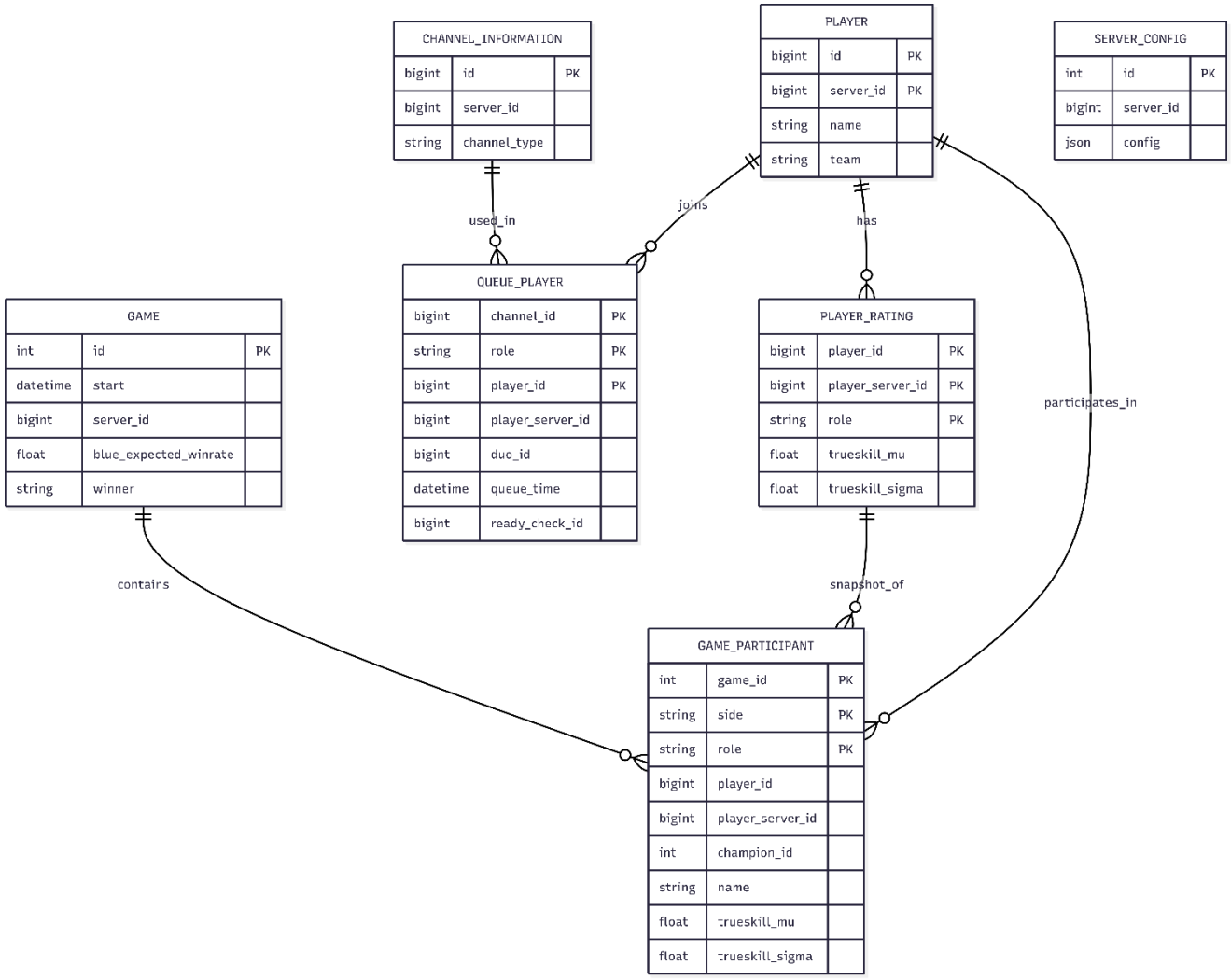



Рисунок 2.3 – Схема бази даних

Логічна структура бази даних (рис. 2.3) може бути подана такими основними таблицями:

1. `player` – зберігає дані про гравців Discord;

2. `player_rating` – зберігає рейтингові параметри гравців для кожної ролі;

3. `queue_player` – зберігає інформацію про гравців, які перебувають у черзі;

4. `game` – зберігає інформацію про створені матчі;

5. `game_participant` – зберігає інформацію про участь гравців у конкретних матчах;

6. `channel_information` – зберігає інформацію про Discord-канали, які використовуються ботом;

7. `server_config` – зберігає конфігурацію Discord-серверу.

Зв’язок між таблицями реалізується за допомогою зовнішніх ключів. Один гравець може мати кілька рейтингових записів, оскільки рейтинг зберігається окремо для кожної ролі. Один матч може мати багато учасників, а кожен учасник матчу пов’язаний з конкретним гравцем і його рейтинговими параметрами. Записи черги також пов’язані з гравцями та каналами Discord.

Важливою особливістю структури є збереження значень `trueskill_mu` та `trueskill_sigma` у таблиці `game_participant`. Під час створення матчу система фіксує рейтингові показники кожного учасника саме на момент початку гри. Це дозволяє зберігати історичний стан рейтингу та коректно аналізувати минулі матчі.

Для роботи з базою даних використовується SQLAlchemy ORM. Ця бібліотека дозволяє описувати таблиці у вигляді Python-моделей і виконувати операції з базою даних без прямого написання SQL-запитів у бізнес-логіці бота. Такий підхід спрощує підтримку програмного забезпечення та робить структуру проєкту більш зрозумілою.

Отже, спроєктована база даних забезпечує збереження основних даних системи: профілів гравців, рейтингу за ролями, активних черг, історії матчів, складів команд, результатів і конфігурації серверу. Така структура є достатньо гнучкою для поточного функціоналу чат-бота та може бути розширена в майбутньому для додавання нових ігрових режимів, статистики або адміністративних можливостей.

## 2.3 Проєктування алгоритму формування та балансування команд

Однією з основних задач чат-бота є автоматичне формування збалансованого in-house матчу. У розроблюваній системі цей процес виконується на основі рольових черг, рейтингової моделі TrueSkill та перевірки можливих варіантів розподілу гравців між двома командами.

У проєкті логіка матчмейкінгу винесена в окремий пакет `matchmaking_logic`. Основними його компонентами є функції оцінювання якості гри, пошуку найкращого складу команд та оновлення рейтингу після завершення

матчу. Такий поділ дозволяє відокремити алгоритмічну частину від обробки Discord-команд і роботи з базою даних.

Перед створенням матчу система перевіряє наявність достатньої кількості гравців у черзі. Для формату League of Legends необхідно мати щонайменше двох гравців на кожну з п’яти ролей: Top, Jungle, Mid, ADC та Support. Якщо хоча б для однієї ролі не вистачає учасників, матч не створюється.

Після проходження цієї перевірки система починає пошук оптимального складу гри. Спочатку розглядаються десять гравців, які перебувають у черзі найдовше. Якщо серед них не вдається знайти достатньо якісний варіант, алгоритм поступово розширює набір кандидатів, додаючи наступних гравців із черги. Це дозволяє враховувати як баланс команд, так і час очікування учасників.

Для кожної ролі формуються можливі пари гравців. Далі алгоритм перебирає комбінації цих пар і розподіляє їх між синьою та червоною командами. У результаті створюються варіанти команд, у яких кожна сторона має по одному гравцю на кожній ролі. Додатково перевіряється, щоб один і той самий гравець не потрапив у матч кілька разів.

Окремо враховується підтримка duo-черги. Якщо гравець зареєструвався разом із напарником, система намагається сформувати матч так, щоб обидва учасники потрапили в одну команду. Якщо умова не виконується, такий варіант складу відкидається.

Для оцінювання балансу команд використовується модель TrueSkill. Для кожного гравця система зберігає два параметри: `trueskill_mu`, що відображає оцінку навички, та `trueskill_sigma`, що відображає невизначеність цієї оцінки. На основі цих значень обчислюється очікувана ймовірність перемоги синьої команди над червоною [10, 16].

Якість матчу визначається за відхиленням очікуваної ймовірності перемоги від 50%. Чим ближче це значення до 50%, тим рівнішими вважаються команди. Для цього використовується показник `matchmaking_score`, який обчислюється як модуль різниці між 0.5 та очікуваною ймовірністю перемоги синьої команди.

Під час пошуку система зберігає найкращий знайдений варіант. Якщо

показник якості матчу стає достатньо малим, пошук може завершуватися достроково. Це зменшує кількість обчислень і дозволяє швидше сформувати гру навіть за наявності великої черги.

Після знаходження оптимального складу створюється об’єкт матчу. У ньому фіксуються гравці обох команд, їхні ролі, сторона, поточні значення TrueSkill та очікувана ймовірність перемоги синьої команди. Ці дані надалі зберігаються в базі даних і використовуються для історії матчів та оновлення рейтингу.

Алгоритм формування матчу можна подати у вигляді такої послідовності дій: гравці додаються до рольових черг, система перевіряє кількість учасників, формує можливі склади команд, оцінює їх за TrueSkill, вибирає найкращий варіант та створює матч. Діаграму діяльності алгоритму формування матчу наведено на (рис. 2.4).

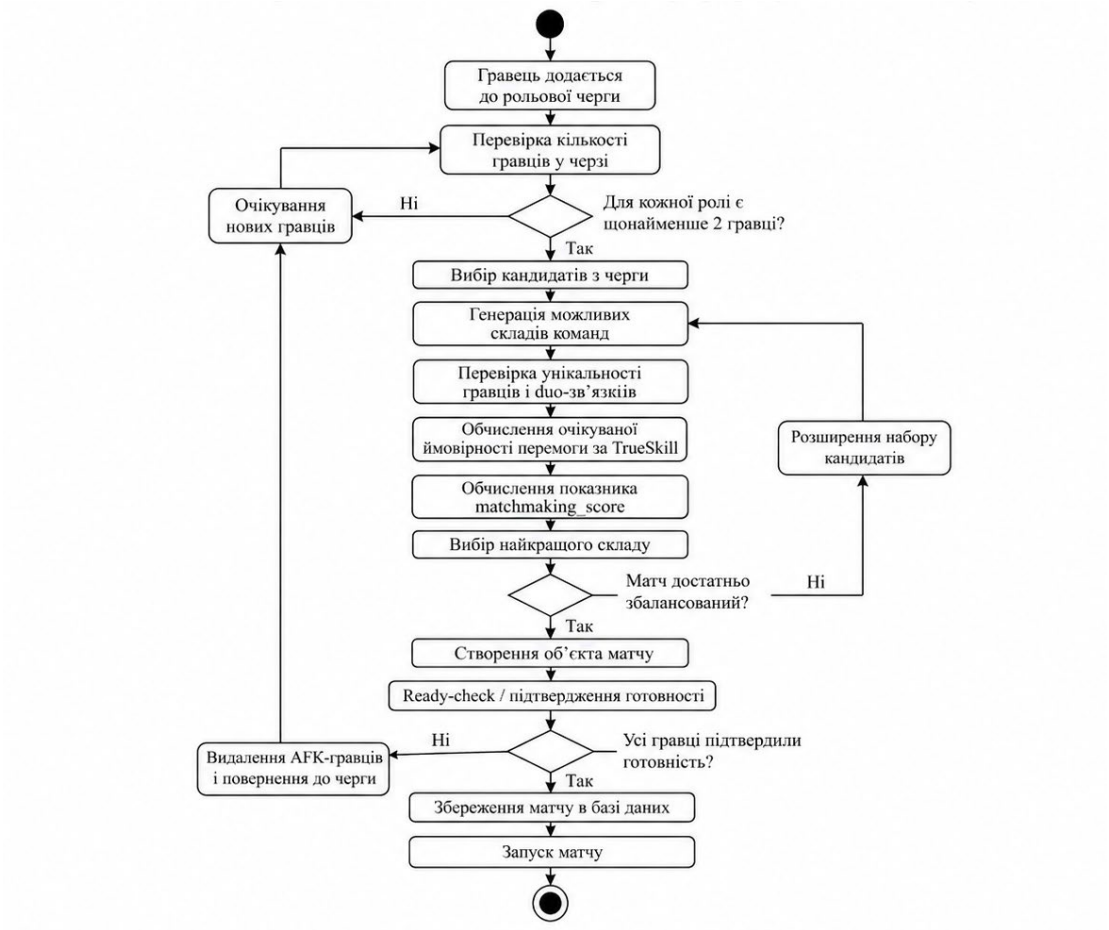



Рисунок 2.4 – Діаграма діяльності алгоритму формування матчу

Після завершення гри адміністратор або переможець фіксує результат. Система визначає сторону переможця та оновлює рейтинги учасників за допомогою функції `trueskill.rate`. Значення `trueskill_mu` та `trueskill_sigma` змінюються відповідно до результату гри, після чого оновлені дані зберігаються у базі даних.

Життєвий цикл матчу складається з кількох станів: очікування достатньої кількості гравців, пошук збалансованого складу, підтвердження готовності, активний матч, завершення або скасування. Діаграму станів матчу наведено на (рис. 2.5).

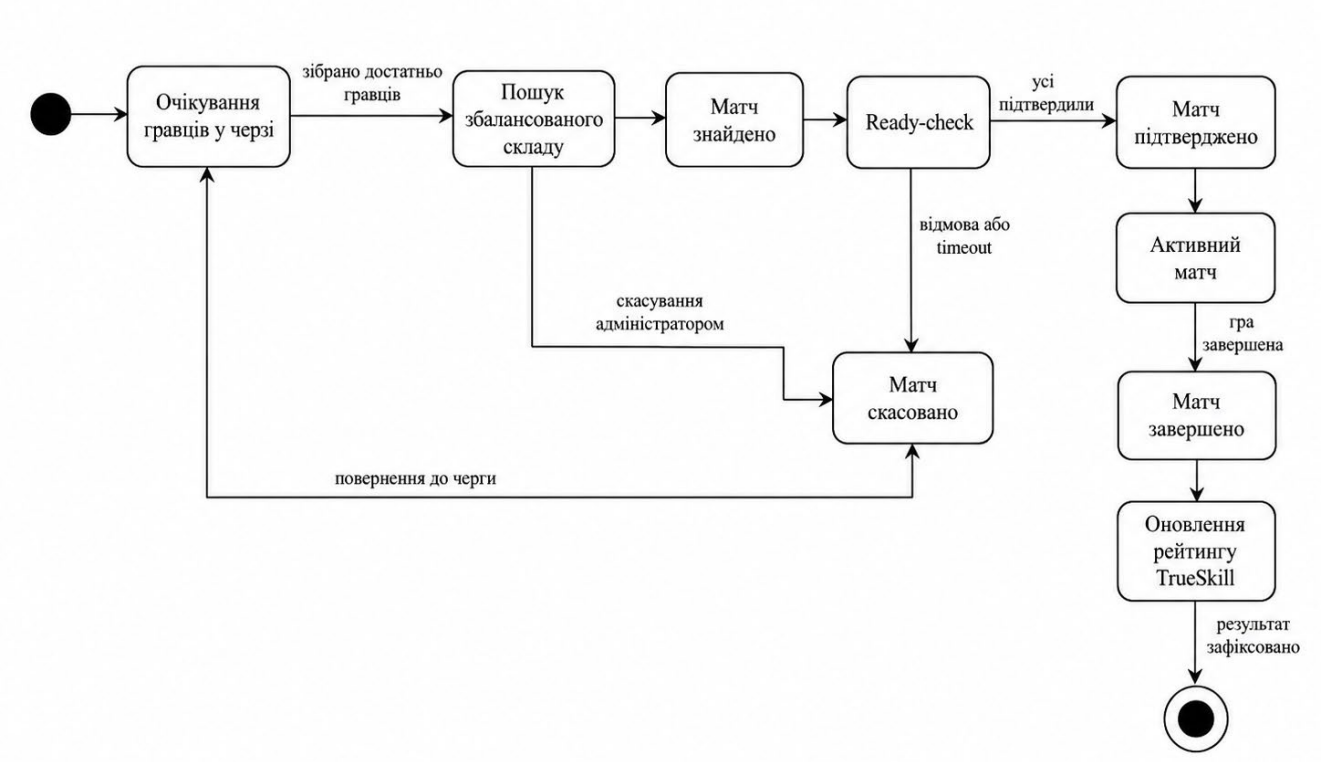



Рисунок 2.5 – Діаграма станів матчу

Отже, алгоритм формування та балансування команд дозволяє автоматизувати процес створення in-house матчів. На відміну від ручного розподілу або вибору капітанами, система використовує математичну оцінку сили гравців, враховує ролі, час перебування в черзі та можливі duo-зв’язки. Це зменшує вплив суб’єктивного фактора та забезпечує більш справедливий розподіл учасників між командами.

## 2.4 Проєктування структури програмних модулів

Основна структура проєкту містить кілька логічних пакетів: `cogs`, `common_utils`, `database_orm` та `matchmaking_logic`. Кожен із них відповідає за окрему частину роботи системи.

Пакет `cogs` (рис. 2.6) використовується для реалізації команд Discord-бота. У ньому розміщуються обробники команд, через які користувачі та адміністратори взаємодіють із системою. Саме цей рівень приймає дії з Discord і передає їх до відповідних сервісів або алгоритмів.

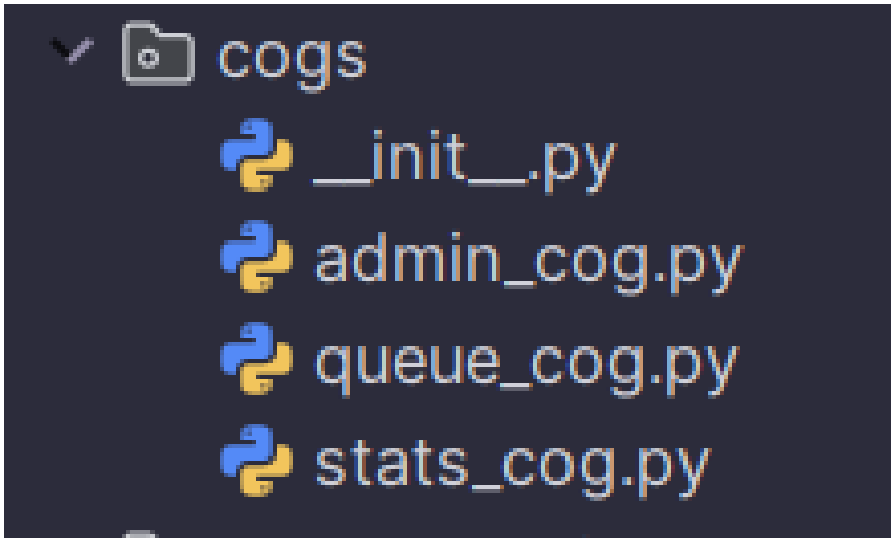



Рисунок 2.6 – Пакет cogs

Пакет `common_utils` (рис. 2.7) містить допоміжні функції та спільні налаштування, які використовуються в різних частинах проєкту. До них належать списки ролей, службові поля, емодзі, функції для отримання даних про останній матч та інші утиліти, що не належать до окремої бізнес-логіки.

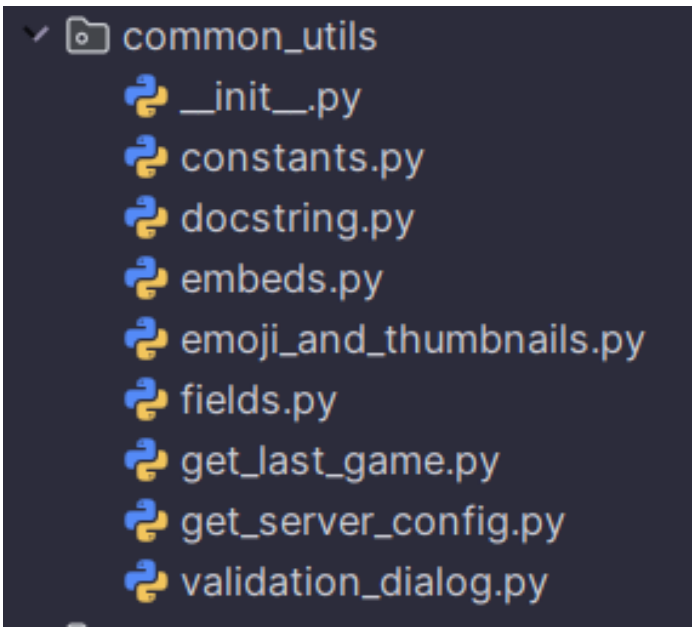



Рисунок 2.7 – Пакет common_utils

Пакет `database_orm` (рис. 2.8) відповідає за роботу з базою даних. У ньому розміщені моделі SQLAlchemy ORM, які описують основні таблиці системи: `Player`, `PlayerRating`, `QueuePlayer`, `Game`, `GameParticipant`, `ChannelInformation` та `ServerConfig`. Також у цьому пакеті організовано роботу із сесіями бази даних.

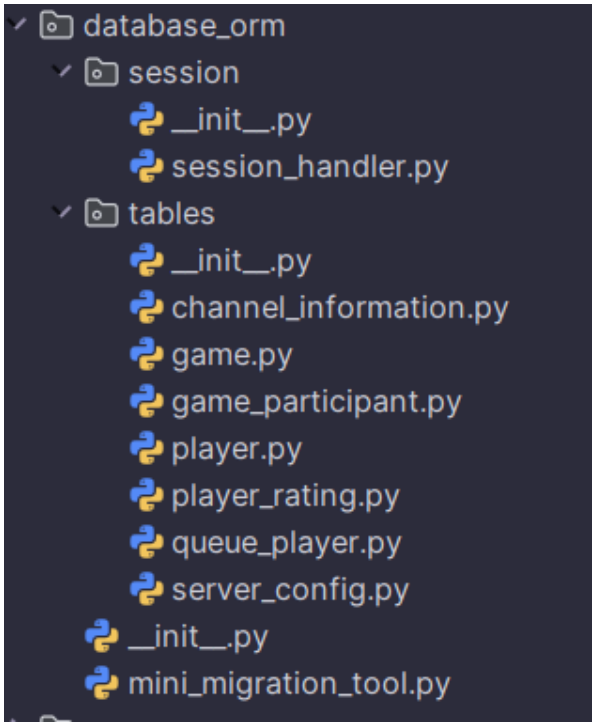



Рисунок 2.8 – Пакет database_orm

Клас `Player` описує гравця Discord-серверу. Він містить ідентифікатор користувача, ідентифікатор серверу, ім’я гравця та назву команди. З ним пов’язані рейтингові записи, які зберігаються окремо для кожної ролі (рис. 2.9).

```python
class Player(bot_declarative_base):
    __tablename__ = "player"

    # Discord account info
    id = Column(BigInteger, primary_key=True)

    # One player object per server_id
    server_id = Column(BigInteger, primary_key=True)

    # Player nickname and team as defined by themselves
    name = Column(String)
    team = Column(String)

    # We automatically load the ratings when loading a Player object
    ratings = relationship(
        "PlayerRating",
        collection_class=attribute_mapped_collection("role"),
        backref="player",
        cascade="all, delete-orphan",
        lazy="selectin",
    )

    # ORM relationship to the GameParticipant table
    participant_objects = relationship("GameParticipant", viewonly=True)

    @hybrid_property  8 usages (8 dynamic)
    def short_name(self):
        return self.name[:15]

    def __repr__(self):
        return f"<Player: {self.id=} | {self.name=}>"
```

Рисунок 2.9 – Клас Player

Клас `PlayerRating` (рис. 2.10) описує рольовий рейтинг гравця. Для кожної ролі зберігаються параметри TrueSkill: `trueskill_mu` та `trueskill_sigma`. На їх основі формується внутрішній MMR, який використовується для відображення сили гравця та балансування команд.

```python
class PlayerRating(bot_declarative_base):  34 usages (6 dynamic)
    """Represents the role-specific rating for a player taking part in in-house games"""

    __tablename__ = "player_rating"

    # Just like Player
    player_id = Column(BigInteger, primary_key=True)
    player_server_id = Column(BigInteger, primary_key=True)

    # We will get one row per role
    role = Column(role_enum, primary_key=True)

    # Current TrueSkill rating
    trueskill_mu = Column(Float)
    trueskill_sigma = Column(Float)

    # Foreign key to Player
    __table_args__ = (
        ...
    )

    # Conservative rating for MMR display
    @hybrid_property  8 usages (4 dynamic)
    def mmr(self):...

    def __repr__(self):...

    def __init__(self, player: Player, role):...
```

Рисунок 2.10 – Клас PlayerRating

Клас `QueuePlayer` (рис. 2.11) описує гравця, який перебуває у черзі на матч. Він містить інформацію про канал, роль, гравця, час входу в чергу, ready-check та можливий duo-зв’язок. Цей клас використовується під час пошуку кандидатів для створення матчу.

```python
class QueuePlayer(bot_declarative_base):  59 usages
    """..."""

    __tablename__ = "queue_player"

    channel_id = Column(
        BigInteger,
        ForeignKey("channel_information.id", **foreignkey_cascade_options),
        primary_key=True,
        index=True,
    )

    channel_information = relationship(
        "ChannelInformation", viewonly=True, backref="game_participant_objects", sync_backref=False
    )

    role = Column(role_enum, primary_key=True)

    # Saving both allows us to going to the Player table
    player_id = Column(BigInteger, primary_key=True, index=True)
    player_server_id = Column(BigInteger)
```

Рисунок 2.11 – Клас QueuePlayer

Клас `Game` (рис. 2.12) описує створений in-house матч. Він зберігає дату створення, сервер, очікувану ймовірність перемоги синьої команди та переможця гри. Також у ньому визначено властивості для отримання складів синьої та червоної команд.

```python
class Game(bot_declarative_base):
    id = Column(Integer, primary_key=True)

    # Game creation date
    start = Column(DateTime)

    # Server the game was played from
    server_id = Column(BigInteger)

    # Predicted outcome before the game was played
    blue_expected_winrate = Column(Float)

    # Winner, updated at the end of the game
    winner = Column(side_enum)
```

Рисунок 2.12 – Клас Game

Клас `GameParticipant` (рис. 2.13) описує участь конкретного гравця у матчі. Він зберігає сторону, роль, ідентифікатор гравця, ім’я, обраного чемпіона та значення TrueSkill на момент створення матчу. Це дозволяє зафіксувати стан учасника саме під час проведення гри.

```
class GameParticipant(bot_declarative_base):  29 usages
    name = Column(String)

    # Pre-game TrueSkill values
    trueskill_mu = Column(Float)
    trueskill_sigma = Column(Float)

    # Foreign key to Player
    __table_args__ = (
        ForeignKeyConstraint((player_id, player_server_id), (Player.id, Player.server_id)),
        ForeignKeyConstraint(
            (player_id, player_server_id, role),
            (PlayerRating.player_id, PlayerRating.player_server_id, PlayerRating.role),
        ),
        {},
    )

    # Conservative rating for MMR display
    @hybrid_property  5 usages (4 dynamic)
    def mmr(self):...

    @hybrid_property  8 usages (8 dynamic)
    def short_name(self):...

    # Called only from the Game constructor itself
    def __init__(self, side: str, role: str, player: Player):...
```

Рисунок 2.13 – Клас GameParticipant

Клас `ChannelInformation` (рис. 2.14) зберігає інформацію про Discord-канали, які використовуються ботом. Канал може бути пов’язаний із чергою, рейтингом або іншими службовими функціями. Клас `ServerConfig` використовується для збереження налаштувань конкретного Discord-серверу у JSON-форматі.

```python
class ChannelInformation(bot_declarative_base):  15 usages
    """Represents a channel used by the inhouse bot"""

    __tablename__ = "channel_information"

    # Discord ID
    id = Column(BigInteger, primary_key=True)

    # Useful for querying
    server_id = Column(BigInteger)

    # Current accepted values are "QUEUE" and "RANKING",
    channel_type = Column(String)

    def __repr__(self):...
```

Рисунок 2.14 – Клас ChannelInformation

Пакет `matchmaking_logic` (рис. 2.15) містить алгоритмічну частину системи. У ньому реалізовано оцінювання якості матчу, пошук найкращого складу команд і оновлення рейтингу після завершення гри. Функція `evaluate_game` обчислює очікувану ймовірність перемоги синьої команди. Функція `find_best_game` виконує пошук оптимального розподілу гравців. Функція `update_trueskill` оновлює рейтингові параметри учасників після фіксації результату.

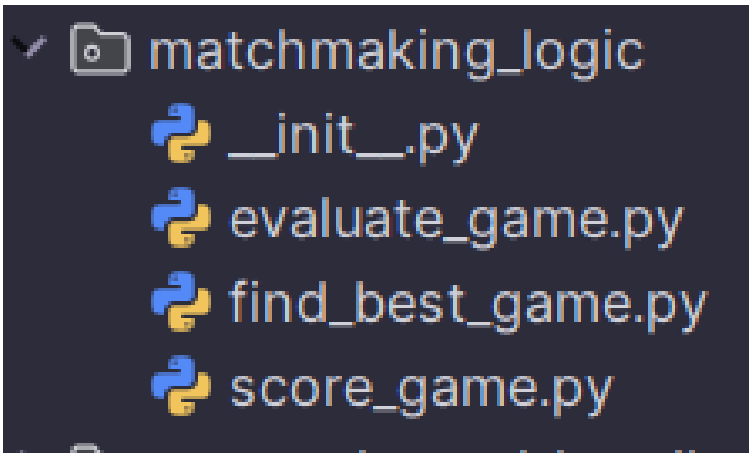



Рисунок 2.15 – Пакет matchmaking_logic

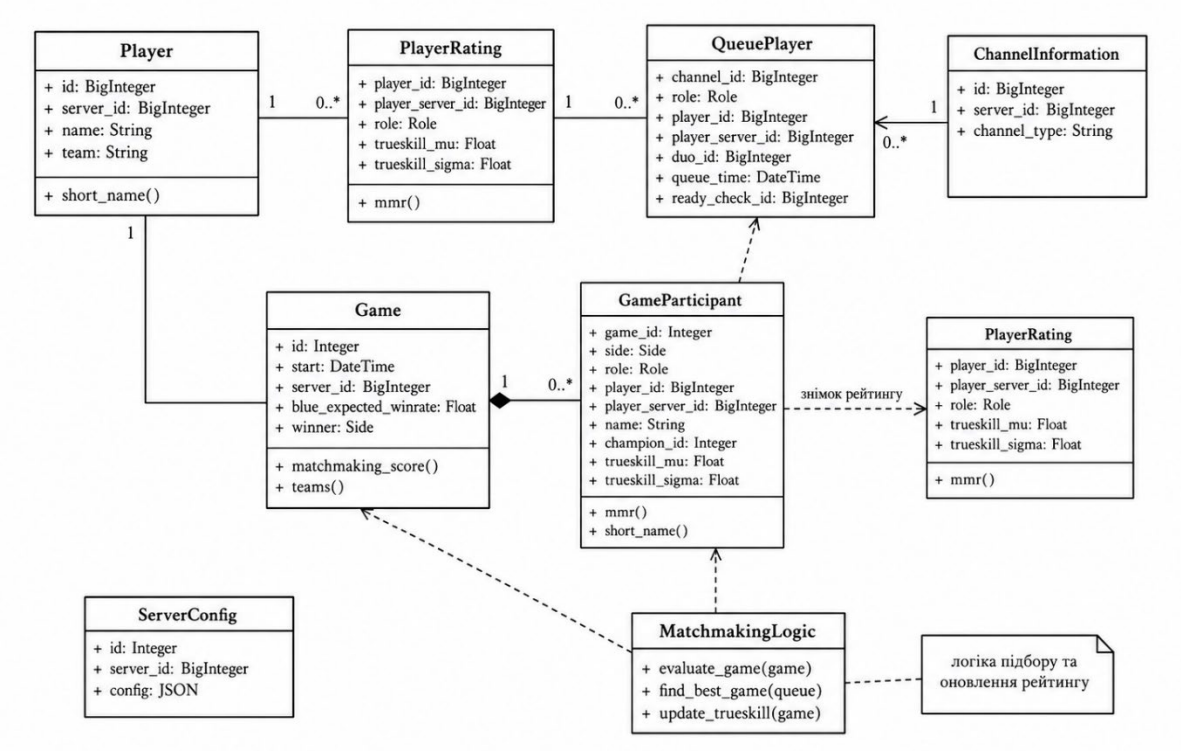



Рисунок 2.16 – Діаграма класів основних компонентів чат-бота

Діаграма класів (рис. 2.16) відображає зв’язки між ORM-моделями та основними логічними компонентами системи. Вона показує, що гравець може мати кілька рейтингових записів, брати участь у чергах і матчах, а кожен матч складається з набору учасників, розподілених за сторонами та ролями.

Отже, модульна структура проєкту дозволяє відокремити обробку команд Discord, роботу з базою даних, допоміжні функції та алгоритми матчмейкінгу. Такий підхід робить систему зрозумілішою, спрощує тестування окремих частин і створює основу для подальшого розширення функціоналу чат-бота.

# 3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ЧАТ-БОТА

У цьому розділі розглянуто програмну реалізацію інтерактивного чат-бота для організації кіберспортивних матчів. Описано структуру програмного проєкту, основні модулі системи, реалізацію команд для користувачів і адміністраторів, а також механізми роботи з чергами, матчами, рейтингом і статистикою.

Окрему увагу приділено перевірці працездатності розробленого програмного забезпечення. Для цього розглянуто результати ручного тестування основних сценаріїв роботи чат-бота та автоматизовані тести, які перевіряють логіку черг, матчмейкінгу й командного шару системи.

## 3.1 Організація структури програмного проєкту

Програмна реалізація інтерактивного чат-бота виконана мовою Python з використанням бібліотеки discord.py. Проєкт має модульну структуру (рис. 3.1), що дозволяє розділити логіку обробки команд, роботу з базою даних, допоміжні функції та алгоритми матчмейкінгу.

Основний код системи розміщено у пакеті `inhouse_bot`. У ньому виділено кілька логічних каталогів: `cogs`, `common_utils`, `database_orm` та `matchmaking_logic`. Такий поділ спрощує супровід програмного забезпечення та дозволяє вносити зміни в окремі частини системи без повної переробки проєкту.

Каталог `cogs` містить обробники команд Discord-бота. У discord.py такі компоненти використовуються для групування команд і подій за функціональним призначенням. Через них реалізується взаємодія користувачів із ботом: реєстрація у черзі, вихід з неї, перегляд інформації, адміністративне керування та фіксація результатів матчів.

Каталог `common_utils` містить допоміжні функції та спільні налаштування, які використовуються різними частинами системи. До них належать списки ігрових ролей, службові поля, робота з емодзі, форматування повідомлень і функції для отримання інформації про останні матчі.

Каталог `database_orm` відповідає за роботу з базою даних. У ньому розміщено моделі SQLAlchemy ORM, що описують таблиці `player`, `player_rating`, `queue_player`, `game`, `game_participant`, `channel_information` та `server_config`. Також у цьому пакеті організовано створення сесій для виконання операцій читання та запису.

Каталог `matchmaking_logic` містить алгоритмічну частину системи. У ньому реалізовано оцінювання якості гри, пошук найкращого складу команд і оновлення рейтингу після завершення матчу. Саме ця частина відповідає за формування збалансованих in-house ігор на основі ролей і параметрів TrueSkill.

Для розгортання системи використовується Docker та Docker Compose. Завдяки цьому бот і база даних PostgreSQL можуть запускатися в окремих контейнерах, що спрощує налаштування середовища та забезпечує однакові умови запуску на різних пристроях. Розробку та налагодження виконано в інтегрованому середовищі PyCharm [4], а керування версіями коду – за допомогою системи Git [13].

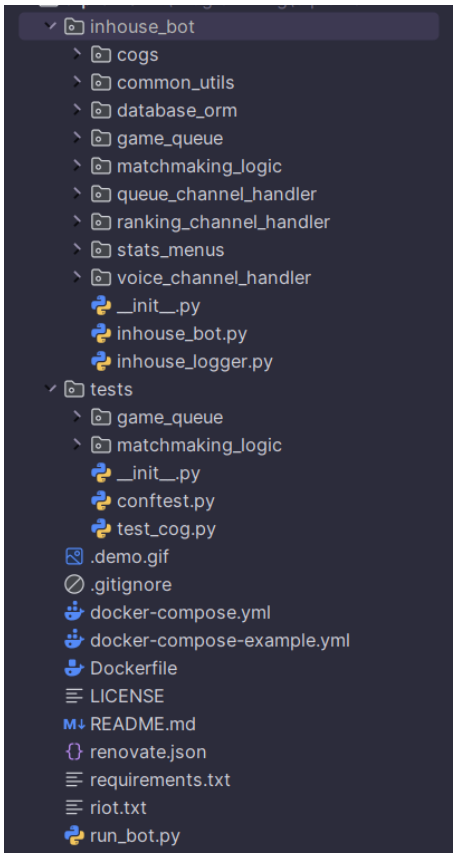



Рисунок 3.1 – Стурктура проекту у середовищі розробки

Отже, структура програмного проєкту побудована за модульним принципом. Обробка Discord-команд, робота з даними, допоміжні функції та алгоритми матчмейкінгу винесені в окремі частини, що робить код зрозумілим, зручним для підтримки та готовим до подальшого розширення.

### 3.2 Реалізація команд Discord-бота

Взаємодія користувачів із системою реалізована безпосередньо в середовищі Discord. Для цього використовується бібліотека `discord.py`, яка забезпечує підключення бота до Discord API, обробку команд, подій та відправлення відповідей користувачам.

Команди бота організовано у вигляді окремих обробників у пакеті `cogs`. Такий підхід дозволяє розділити функціонал за призначенням і не зберігати всю логіку в одному файлі. Наприклад, окремі групи команд можуть відповідати за роботу з профілем гравця, чергами, матчами, рейтингом та адміністративними діями.

Основною групою користувачів системи є гравці Discord-серверу. Вони можуть реєструватися у рольовій черзі, виходити з неї, переглядати власний профіль, рейтинг та інформацію про активний матч. Для командної дисципліни League of Legends у системі передбачено п’ять ролей: Top, Jungle, Mid, ADC та Support. Під час входу в чергу користувач обирає одну з доступних ролей, після чого інформація про нього зберігається у таблиці `queue_player` (рис. 3.2).

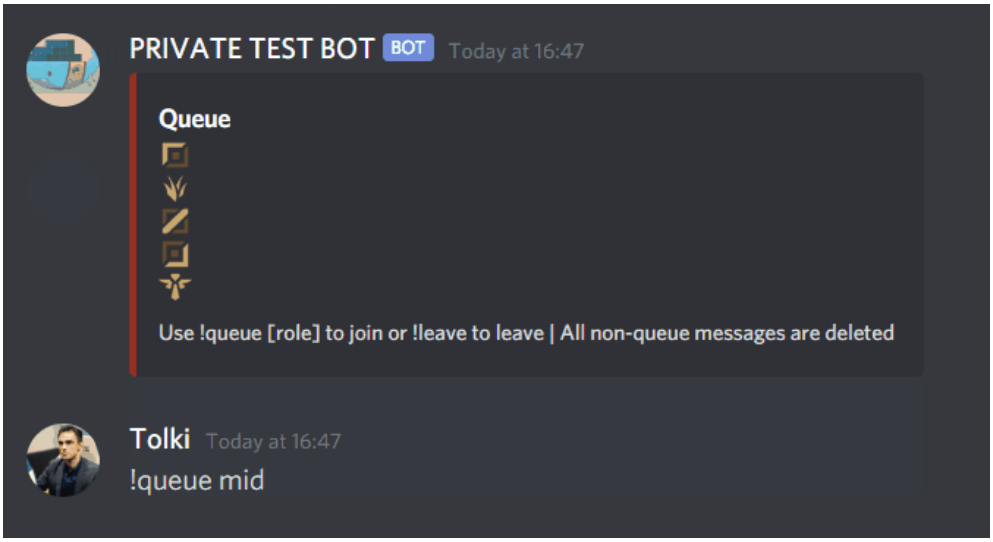



Рисунок 3.2 – Приклад додавання гравця до рольової черги

Після додавання гравця до черги бот перевіряє, чи є достатня кількість учасників для створення матчу. Якщо для кожної ролі доступно щонайменше два гравці, система запускає алгоритм формування команд. Якщо учасників недостатньо, бот залишає гравця в черзі та очікує нових користувачів.

Для перевірки роботи рольових черг також використовується службова команда !test queue (рис. 3.3). Вона дозволяє швидко заповнити чергу тестовими гравцями без необхідності вручну додавати кожного учасника окремо. Така команда спрощує перевірку алгоритму формування матчу, оскільки дає змогу швидко створити ситуацію, коли в черзі вже є достатня кількість гравців для всіх ролей.

Після виконання тестової команди бот виводить повідомлення про заповнення черги та показує її поточний стан. У повідомленні відображаються ролі та гравці, які перебувають у черзі. Це дозволяє перевірити, чи правильно система групує учасників за ролями та чи може надалі перейти до формування матчу.


Рисунок 3.3 – Приклад заповнення тестової рольової черги

Окремо реалізовано команди для перегляду інформації. Користувач може отримати дані про свій профіль, рольовий рейтинг, поточний стан черги або останні матчі. Такі команди не змінюють стан системи, а лише отримують інформацію з бази даних і відображають її у зручному форматі в Discord (рис. 3.4).

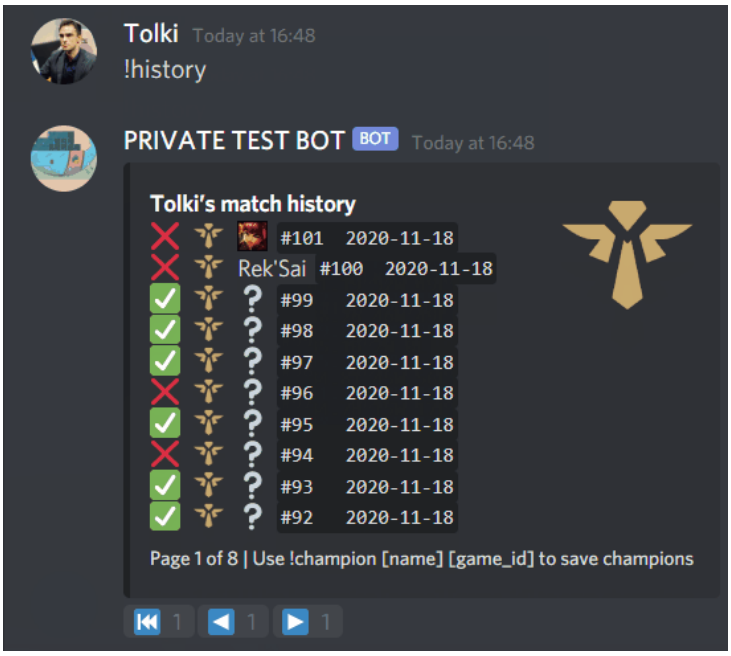



Рисунок 3.4 – Перегляд історії матчів гравця

Для підтвердження участі в матчі використовується механізм ready-check. Після знаходження збалансованого складу бот надсилає повідомлення з інформацією про команди та очікує підтвердження готовності від гравців. Якщо всі учасники підтверджують готовність, матч вважається прийнятим і зберігається у базі даних. Якщо частина гравців не відповідає або відмовляється, система може повернути учасників до черги або скасувати створення матчу (рис. 3.5).

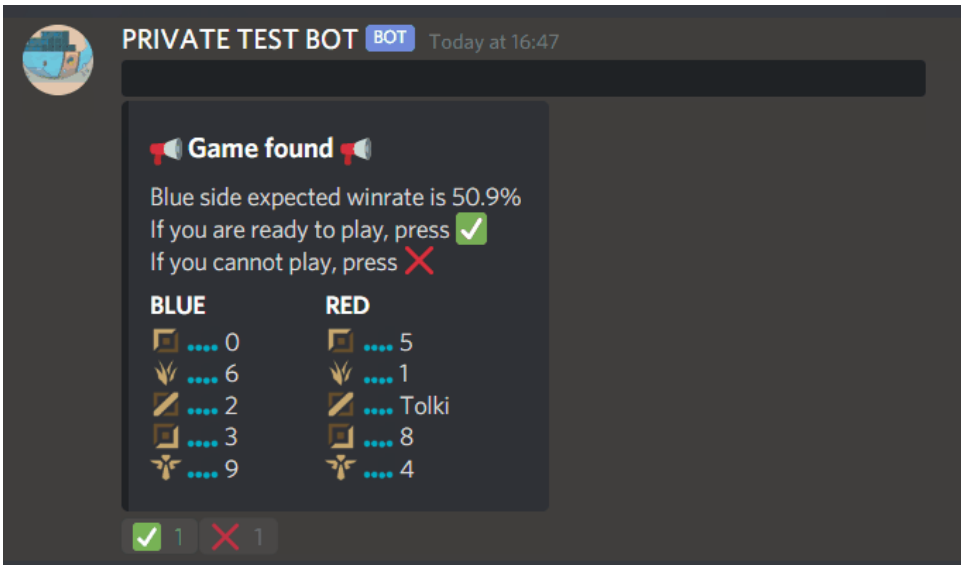



Рисунок 3.5 – Повідомлення боті про знайдений матч

Адміністративні команди (рис. 3.6) призначені для контролю нестандартних ситуацій. Адміністратор може очистити чергу, видалити конкретного гравця, скасувати активний матч або зафіксувати переможця після завершення гри. Такі дії необхідні для підтримки коректної роботи бота, особливо у випадках, коли гравець залишив матч, не підтвердив готовність або результат потрібно внести вручну.

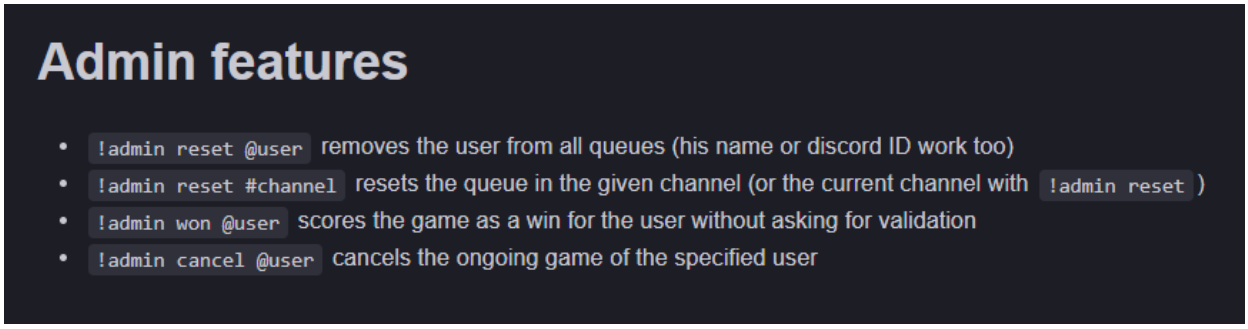



Рисунок 3.6 – Список адміністративних команд

Після фіксації переможця бот оновлює рейтингові параметри учасників. Для цього використовується модель TrueSkill, яка змінює значення `trueskill_mu` та `trueskill_sigma` залежно від результату матчу. Оновлені дані зберігаються у таблиці `player_rating` (рис. 3.7).


Рисунок 3.7 – Таблиця рейтингу гравців

Для виведення інформації бот використовує повідомлення Discord, зокрема вбудовані повідомлення типу `Embed`. Це дозволяє зручно відображати склад команд, очікувану ймовірність перемоги, статус ready-check та іншу службову інформацію. Такий формат є зрозумілим для користувачів і добре підходить для командної взаємодії всередині Discord-серверу.

Окрім перегляду профілю та історії матчів, у боті реалізовано можливість перегляду зміни рейтингу гравця за певний період. Для цього використовується команда !mmr_history (рис. 3.8), яка формує графік зміни MMR на основі збережених результатів матчів. На графіку відображається динаміка рейтингу гравця протягом останнього місяця для конкретної ролі.

Такий функціонал є корисним для учасників спільноти, оскільки дозволяє не лише бачити поточне значення рейтингу, а й аналізувати його зміну після зіграних

матчів. Якщо рейтинг гравця зростає, це свідчить про успішні результати в попередніх іграх. Якщо значення знижується або залишається стабільним, користувач може оцінити власний прогрес і результати участі в in-house матчах.

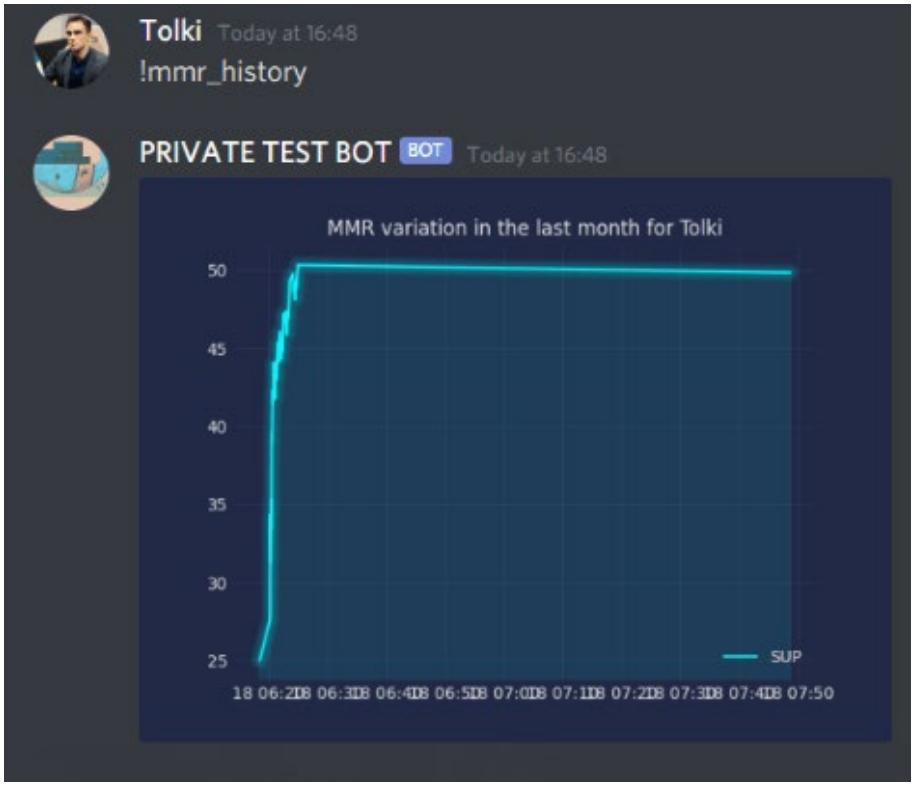



Рисунок 3.8 – Графік зміни MMR гравця

Окрім ручного тестування роботи чат-бота в Discord, у проєкті передбачено каталог автоматизованих тестів (рис. 3.9) [14].

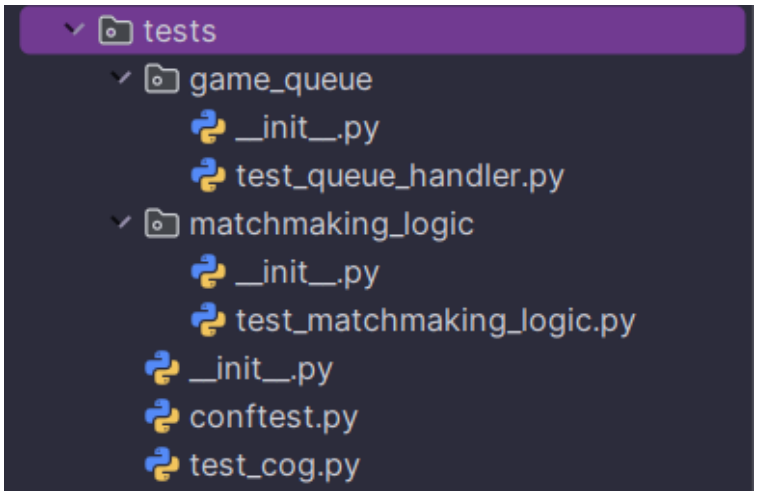



Рисунок 3.9 – Структура каталогу автоматизованих тестів

Автоматизовані тести розміщено в каталозі `tests`. Вони призначені для перевірки окремих компонентів системи без необхідності щоразу вручну відтворювати всі сценарії в Discord. Такий підхід дозволяє швидше виявляти помилки після внесення змін у код і перевіряти стабільність основної логіки бота.

Каталог `tests/game_queue` містить тести для перевірки роботи черг. Файл `test_queue_handler.py` (рис. 3.10) використовується для перевірки сценаріїв додавання гравців до черги, виходу з черги та обробки стану черги. Такі тести є важливими, оскільки рольова черга є початковим етапом усього процесу створення in-house матчу.

```python
def test_queue_full():
    game_queue.reset_queue()

    for player_id in range(0, 10):
        game_queue.add_player(player_id, roles_list[player_id % 5], channel_id: 0, server_id: 0, name=str(player_id))

    assert len(GameQueue(0)) == 10

    # We queue our player 0 in channel 1, which he should be allowed to do
    game_queue.add_player( player_id: 0, roles_list[0], channel_id: 1, server_id: 0, name="0")
    assert len(GameQueue(1)) == 1

    # We queue our player 0 for every other role in channel 0
    for role in roles_list:
        game_queue.add_player( player_id: 0, role, channel_id: 0, server_id: 0, name="0")

    assert len(GameQueue(0)) == 14  # Every role should count as a different QueuePlayer

    # Assuming our matchmaking logic found a good game (id 0)
    game_queue.start_ready_check(list(range(0, 10)), channel_id: 0, ready_check_message_id: 0)
    assert len(GameQueue(0)) == 0  # Player 0 should not be counted in queue for any role anymore

    # Our player 0 in channel 1 should not be counted in queue either
    assert len(GameQueue(1)) == 0
```

Рисунок 3.10 – Вміст файлу test_queue_handler.py

Каталог `tests/matchmaking_logic` містить тести алгоритму матчмейкінгу. Файл `test_matchmaking_logic.py` використовується для перевірки логіки формування гри, оцінювання балансу команд і вибору найкращого складу. Ці тести дають змогу переконатися, що алгоритм коректно працює з різними наборами гравців і не порушує основні правила формування матчу.

```python
def test_matchmaking_logic():
    game_queue.reset_queue()

    # We queue for everything except the red support
    for player_id in range(0, 9):
        game_queue.add_player(player_id, roles_list[player_id % 5], channel_id: 0, server_id: 0, name=str(player_id))

        assert not find_best_game(GameQueue(0))

    # We add the last player
    game_queue.add_player( player_id: 9, role: "SUP", channel_id: 0, server_id: 0, name="9")

    game = find_best_game(GameQueue(0))

    assert game

    # We commit the game to the database
    with session_scope() as session:
        session.add(game)

    # We say player 0 won his last game on server 0
    score_game_from_winning_player( player_id: 0, server_id: 0)
```

Рисунок 3.11 – Вміст файлу test_matchmaking_logic.py

Окремий файл `conftest.py` (рис.3.12) використовується для спільних налаштувань тестового середовища. У ньому можуть розміщуватися фікстури, які потрібні кільком тестам одночасно: тестові гравці, черги, об’єкти матчів або підготовлені дані для перевірки алгоритмів. Це дозволяє не дублювати однаковий підготовчий код у різних тестових файлах.

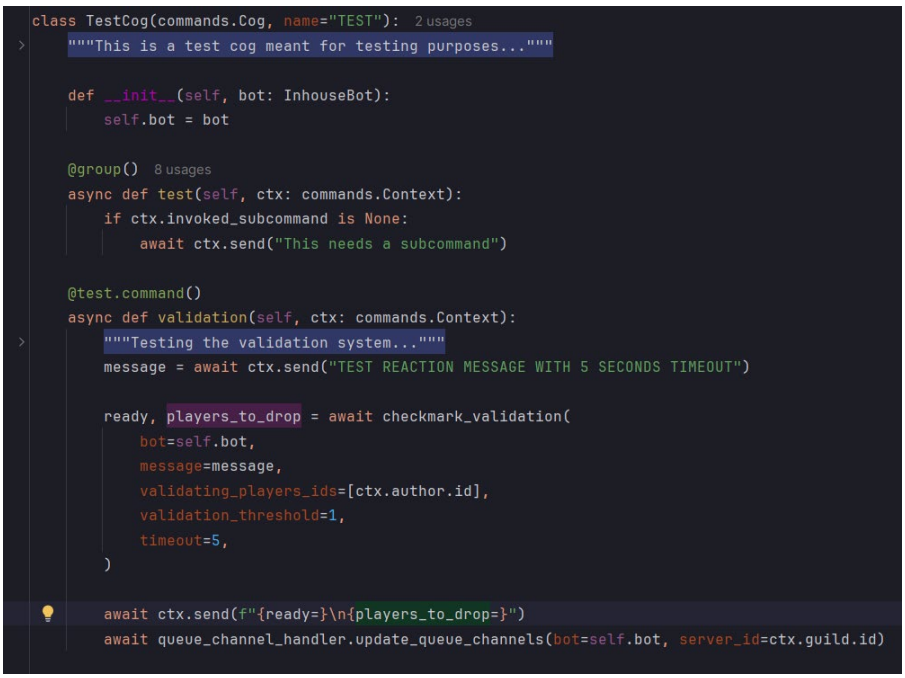
```python
import os
import sqlalchemy

db_name = "inhouse_bot"

# We need to create an engine that's not linked to the database
no_db_engine = sqlalchemy.create_engine(os.environ["INHOUSE_BOT_CONNECTION_STRING"][: -len(db_name)])
no_db_engine.execution_options(isolation_level="AUTOCOMMIT").execute(f"DROP DATABASE IF EXISTS {db_name};")
no_db_engine.execution_options(isolation_level="AUTOCOMMIT").execute(f"CREATE DATABASE {db_name};")
del no_db_engine
```

Рисунок 3.12 – Вміст conftest.py

Файл `test_cog.py` використовується для перевірки логіки, пов’язаної з командами бота. Такі тести дозволяють перевірити, чи правильно обробники команд викликають потрібні функції та чи коректно реагують на основні сценарії взаємодії користувача із системою.



```python
class TestCog(commands.Cog, name="TEST"):  2 usages
    """This is a test cog meant for testing purposes..."""

    def __init__(self, bot: InhouseBot):
        self.bot = bot

    @group()  8 usages
    async def test(self, ctx: commands.Context):
        if ctx.invoked_subcommand is None:
            await ctx.send("This needs a subcommand")

    @test.command()
    async def validation(self, ctx: commands.Context):
        """Testing the validation system..."""
        message = await ctx.send("TEST REACTION MESSAGE WITH 5 SECONDS TIMEOUT")

        ready, players_to_drop = await checkmark_validation(
            bot=self.bot,
            message=message,
            validating_players_ids=[ctx.author.id],
            validation_threshold=1,
            timeout=5,
        )

        await ctx.send(f"{ready=}\n{players_to_drop=}")
        await queue_channel_handler.update_queue_channels(bot=self.bot, server_id=ctx.guild.id)
```

Рисунок 3.13 – Вміст файлу test_cog.py

Автоматизоване тестування доповнює ручну перевірку. Ручне тестування дозволяє переконатися, що бот правильно працює в інтерфейсі Discord, а автоматизовані тести перевіряють внутрішню логіку окремих компонентів. У поєднанні ці два підходи дають змогу перевірити як зовнішню поведінку системи, так і коректність її програмної реалізації.

Основними перевагами автоматизованого тестування в межах проєкту є:

– швидка перевірка логіки після внесення змін;

– можливість повторного запуску тестів без ручного введення команд;

– зменшення ризику появи помилок у чергах і матчмейкінгу;

– ізольована перевірка окремих функціональних блоків;

– підвищення надійності програмного забезпечення.

Зведені результати ручного та автоматизованого тестування основних сценаріїв роботи чат-бота наведено в додатку Б (таблиця Б.1).

Автоматизовані тести доповнили ручну перевірку, оскільки дали змогу окремо перевірити внутрішню логіку системи без виконання всіх дій через інтерфейс Discord. У результаті можна зробити висновок, що розроблений чат-бот готовий до використання в тестовому середовищі Discord-спільноти.

# 4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

Розробка інтерактивного Discord-чат-бота для організації кіберспортивних in-house матчів виконується переважно за персональним комп’ютером. Такий вид діяльності пов’язаний із тривалою розумовою працею, статичним положенням тіла, навантаженням на зоровий аналізатор, нервово-емоційним напруженням, а також використанням електрообладнання. Тому під час виконання роботи необхідно враховувати вимоги безпеки життєдіяльності та охорони праці для користувачів персональних комп’ютерів.

У цьому розділі розглянуто ризик як кількісну оцінку небезпек, а також загальні вимоги безпеки з охорони праці для користувачів ПК. Дотримання цих вимог дає змогу зменшити ймовірність професійного перевтомлення, порушень зору, захворювань опорно-рухового апарату, ураження електричним струмом і виникнення пожежонебезпечних ситуацій. Нормативною основою є Закон України «Про охорону праці», вимоги щодо роботи з екранними пристроями та санітарні норми для виробничих приміщень [16], [19], [21].

## 4.1 Ризик як кількісна оцінка небезпек

У безпеці життєдіяльності небезпека розглядається як явище, процес або чинник, що потенційно може завдати шкоди людині, обладнанню, інформаційним ресурсам або середовищу. Однак сама наявність небезпеки ще не означає обов’язкове виникнення негативних наслідків. Для оцінювання реального рівня загрози використовують поняття ризику [17].

Ризик – це кількісна оцінка небезпеки, яка враховує ймовірність настання небезпечної події та тяжкість її можливих наслідків. У найпростішому вигляді ризик можна подати формулою:

$$R = P \cdot C, \tag{4.1}$$

де R – рівень ризику, P – імовірність виникнення небезпечної події, C – тяжкість або величина можливих наслідків.

Для діяльності розробника програмного забезпечення небезпеки не завжди мають миттєвий прояв. Часто вони накопичуються поступово: тривале сидіння за комп’ютером може спричинити втому м’язів спини та шиї, неправильне освітлення – перенапруження зору, а відсутність перерв – зниження уваги й працездатності. Тому ризик у такій діяльності оцінюється не лише як імовірність аварійної ситуації, а й як імовірність поступового погіршення стану здоров’я.

Основними небезпечними та шкідливими чинниками під час роботи над програмним забезпеченням є:

1. тривале статичне навантаження через вимушене сидяче положення;
2. напруження зору під час роботи з екраном;
3. нервово-емоційне навантаження, пов’язане з розумовою працею;
4. монотонність роботи та висока концентрація уваги;
5. ризик ураження електричним струмом;
6. пожежна небезпека через використання електрообладнання;
7. несприятливі параметри мікроклімату та освітлення.

Оцінювання ризику дає змогу визначити, які небезпеки є найбільш значущими та які заходи потрібно застосувати першочергово. Наприклад, якщо ймовірність перевтоми під час тривалої роботи за комп’ютером є високою, а наслідки можуть призвести до зниження працездатності та погіршення здоров’я, такий ризик потрібно зменшувати за допомогою регламентованих перерв, правильної організації робочого місця та виконання вправ для очей і м’язів.

Для практичного оцінювання ризиків часто застосовують бальну шкалу. Імовірність небезпечної події та тяжкість її наслідків оцінюють, наприклад, від 1 до 5 балів. Після множення цих значень отримують загальний рівень ризику. Низький ризик може бути прийнятним, середній потребує контролю та профілактичних заходів, а високий вимагає обов’язкового усунення або зменшення небезпечного чинника [18].

Таблиця 4.1 – Приклад оцінювання ризиків під час роботи користувача ПК

| **Небезпечний або шкідливий чинник** | **Можливі наслідки** | **Імовірність, P** | **Тяжкість наслідків, C** | **Рівень ризику, R = P · C** | **Заходи зниження ризику** |
|---|---|---|---|---|---|
| Тривала робота в сидячому положенні | Біль у спині, шиї, порушення постави | 3 | 3 | 9 | Використання ергономічного крісла, правильна посадка, короткі перерви |
| Напруження зору під час роботи з екраном | Втома очей, головний біль, зниження концентрації | 4 | 3 | 12 | Розташування екрана на відстані 50–70 см, належне освітлення, вправи для очей |
| Нервово-емоційне напруження | Перевтома, дратівливість, зниження працездатності | 3 | 3 | 9 | Раціональний режим праці та відпочинку, чергування видів діяльності |
| Несправне електрообладнання | Ураження електричним струмом | 2 | 5 | 10 | Перевірка кабелів, справні розетки, захисне заземлення |
| Перевантаження електромережі | Коротке замикання, пожежа | 2 | 5 | 10 | Заборона використання пошкоджених подовжувачів, контроль навантаження на мережу |
| Недостатнє або неправильне освітлення | Перенапруження зору, швидка втома | 3 | 3 | 9 | Забезпечення рівномірного освітлення, усунення відблисків на екрані |
| Несприятливий мікроклімат у приміщенні | Дискомфорт, зниження уваги та працездатності | 3 | 2 | 6 | Провітрювання приміщення, підтримання оптимальної температури та вологості |

Отже, ризик є важливим показником безпеки, оскільки дає змогу не лише виявити небезпеку, а й оцінити її реальну значущість. Для роботи розробника програмного забезпечення основним завданням є зниження ризиків, пов’язаних із тривалим використанням ПК, шляхом правильної організації робочого місця, дотримання режиму праці та відпочинку, а також забезпечення електро- і пожежної безпеки.

## 4.2 Загальні вимоги безпеки з охорони праці для користувачів ПК

Робота користувача персонального комп’ютера повинна бути організована так, щоб забезпечити безпечні умови праці, зменшити вплив шкідливих чинників і запобігти перевтомі. Вимоги до роботи з екранними пристроями передбачають належну організацію робочого місця, використання справного обладнання, дотримання регламентованих перерв і створення безпечного виробничого середовища [19].

До роботи з персональним комп’ютером допускаються користувачі, які ознайомлені з правилами безпечної експлуатації обладнання, вимогами електробезпеки та правилами поведінки у разі аварійної ситуації. Перед початком роботи необхідно перевірити справність комп’ютера, монітора, клавіатури, миші, кабелів живлення, розеток і подовжувачів. Забороняється працювати з обладнанням, яке має пошкоджені кабелі, нестійке підключення або ознаки перегрівання.

Робоче місце користувача ПК має відповідати ергономічним вимогам. Стіл повинен мати достатню площу для розміщення монітора, клавіатури, миші та інших необхідних засобів. Крісло повинно бути стійким, зручним і бажано регульованим за висотою. Положення тіла під час роботи має бути природним: спина пряма, плечі розслаблені, ноги розміщені на підлозі або підставці, кисті рук перебувають у зручному положенні без надмірного згинання.

Екран монітора потрібно розташовувати на відстані приблизно 50–70 см від очей користувача. Верхній край екрана має бути на рівні очей або трохи нижче. Таке розташування зменшує навантаження на очі та шийний відділ хребта. На поверхні екрана не повинно бути відблисків від вікон або світильників, оскільки вони підвищують зорове напруження та сприяють швидкій втомі.

Освітлення робочого місця повинно бути достатнім, рівномірним і не створювати засліплення. Бажано поєднувати природне та штучне освітлення, розташовуючи робоче місце так, щоб світло не потрапляло безпосередньо на екран і не створювало різких тіней. Для роботи з ПК важливими також є параметри

мікроклімату: температура, відносна вологість і швидкість руху повітря. Санітарні норми визначають мікроклімат виробничих приміщень як умови внутрішнього середовища, що впливають на тепловий обмін людини з навколишнім середовищем [20].

Під час роботи за комп’ютером необхідно дотримуватися раціонального режиму праці та відпочинку. Тривалі безперервні періоди роботи слід чергувати з короткими перервами. Під час перерв доцільно виконувати вправи для очей, шиї, спини та кистей рук. Це зменшує статичне навантаження, покращує кровообіг і сприяє відновленню працездатності.

Під час роботи користувачу ПК забороняється:

1. торкатися оголених проводів і пошкоджених кабелів;

2. самостійно розбирати системний блок, монітор або блок живлення;

3. працювати з обладнанням у разі появи запаху гару, іскріння або перегрівання;

4. ставити напої біля комп’ютера та електричних пристроїв;

5. закривати вентиляційні отвори обладнання;

6. перевантажувати електромережу великою кількістю пристроїв;

7. працювати за комп’ютером без перерв протягом тривалого часу.

У разі виникнення несправності необхідно негайно припинити роботу, вимкнути обладнання з електромережі, повідомити відповідальну особу або керівника та не намагатися самостійно ремонтувати електрообладнання без відповідної підготовки. У разі пожежі потрібно відключити електроживлення, скористатися первинними засобами пожежогасіння та діяти відповідно до інструкції з евакуації.

Після завершення роботи необхідно коректно завершити роботу програм, вимкнути комп’ютер і периферійні пристрої, привести робоче місце до порядку та переконатися, що електрообладнання не залишене у небезпечному стані. Дотримання цих вимог забезпечує безпечні умови праці користувача ПК і знижує ризик негативного впливу тривалої роботи за комп’ютером на здоров’я.

### 4.3 Висновки до розділу 4

У розділі розглянуто ризик як кількісну оцінку небезпек, що враховує ймовірність виникнення небезпечної події та тяжкість її наслідків. Визначено основні небезпечні та шкідливі чинники, які можуть впливати на розробника програмного забезпечення під час тривалої роботи за персональним комп’ютером.

Також сформульовано загальні вимоги безпеки з охорони праці для користувачів ПК. Основну увагу приділено правильній організації робочого місця, ергономічному розташуванню обладнання, забезпеченню належного освітлення та мікроклімату, дотриманню режиму праці й відпочинку, а також вимогам електро- та пожежної безпеки. Дотримання цих вимог дає змогу знизити ризики для здоров’я користувача та забезпечити безпечні умови виконання роботи.

# ВИСНОВКИ

У даній кваліфікаційній роботі було досягнуто поставленої мети: розроблено програмне забезпечення інтерактивного Discord-чат-бота для організації кіберспортивних in-house матчів. Створена система автоматизує процес збору гравців, формування команд, підтвердження готовності, збереження історії матчів та ведення внутрішнього рейтингу учасників.

У результаті виконання роботи були отримані такі основні результати та зроблені висновки:

1. Проведено аналіз предметної області організації кіберспортивних матчів та визначено особливості in-house формату. Встановлено, що для локальних ігрових спільнот важливими є швидкий збір гравців, підтримка рольових черг, прозоре формування команд і збереження результатів матчів. Обґрунтовано актуальність створення власного Discord-бота.

2. Проаналізовано існуючі технічні рішення, зокрема FACEIT та InHouseQueue. Визначено, що FACEIT є потужною платформою для публічного матчмейкінгу, але є надлишковою для невеликих Discord-спільнот. InHouseQueue є ближчим аналогом, однак має обмежену гнучкість для адаптації під власну структуру даних, правила рейтингу та алгоритм балансування.

3. Сформовано функціональні та нефункціональні вимоги до програмного забезпечення. До основних функціональних вимог віднесено роботу з рольовими чергами, формування матчу у форматі 5 на 5, балансування команд, ready-check, перегляд історії матчів, оновлення рейтингу та адміністративне керування. До нефункціональних вимог віднесено швидкодію, надійність, зручність використання, можливість розширення та контейнеризоване розгортання.

4. Спроєктовано архітектуру інтерактивного чат-бота. Система складається з Discord-серверу як середовища взаємодії, backend-частини на Python, бази даних PostgreSQL та контейнеризованого середовища запуску Docker / Docker Compose. Для взаємодії з ботом використовуються Discord-команди та повідомлення.

5. Спроєктовано структуру бази даних для збереження інформації про

гравців, рейтинги за ролями, активні черги, створені матчі, учасників матчів, Discord-канали та конфігурацію серверу. Така структура дозволяє зберігати поточний стан системи, історію матчів і рейтингові показники після перезапуску бота.

6. Розроблено алгоритм формування та балансування команд. Для цього використано рейтингову модель TrueSkill, яка враховує оцінку навички гравця та рівень невизначеності цієї оцінки. Рейтинг зберігається окремо для кожної ігрової ролі, що дозволяє точніше оцінювати гравців під час формування складів команд.

7. Реалізовано основні функціональні компоненти чат-бота: керування рольовими чергами, формування матчу, ready-check, інформаційні команди, адміністративні команди, перегляд історії матчів, побудову графіка зміни MMR та оновлення рейтингу після завершення гри. Для зручності користувачів результати команд виводяться у форматі повідомлень Discord.

8. Проведено ручне та автоматизоване тестування програмного забезпечення. Ручне тестування виконувалося в тестовому Discord-сервері, а автоматизовані тести використовувалися для перевірки логіки черг, матчмейкінгу та командного шару. За результатами тестування встановлено, що основні функціональні сценарії працюють коректно.

У розділі з безпеки життєдіяльності та охорони праці було розглянуто фактори, які можуть впливати на користувача під час роботи з комп’ютерною технікою, а також наведено рекомендації щодо безпечної організації робочого місця.

Таким чином, усі поставлені у кваліфікаційній роботі завдання були виконані, а мета роботи – досягнута. Розроблений Discord-чат-бот демонструє працездатність обраного підходу та може використовуватися для автоматизації організації in-house матчів у локальних кіберспортивних спільнотах. Подальший розвиток системи може передбачати додавання сезонів, розширеної статистики, підтримки нових ігрових режимів та гнучкіших налаштувань для різних Discord-серверів.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Guide to the Software Engineering Body of Knowledge (SWEBOK Guide). Version 4.0 / ed. H. Washizaki. IEEE Computer Society, 2024. 411 p.

2. Python 3 Documentation. The Python Standard Library. – Режим доступу: https://docs.python.org/3/library/ – (дата звернення: 30.04.2026).

3. SQLAlchemy Documentation. – Режим доступу: https://docs.sqlalchemy.org/ (дата звернення: 30.04.2026).

4. Pycharm. URL: https://www.jetbrains.com/pycharm/ – (дата звернення: 30.04.2026).

5. Docker Documentation. – Режим доступу: https://docs.docker.com/ – ( дата звернення: 30.04.2026).

6. Docker Documentation. Docker Compose. – Режим доступу: https://docs.docker.com/compose/ – ( дата звернення: 30.04.2026).

7. Discord Developer Documentation. API Reference. – Режим доступу: https://docs.discord.com/developers/reference – (дата звернення: 01.04.2026.).

8. Discord.py: A Python wrapper for the Discord API. – PyPI. – Режим доступу: https://pypi.org/project/discord.py / – ( дата звернення: 30.04.2026).

9. TrueSkill Ranking System. – Microsoft Research. – Режим доступу: https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/ – (дата звернення: 19.05.2026).

10. TrueSkill Documentation. – Python package documentation. – Режим доступу: https://trueskill.org/ – (дата звернення: 17.05.2026).

11. In-House Queue Documentation. – Режим доступу: https://docs.inhousequeue.xyz/ – (дата звернення: 12.05.2026).

12. PostgreSQL Documentation. – The PostgreSQL Global Development Group. – Режим доступу: https://www.postgresql.org/docs/ – (дата звернення: 19.05.2026).

13. Git Documentation. – Режим доступу: https://git-scm.com/doc – (дата звернення: 19.05.2026).

14. Види функціонального та нефункціонального тестування. URL: https://dan-it.com.ua/uk/blog/vidy-funkcionalnogo-i-nefunkcionalnogo-testirovanija/ (дата звернення: 29.05.2026).

15. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі Михалик Д.М., Цуприк Г.Б., Бревус В.М. Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. 45 с.

16. Кодекс законів про працю України : Кодекс України від 10.12.1971 № 322-VIII. URL: https://zakon.rada.gov.ua/laws/show/322-08 (дата звернення: 18.06.2026).

17. ДСТУ ISO 31000:2018. Менеджмент ризиків. Принципи та настанови. Київ : ДП «УкрНДНЦ», 2018. (дата звернення: 18.06.2026).

18. ДСТУ IEC/ISO 31010:2013. Керування ризиком. Методи загального оцінювання ризику. Київ : Мінекономрозвитку України, 2015. 80 с. (дата звернення: 18.06.2026).

19. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин : Правила МОЗ України від 10.12.1998 № 7. URL: https://zakon.rada.gov.ua/rada/show/v0007282-98 (дата звернення: 18.06.2026).

20. Зеркалов Д.В. Безпека життєдіяльності та основи охорони праці. Навчальний посібник. К.: «Основа». 2016. 267 с.

21. Про затвердження Правил пожежної безпеки в Україні : наказ Міністерства внутрішніх справ України від 30.12.2014 № 1417. URL: https://zakon.rada.gov.ua/laws/show/z0252-15 (дата звернення: 18.06.2026).

# ДОДАТКИ

# ДОДАТОК А

Лістинг коду основних компонентів бота

Лістинг А.1 – Реалізація входу гравця до рольової черги

```python
class QueueCog(commands.Cog, name="Queue"):
    """
    Manage your queue status and score games
    """

    def __init__(self, bot: InhouseBot):
        self.bot = bot
        self.players_whose_last_game_got_cancelled = {}
        self.games_getting_scored_ids = set()

    @commands.command()
    @queue_channel_only()
    @doc(f"""
        Adds you to the current channel’s queue for the given role

        To duo queue, add @player role at the end.

        Roles are TOP, JGL, MID, BOT/ADC, and SUP.

        Example:
            {PREFIX}queue SUP
            {PREFIX}queue adc @CoreJJ support
    """)
    async def queue(
        self,
        ctx: commands.Context,
        role: RoleConverter(),
        duo: discord.Member = None,
        duo_role: RoleConverter() = None,
    ):
        jump_ahead = False

        if cancel_timestamp :=
self.players_whose_last_game_got_cancelled.pop(ctx.author.id, None):
            if datetime.now() - cancel_timestamp <
timedelta(hours=1):
                jump_ahead = True

        if not duo:
            game_queue.add_player(
                player_id=ctx.author.id,
                name=ctx.author.display_name,
                role=role,
                channel_id=ctx.channel.id,
                server_id=ctx.guild.id,
```

```
                jump_ahead=jump_ahead,
            )
        else:
            if not duo_role:
                await ctx.send("You need to input a role for your
duo partner")
                return

            duo_validation_message = await ctx.send(
                f"<@{ctx.author.id}> {get_role_emoji(role)} wants to
duo with "
                f"<@{duo.id}> {get_role_emoji(duo_role)}\n"
                f"Press ✅ to accept the duo queue"
            )

            validated, players_who_refused = await
checkmark_validation(
                bot=self.bot,
                message=duo_validation_message,
                validating_players_ids=[duo.id],
                validation_threshold=1,
            )

            if not validated:
                await ctx.send(f"<@{ctx.author.id}>: Duo queue was
refused")
                return

            game_queue.add_duo(
                first_player_id=ctx.author.id,
                first_player_role=role,
                first_player_name=ctx.author.display_name,
                second_player_id=duo.id,
                second_player_role=duo_role,
                second_player_name=duo.display_name,
                channel_id=ctx.channel.id,
                server_id=ctx.guild.id,
                jump_ahead=jump_ahead,
            )

        await self.run_matchmaking_logic(ctx=ctx)
        await queue_channel_handler.update_queue_channels(
            bot=self.bot,
            server_id=ctx.guild.id,
        )
```

Лістинг А.2 – Запуск матчмейкінгу та ready-check

```python
async def run_matchmaking_logic(self, ctx: commands.Context):
    """
```

```
    Runs the matchmaking logic in the channel defined by the
context.
    """
    queue = game_queue.GameQueue(ctx.channel.id)
    game = matchmaking_logic.find_best_game(queue)

    if not game:
        return

    elif game and game.matchmaking_score < 0.2:
        embed = game.get_embed(
            embed_type="GAME_FOUND",
            validated_players=[],
            bot=self.bot,
        )

        ready_check_message = await ctx.send(
            content=game.players_ping,
            embed=embed,
            delete_after=60 * 15,
        )

        game_queue.start_ready_check(
            player_ids=game.player_ids_list,
            channel_id=ctx.channel.id,
            ready_check_message_id=ready_check_message.id,
        )

        await queue_channel_handler.update_queue_channels(
            bot=self.bot,
            server_id=ctx.guild.id,
        )

        try:
            ready, players_to_drop = await checkmark_validation(
                bot=self.bot,
                message=ready_check_message,
                validating_players_ids=game.player_ids_list,
                validation_threshold=10,
                game=game,
            )
        except Exception as e:
            self.bot.logger.error(e)
            game_queue.cancel_ready_check(
                ready_check_id=ready_check_message.id,
                ids_to_drop=game.player_ids_list,
                server_id=ctx.guild.id,
            )

            await ctx.send(
                "There was a bug with the ready-check message, "
                "all players have been dropped from queue"
            )
```

```
            await queue_channel_handler.update_queue_channels(
                bot=self.bot,
                server_id=ctx.guild.id,
            )
            return

        if ready is True:
            game_queue.validate_ready_check(ready_check_message.id)

            with session_scope() as session:
                session.expire_on_commit = False
                game = session.merge(game)

            queue_channel_handler.mark_queue_related_message(
                await
ctx.send(embed=game.get_embed("GAME_ACCEPTED"))
            )

            await create_voice_channels(ctx, game)

        elif ready is False:
            game_queue.cancel_ready_check(
                ready_check_id=ready_check_message.id,
                ids_to_drop=players_to_drop,
                channel_id=ctx.channel.id,
            )

            await ctx.send(
                "A player cancelled the game and was removed from
the queue. "
                "All other players have been put back in the queue"
            )

            await self.run_matchmaking_logic(ctx)

        elif ready is None:
            game_queue.cancel_ready_check(
                ready_check_id=ready_check_message.id,
                ids_to_drop=players_to_drop,
                server_id=ctx.guild.id,
            )

            await ctx.send(
                "The check timed out and players who did not answer
"
                "have been dropped from all queues"
            )

            await self.run_matchmaking_logic(ctx)

    elif game and game.matchmaking_score >= 0.2:
        await ctx.send(
```

```
            f"The best match found had a side with a "
            f"{(.5 + game.matchmaking_score) * 100:.1f}% predicted
winrate "
            f"and was not started"
        )
```

Лістинг А.3 – Оцінювання очікуваної ймовірності перемоги команди

```python
import itertools
import math
import trueskill

from inhouse_bot.database_orm import Game


def evaluate_game(game: Game) -> float:
    """
    Returns the expected win probability of the blue team over the
red team.
    """

    blue_team_ratings = [
        trueskill.Rating(
            mu=p.trueskill_mu,
            sigma=p.trueskill_sigma,
        )
        for p in game.teams.BLUE
    ]

    red_team_ratings = [
        trueskill.Rating(
            mu=p.trueskill_mu,
            sigma=p.trueskill_sigma,
        )
        for p in game.teams.RED
    ]

    delta_mu = (
        sum(r.mu for r in blue_team_ratings)
        - sum(r.mu for r in red_team_ratings)
    )

    sum_sigma = sum(
        r.sigma ** 2
        for r in itertools.chain(blue_team_ratings,
red_team_ratings)
    )

    size = len(blue_team_ratings) + len(red_team_ratings)

    denominator = math.sqrt(
```

```
        size * (trueskill.BETA * trueskill.BETA) + sum_sigma
    )

    ts = trueskill.global_env()

    return ts.cdf(delta_mu / denominator)
```

Лістинг А.4 – Пошук найкращого складу команд

```python
def find_best_game(queue: GameQueue, game_quality_threshold=0.1) ->
Optional[Game]:
    """
    Finds the best possible game from the current queue.
    """

    for role_queue in queue.queue_players_dict.values():
        if len(role_queue) < 2:
            return None

    best_game = None

    for players_threshold in range(10, len(queue) + 1):
        best_game = find_best_game_for_queue_players(
            queue.queue_players[:players_threshold]
        )

        if best_game and best_game.matchmaking_score <
game_quality_threshold:
            return best_game

    return best_game


def find_best_game_for_queue_players(queue_players:
List[QueuePlayer]) -> Game:
    """
    Iterates over possible role-based team compositions.
    """

    role_permutations = []

    for role in roles_list:
        role_permutations.append(
            [
                queue_player
                for queue_player in itertools.permutations(
                    [qp for qp in queue_players if qp.role == role],
                    2,
                )
            ]
        )
```

```
    best_score = 1
    best_game = None

    for team_composition in itertools.product(*role_permutations):
        shuffle = bool(random.getrandbits(1))

        queue_players_dict = {
            (
                "BLUE" if bool(tuple_idx) == shuffle else "RED",
                roles_list[role_idx],
            ): queue_players_tuple[tuple_idx]
            for role_idx, queue_players_tuple in
enumerate(team_composition)
            for tuple_idx in (0, 1)
        }

        duos_not_in_same_team = False

        for team_tuple, qp in queue_players_dict.items():
            if qp.duo_id is not None:
                try:
                    next(
                        duo_qp
                        for duo_team_tuple, duo_qp in
queue_players_dict.items()
                        if duo_team_tuple[0] == team_tuple[0]
                        and duo_qp.player_id == qp.duo_id
                    )
                except StopIteration:
                    duos_not_in_same_team = True
                    continue

        if duos_not_in_same_team:
            continue

        players = {k: qp.player for k, qp in
queue_players_dict.items()}

        if len(set(players.values())) != 10:
            continue

        game = Game(players)

        if game.matchmaking_score < best_score:
            best_game = game
            best_score = game.matchmaking_score

            if best_score < 0.01:
                break

    return best_game
```

Лістинг А.5 – Оновлення рейтингу TrueSkill після завершення матчу

```python
import trueskill

from inhouse_bot.database_orm import session_scope
from inhouse_bot.database_orm import Game
from inhouse_bot.common_utils.get_last_game import get_last_game


def update_trueskill(game: Game, session):
    """
    Updates the player ratings based on the game result.
    """

    blue_team_ratings = {
        participant.player.ratings[participant.role]:
trueskill.Rating(
            mu=participant.trueskill_mu,
            sigma=participant.trueskill_sigma,
        )
        for participant in game.teams.BLUE
    }

    red_team_ratings = {
        participant.player.ratings[participant.role]:
trueskill.Rating(
            mu=participant.trueskill_mu,
            sigma=participant.trueskill_sigma,
        )
        for participant in game.teams.RED
    }

    if game.winner == "BLUE":
        new_ratings = trueskill.rate([blue_team_ratings,
red_team_ratings])
    else:
        new_ratings = trueskill.rate([red_team_ratings,
blue_team_ratings])

    for ratings in new_ratings:
        for player_rating in ratings:
            player_rating.trueskill_mu = ratings[player_rating].mu
            player_rating.trueskill_sigma =
ratings[player_rating].sigma
            session.merge(player_rating)


def score_game_from_winning_player(player_id: int, server_id: int):
    """
    Scores the last game of the player on the server as a win.
    """
```

```
    with session_scope() as session:
        game, participant = get_last_game(player_id, server_id,
session)

        game.winner = participant.side

        update_trueskill(game, session)
```

# ДОДАТОК Б

## Результати тестування програмного забезпечення

Таблиця Б.1 – Результати ручного та автоматизованого тестування чат-бота

| № | Тестовий сценарій | Очікуваний результат | Фактичний результат | Статус |
|---|---|---|---|---|
| 1 | Додавання гравця до рольової черги | Гравець додається до черги з вибраною роллю | Гравець успішно відображається у черзі | Пройдено |
| 2 | Вихід гравця із черги | Запис гравця видаляється з черги | Гравець успішно видаляється зі списку очікування | Пройдено |
| 3 | Перегляд стану черги | Бот показує актуальний список гравців за ролями | Черга відображається коректно | Пройдено |
| 4 | Автоматичне тестове заповнення черги | Черга заповнюється тестовими учасниками | Бот формує тестову чергу для перевірки матчмейкінгу | Пройдено |
| 5 | Формування матчу за недостатньої кількості гравців | Матч не створюється | Бот очікує нових гравців у черзі | Пройдено |
| 6 | Формування матчу за наявності всіх ролей | Створюються дві команди по 5 гравців | Команди формуються відповідно до ролей | Пройдено |
| 7 | Перевірка унікальності гравців у матчі | Один гравець не може потрапити в матч двічі | Дублювання гравців не виявлено | Пройдено |
| 8 | Перевірка роботи duo-зв’язків | Гравці з duo-зв’язком потрапляють в одну команду | Некоректні варіанти складу відкидаються | Пройдено |

Продовження таблиці Б.1

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 9 | Ready-check після створення матчу | Бот очікує підтвердження від усіх учасників | Повідомлення ready-check працює коректно | Пройдено |
| 10 | Відмова або відсутність підтвердження | Матч не переходить в активний стан | Система не запускає матч без підтвердження | Пройдено |
| 11 | Перегляд історії матчів | Бот показує попередні матчі гравця | Історія матчів відображається коректно | Пройдено |
| 12 | Перегляд графіка MMR | Бот формує графік зміни рейтингу | Графік будується та відображається коректно | Пройдено |
| 13 | Перегляд профілю гравця | Бот показує інформацію про гравця та його статистику | Профіль гравця відображається у Discord | Пройдено |
| 14 | Фіксація переможця | Результат матчу зберігається | Переможець успішно фіксується у системі | Пройдено |
| 15 | Оновлення рейтингу після матчу | Значення рейтингу гравців змінюються | Рейтинг оновлюється після завершення гри | Пройдено |
| 16 | Адміністративне очищення черги | Черга очищується адміністратором | Усі записи черги видаляються | Пройдено |
| 17 | Адміністративне видалення гравця з черги | Конкретний гравець видаляється з черги | Гравець успішно видаляється адміністратором | Пройдено |
| 18 | Скасування активного матчу | Матч переходить у скасований стан | Активний матч скасовується командою адміністратора | Пройдено |
| 19 | Перезапуск бота після запису даних | Дані залишаються доступними після перезапуску | Дані про гравців, матчі та рейтинг зберігаються | Пройдено |
| 20 | Автоматизована перевірка роботи черги | Черга коректно додає та видаляє гравців | Тести test_queue_handler.py виконуються успішно | Пройдено |

Кінець таблиці Б.1

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 21 | Автоматизована перевірка матчмейкінгу | Алгоритм формує коректний склад матчу | Тести test_matchmaking_logic.py виконуються успішно | Пройдено |
| 22 | Автоматизована перевірка командного шару бота | Обробники команд викликають потрібну логіку | Тести test_cog.py виконуються успішно | Пройдено |
| 23 | Перевірка спільних тестових налаштувань | Фікстури та тестові дані доступні для різних тестів | Файл conftest.py забезпечує підготовку тестового середовища | Пройдено |