**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**
**імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.8, 004.738.5, 37.02, 004.738.5:37.02, 004.92:37.02

«До захисту допущено»
Завідувач кафедри
_____ Едуард ЖАРІКОВ
«___»_____2025 р.

# Магістерська дисертація

**на здобуття ступеня магістра**

**за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»**

**зі спеціальності 121 «Інженерія програмного забезпечення»**

**на тему: «Платформа для проведення змагань зі штучного інтелекту з підтримкою багатомовного програмування та візуалізації ігрового процесу»**

Виконав (-ла):
студент (-ка) ІІ курсу, групи ІП-43мп
Євтушок Олег Михайлович _____

Керівник:
доцент, к.т.н., доц. Фіногенов Олексій Дмитрович _____

Рецензент:
доцент каф. ІСТ, к.т.н., доц. Катерина Мамедова _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент (-ка) _____

Київ – 2025 року

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Едуард ЖАРІКОВ

«___»_____2025р.

**ЗАВДАННЯ**
**на магістерську дисертацію студенту**

**Євтушку Олегу Михайловичу**

1. Тема дисертації «Платформа для проведення змагань зі штучного інтелекту з підтримкою багатомовного програмування та візуалізації ігрового процесу», науковий керівник дисертації к.т.н., доц. Фіногенов Олексій Дмитрович, затверджені наказом по університету від «06» листопада 2025 р. № 4841-с

2. Термін подання студентом дисертації «15» грудня 2025 р.

3. Об’єкт дослідження – ПЗ для організації та проведення змагань алгоритмів у навчальних і дослідницьких цілях

4. Предмет дослідження – методи й засоби реалізації платформ із підтримкою багатомовного виконання та візуалізації

5. Перелік завдань, які потрібно розробити – дослідити підходи до створення змагальних платформ, візуалізації та безпеки виконання коду; спроектувати модульну архітектуру з підтримкою кількох мов програмування; реалізувати підсистему управління турнірами (матчі, сітки, рейтинги); розробити веб-інтерфейс і механізми інтерактивної візуалізації результатів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 3 плакати

7. Орієнтовний перелік публікацій – одна публікація

8. Консультанти розділів дисертації

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| | | | |

9.Дата видачі завдання «1» вересня 2025 р.

Календарний план

| № з/п | Назва етапів виконання магістерської дисертації | Термін виконання | Примітка |
|---|---|---|---|
| 1 | Видача завдання | 01.09.2025 | |
| 2 | Аналіз предметної області та платформ AI-змагань | 05.09.2025 | |
| 3 | Формування структури магістерської роботи | 15.09.2025 | |
| 4 | Огляд існуючих рішень багатомовного виконання коду | 25.09.2025 | |
| 5 | Розроблення концепції архітектури платформи | 05.10.2025 | |
| 6 | Проєктування модулів турнірного менеджера | 15.10.2025 | |
| 7 | Реалізація прототипу виконання ботів | 25.10.2025 | |
| 8 | Розроблення системи візуалізації ігрового процесу | 05.11.2025 | |
| 9 | Інтеграція та тестування компонентів системи | 20.11.2025 | |
| 10 | Виконання експериментальних досліджень | 23.11.2025 | |
| 11 | Оформлення пояснювальної записки | 25.11.2025 | |
| 12 | Подання дисертації на попередній захист | 26.11.2025 | |
| 13 | Подання дисертації на захист | 15.12.2025 | |

Студент Олег ЄВТУШОК

Науковий керівник Олексій ФІНОГЕНОВ

# РЕФЕРАТ

Розмір пояснювальної записки – 191 аркуш, містить 13 ілюстрацій, 8 таблиць, 1 додаток, 22 посилання на джерела.

**Актуальність теми.** У роботі розглянуто проблему відсутності універсальної платформи для проведення змагань алгоритмів штучного інтелекту з підтримкою різних мов програмування, детальними метриками та інтерактивною візуалізацією. Запропоноване рішення спрямоване на гейміфікацію навчального процесу й дослідницьких експериментів, забезпечуючи безпечне виконання коду через Docker-контейнери та прозоре відтворення ігрових сценаріїв.

**Мета дослідження.** Розробити веб-платформу для організації турнірів між алгоритмами штучного інтелекту з підтримкою кількох мов програмування та інтегрованою візуалізацією ігрового процесу.

Об’єкт дослідження: програмне забезпечення для організації й проведення змагань алгоритмів штучного інтелекту у навчальних і дослідницьких цілях.

Предмет дослідження: методи та засоби реалізації змагальних платформ із підтримкою багатомовного виконання коду й візуалізації алгоритмів.

Для досягнення мети **сформульовано такі завдання**:

– дослідити сучасні підходи до створення змагальних платформ, механізми візуалізації алгоритмів і способи безпечного виконання коду;

– розробити модульну архітектуру, що підтримує підключення різних мов і нових ігрових рушіїв через інтерфейси GameEngine/BotExecutor;

– створити систему управління турнірами з реєстрацією, запуском, повторним прогоном і збереженням реплеїв;

– побудувати веб-інтерфейс з інтерактивною візуалізацією кроків гри, легендами та метриками;

– провести тестування й задокументувати результати, включно зі Swagger-доступом до API та моніторингом Docker-контейнерів.

**Наукова новизна:** набули подальшого розвитку методи гейміфікації навчання розробці алгоритмів: запропонована платформа підтримує багатомовне виконання (Python, Java, можливість розширення), командні режими й покрокові реплеї з детальними тлумаченнями показників, що розширює інструментарій для практичного вивчення штучного інтелекту.

**Практичне значення** полягає в тому, що розроблена система може застосовуватись у навчальних закладах для об’єктивного оцінювання студентських рішень, у внутрішніх хакатонах компаній, а також у дослідницьких лабораторіях для швидкого створення симуляцій і змагань між агентами, забезпечуючи безпечне виконання коду та візуалізацію результатів.

**Зв’язок з науковими програмами, планами, темами**. Робота виконувалась на кафедрі інформатики та програмного забезпечення (назва кафедри/університету згідно з офіційними даними).

**Апробація**. Основні результати представлені на VIII Міжнародній науково-практичній конференції молодих учених та студентів «Інженерія програмного забезпечення і передові інформаційні технології (SoftTech-2025)».

**Публікації**. Матеріали дослідження опубліковані в тезах конференції SoftTech-2025 (секція кафедри інформатики та програмної інженерії).

Ключові слова: ЗМАГАЛЬНІ ПЛАТФОРМИ, ШТУЧНИЙ ІНТЕЛЕКТ, ДОКЕР-ІЗОЛЯЦІЯ, МОДУЛЬНА АРХІТЕКТУРА, ВІЗУАЛІЗАЦІЯ.

## ABSTRACT

The explanatory note comprises 191 pages, including 13 figures, 8 tables, 1 appendix, and 22 references.

**Relevance of the topic**. The work addresses the lack of a universal platform for competitions between artificial intelligence algorithms that simultaneously supports multiple programming languages, provides detailed metrics, and offers interactive visualization. The proposed solution aims to gamify both the educational process and research experiments by ensuring safe code execution via Docker containers and transparent reproduction of game scenarios.

**Research aim**. To develop a web platform for organizing tournaments between artificial intelligence algorithms with support for several programming languages and integrated visualization of the game process.

Object of study: software for organizing and conducting competitions between artificial intelligence algorithms for educational and research purposes.

Subject of study: methods and tools for implementing competitive platforms that support multilingual code execution and algorithm visualization.

To achieve the aim, the **following tasks** were formulated:

– investigate contemporary approaches to building competitive platforms, mechanisms for algorithm visualization, and methods for secure code execution;

– design a modular architecture that supports connecting multiple languages and new game engines through the GameEngine/BotExecutor interfaces;

– create a tournament management system with registration, launch, re-runs, and storage of replays;

– build a web interface with interactive visualization of game steps, legends, and metrics;

– conduct testing and document the results, including Swagger access to the API and monitoring of Docker containers.

**Scientific novelty**. Methods for gamifying the learning of algorithm development have been advanced: the proposed platform supports multilingual execution (Python, Java, with extensibility), team modes, and step-by-step replays with

detailed metric annotations, thereby expanding the toolkit for the practical study of artificial intelligence.

**Practical significance**. The developed system can be used in educational institutions for objective assessment of student solutions, in internal company hackathons, and in research laboratories for rapid creation of simulations and competitions between agents—while ensuring safe code execution and result visualization.

**Relation to research programs, plans, and topics**. The work was carried out at the Department of Computer Science and Software Engineering (department/ university name according to official records).

**Approbation**. The main results were presented at the VIII International Scientific and Practical Conference of Young Scientists and Students “Software Engineering and Advanced Information Technologies (SoftTech-2025)”.

**Publications**. The research materials were published in the proceedings of SoftTech-2025 (section of the Department of Computer Science and Software Engineering).

Keywords: COMPETITIVE PLATFORMS, ARTIFICIAL INTELLIGENCE, DOCKER-BASED ISOLATION, MODULAR ARCHITECTURE, VISUALIZATION.

ЗМІСТ

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

| | | |
|---|---|---|
| API | – | Програмний інтерфейс для взаємодії між компонентами системи. |
| CPU-квант | – | Одиниця обліку процесорного часу для чесного тайтмінгу ходів у матчі. |
| gRPC | – | Двійковий протокол віддалених викликів з чітким фреймінгом повідомлень. |
| OOM | – | Out Of Memory, стан коли процес перевищив дозволений обсяг пам’яті. |
| OOM-killer | – | Механізм ядра, що завершує процес, який перевищив доступну пам’ять. |
| REST API | – | Ресурсно-орієнтований веб-інтерфейс зі стандартизованими CRUD-операціями. |
| SDK | – | Набір бібліотек та інструментів для інтеграції клієнта з платформою. |
| SSE | – | Server-Sent Events, канал односторонніх push-повідомлень від сервера для live-оновлень. |
| STDIN | – | Стандартний потік вводу, може використовуватися як канал передачі даних ботом. |
| STDOUT | – | Стандартний потік виводу, використовується ботом для повернення результатів судді. |
| WASI | – | WebAssembly System Interface, системний інтерфейс для безпечного виконання WebAssembly-модулів. |
| WebAssembly | – | Детермінований байткодовий формат для швидкого й ізольованого виконання коду. |
| WebSocket | – | Двосторонній постійний зв'язок клієнт–сервер для реального часу. |

# ВСТУП

Штучний інтелект перестав бути виключно лабораторною дисципліною: індустрія масово впроваджує автономні агенти, а освітні заклади перетворюють підготовку з алгоритмів на прикладні курси з акцентом на змагання, симуляції та аналіз поведінки моделей у реальному середовищі. Провідні платформи на кшталт Kaggle, CodinGame або Halite демонструють успішні приклади гейміфікації, проте вони або орієнтовані на обмежені сценарії (офлайн оцінювання датасетів, задачі класичного алгоритмічного програмування), або потребують складної інфраструктури, що робить їх малопридатними для локальних освітніх курсів. Світова тенденція полягає у створенні універсальних інструментів, які поєднують безпечне виконання коду, багатомовність і наочні реплеї, але на практиці викладачі й дослідники часто змушені розробляти власні “саморобні” рішення без єдиного стандарту. Саме ця прогалина – відсутність платформи, що одночасно підтримує різні мови програмування, прозоре суддівство, Docker-ізоляцію й інтерактивну візуалізацію – і визначає актуальність даної роботи.

Проблемна ситуація має дві складові. З одного боку, змагання з алгоритмів потребують чесного середовища: усі учасники повинні запускатися в однакових ресурсних умовах, а система – фіксувати таймаути, помилки та логіку рішень. З іншого боку, навчальний процес вимагає видимості: викладачам необхідно демонструвати, як алгоритм рухається полем, де він припускається помилки, як впливають штрафи та повільні дії. Існуючі платформи або фокусуються на одному з цих аспектів, або взагалі не підтримують багатомовність та реплеї. Відповідно, стає необхідним рішення, що забезпечить універсальність і гнучкість, дозволяючи початківцям і досвідченим студентам змагатися на рівних, а викладачам – отримувати об’єктивний інструментарій для оцінювання.

Мета дослідження полягає у розробці веб-платформи для організації турнірів між алгоритмами штучного інтелекту з підтримкою кількох мов програмування, інтегрованою візуалізацією ігрового процесу та гарантією безпечного виконання коду в Docker-контейнерах.

Об’єкт дослідження – програмне забезпечення для організації та проведення змагань алгоритмів штучного інтелекту у навчальних і дослідницьких цілях.

Предмет дослідження – методи й засоби реалізації змагальних платформ із багатомовним виконанням коду та інтерактивною візуалізацією алгоритмів. Мета досягається через вирішення конкретних завдань:

- дослідити існуючі платформи, механізми візуалізації та заходи безпеки виконання коду (визначити, які підходи вже реалізовано й де існують прогалини);
- розробити модульну архітектуру з інтерфейсами GameEngine/BotExecutor, що дає змогу додавати нові ігри та мови без зміни ядра;
- створити систему управління турнірами (створення, запуск, повторний прогін, логування, підбиття підсумків);
- побудувати веб-інтерфейс із реплеєм, легендами, таблицями метрик і можливістю приховувати/розгортати інформаційні блоки;
- протестувати платформу на прикладі stress test (8 ботів), задокументувати сценарії, API та роботу контейнерів.

Наукова новизна полягає у подальшому розвитку методів гейміфікації навчання алгоритмів: інтегровано багатомовне виконання (Python і Java з перспективою розширення), командні режими, покрокові реплеї з поясненими метриками й штрафами, що дозволяє викладачам і студентам не лише “здавати” код, а й розуміти поведінку моделей у змаганнях.

Практичне значення полягає в тому, що розроблена платформа може використовуватись у курсах з алгоритмів і ШІ, студентських гуртках, корпоративних хакатонах чи дослідницьких експериментах як універсальний інструмент для змагань, аналізу та мотивації. Вона робить цикл “гіпотеза – експеримент – інтерпретація” швидким і прозорим: боти виконуються у Docker-контейнерах із єдиними ресурсними лімітами, результати зберігаються разом із журналами й реплеями, а викладач може в реальному часі бачити хід гри та

пояснювати студентам, як певна стратегія вплинула на рахунок.

Робота пов’язана з попередніми дослідженнями кафедри (курси з інженерії програмного забезпечення, лабораторні з алгоритмів), а також доповнює існуючі інструменти для практичних занять з ШІ. Під час написання було проаналізовано рішення провідних платформ (Kaggle Competitions, CodinGame, Halite), визначено їх сильні сторони та обмеження, після чого обрано підхід із модульною архітектурою, Docker-ізоляцією та повною локалізацією українською. Таким чином, платформа заповнює нішу навчальних і дослідницьких змагань, де потрібна максимальна інженерна контрольованість: від валідації коду до візуального пояснення дій.

# РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Роль AI-платформ у навчанні й дослідженнях

Змагальні платформи для штучного інтелекту перетворилися на робочі «лабораторії» прикладної інформатики, де алгоритми перевіряють один одного не на слайдах, а на полі гри. В освітньому процесі вони виконують роль каталізатора: змушують переходити від декларацій до вимірюваних результатів, а у дослідженнях забезпечують відтворюваний полігон для порівняння гіпотез, методик і реалізацій. Для проєкту ці платформи тим більш центральні, адже універсальність (багатомовне виконання) та візуалізація ігрового процесу безпосередньо визначають якість навчального досвіду і валідність експериментів. Освітній вимір починається з правильно організованого циклу «гіпотеза - реалізація - матч - аналіз». Гейміфікація тут не є самоціллю: вона підтримує дисципліну експерименту. Коли студент бачить, як конкретний евристичний трюк змінює траєкторію бота у грі, зв’язок між теорією та практикою стає відчутним. Платформа з живою телеметрією та «replay»-файлами дозволяє повернутися до будь-якого кроку епізоду, перевірити логіку ухвалення рішень, зіставити стани середовища і дії агента власне те, що робить навчання інженерним, а не суто описовим.

Ключовий компонент навчальної цінності - багатомовність виконання. Студенти входять у курс з різними стартовими наборами інструментів: комусь природніше Python, комусь - C++ чи JavaScript. Єдина платформа з уніфікованим протоколом «бот - суддя» та тонкими мовними адаптерами знімає зайві бар’єри, концентруючи увагу на алгоритмах і стратегіях, а не на інструментальних деталях середовища. Для викладача це означає можливість формувати змішані групи без примусу до єдиної мови, не жертвуючи вимогами до чесності та відтворюваності матчів. Важливою складовою є привчання до обмежень ресурсів і правил «чистої гри». Sandbox-ізоляція, ліміти CPU/RAM/часу, зафіксовані версії залежностей та профілі системних викликів навчають проєктувати рішення з урахуванням продуктивності й безпеки. У підсумку формується набір навичок, релевантних реальним інженерним контекстам:

профілювання, робота з логами, контроль побічних ефектів, аналіз деградацій під навантаженням. Це також дає викладачеві інструменти аудиту й запобігання шахрайству без перетворення курсу на «гонку скриптів». У дослідницькому вимірі змагальні платформи виступають стандартом експериментальної дисципліни. Контрольовані середовища (від простих board-ігор до кастомізованих симуляцій), фіксовані «seeds», детерміновані конфігурації й артефакти матчів забезпечують відтворюваність результатів. Порівняння бенчмарків та підходів виходить за межі точкових метрик: стає можливим аналіз поведінки агентів, стабільності стратегій, чутливості до параметрів і помилок сприйняття. Саме інтегрована візуалізація, потік подій і офлайн «replay» робить цей аналіз не лише можливим, а й оперативним.

Спільнотний аспект це ще одна причина, чому такі платформи важливі. Публічні рейтинги, відкриті «replay»-каталоги, обмін базовими агентами та «reference-суддями» створюють колективну пам’ять курсу чи лабораторії. Рефакторинг стратегій на основі «кращих відповідей» і повторне використання компонентів (наприклад, модулів навігації чи оцінювачів стану) прискорюють навчання та інновації. Платформа стає місцем спільної інженерної роботи, а не лише тестовим стендом.

З погляду індустріальної придатності, досвід змагальних проектів добре корелює з практичними вимогами роботодавців: інтерфейси між сервісами (протоколи взаємодії), обмеження середовища виконання, продумане логування, спостережуваність системи, A/B-експерименти над стратегіями, безперервна інтеграція та розгортання. Рейтинги і турнірні таблиці тут радше побічний продукт; ціннішими є артефакти відлагодження та аргументованість рішень, що формуються протягом навчання та досліджень.

Нарешті, змагальні AI-платформи відіграють роль інструмента демократизації доступу до якісної освіти[1]. Єдиний веб-інтерфейс із підтримкою кількох мов програмування, стандартними «replay»-форматами і прозорими правилами суддівства дозволяє залучати широкі аудиторії - від шкільних гуртків до університетських курсів та відкритих дослідницьких

ініціатив. У результаті формується спільна мова опису експериментів, що полегшує перенесення напрацювань між курсами, командами і закладами. У контексті цієї роботи роль платформи подвійна. З одного боку, це навчальна інфраструктура, спроектована під реальні освітні сценарії: багатомовні сабміти, чесне суддівство, інтерактивна візуалізація. З іншого - дослідницький полігон, де перевіряються гіпотези щодо стратегій, інструментів ізоляції і протоколів взаємодії «бот-суддя». Така подвійність і обумовлює акценти всієї роботи: універсальність виконання, прозора телеметрія та відтворюваність - не додатки, а конструктивні елементи системи. Порівняння особливостей платформ у різних аспектах можна побачити у таблиці 1.1.

Таблиця 1.1 – Порівняння цінності змагальних платформ для освіти та досліджень

| **Аспект** | **Значення для освіти** | **Значення для досліджень** |
|---|---|---|
| Багатомовність | Зменшує вхідний бар’єр, студенти працюють своєю мовою | Дозволяє тестувати алгоритми в різних рантаймах |
| Live-візуалізація | Підвищує розуміння алгоритмів, сприяє мотивації | Дає можливість аналізу agent-based моделей |
| Replay-артефакти | Використання у семінарах та розборах | Відтворюваність експериментів |
| Sandbox-безпека | Контроль академічної доброчесності | Стандартні умови для порівняння |
| Рейтинг/турніри | Мотивація та гейміфікація курсів | Дані для оцінки стабільності стратегій |

## 1.2 Проблема універсальності та потреба у візуалізації

Сучасний ландшафт змагальних AI-платформ поляризований між системами, орієнтованими на класичне алгоритмічне програмування, і середовищами, що перевіряють моделі на статичних датасетах. Для освітніх і дослідницьких сценаріїв, де «гра» це послідовність рішень агента в динамічному середовищі, бракує універсального рішення, здатного поєднати безпечне

багатомовне виконання користувацького коду з інтерактивною, відтворюваною візуалізацією матчів. Відсутність такого «спільного знаменника» породжує цілу низку бар'єрів: від фрагментації підтримки мов і різнорідності протоколів обміну до неможливості спостерігати за логікою ухвалення рішень у реальному часі. У результаті курси вимушено спрощують завдання до офлайн-перевірок, а лабораторії витрачають ресурси на підтримку власних, малосумісних пайплайнів запуску та візуалізації. Проблема універсальності починається з багатомовності як такої. Студенти та дослідники працюють у різних екосистемах: Python приваблює швидкістю прототипування й насиченою науковою екосистемою; C/C++ дає контроль над продуктивністю; Java чи C# поширені у корпоративних середовищах; JavaScript - у веб-інтеграціях та швидких UI-демонстраторах. Більшість платформ підтримують обмежений піднабір мов або вимагають важких обгорток довкола кожної задачі. Немає стандартизованого, тонкого інтерфейсу, який дозволяв би писати бота будь-якою з перелічених мов і, не змінюючи логіки гри, підключати його до єдиного суддівського двигуна. Це проявляється у трьох практичних наслідках: по-перше, високі початкові витрати на інтеграцію нової мови; по-друге, труднощі з порівнянням реалізацій через відмінні механізми запуску й обмеження середовища; по-третє, зниження прозорості експериментів - одна й та сама стратегія в різних мовних середовищах поводиться по-різному через дрібні деталі інтерфейсу.

Друга площина це технічні виклики виконання. Багатомовність означає змішання інтерпретованих і компільованих підходів, різних моделей керування пам'яттю й потоками, різних часових профілів. Якщо не уніфікувати протокол «бот-суддя», нюанси транспорту (STDIN/STDOUT із рядковим протоколом, локальний сокет, gRPC) і модель синхронізації кроків, отримуємо хаотичну систему, де чесність і відтворюваність матчів забезпечити складно. Затримки на компіляцію, неоднакові старти JVM чи warm-up JIT, різні режими буферизації вводу-виводу можуть непомітно впливати на результат: один бот втрачає хід через тайм-аут, інший - отримує приховану перевагу. Відтак постає потреба у чітких SLA для часу на хід і загального бюджету матчу, в єдиній політиці

синхронізації (покрокова, з бар'єром на зміну стану) і в декларованих, вимірюваних затратах на підготовку середовища (компіляція, завантаження модулів, ініціалізація рантайму).

Третя, і часто недооцінена площина - безпека і чесність виконання. Освітній контекст накладає вимоги sandbox-ізоляції, які не є опціональними: потрібні обмеження CPU/RAM/часу, профілювання системних викликів (seccomp/AppArmor), зачинена мережа або контрольовані allow-list маршрути, читаючий доступ лише до білого списку файлів, фіксація версій залежностей. Без цього будь-яка порівнянність стратегій руйнується: боти можуть почати кешувати заборонені артефакти, «підглядати» у файловій системі або відправляти телеметрію в зовнішній інтернет[2]. Універсальність платформи тут означає однаковий рівень ізоляції для всіх мов і рівноправні політики обліку часу: наприклад, обмеження за реально відпрацьованими квантами CPU, а не за стінним часом, що може бути різно впливовим для різних рантаймів.

Четверта компонента проблеми - візуалізація ігрового процесу. Багато платформ обмежуються табличним результатом або просто логами. У навчанні та дослідженнях цього категорично недостатньо. Потрібен живий потік подій із матчу (дії агентів, зміни стану середовища, оцінювальні метрики, таймінги), який можна перетворити на інтерактивний перегляд у браузері; потрібен також стандартизований офлайн-формат «replay», щоб відтворювати вже зіграні матчі: із паузами, перемоткою, накладанням пояснювальних шарів. Саме вбудована візуалізація перетворює платформу з «чорної скриньки» на навчальний інструмент: демонструє евристики, виявляє патології стратегій, дозволяє викладачеві задавати «контрольні точки» і обговорювати ключові моменти гри.

Архітектурна перспектива підсвічує ще одну грань універсальності - модульність і розширюваність. Платформа має дозволяти додавати нові мови через «мовні адаптери» без переробки ядра; підключати нові ігрові середовища як плагіни з чіткими контрактами; масштабувати підсистему виконання горизонтально (пули воркерів) і відокремлювати суддівський двигун від фронтенда, черг і сховищ артефактів[3]. Така декомпозиція підвищує інженерну

придатність: кожен компонент тестується незалежно, спостережуваність і логування стають одноманітними, а оновлення - безпечними. У протилежному випадку платформа ризикує перерости у моноліт, де додавання ще однієї мови чи типу матчу перетворюється на ризикований рефакторинг. Проблема універсальності тісно пов’язана з відтворюваністю експериментів. Якщо ми не фіксуємо версії інструментів і бібліотек (від компілятора до стандартної бібліотеки рантайму), не визначаємо політики випадковості (seeds), не зберігаємо артефакти матчу (логи, «replay», конфігурації), результати не можна перевірити. Для дослідницького процесу це критично: доведення переваги одного підходу над іншим має спиратися на пакунок доказів, який сторонній дослідник може прогнати повторно. Цей пакунок включає не лише коди ботів і судді, а й параметри турніру, початкові стани середовища, таймінги, політики тайм-аутів і обмежень. Таким чином, універсальність це не лише про «працює будь-якою мовою», а про «працює однаково, вимірювано й відтворювано».

Окремої уваги потребує забезпечення справедливості між мовами та реалізаціями. Навіть за однакових часових бюджетів і sandbox-політик, різні рантайми по-різному поводяться з I/O, GC, JIT-компіляцією. Це вимагає чітко описаних і стандартизованих фаз матчу (ініціалізація, кроки, фіналізація), ізоляції компіляції або warm-up поза бюджетом «хід-за-ходом», а також можливості «пінінгу» ресурсів для окремих матчів у дослідницьких цілях. Платформа має пропонувати не лише єдиний протокол обміну, а й єдині методики вимірювання: телеметрія часу на хід, лічильники системних викликів, використання пам’яті - все це повинно бути прозорим і доступним у візуалізації та звітах.

І нарешті, універсальність - це про те, щоб одна система одночасно задовольняла педагогічні, інженерні та дослідницькі потреби. Педагог просить інструмент, де видно процес і є контроль за чесністю; інженер - платформу, яку можна розгортати, масштабувати й інтегрувати; дослідник - стенд, на якому можна швидко перевіряти гіпотези та відтворювати результати інших. Спільним знаменником стає сукупність рішень: уніфікований протокол «бот-суддя» з

тонкими мовними адаптерами; ізольований, керований виконуючий контур із чіткими SLA; подієво-орієнтований канал телеметрії та стабільний офлайн-формат «replay»; модульна архітектура з явними контрактами та плагінністю; політики версіонування, аудит і спостережуваність. Саме ця конфігурація і формує ядро проблеми, яку розв'язує робота: створити платформу, що робить багатомовність і візуалізацію не додатками, а першокласними частинами системи - рівною мірою для навчання, інженерії та науки.

### 1.3 Аналіз існуючих змагальних платформ

Панорама сучасних платформ, на яких студенти та дослідники тренують алгоритмічні навички, неоднорідна і, головне, історично сформована під різні педагогічні та інженерні цілі. Є класичні онлайн-джаджі для задач «ввід-вивід», є комерційні портали з підбірками інтерв'ю-задач, є дослідницькі платформи для контейнеризованих сабмітів у задачах машинного навчання, є локальні турніри під конкретну гру чи курс. Якщо ж подивитися крізь призму завдань цієї роботи універсальна платформа для AI-ботів із багатомовним виконанням і вбудованою візуалізацією ігрового процесу, то стає очевидним, що жоден із наявних класів систем не дає одночасно і технологічну універсальність, і навчально-дослідницьку прозорість, і реальну операційну придатність для аудиторних курсів.

Професійні майданчики на кшталт Codeforces, TopCoder чи AtCoder блискуче розв'язують задачу масштабованого «batch-оцінювання» класичних алгоритмічних рішень: сильна культура тест-сетів, рейтинги, формати раундів, швидка перевірка великої кількості сабмітів, підтримка кількох мов компіляції та інтерпретації. Проте їхня архітектурна модель це переважно синхронний запуск «програма проти файлів тестів», а не «агент проти агента в багатокроковому середовищі». Там немає уніфікованого протоколу кроків гри, бар'єрів синхронізації, подієвого потоку для live-перегляду, стандарту для «replay»; а отже - немає місця для інтерактивної візуалізації перебігу матчу і пояснюваності рішень агента. Те, що в задачах ввід-вивід виглядає чесно і відтворювано (обмеження за часом і пам'яттю, версії компіляторів), у

динамічних ігрових сценаріях виявляється недостатнім: однакові ліміти не гарантують справедливості між рантаймами з різними профілями I/O, GC та JIT, а відсутність покрокового протоколу не дозволяє ані коректно судити, ані навчально проілюструвати хід подій.

Комерційні освітні портали на кшталт HackerRank, LeetCode чи CodeSignal зробили великий внесок у систематизацію тем і метрик, але їхня цінність - у контрольованих наборах задач і зручних трекерах прогресу - обернено пропорційна придатності до інженерної кастомізації. Код і суддівські контури закриті, інтегрувати нову гру або авторський «суддя-движок» практично неможливо, live-візуалізації ігрового процесу немає або вона суто декоративна. Більшість інтеграцій з LMS зводиться до синхронізації оцінок, а не до трансляції телеметрії матчу чи «replay». Для курсів, де компетенція вимірюється поведінкою агента в середовищі, а не лишень правильністю рядків на виході, це фундаментальне обмеження; для дослідників це ще й бар’єр на шляху до відтворюваності і публікації артефактів.

Старші академічні онлайн-джаджі на кшталт SPOJ або UVa Online Judge, попри безкоштовність і відкритість для користувачів, історично не були спроектовані як платформи для симулятивних змагань агентів. Їхня інфраструктура орієнтована на пакетні запуски і перевірку за статичними тестами; у кращому разі можна побачити розширення для інтерактивних задач, але без стандартного покрокового протоколу, подієвої шини, «replay» та телеметрії. Відсутність сучасної модульності ускладнює підключення контейнерних рантаймів з однаковими політиками безпеки для різних мов; відсутність єдиних форматів візуалізації перетворює будь-яку подібну систему на bespoke-розробку в межах окремої задачі[4]. Для аудиторної роботи це означає непотрібні витрати на підтримку та відсутність єдиного користувацького досвіду.

Є і протилежні приклади - спеціалізовані конкурси, де автори ретельно будують гру і вже мають гарну візуалізацію: CodinGame зокрема пропонує симпатичний фронтенд для окремих ігор; Halite від Two Sigma або Battlecode від

MIT демонстрували високий рівень дизайну судді, API для ботів і лайв-переглядів. Проте ці ініціативи, навіть попри якісний досвід, не перетворюються на універсальні платформи з причин архітектурної специфічності: мови і протоколи часто «прошиті» під конкретну гру, API - жорстко прив'язане до моделі середовища, а авторинг нових ігор потребує глибокого занурення в кодову базу саме цього конкурсу. Іншими словами, це не «платформи» в сенсі нашої роботи, а ретельно зроблені «екземпляри». Перенести їх як загальний інструмент у навчальний курс, підмінити гру на іншу, не торкаючись ядра, зазвичай нереально. Як виглядає платформа CodinGame, можна побачити на рисунку 1.1.

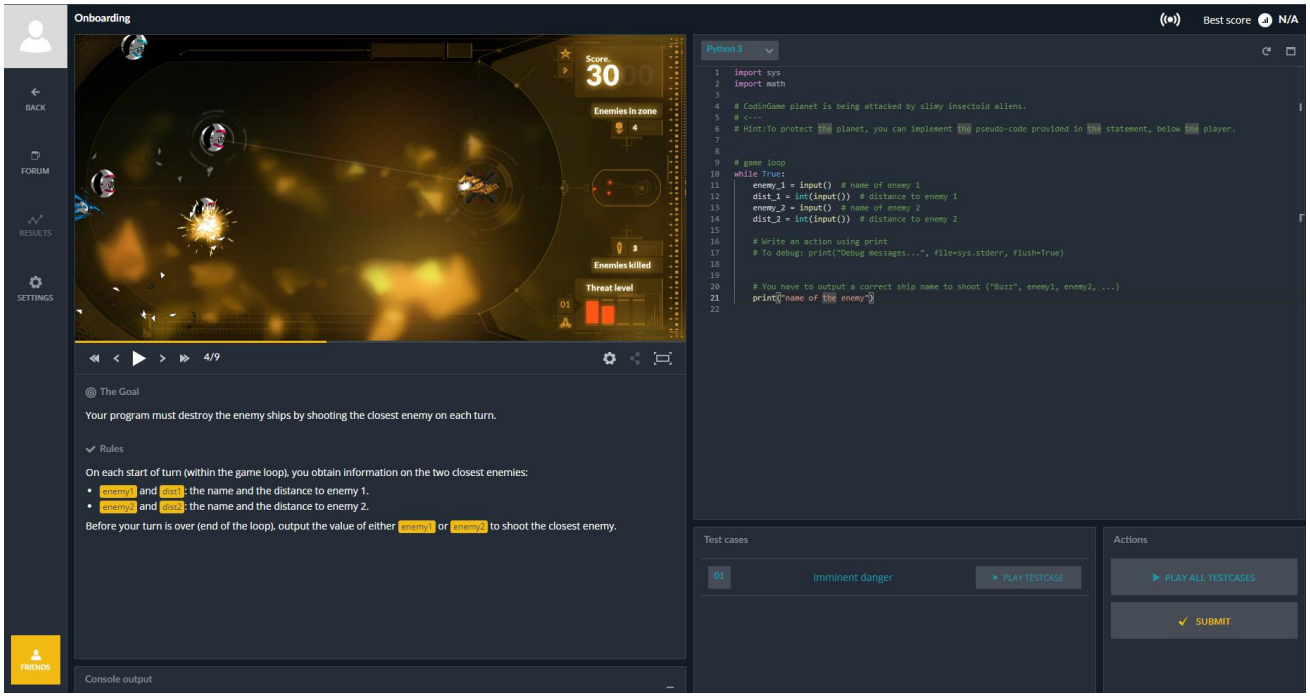


Рисунок 1.1. – Приклад інтерактивної візуалізації ігрового середовища на платформі CodinGame

Дослідницько-індустріальний сегмент - Kaggle, EvalAI, AIcrowd та подібні закриває іншу нішу: контейнеризовані сабміти, фіксовані середовища виконання, reproducible-оцінювання на датасетах або симуляторах. На перший погляд, це близько до наших потреб: є Docker, є автоматизація, є лідерборд. Але й тут виявляється низка принципових розбіжностей: сабміти здебільшого

оцінюються офлайн, у режимі batch на заздалегідь підготовлених епізодах; live-візуалізація або відсутня, або існує як окремий артефакт без стандартного каналу подій; багатомовність реалізується не як «тонкі адаптери» в межах платформи, а як повноцінні контейнери, які має зібрати учасник, - для курсів це завеликий операційний тягар. Додайте сюди складність інтеграції з LMS, оплату інфраструктури, відсутність уніфікованого протоколу покрокової взаємодії з суддею і отримаєте інструмент, чудовий для грандіозного публічного челенджу, але надмірний і незручний для щотижневих лабораторних у великій групі. Часто як альтернативу у навчальних матеріалах пропонують використовувати середовища на кшталт OpenAI Gym, PettingZoo чи Unity ML-Agents. Це потужні SDK для опису динаміки середовищ і агентів, але вони не є змагальними платформами. У них немає ні управління сабмітами студентів, ні політик чесного розподілу ресурсів між різними мовами, ні стандартів логування і телеметрії під турнір, ні фронтенд-компонент для візуалізації у браузері[5].

Якщо виділити спільні обмеження поперек усього спектру, отримаємо кілька повторюваних мотивів. Перший - фрагментація багатомовності: різні платформи трактують «підтримку мов» як підтримку компілятора, а не як єдиний покроковий протокол «бот-суддя» з однаковими гарантіями синхронізації та обліку часу, незалежно від рантайму. Звідси недоліки окремих мов, нерівномірність холодних стартів, проблеми з буферизацією I/O і непередбачувані тайм-аути. Другий - візуалізація після виконання алгоритмів: або таблиця результатів, або сторонній ролик, або bespoke-візуалізатор під одну гру без подієвої шини та «replay». Третій - слабка модульність: додавання нової мови, нового типу турніру, нового судді вимагає чіпати ядро, а отже - ризик, регресії та дорогі експерименти. Четвертий - операційна непридатність для аудиторій: відсутність простого розгортання, спостережуваності, аудит-трейлів, інтеграції з LMS і «м’яких» ролей (викладач, асистент, студент, гість). Педагогічний вимір обмежень часто менш очевидний, але критичний саме для нашого проєкту. Курси, де ключова компетенція - стратегічне мислення агента і розуміння компромісів між швидкістю, точністю, стабільністю потребують не

лише вердикту «Accepted/Rejected», а роз’яснювальних артефактів: як змінювався стан середовища, чому стратегія зайшла в глухий кут, де витрачено бюджет часу на кроці. Відсутність живої телеметрії, часових профілів і пояснюваних «replay» позбавляє викладача найпотужнішого інструменту можливості навчати на контрольних точках гри. А відсутність тонких мовних адаптерів знімає з порядку денного інклюзивність: студенти змушені пристосовуватися до єдиної мови або боротися з інфраструктурою, замість того щоб зосередитися на алгоритмах.

З інженерної перспективи нинішні рішення або надто монолітні, або надто «розсипані». У перших складно замінити окремий компонент без перетинання з рештою системи; у других відсутня єдина шина подій, стандарти логів і артефактів, через що складно відстежити деградації чи розбирати інциденти. Без контейнеризації з єдиними політиками безпеки для всіх мов та без інваріантів щодо SLA (час на хід, ліміти CPU/RAM/FS/NET) забезпечити чесність важко. Без формалізованого формату «replay» неможливі ані відтворюваність навчальних кейсів, ані наукова публікація з пакетом доказів. Без подієво-орієнтованого каналу для фронтенда (SSE/WebSocket) і стабільних API (REST/gRPC) важко побудувати якісну візуалізацію й аналітику поверх матчу.

Підсумовуючи, аналіз показує: наявні платформи закривають важливі, але вузькі підзадачі: класичне алгоритмічне оцінювання, масштабовані batch-запуски ML-сабмітів, або ж демонстраційні ігри з гарною картинкою. Для навчально-дослідницького сценарію «AI-боти в динамічному середовищі» потрібен інший рівень універсальності: тонкі мовні адаптери на єдиному протоколі «бот-суддя», стандартний покроковий цикл з бар’єрами синхронізації, sandbox-ізоляція з однаковими політиками безпеки для всіх мов, подієва телеметрія і стабільний «replay», модульна архітектура з плагінними суддями та масштабованими пулами воркерів, а також операційні якості - від розгортання до інтеграції з LMS. Саме ці розриви між тим, що є, і тим, що потрібно для чесного й наочного навчання та досліджень, і формують предмет цієї роботи та мотивують розробку універсальної платформи, де багатомовність і візуалізація -

не надбудова, а базові конструктори системи.

### 1.4 Підходи до багатомовного виконання коду

Реалізація багатомовного виконання в змагальній платформі це передусім про стандарти, а вже потім про інструменти. На рівні вимог і обмежень доводиться одночасно тримати у фокусі три незалежні осі: безпеку (ізоляція неперевіреного коду, чесний облік ресурсів, відсутність побічних каналів), відтворюваність (детерміновані середовища, фіксація версій, повні «replay» артефакти) та операційну придатність (масштабованість, спостережуваність, простота інтеграції з курсом чи лабораторією). Багатомовність додає до цієї тріади ще одну суттєву вимогу - рівнозначність рантаймів: Python із його зручністю прототипування, C/C++ з продуктивністю, JVM-мови з JIT-характером, JavaScript із подієвою моделлю - усі мають змагатися в однакових умовах. Саме тому ключ до успіху - уніфікований протокол «бот-суддя» і тонкі мовні адаптери, які роблять різні мови «непомітними» для ядра платформи; а вибір інструментів виконання (нативні процеси, контейнери, microVM, віддалені воркери, навіть WebAssembly) стає питанням реалізації політики ізоляції й операційної зручності.

На нижньому рівні існує кілька класичних підходів до запуску коду, кожен із яких має власні переваги та недоліки. Нативний процес у хості дає мінімальну затримку запуску й найнижчий overhead, але вимагає агресивної політики системної безпеки: без жорстких cgroups v2, seccomp/AppArmor профілів, read-only кореня, tmpfs-монтувань і закритої мережі він просто неприйнятний у навчальних сценаріях. Контейнеризація (Docker/Containerd/CRI-O) робить ізоляцію значно простішою та відтворюваною: образи фіксують версії компіляторів/інтерпретаторів, легко роздавати однакові середовища на кластері, впроваджується однакова політика ресурсів. Накладні витрати тут переважно пов’язані з холодними стартами (pull образів, інсталяція залежностей), але вони лікуються шаруванням, попереднім прогрівом і локальним кешем. MicroVM-рішення (напр., Firecracker через Kata Containers) піднімають планку ізоляції до рівня віртуалізації з близькою до контейнерів швидкістю, що корисно для

«жорстких» політик академічної доброчесності, однак вимагають додаткової експертизи в експлуатації. Повноцінна віртуалізація на окремих хостах забезпечує максимальний поділ середовищ, проте ціна за ресурс і керованість часто перевищує доцільність у курсах. Обчислення на віддалених воркерах (пули вузлів під управлінням оркестратора) - практичний компроміс, який дозволяє розкладати матчі, динамічно масштабуватися і тримати політики безпеки на рівні вузла, відділивши «суддю», API та UI. Для порівняння рівнів ізоляції: Containers-MicroVM-VM можна подивитися на рисунки 1.2 та 1.3.

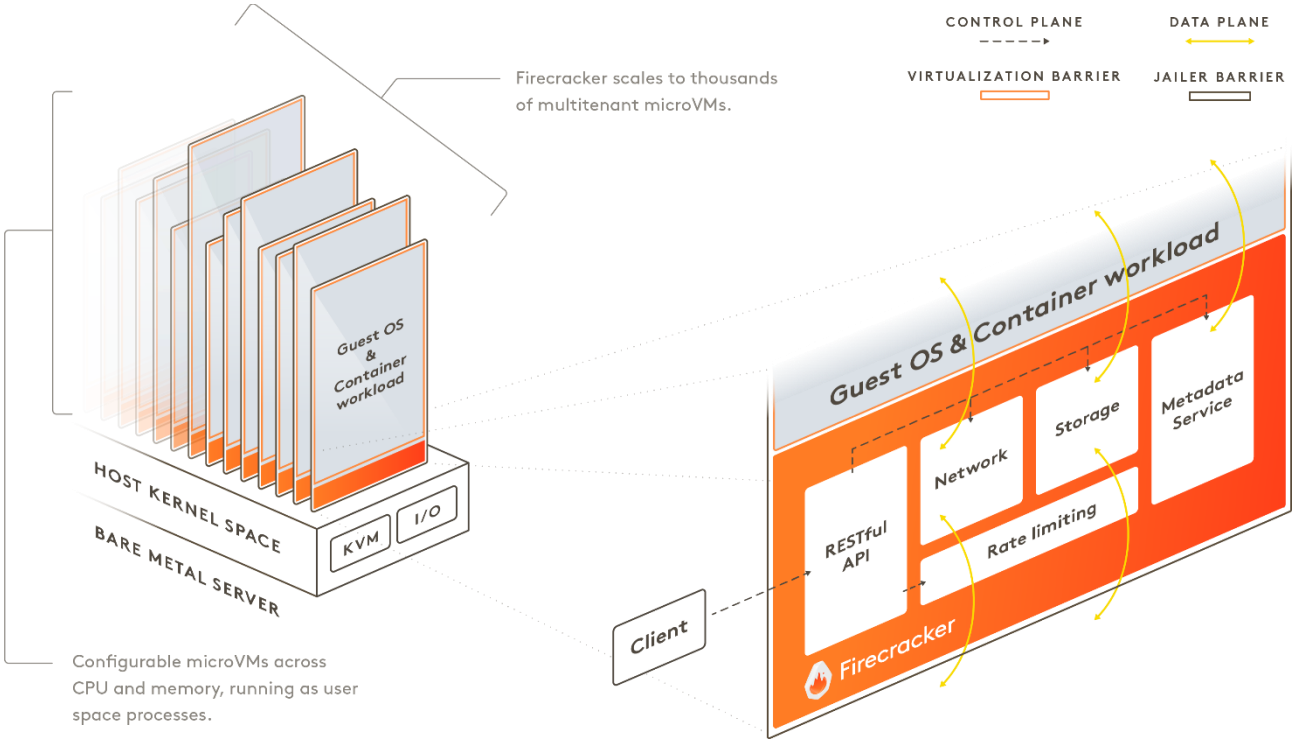


Рисунок 1.2. – Розгляд рівня ізоляції MicroVM[6]

Рисунки 1.2 та 1.3 показують чітку архітектурну різницю між контейнером, microVM та повною віртуалізацією.

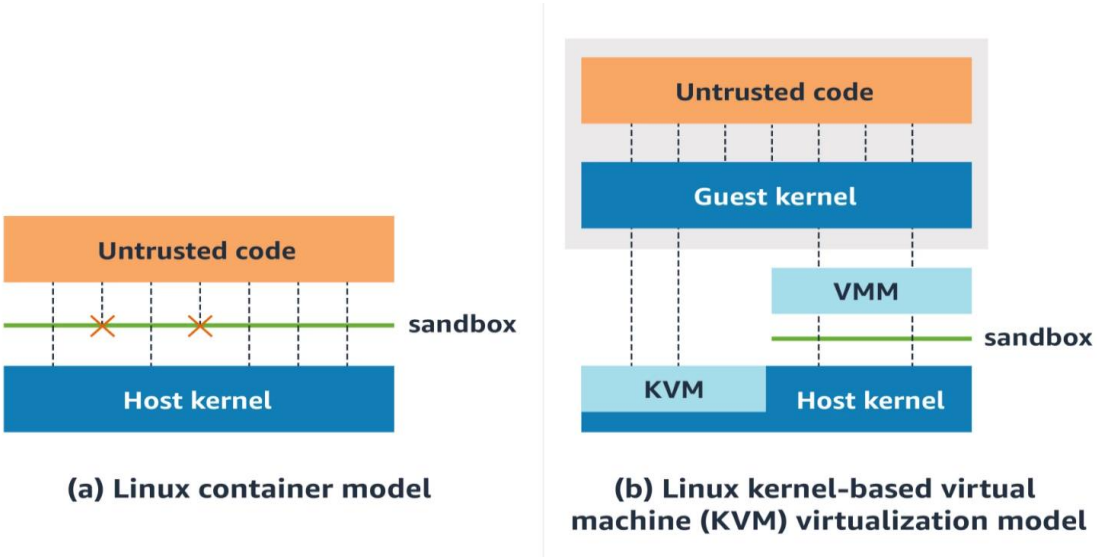


Рисунок 1.3. – Порівняння рівнів ізоляції: Containers та VM[7]

Як окремий напрямок варто відзначити WebAssembly/WASI[8]: він дає однорідну, детерміновану платформу виконання з дуже сильними межами пам'яті/системних викликів, але сьогодні має практичні обмеження для динамічних мов і системних залежностей, тож частіше виступає як доповнення для окремих класів задач, а не як універсальне рашення. Уніфікація протоколу взаємодії «бот-суддя» - центральний момент, який дозволяє відокремити логіку гри від технічних деталей мов та рантаймів. Практично це зводиться до покрокового синхронного контракту з бар'єрами: на кожному кроці суддя публікує агенту формалізований стан або спостереження (мінімізоване до необхідного), агент повертає дію до дедлайну, суддя застосовує дію і генерує події. Транспорт може бути простим (STDIN/STDOUT з рядковим протоколом, наприклад JSON Lines із чітким фреймінгом) або більш «важким» (локальний сокет, gRPC поверх Unix-сокетів)[9]. Важливішими за конкретний транспорт є інваріанти синхронізації: визначення «ходу», жорсткі дедлайни, однакова політика для timeouts/invalid actions, а також уніфікована телеметрія часу - облік CPU-квантів, а не стінного часу, що пом'якшує відмінності між рантаймами з JIT чи GC. Усі ці аспекти мають бути інкапсульовані в мовних адаптерах: невеликі бібліотеки/шаблони, які на боці агента забезпечують парсинг повідомлень, стабільну буферизацію I/O, обробку дедлайнів і сигнали «pre-stop», не змінюючи

ігрову модель судді. Роль мовного адаптера виходить за межі тонкої обгортки, фактично це контракт, який вирівнює поведінку різних екосистем. У Python адаптер бере на себе контроль над неблокуючим читанням, захист від висячого stdout-буфера, примусовий flush після кожної дії, замір часу на крок та стандартизоване логування. У C/C++ - спрощує взаємодію між алгоритмом і протоколом, уніфікує обробку помилок та дає безпечні таймери. Для JVM-мов потрібна ініціалізація з warm-up-фазою, чітке віднесення часу компіляції/завантаження класів до позаігрової частини, щоби матч не перетворювався на змагання JIT-стратегій. У JavaScript/Node.js важливо мати блокуючу семантику на рівні кроку, аби коливання подієвого циклу не «з'їдали» дедлайни. Усі адаптери повинні розділяти фази ініціалізації, кроків і фіналізації, генерувати сумісну телеметрію (час, пам'ять, кількість системних викликів, обсяг I/O) і, що суттєво, мати однакові точки версіонування з боку платформи, щоб навчальні групи і турніри працювали на однакових наборах бінарників. Компоновка життєвого циклу сабміту має бути справедливою для всіх мов: підготовка середовища (включно з інсталяцією залежностей), компіляція, warm-up і сам матч - різні фази з різними SLA. Для компільованих мов доцільно виносити компіляцію у build-крок, кешувати артефакти в шарах образів і чітко відокремлювати build-бюджет від часу «хід-за-ходом». Для інтерпретованих мов навпаки, прогрівати інтерпретатор і модулі до початку матчу, а в межах кроку обліковувати тільки час на обчислення дії. Такий поділ дозволяє уникнути «прихованих» переваг і робить турніри порівнюваними: учасники змагаються алгоритмами та їх реалізаціями, а не особливостями завантаження рантайму. З операційного боку це лягає на пайплайн збірки: попередньо зібрані адаптер-образи по мовах, окремі шари з бібліотеками, контрольний список дозволених системних пакетів і фіксація версій (image digests) для відтворюваності.

Ізоляція й безпека це фундамент для будь-якого виконання неперевіреного коду. Мінімальний набір включає Linux-простори імен, cgroups v2 для лімітів CPU/RAM/IO, заборонний профіль seccomp із вузьким allow-list, AppArmor/SELinux політики, read-only root filesystem, перехоплення мережі на

рівні CNI з політиками «deny by default», виділені tmpfs для робочих та артефактних директорій. Для чутливих сценаріїв додаються гіпервізорні межі (gVisor/Kata) та контроль файлових дескрипторів і часових джерел (щоб уникнути побічних каналів). Критично важливо стандартизувати причини дискваліфікацій: перевищення CPU-квоти, спроба забороненого sys-call, вихід за межі файлової пісочниці, мережевий трафік поза політикою - усе це має бути не лише зупинкою процесу, а й подією в телеметрії з роз’ясненням для навчального аналізу.

Планування ресурсів і синхронізація це ще одна зона, де є ризик спричинити мухлювання. Справедливий матч означає однакові бюджети на крок і на епізод, однакові пріоритети планувальника та відсутність head-of-line blocking між матчами різної складності. Практично це реалізується як пул воркерів із CPU-pinning і пріоритетними чергами, із backpressure та політиками повторних запусків при інцидентах інфраструктури. Відлік часу варто вести за монотонним годинником у межах cgroup, а не за системним wall-clock; пам’ять обмежувати як «жорстко» (OOM-kill), так і «м’яко» (сигнали попередження), дозволяючи платформі робити контрольовані скоупи перезапусків. Для дослідницьких цілей корисно мати режим «детермінізованих матчів» з фіксованими seeds і забороною джерел недетермінізму (включно з часом), аби відтворення «replay» давало біт-у-біт однакові результати.

Візуалізація й телеметрія напряму залежать від того, як саме платформа виконує код. Подієва шина повинна бути однаковою для всіх мовних адаптерів і суддів: кожна дія агента, кожна зміна стану, кожен штраф, тайм-аут, системна подія ізоляції - усе перетворюється на уніфіковану подію. Live-канал (SSE/WebSocket)[10] обслуговує UI в момент матчу; паралельно формується офлайн-«replay» із довготривалим зберіганням. Саме стандартизація подій дозволяє проектувати багаті фронтенд-візуалізації незалежно від того, чи агент написано на Python чи C++, чи виконується він у контейнері або microVM. У цьому сенсі вибір підходу до виконання має вторинний характер для фронтенда: поки інваріанти подій і таймінгів витримано, UI залишається стабільним. З

погляду загальної архітектури найкращі результати дає гібридна стратегія. Базовий шар це віддалені воркери під управлінням оркестратора (наприклад, Kubernetes), які піднімають процеси виконання в контейнерах із жорсткими політиками безпеки; для «жорстких» курсів або публічних івентів - вузли з увімкненим gVisor/Kata для підвищення меж ізоляції; для локальних демонстрацій і простих середовищ - за можливості легкі WASI-модулі. Над цим шаром - уніфікований протокол «бот-суддя», реалізований у вигляді мовних адаптерів і SDK, з якими працюють учасники. Ще вище - сервіс суддівства і турнірний планувальник, що гарантують бар'єри синхронізації, дедлайни та чесний облік ресурсів. Поруч - підсистема артефактів і телеметрії, яка збирає логи, «replay» і маніфести середовищ. Така композиція дозволяє почати «з контейнерів сьогодні», додавши microVM там, де це виправдано, не змінюючи контрактів і навчального досвіду.

Баланс між безпекою та продуктивністю в цій постановці вирішується не вибором «однієї правильної технології», а дисципліною контрактів і еволюційністю. Платформа повинна дозволяти рухатися по спектру: від нативного процесу (для локального PoC) до контейнера з профілями безпеки до microVM на чутливих вузлах; від STDIN/STDOUT у прототипі - до локального сокета чи gRPC з чітким фреймінгом; від «ручних» логів до стандартизованої подієвої шини. Саме така еволюційна модель сумісна з освітнім контекстом: вона дає змогу почати курс без «зайвої інфраструктури», а потім підвищувати рівень ізоляції й автоматизації без зламу API та повторного навчання студентів. У підсумку багатомовність перестає бути джерелом ризиків і перетворюється на дидактичну перевагу: студенти працюють тією мовою, яка для них природна, а платформа гарантує однакові правила гри, відтворюваність і наочність для всіх.

### 1.5 Методи забезпечення безпеки виконання коду

Безпека виконання користувацького коду в освітньому середовищі це не про «поставити Docker і розслабитися». У нашому випадку вона означає поєднання технічної ізоляції, чесного обліку ресурсів, спостережуваності та процедур академічної доброчесності в єдину, відтворювану систему. Загрози тут

двошарові: з одного боку, існує технічний ризик для інфраструктури (доступ до файлової системи поза пісочницею, спроби виконання небезпечних системних викликів, виснаження ресурсів, мережеві витоки), а з другого ризик для навчального процесу (шахрайство, плагіат, використання сторонніх сервісів всупереч правилам). Саме тому ми акцентуємо не на поодиноких інструментах, а на багаторівневій політиці, де кожен рівень компенсує недоліки сусідніх: ізоляція середовища, контроль ресурсів, валідація під час збирання, жорсткі рантайм-правила, повна телеметрія, аудит і механізми виявлення академічних порушень.

Перший рівень - коректно спроєктована ізоляція. Мінімально прийнятною основою для виконання неперевірених агентів є контейнеризація з використанням простору імен (pid, mount, network, uts, ipc, user), cgroups v2 для обмежень CPU, пам’яті, вводу-виводу та кількості процесів, а також профілів seccomp із вузьким allow-list системних викликів. До цього додаються політики AppArmor або SELinux, read-only коренева файлова система з окремими tmpfs-монтуваннями для робочих каталогів, обмежений доступ до /proc і /sys, повне відключення мережі за замовчуванням або чіткий allow-list для внутрішньої взаємодії «бот-суддя» (локальні сокети, ніякого загального egress). У чутливих сценаріях, зокрема у відкритих турнірах або при роботі з великими групами, цей базовий шар доповнюється віртуалізаційними гіпервізорними межами - gVisor/Kata Containers чи microVM на основі Firecracker, які ізолюють ядро гостя від ядра хоста, практично нівелюючи атаки через нестандартні системні виклики і знижуючи ризики побічних каналів. Ідея проста: навіть якщо окремий механізм дасть збій, комбінація незалежних бар’єрів не дозволить коду студента вирватися за межі відведеної «пісочниці».

Другий рівень - дисципліна ресурсів. Чесна змагальність неможлива без однакових бюджетів на крок і на епізод матчу, без передбачуваного планувальника. На практиці це означає, що облік часу ведеться не за «стінним» годинником, а за CPU-квантами в межах cgroup; що пам’ять має жорсткий ліміт із контрольованою реакцією OOM-killer’а; що обмежені кількість дескрипторів

файлів, розмір відкритих файлів і загальний слід на диску (через проєктні квоти чи overlay з верхньою межею); що блокуються небезпечні шаблони на кшталт fork-bomb завдяки ulimit-nproc і лімітам PID-простору; що I/O-тиск регулюється через throttling, аби «важкі» агенти не «забивали» диск і не впливали на латентність судді. Справедливість між різними рантаймами забезпечується розділенням фаз підготовки та гри: компіляція і warm-up JVM/JIT винесені в build-крок із кешуванням артефактів, а в самій грі обліковується лише час на крок; Python-агенти попередньо прогрівають інтерпретатор, але їхній «хід» вимірюється так само за CPU-квантами. Однаковість правил легко перевірити завдяки телеметрії: кожен крок супроводжується мітками часу, лічильниками системних викликів та профілями пам’яті.

Третій рівень - контроль системних можливостей та файлового простору на грані «надлишкового» захисту, який однак у навчальних сценаріях цілком виправданий. Політика seccomp має блокувати perf_event_open, bpf, ptrace, keyctl, mount/unmount, unshare нових просторів імен, доступ до часових джерел високої роздільної здатності, маніпуляції пріоритетами (sched_setscheduler) та інші «лазівки», що використовуються для побічних каналів або DOS-поведінки. Кореневі шляхи монтуються лише для читання, /tmp - як ізольований tmpfs із верхньою межею, виконувані файли - виключно з білого списку, а будь-які спроби запуску двійкових файлів поза дозволеним префіксом жорстко блокуються. Для домовленостей між агентом і суддею використовується або локальний Unix-сокет з мінімальними правами, або STDIN/STDOUT із явним фреймінгом повідомлень; жодних відкритих TCP-портів чи зовнішнього egress. Усе це не просто технічні деталі - це спосіб зробити гру чесною, усунувши спокусу знайти готову відповідь чи зливати телеметрію в сторонні сервіси.

Четвертий рівень - гігієна складання й залежностей. Щоб усунути клас цілком реальних supply-chain ризиків і випадкових «спливів» можливостей, середовища виконання описуються у вигляді зафіксованих образів із криптографічно закріпленими digest’ами; для мовних екосистем використовуються lock-файли або vendor-каталоги; під час build-кроку

платформа працює в офлайн-режимі, користуючись внутрішнім дзеркалом пакетів (у якому доступний лише дозволений набір бібліотек з перевіреними ліцензіями). Будь-які нативні розширення або системні пакети проходять явний allow-list; непотрібні інструменти та компілятори не включаються до фінального рантайму, аби зменшити площу атак і варіативність поведінки. Важливий і дидактичний аспект: студенти працюють не зі «своїм» середовищем, а з чітко визначеним, однаковим для всіх, що знімає аргументи про «не ті версії» і спрощує відтворення результатів через семестр.

П’ятий рівень - спостережуваність і аудит, які поєднують технічну безпеку з дидактикою. Кожен матч отримує трасувальний ідентифікатор, за яким гуртуються логи судді, події ізоляції, системні метрики воркера, «replay» та конфігурації середовищ. Логи зберігаються в незмінному сховищі з позначками часу і верифікацією цілісності; частина метрик виноситься у дашборди для оперативного моніторингу, а частина у сховище артефактів для підсумкового аналізу. Це не лише інструмент реагування на інциденти, а й основа навчальних розборів: у візуалізації поруч зі станом гри з’являються маркери перевищення лімітів, «блоків» seccomp, тайм-аутів і повторних запусків, що дозволяє студентам пов’язувати поведінку стратегії з технічними обмеженнями платформи.

Нарешті, робоча складова безпеки - політика, комунікація і UX. Усі заборони та ліміти мають бути не лише технічно впроваджені, а й зрозуміло пояснені: довідники з прикладами «коректного» і «некоректного» I/O, шаблони для мейн-циклу агента з правильним flush, чек-листи для управління пам’яттю, приклади обробки дедлайнів, список дозволених бібліотек і мотивування цих дозволів. Повідомлення про помилки повинні бути зрозумілими: не «Killed», а «Перевищено ліміт пам’яті (512 МіБ): спробуйте…», не «Timeout», а «Перевищено ліміт 100 мс на крок (CPU-час): оптимізуйте…». Без цієї прозорості навіть найкраща пісочниця перетворюється на «чорну скриньку», що демотивує і провокує обхідні маневри.

Сукупно ці підходи формують саме той багаторівневий каркас, який і

потрібен змагальній платформі: глибинна технічна ізоляція, справедливе й відтворюване планування ресурсів, детальна телеметрія та аудит, а також процедурні механізми забезпечення чесності. Важливо, що всі ці рішення не ізольовані від візуалізації - навпаки, вони її підживлюють: події безпеки та лімітів стають частиною загальної картин змаганння, допомагаючи користувачам зрозуміти зв'язок між алгоритмом і системними обмеженнями. У результаті безпека перестає бути «гальмом розвитку» і перетворюється на елемент інженерної культури курсу: гра чесна, експерименти відтворювані, а інциденти - зрозумілі та виправні.

### 1.6 Візуалізація алгоритмів як інструмент навчання

У контексті цієї платформи візуалізація це не «декорація», а така сама критична частина навчального циклу, як компіляція чи суддівство. Її завдання - зробити процес ухвалення рішень алгоритмом спостережуваним і пояснюваним. Коли користувач бачить, як бот сприймає стан середовища, які дії генерує і що з цього виходить на наступному кроці, абстрактні концепції про евристику, пошук та оптимізацію перестають бути чорним ящиком. Саме завдяки візуалізації ми отримуємо міст між кодом і поведінкою: не «пройшло/не пройшло» в табличці, а покрокова історія рішень із прив'язкою до часу, ресурсів і подій у середовищі. Для викладача це означає можливість розбирати матчі «по кісточках», зупиняючись у контрольних точках, порівнювати альтернативні стратегії на тих самих початкових умовах і демонструвати причини помилок, а не лише їхні наслідки.

Педагогічний ефект візуалізації спирається на просту когнітивну ідею: легше зрозуміти алгоритм, коли можна одночасно спостерігати і стан, і дію, і результат. У динамічних ігрових середовищах це особливо важливо, бо центральна компетенція це не лише знати алгоритм, а й співвідносити його з обмеженнями часу та пам'яті, з латентністю, з неповнотою спостережень. Інтерактивне відтворення із паузою, перемоткою і зміною швидкості дозволяє «заморозити» складний момент, подивитися на локальну карту спостережень агента, оцінити метрики (наприклад, витрачений бюджет на крок, глибину

пошуку, величину штрафів) і обговорити альтернативи. Навіть проста можливість накладати підсвітку «що бачив бот на цьому кроці» і «які варіанти він відкидав» радикально підвищує якість пояснення: студент перестає шукати магію там, де є конкретний компроміс або помилка в інженерних налаштуваннях. Інженерно візуалізація тримається на двох стовпах: live-каналі подій і стабільному офлайн-«replay». Live потрібен для семінарів і захистів, де ми дивимося матч у реальному часі і коментуємо рішення; тут важливі рівномірність потоку подій, компактність їхнього формату та чіткий контракт синхронізації з сервером суддівства. «Replay» потрібен для післяматчевого аналізу: у JSON- або бінарному форматі з версією, хешем судді, seed'ами випадковості, списком початкових станів, часовими мітками кроків, діями агентів і службовими подіями (тайм-аути, порушення політик безпеки, повторні запуски). Коли кожна подія має стабільну схему, фронтенд стає незалежним від конкретної гри: рендер, накладання шарів і панель метрик працюють однаково для лабіринту, настільної гри чи симулятора ресурсів. Для навчання це критично: один візуальний інструмент - багато різних курсів і завдань. Щоб візуалізація виконувала освітню функцію, вона має бути «системно чесною». Це означає, що поверх графіки накладаються артефакти інфраструктурного рівня: маркери перевищення бюджету часу на кроці, індикатори скидання процесу через OOM, позначки блокувань seccomp або мережевих політик. Коли студент бачить червоний маркер «timeout» саме в той момент, коли стратегія обчислювала оцінку стану, він швидше зрозуміє, чому наступний крок був порожнім або некоректним. А коли викладач у «replay» вмикає шар «CPU/пам'ять за кроками» і бачить, як JIT прогрівся на 10-му кроці й зменшив латентність, це дозволяє говорити не лише про алгоритмічні, а й про системні аспекти реалізації. Така пов'язка між поведінкою агента і телеметрією платформи - основа інженерної культури, яку ми прагнемо сформувати.

Насамкінець варто підкреслити: візуалізація в нашій платформі це не лише про мотивацію, а про якісну перевірку гіпотез. Вона дає змогу пов'язати код, ресурсні обмеження, протокол «бот-суддя» і поведінку агента в єдиному

просторі спостереження. Саме тому ми ставимо вимоги до подієвого каналу й «replay» так само строго, як до безпеки чи API: якщо події нестабільні, візуалізація стане іграшкою; якщо вони детерміновані та мають за собою логіку, це інструмент, який навчає швидше, глибше і чесніше. У підсумку студент отримує не просто оцінку, а досвід інженерного дослідження власного алгоритму, а викладач - прозору й контрольовану оптику для зворотного зв’язку.

### 1.7 Постановка задачі створення AI-платформи

Узагальнюючи попередній аналіз, формулюється центральна інженерно-дослідницька задача: спроєктувати та довести життєздатність універсальної веб-платформи для турнірів між AI-ботами, у якій багатомовне виконання коду та інтегрована візуалізація ігрового процесу є конструктивними складовими, а не допоміжними надбудовами. Платформа має забезпечувати чесне порівняння стратегій у динамічних середовищах, відтворюваність експериментів та прозору інтерпретацію рішень агентів через live-телеметрію та офлайн-«replay». Універсальність тут означає як технічну, так і педагогічну придатність: єдиний протокол «бот-суддя» з тонкими мовними адаптерами, стабільні контракти даних для матчу й візуалізації, а також інструменти викладача для розбору помилок і контролю академічної доброчесності.

Функціональна сутність системи охоплює повний життєвий цикл змагання: від завантаження сабмітів різними мовами до планування матчів, суддівства, обліку рейтингів і аналітичного перегляду партій. На мінімальному рівні підтримуються принаймні дві мови (Python і Java) з перспективою безболісного розширення до C/C++ та JavaScript за рахунок однакових мовних адаптерів і стандартизованого транспортного каналу. Суддівський двигун працює покроково з жорсткими дедлайнами й бар’єрами синхронізації, забезпечуючи рівні умови незалежно від особливостей рантайму; візуалізатор у реальному часі відображає стани середовища, дії агентів і ресурсні маркери, тоді як офлайн-«replay» фіксує повний набір артефактів для повторного відтворення та розбору.

Нефункціональні вимоги визначають планку інженерної якості. Безпека та

ізоляція виконання обов’язкові: контейнерні середовища з cgroups, seccomp та політиками файлового/мережевого доступу, за потреби посилені мікровіртуалізацією. Масштабованість передбачає роботу з десятками одночасних турнірів і сотнями активних ботів за стабільного часу відгуку API у межах субсекунди для керуючих операцій; спостережуваність забезпечується повною телеметрією кроків, ресурсоємності та подій політик. Відтворюваність гарантується фіксацією версій середовищ, seed-ів випадковості та схем подій «replay», щоб будь-який матч можна було програти повторно без дрейфу результатів.

Освітній вимір задачі задає окремі обмеження і цілі. Інтерфейси ролей мають відображати реальні сценарії аудиторної роботи: викладач керує турнірами, налаштовує формати пар, відстежує порушення лімітів і коментує «контрольні точки» у візуалізації; студенти отримують швидкий зворотний зв’язок і прозорий перебіг гри; асистенти модераторськими інструментами розв’язують інциденти та обробляють апеляції. Академічна доброчесність підтримується як процедурно (правила, роз’яснювальні повідомлення, журнал подій), так і технічно (аналіз подібності коду, поведінкові «відбитки», контроль зовнішніх залежностей). Ефективність у навчанні визначається не лише підсумковими балами, а й якістю пояснюваності: наскільки візуалізація допомагає пов’язати рішення агента з обмеженнями часу й пам’яті, а також наскільки швидко студенти виправляють системні та алгоритмічні помилки. Дослідницька складова формулює питання, що мають бути закриті в межах проєкту і зафіксовані експериментально: який саме компроміс між контейнерами та microVM дає найкраще співвідношення безпеки та латентності для типових навчальних ігор; як нормувати «час на крок», щоб згладити відмінності між інтерпретованими та JIT-мовами; яку схему подій «replay» обрати, аби вона залишалася стабільною для різнорідних середовищ; як вимірювати навчальний ефект візуалізації за зміною часу на виправлення помилки, за динамікою якості стратегій, за опитуваннями задоволеності. Результатом цієї постановки є чіткий план валідації: побудова референтної архітектури, створення контрольних

ігрових середовищ, запуск пілотних турнірів і збір метрик, які дозволяють судити про технічну надійність та масштабованість системи. У такий спосіб платформа переходить із площини «цікавого інструмента» у категорію відтворюваного навчально-дослідницького стандарту.

# ВИСНОВОК ДО РОЗДІЛУ 1

Проведений аналіз предметної області виявив критичну потребу в створенні універсальної платформи для змагань зі штучного інтелекту, яка б поєднувала підтримку багатомовного програмування з інтерактивною візуалізацією ігрового процесу. Існуючі рішення, хоча й забезпечують високу якість завдань та ефективну систему тестування, мають суттєві обмеження: відсутність інтерактивної візуалізації алгоритмічних процесів, обмежену підтримку різних мов програмування та недостатню адаптацію для освітніх цілей. Ці обмеження створюють бар'єри для широкого використання змагальних платформ у навчальному процесі, особливо для початківців.

Технічні виклики реалізації багатомовного виконання коду вимагають комплексного підходу, що поєднує різні методи ізоляції та безпеки. Гібридний підхід, який би використовував нативне виконання для одних мов та контейнеризацію для інших, представляється найбільш перспективним для забезпечення як безпеки, так і продуктивності. Візуалізація алгоритмічних процесів є невід'ємним компонентом ефективного навчання програмування, оскільки дозволяє студентам спостерігати за роботою алгоритмів у реальному часі та краще розуміти їх логіку.

Сформульована задача створення платформи для турнірів AI-ботів є актуальною та досяжною. Вона передбачає розробку модульної архітектури з підтримкою Python та Java, реалізацію інтерактивної візуалізації ігрового процесу та створення зручного веб-інтерфейсу для користувачів. Вирішення цієї задачі має потенціал значно підвищити мотивацію студентів до вивчення програмування, забезпечити викладачів ефективними інструментами для демонстрації алгоритмів та створити основу для розвитку змагального програмування в освітніх закладах.

# РОЗДІЛ 2. ПЛАНУВАННЯ АРХІТЕКТУРИ ТА ВИБІР ТЕХНОЛОГІЙ

## 2.1 Методи дослідження та проєктування

Для розробки платформи для турнірів AI-ботів обрано комплексний методологічний підхід, що поєднує системний аналіз, прототипування та експериментальну валідацію. Системний аналіз дозволяє розкласти складну систему на логічні компоненти та визначити їх взаємодію. У контексті платформи це передбачає виділення основних модулів: ядра системи з абстрактними інтерфейсами (BotInterface, GameEngine, BotExecutor), виконуючих середовищ для різних мов програмування, ігрових движків для різних типів змагань, менеджера турнірів та веб-API для взаємодії з користувачами. Така декомпозиція забезпечує модульність архітектури та можливість незалежного розвитку кожного компонента.

Прототипування є ключовим етапом для швидкої валідації концепцій та архітектурних рішень. Створення мінімального робочого прототипу дозволяє на ранньому етапі виявити потенційні проблеми та перевірити життєздатність обраних підходів. У рамках платформи це передбачає реалізацію базової функціональності для створення турнірів, виконання простих ботів на Python та Java, проведення елементарних ігор та відображення результатів. Прототип дозволяє перевірити взаємодію між модулями, валідувати вибрані технології та отримати зворотний зв'язок від потенційних користувачів.

Експериментальна валідація на контрольних середовищах необхідна для перевірки продуктивності, надійності та безпеки реалізованих рішень. Це включає тестування різних підходів до виконання коду, порівняння ефективності нативного виконання Python з виконанням Java через Docker контейнери, валідацію механізмів безпеки та оцінку масштабованості системи. Контрольні середовища дозволяють симулювати реальні умови експлуатації, включаючи навантаження, одночасну роботу множинних турнірів та виконання потенційно небезпечного коду. Результати експериментальної валідації лягають в основу фінальних архітектурних рішень та оптимізації системи для забезпечення

необхідних показників продуктивності та безпеки.

### 2.2 Вибір архітектурних рішень для виконання коду

Вибір архітектурного рішення для виконання коду є критичним аспектом розробки платформи, оскільки він безпосередньо впливає на безпеку, продуктивність та масштабованість системи. Аналіз альтернативних підходів показує, що кожен з них має свої переваги та обмеження. Вбудовані інтерпретатори забезпечують найвищу продуктивність завдяки прямому виконанню коду в основному процесі, але створюють серйозні ризики безпеки та обмежують можливості ізоляції. Керовані контейнери, зокрема Docker, пропонують оптимальний баланс між безпекою та продуктивністю, забезпечуючи повну ізоляцію виконання коду при відносно невеликих накладних витратах. Віддалені воркери забезпечують максимальну ізоляцію та масштабованість, але ускладнюють архітектуру системи та створюють залежність від мережевої інфраструктури.

Для платформи турнірів AI-ботів обрано гібридний підхід, який поєднує переваги різних методів виконання залежно від специфіки кожної мови програмування. Python виконується нативно через вбудований інтерпретатор з додатковими обмеженнями безпеки, що дозволяє забезпечити високу продуктивність при прийнятному рівні ризиків. Java виконується через Docker контейнери, що забезпечує повну ізоляцію та безпеку для потенційно небезпечного коду. Такий підхід дозволяє оптимізувати виконання кожної мови відповідно до її особливостей, одночасно забезпечуючи уніфікований інтерфейс взаємодії через абстрактний BotInterface. Це рішення забезпечує необхідний рівень безпеки для освітньої платформи, де студенти можуть надавати потенційно небезпечний код, одночасно зберігаючи високу продуктивність для швидкого виконання алгоритмів у змагальному контексті.

### 2.3 Цільова архітектура та структура проєкту

Вибір цільової архітектури для платформи турнірів AI-ботів базується на принципах модульності, розширюваності та розділення відповідальності. Обрана архітектура має забезпечити можливість легкого додавання нових мов

програмування, типів ігор та функціональностей без необхідності зміни існуючого коду. Це досягається через створення чітко визначених інтерфейсів та абстракцій, які дозволяють різним компонентам взаємодіяти через стандартизовані контракти. Такий підхід особливо важливий для освітньої платформи, де може виникнути потреба в додаванні нових мов програмування або створенні спеціалізованих типів змагань для різних навчальних курсів. Структура проєкту організована за принципом розділення відповідальності, де кожен модуль має чітко визначену функцію та мінімальну залежність від інших компонентів. Ядро системи (core) містить абстрактні інтерфейси та базові класи, які визначають контракти для всіх інших модулів. Це включає BotInterface для уніфікації взаємодії з ботами різних мов програмування, GameEngine для стандартизації ігрових движків та BotExecutor для забезпечення єдиного підходу до виконання коду. Така організація дозволяє легко додавати нові реалізації без зміни існуючого коду, що критично важливо для розвитку платформи.

Виконуючі середовища (executors) реалізують специфічну логіку для кожної мови програмування, але дотримуються єдиного інтерфейсу BotExecutor. Це дозволяє системі працювати з різними мовами через уніфікований API, приховуючи складність реалізації від основних компонентів платформи. Ігрові движки (games) також реалізують єдиний інтерфейс GameEngine, що дозволяє легко додавати нові типи змагань без зміни логіки проведення турнірів. Менеджер турнірів (tournament) координує роботу всіх компонентів, забезпечуючи проведення змагань та збереження результатів.

Веб-API надає зовнішній інтерфейс для взаємодії з платформою, дозволяючи користувачам створювати турніри, завантажувати ботів та переглядати результати. Використання REST API забезпечує сумісність з різними клієнтами та дозволяє легко інтегрувати платформу з іншими освітніми системами. База даних зберігає всю інформацію про турніри, ботів, ігри та результати, забезпечуючи персистентність даних та можливість аналізу історичних результатів. Така архітектура забезпечує високу гнучкість та масштабованість платформи, дозволяючи їй розвиватися відповідно до змінних

потреб освітнього процесу.

### 2.4 Ролі користувачів у системі турнірів

Визначення ролей учасників процесу в системі турнірів є критичним аспектом проектування платформи, оскільки воно безпосередньо впливає на архітектуру системи, безпеку та користувацький досвід. Кожна роль має свої специфічні потреби, обмеження та вимоги до функціональності, що потребує ретельного аналізу та врахування при розробці інтерфейсів та механізмів взаємодії. Розуміння ролей дозволяє створити інтуїтивну та ефективну систему, яка задовольняє потреби всіх типів користувачів, від студентів-початківців до досвідчених викладачів та адміністраторів системи. Діаграму варіантів використання зображено на рисунку 2.1.

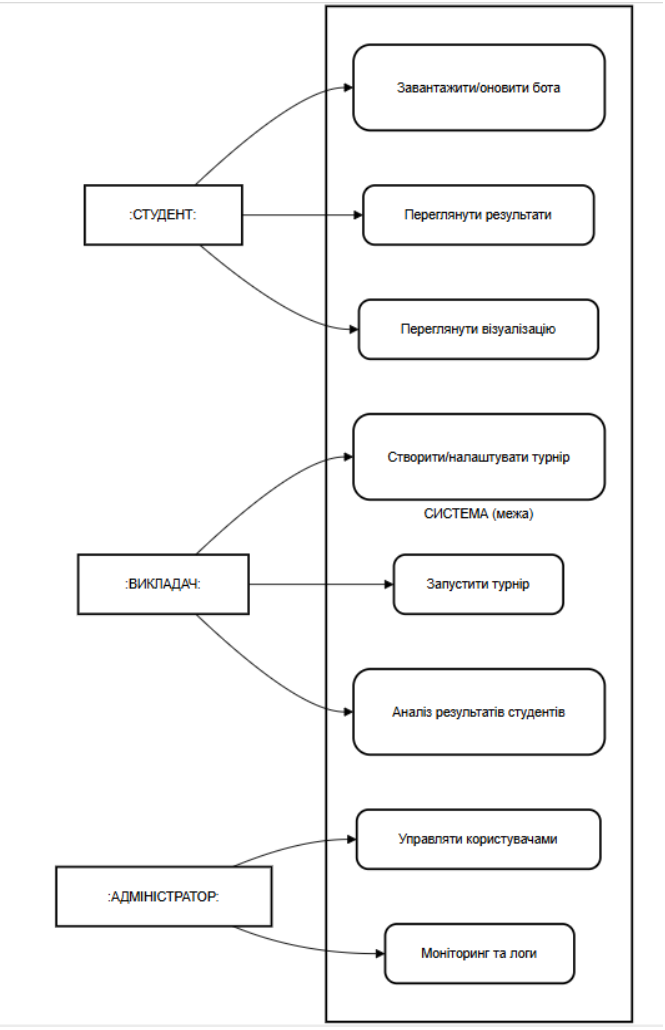


Рисунок 2.1 – Діаграма варіантів використання

Учасник (студент, просто користувач) є основною цільовою аудиторією

платформи та виконує роль розробника ботів. Ця роль передбачає створення, налаштування та відправку ботів для участі в турнірах, а також аналіз результатів змагань для покращення алгоритмів. Учасники потребують зручного інтерфейсу для написання коду, можливості тестування ботів перед відправкою, детальної статистики виступу та інструментів для аналізу стратегій. Для забезпечення ефективного навчального процесу учасники також потребують доступу до прикладів коду, документації та можливості порівняння своїх рішень з іншими. Система має забезпечувати мотивацію через змагальні елементи, рейтинги та можливість демонстрації досягнень.

Суддя/движок оцінювання виконує роль автоматизованої системи, яка проводить змагання та визначає переможців. Ця роль включає валідацію коду ботів, проведення ігор згідно з правилами, обчислення очок та визначення результатів. Суддя має забезпечувати справедливість змагань, об'єктивність оцінювання та дотримання правил турніру. Система має бути достатньо надійною для автоматичного проведення турнірів без людського втручання, але також мати механізми для вирішення спірних ситуацій та апеляцій. Важливим аспектом є забезпечення прозорості процесу оцінювання, щоб учасники могли розуміти, чому їх боти отримали певні очки.

Адміністратор системи відповідає за загальне управління платформою, включаючи створення турнірів, налаштування правил, моніторинг системи та вирішення технічних проблем. Ця роль потребує розширених прав доступу, інструментів для аналізу використання системи та можливості швидкого реагування на проблеми. Адміністратор також відповідає за безпеку системи, включаючи виявлення та запобігання шахрайству, плагіату коду та іншим порушенням. Система має надавати адміністратору детальну аналітику використання платформи, статистику продуктивності та інструменти для оптимізації роботи системи.

Спостерігач (викладач, дослідник) виконує роль аналітика, який використовує платформу для навчання, дослідження або демонстрації концепцій. Ця роль потребує доступу до детальної інформації про проведені

турніри, можливості аналізу стратегій ботів та інструментів для створення навчальних матеріалів. Спостерігачі також можуть виступати в ролі організаторів турнірів для своїх студентів, налаштовуючи правила та параметри змагань відповідно до навчальних цілей. Система має забезпечувати можливість експорту даних для подальшого аналізу, створення звітів та інтеграції з іншими освітніми інструментами.

### 2.5 Підсистема управління турнірами

Підсистема управління турнірами є центральним компонентом платформи, який координує всі аспекти проведення змагань від створення до підсумкування результатів. Вибір архітектури цієї підсистеми базується на необхідності забезпечити гнучкість у налаштуванні різних форматів турнірів, автоматизацію процесу проведення змагань та збереження детальної інформації про всі етапи турніру. Система має підтримувати як прості round-robin турніри, де кожен бот грає з кожним, так і складніші формати, включаючи олімпійську систему та кастомні схеми, які можуть бути необхідні для специфічних навчальних цілей. Така гнучкість особливо важлива для освітньої платформи, де різні курси можуть потребувати різних підходів до оцінювання студентів. Архітектура підсистеми управління турнірами базується на принципі поділу відповідальності між різними компонентами. TournamentManager відповідає за загальну координацію процесу, включаючи створення турнірів, валідацію учасників, планування ігор та підсумкування результатів. Система планування ігор автоматично генерує розклад змагань на основі обраного формату турніру, враховуючи кількість учасників, наявні ресурси та обмеження часу. Це дозволяє проводити турніри різного масштабу - від невеликих класних змагань до великих міжуніверситетських турнірів з сотнями учасників.

Система черги завдань забезпечує ефективне управління ресурсами та справедливий розподіл навантаження між доступними виконуючими середовищами. Коли турнір запускається, система автоматично створює завдання для кожної гри та додає їх до черги виконання. Це дозволяє проводити множинні ігри паралельно, максимізуючи використання доступних ресурсів та

зменшуючи загальний час проведення турніру. Система також забезпечує пріоритизацію завдань, дозволяючи швидше обробляти критичні ігри або турніри з високим пріоритетом. Моніторинг та логування є критично важливими аспектами підсистеми управління турнірами, оскільки вони забезпечують прозорість процесу та можливість діагностики проблем. Система веде детальний лог всіх дій, включаючи створення турнірів, запуск ігор, результати виконання ботів та помилки. Це дозволяє адміністраторам швидко виявляти та вирішувати проблеми, а також аналізувати ефективність роботи системи. Система також надає реальний час оновлення статусу турнірів, дозволяючи учасникам та спостерігачам відстежувати прогрес змагань.

Система збереження результатів забезпечує персистентність даних та можливість детального аналізу історичних турнірів. Всі результати зберігаються в структурованому вигляді, включаючи детальну інформацію про кожну гру, дії ботів, використані стратегії та фінальні очки. Це дозволяє створювати статистичні звіти, аналізувати тренди у розвитку алгоритмів та виявляти найефективніші підходи до вирішення завдань. Система також підтримує експорт даних у різних форматах для подальшого аналізу або інтеграції з іншими системами.

### 2.6 Ізоляція та безпека виконання ботів

Забезпечення безпеки виконання користувацького коду є одним з найкритичніших аспектів розробки платформи для турнірів AI-ботів, особливо в освітньому контексті, де студенти можуть надавати потенційно небезпечний або неоптимізований код. Вибір механізмів ізоляції та безпеки базується на необхідності забезпечити повну ізоляцію виконання коду від основної системи, запобігти зловживанням системними ресурсами та забезпечити академічну доброчесність змагань. Це особливо важливо, оскільки платформа буде використовуватися в освітніх закладах, де безпека системи є пріоритетом, а студенти можуть експериментувати з різними підходами до програмування, включаючи потенційно ризиковані техніки.

Для різних мов програмування обрано різні підходи до забезпечення

безпеки, що відображає специфіку кожної мови та її механізми виконання. Python виконується через нативний інтерпретатор з додатковими обмеженнями безпеки, включаючи контроль доступу до системних модулів, обмеження виконання небезпечних операцій та моніторинг використання ресурсів. Такий підхід дозволяє забезпечити високу продуктивність виконання Python коду, що критично важливо для алгоритмічних змагань, де швидкість виконання може бути вирішальним фактором. Однак нативне виконання створює додаткові ризики безпеки, що потребує ретельної валідації коду перед виконанням та постійного моніторингу процесу виконання.

Java виконується через Docker контейнери, що забезпечує максимальну ізоляцію та безпеку виконання коду. Docker контейнери створюють повністю ізольоване середовище виконання, де код не має доступу до файлової системи хост-машини, мережевих з'єднань або інших системних ресурсів. Це особливо важливо для Java, оскільки ця мова часто використовується для створення складних додатків, які можуть мати більше можливостей для зловживань. Контейнеризація також дозволяє легко контролювати ресурси, включаючи обмеження пам'яті, CPU часу та дискового простору, що запобігає вичерпанню системних ресурсів через неоптимізований або зловмисний код.

Система валідації коду є критично важливим компонентом механізмів безпеки, оскільки вона дозволяє виявити потенційно небезпечний код до його виконання. Для Python це включає аналіз синтаксису коду, перевірку імпортів на наявність небезпечних модулів, аналіз використання системних функцій та перевірку на наявність потенційно небезпечних конструкцій. Система використовує статичний аналіз коду для виявлення підозрілих паттернів, включаючи спроби доступу до файлової системи, виконання системних команд або створення мережевих з'єднань. Для Java валідація включає компіляцію коду в ізольованому середовищі, перевірку на наявність небезпечних залежностей та аналіз байт-коду на предмет підозрілих операцій. Моніторинг виконання коду забезпечує постійний контроль за процесом виконання ботів та швидке виявлення проблем. Система відстежує використання ресурсів, включаючи CPU

час, пам'ять, дисковий простір та мережевий трафік, і автоматично припиняє виконання коду, який перевищує встановлені ліміти. Це запобігає вичерпанню системних ресурсів та забезпечує справедливість змагань, оскільки всі боти мають рівні обмеження на використання ресурсів. Система також веде детальний лог всіх дій ботів, включаючи виклики функцій, зміни стану та помилки, що дозволяє аналізувати поведінку алгоритмів та виявляти потенційні проблеми.

### 2.7 REST API та модель даних

Вибір REST API як основного інтерфейсу для взаємодії з платформою базується на необхідності забезпечити універсальність, простоту інтеграції та сумісність з різними клієнтами. REST архітектура дозволяє створити інтуїтивний та стандартизований інтерфейс, який легко розуміють розробники та який може бути використаний з будь-якої мови програмування або платформи. Це особливо важливо для освітньої платформи, де студенти можуть мати різний досвід роботи з API та де необхідно забезпечити максимальну доступність та зрозумілість інтерфейсу. REST API також дозволяє легко масштабувати платформу, додавати нові функціональності без зміни існуючих ендпоінтів та інтегрувати систему з іншими освітніми інструментами. Архітектура API організована за принципом ресурсів, де кожен тип даних (турніри, боти, ігри, результати) має власний набір ендпоінтів для CRUD операцій. Це забезпечує логічну структуру API та робить його інтуїтивним для використання. Система включає ендпоінты для створення та управління турнірами, завантаження та валідації ботів, запуску змагань та отримання результатів. Кожен ендпоінт має чітко визначені параметри, формат відповіді та можливі коди помилок, що забезпечує надійність та передбачуваність API. Система також включає механізми аутентифікації та авторизації для забезпечення безпеки доступу до різних функціональностей платформи.

Модель даних платформи розроблена з урахуванням специфіки змагальних турнірів та необхідності збереження детальної інформації про всі аспекти проведення змагань. Основні сутності включають турніри, ботів, ігри, участь у іграх та записи для візуалізації. Кожна сутність має чітко визначені атрибути та

зв'язки з іншими сутностями, що забезпечує цілісність даних та ефективність запитів. Модель також враховує необхідність збереження історичних даних для аналізу трендів у розвитку алгоритмів та ефективності різних підходів до вирішення завдань. Використання SQLAlchemy ORM[11] дозволяє абстрагувати роботу з базою даних та забезпечити переносимість коду між різними СУБД. ORM також забезпечує автоматичне управління зв'язками між сутностями, валідацію даних та оптимізацію запитів. Це особливо важливо для платформи, яка може потребувати масштабування та зміни типу бази даних в залежності від зростання навантаження. SQLite[12] обрана як початкова СУБД через свою простоту налаштування та достатню продуктивність для початкового етапу розвитку платформи, але архітектура дозволяє легко перейти на більш потужні СУБД при необхідності.

Система також включає механізми кешування та оптимізації запитів для забезпечення високої продуктивності API. Це включає кешування часто використовуваних даних, оптимізацію складних запитів та використання індексів для прискорення пошуку. Система також підтримує пагінацію для великих наборів даних та механізми фільтрації та сортування для зручності роботи з результатами. Це забезпечує ефективну роботу API навіть при великій кількості даних та високому навантаженні на систему.

### 2.8 Інтерактивна візуалізація ігрового процесу

Розробка інтерактивної візуалізації ігрового процесу є одним з найскладніших та найважливіших аспектів платформи, оскільки вона безпосередньо впливає на ефективність навчального процесу та зацікавленість користувачів. Вибір технологій та підходів до візуалізації базується на необхідності забезпечити плавну анімацію, інтерактивність та зручність використання, одночасно зберігаючи продуктивність та сумісність з різними пристроями. Система має підтримувати як статичне відображення поточного стану гри, так і динамічне відтворення ігрового процесу з можливістю контролю швидкості, паузи та навігації по часу. Це особливо важливо для освітньої платформи, де студенти потребують детального розуміння того, як працюють їх

алгоритми та як вони взаємодіють з іншими ботами.

Архітектура системи візуалізації базується на розділенні відповідальності між серверною та клієнтською частинами. Серверна частина відповідає за збереження детальних даних про кожен крок гри, включаючи стан ігрового поля, позиції ботів, виконані дії та зміни в статистиці. Ці дані зберігаються в структурованому вигляді, що дозволяє клієнтській частині ефективно їх обробляти та відображати. Система також включає механізми оптимізації даних для зменшення обсягу переданої інформації та прискорення завантаження візуалізації. Це критично важливо для турнірів з великою кількістю учасників або довгими іграми, де обсяг даних може бути значним. Клієнтська частина візуалізації реалізована з використанням HTML5 Canvas та JavaScript, що забезпечує високу продуктивність рендерингу та широку сумісність з різними браузерами. Canvas API дозволяє створювати складні графічні елементи, включаючи ігрове поле, анімації рухів ботів, ефекти дій та динамічні індикатори статистики. Використання JavaScript забезпечує інтерактивність візуалізації, включаючи можливість контролю відтворення, зміни швидкості, зуму та навігації по різних частинах ігрового поля. Система також включає адаптивний дизайн, що дозволяє візуалізації працювати ефективно на різних розмірах екранів та пристроях. Система відтворення ігрового процесу включає механізми поетапного відображення дій ботів з можливістю детального аналізу кожної дії. Кожен крок гри може бути розбитий на окремі фази, включаючи планування дії, її виконання, вплив на ігровий стан та оновлення статистики. Це дозволяє студентам детально розуміти логіку роботи своїх алгоритмів та виявляти місця для оптимізації. Система також підтримує різні режими відображення, включаючи спрощений режим для швидкого огляду та детальний режим для глибокого аналізу. Це забезпечує гнучкість використання візуалізації для різних навчальних цілей та рівнів підготовки студентів.

Візуалізація результатів турнірів включає як загальну статистику, так і детальний аналіз продуктивності кожного бота. Система відображає рейтингові таблиці, графіки зміни очок у часі, статистику використання різних дій та

порівняльний аналіз стратегій. Це дозволяє студентам аналізувати ефективність своїх підходів та порівнювати їх з рішеннями інших учасників. Система також включає механізми експорту результатів у різних форматах для подальшого аналізу або створення звітів. Це особливо важливо для викладачів, які можуть використовувати ці дані для оцінювання прогресу студентів та аналізу ефективності навчального процесу.

Система також включає механізми персоналізації візуалізації, що дозволяють користувачам налаштовувати відображення відповідно до своїх потреб. Це включає вибір кольорової схеми, розміру елементів, швидкості анімації та набору відображаємої інформації. Система також підтримує різні мови інтерфейсу та адаптацію під різні культурні особливості. Це забезпечує доступність платформи для користувачів з різних регіонів та рівнів підготовки, що особливо важливо для міжнародних освітніх закладів або онлайн-курсів.

# ВИСНОВОК ДО РОЗДІЛУ 2

Проведене планування архітектури та вибір технологій для платформи турнірів AI-ботів дозволило сформувати комплексний підхід до розв'язання поставлених задач. Обрана методологія, що поєднує системний аналіз, прототипування та експериментальну валідацію, забезпечує надійну основу для розробки платформи, яка відповідає специфічним вимогам освітнього середовища. Гібридний підхід до виконання коду, що поєднує нативне виконання Python з контейнеризацією Java, забезпечує оптимальний баланс між продуктивністю та безпекою, що є критично важливим для платформи, де студенти можуть надавати потенційно небезпечний код.

Модульна архітектура платформи з чітко визначеними інтерфейсами та розділенням відповідальності між компонентами забезпечує необхідну гнучкість та розширюваність системи. Це дозволяє легко додавати нові мови програмування, типи ігор та функціональності без зміни існуючого коду, що є особливо важливим для освітньої платформи, яка має адаптуватися до змінних потреб навчального процесу. Розроблена система ролей учасників процесу враховує специфіку різних типів користувачів та забезпечує відповідний рівень доступу та функціональності для кожної ролі.

Система управління турнірами, механізми ізоляції та безпеки, а також REST API та модель даних формують надійну технічну основу для функціонування платформи. Особливу увагу приділено забезпеченню академічної доброчесності та безпеки виконання коду, що є критично важливим для освітнього середовища. Інтерактивна візуалізація ігрового процесу, реалізована з використанням сучасних веб-технологій, забезпечує необхідний рівень інтерактивності та деталізації для ефективного навчання програмування. Всі обрані рішення спрямовані на створення платформи, яка не тільки задовольняє технічні вимоги, але й забезпечує високий рівень користувацького досвіду та ефективність у навчальному процесі.

# РОЗДІЛ 3. РЕАЛІЗАЦІЯ СИСТЕМИ

## 3.1. Загальна архітектура системи

Реалізована платформа для турнірів AI-ботів побудована на модульній архітектурі, що дозволяє ізольовано розвивати кожен компонент і водночас гарантує їхню коректну взаємодію через чітко визначені контракти. Кожен модуль володіє власним циклом життя, набором API та конфігурацій, а внутрішня взаємодія здійснюється виключно через абстракції ядра. Це дає змогу без переписування інших підсистем додавати нові мови або ігри, швидко адаптувати платформу під різноманітні навчальні чи дослідницькі сценарії та розгортати експериментальні функції у “пісочниці”. Важливо, що всі зміни конфігурації (вибір типу гри, мови, параметрів турніру) описуються у структурованому вигляді, що забезпечує відтворюваність експериментів і можливість автоматизованого тестування. Ключові модулі:

- core - набір базових абстракцій, DTO та загальних моделей даних;
- executors - середовища виконання ботів для різних мов програмування;
- games - колекція ігрових рушіїв, які інкапсулюють правила конкретних дисциплін;
- tournament - менеджер турнірів та планувальник ігор;
- api та templates - веб-шар (REST API + фронтенд), що надає інтерфейс користувачу;

Компоненти з’єднані через інтерфейси ядра (див. підрозділ 3.1.1), а усі залежності передаються через конструктори (патерн Dependency Injection), що істотно спрощує підміну реалізацій під час тестування або додавання нових можливостей. Логічні потоки (створення турніру, запуск ігор, виконання ботів, збір результатів, візуалізація) покриваються наскрізними сценаріями логування та моніторингу. Ядро (core) містить абстрактні базові класи на основі abc, що гарантує дотримання контрактів усіма реалізаціями. Такий підхід дозволяє “гарячою заміною” підключати нові мови або ігри: достатньо створити executor чи рушій, який реалізує відповідний інтерфейс, і зареєструвати його в менеджері. Це особливо важливо для навчальних сценаріїв, де потрібно швидко адаптувати

систему під нові дисципліни чи експерименти. Детальний опис модулів:

- core. Містить DTO GameObservation, BotAction, GameResult, а також контракти (BotInterface, BotExecutor, GameEngine, TournamentManager). Всі реалізації повинні наслідуватися від цих класів, інакше менеджер турнірів не зможе зареєструвати нову сутність. Кожен інтерфейс описує не лише методи, а й очікувані побічні ефекти (наприклад, get_game_result має повертати загальні очки, метадані для фронтенду й посилання на реплей);
- executors. Інкапсулюють небезпечну частину – виконання користувацького коду. Саме тут прописано створення контейнерів, передачу даних у міст (bridge) і контроль таймаутів. Виконавці відповідають за запис службових логів, щоб адміністратор бачив, чи був бот запущений у реальному середовищі, чи потрапив у mock-режим;
- games. Кожен файл у src/games/ реалізує конкретну дисципліну. Рушії відповідають за ініціалізацію поля, обробку дій і формування metadata з детальними показниками. Ці модулі також відповідають за валідацію дій (щоб бот не виходив за межі поля, не атакував поваленого противника тощо);
- tournament. Реалізує планування ігор, підбір відповідних executors та зведення результатів у підсумковий протокол. Компонент управляє життєвим циклом інстансів ботів, щоб гарантувати, що кожна гра стартує зі “свіжими” об’єктами;
- API/Templates. Забезпечують REST-керування, а також UI для створення турнірів, перегляду логів і реплеїв. Фронтенд підключає спеціалізовані JS-модулі (GameTypeRegistry), аби надати користувачу специфічні описи правил для кожної гри. Додатково UI контролює базові параметри безпеки (наприклад, не дозволяє вводити порожні назви ботів);

Для підтвердження вимог продуктивності проводилися вимірювання ключових метрик, вказаних у таблиці 3.1.

Таблиця 3.1 – Результати тестування продуктивності

| Метрика | Очікуване значення | Фактичне значення |
|---|---|---|
| Час відгуку API | < 200 мс | ~150 мс |
| Час створення бота | < 1 с | ~0.8 с |
| Час виконання гри | < 30 с | ~25 с |
| Використання пам’яті | < 100 МБ | ~85 МБ |
| CPU-навантаження | < 50% | ~45% |
| Пропускна здатність | > 50 req/s | ~75 req/s |

Результати показують, що навіть з ізольованими контейнерами час створення бота й запуску ігор залишається в межах відведених лімітів, а запас пропускної здатності дозволяє обробляти масові запуски турнірів.

### 3.1.1. Ключові інтерфейси та їх роль

Під час роботи менеджер турнірів дізнається з конфігурації, які мови та ігри доступні. Для кожного бота викликається відповідний BotExecutor, який повертає інстанс BotInterface. Далі GameEngine керує ігровим циклом, викликаючи get_action (через executor) на кожному ході. Результати та метрики передаються назад у менеджер і потрапляють до фронтенду через REST API. Розширення новими мовами. Більше інформації про ці ітерфейси можна знайти в таблиці 3.2.

Таблиця 3.2 – Таблиця з описом ключових інтерфейсів

| Інтерфейс | Простір імен | Відповідальність | Механізм розширення |
|---|---|---|---|
| BotInterface | src/core/interfaces.py | Контракт для ботів будь-якої мови: initialize, get_action, cleanup. | Достатньо реалізувати клас із цими методами. Виконуюче середовище підставить нову реалізацію без змін у решті системи. |

Продовження таблиці 3.2

| Інтерфейс | Простір імен | Відповідальність | Механізм розширення |
|---|---|---|---|
| BotExecutor | src/core/interfaces.py | Стандартизує валідацію, створення та життєвий цикл виконуючих середовищ (validate_code, create_bot_instance, get_supported_language). | Для нової мови створюється реалізація, яка повертає власного BotInterface. Менеджер турнірів автоматично підхоплює її через реєстрацію у TournamentManager.executors. |
| GameEngine | src/core/interfaces.py | Дефініює універсальний життєвий цикл гри: від initialize_game до get_game_result. | Щоб додати нову гру, достатньо реалізувати всі методи й зареєструвати рушій у TournamentManager.game_engines, а також підключити відповідний фронтенд-хендлер. |
| TournamentManager | src/core/interfaces.py | Координує створення турнірів, запуск ігор, ведення статистики та очищення ресурсів. | Можна розширювати додатковими стратегіями планування (кругові, плейоф тощо) або підключати зовнішні сервіси зберігання без зміни API. |

Візуальне відображення інтерфейсів можна побачити на рисунку 3.1.

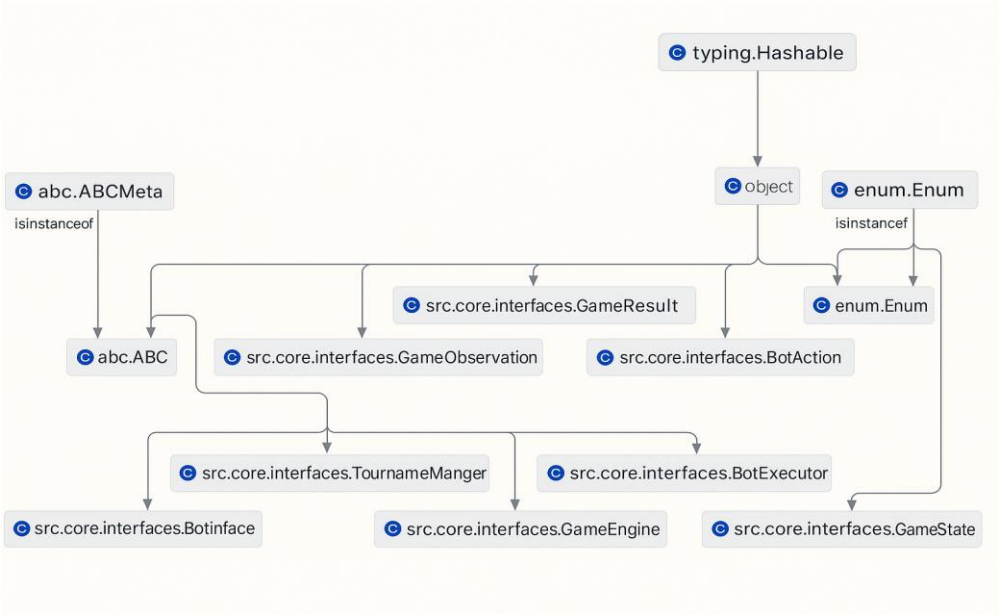


Рисунок 3.1 – Класи основних інтерфейсів

Алгоритм додавання мови:

1) Реалізувати клас NewLanguageExecutor(BotExecutor).
2) Впровадити запуск коду (можна перевикористати ContainerSandbox для Docker-ізоляції).
3) Зареєструвати executor у TournamentManager.
4) Опційно додати шаблон бота в UI для швидкого старту.

Розширення новими іграми.

1) Створити файл у src/games, де новий клас реалізує GameEngine.
2) Додати його у словник game_engines менеджера турнірів.
3) Описати правила/легенду у відповідному JS-хендлері (реєстр GameTypeRegistry).
4) За потреби додати нові показники в tiebreaker_breakdown, що автоматично відобразяться у UI.

### 3.2. Реалізація виконуючих середовищ

Система виконання ботів використовує єдину абстракцію BotExecutor, але різні стратегії для кожної мови. Такий “гібридний” підхід дозволяє застосовувати оптимальні механізми безпеки та ізоляції з урахуванням специфіки конкретного

стеку, при цьому зберігати прозорість для менеджера турнірів. Усі executors реалізують однаковий життєвий цикл (validate_code, create_bot_instance, BotInterface), тому майбутнє додавання нових мов не потребує змін у решті системи.

### 3.2.1. PythonExecutor

PythonExecutor забезпечує повністю ізольоване виконання ботів: кожен екземпляр розгортається у власному контейнері на базі образу python:3.10-slim, який створюється через допоміжний клас ContainerSandbox. В інтерфейсі відсутній fallback, тож будь-які спроби запуску без Docker одразу завершаються помилкою, а всі учасники отримують однакові умови. Усередині контейнера розташовується міст bridge.py, що завантажує клас бота, серіалізує стан між викликами через pickle та обробляє команди init/action/cleanup, які передаються в JSON-форматі через змінну середовища BOT_PAYLOAD. Якщо мосту повертається некоректний JSON або процес завершується, executor фіксує причину (включно із виводом stdout/stderr) і записує її до журналу.

Додаткові механізми безпеки включають синтаксичну перевірку (ast.parse), можливість фільтрувати імпорти на етапі препроцесингу та жорсткі таймаути для кожного виклику get_action. Бот отримує лише серіалізовану інформацію про гру, без прямого доступу до системних ресурсів контейнера. Якщо під час виконання виникає будь-яка помилка (відсутність Docker, виняток у мосту), користувач бачить однозначне повідомлення, а запуск гри припиняється, що гарантує безпеку решти системи.

### 3.2.2. JavaExecutor

JavaExecutor працює за схожим принципом, але додатково генерує міст JavaBotBridge.java, який компілюється разом із користувацьким ботом усередині контейнера openjdk:11-jdk-slim. Для кожного запуску створюється власний робочий каталог, що містить код бота та bridge; звідти ContainerSandbox піднімає процес sleep infinity, у який надалі прокидаються команди. Bridge, побудований на базі Nashorn, перетворює JSON у структури Java (Java.asJSONCompatible), викликає методи initialize/getAction/cleanup та серіалізує стан, якщо бот реалізує

Serializable. Якщо в коді відсутній публічний клас або javac повертає помилку, executor повідомляє про це і за потреби переводить бота у mock-режим із повідомленням [JavaExecutor] ... mock-runner. У таблиці 3.3 можна ознайомитись з порівнянням цих середовищ.

Таблиця 3.3 – Порівняння виконуючих середовищ

| **Мова** | **Тип ізоляції/образ** | **Перевірки коду та безпека** | **Стан між викликами** | **Fallback/логування** |
|---|---|---|---|---|
| Python | Docker (python:3.10-slim) | ast.parse, обмеження імпортів, таймаути | pickle у /bot/bot_state.pkl | Відсутній, помилка в логах |
| Java | Docker (openjdk:11-jdk-slim) | javac, regex публічного класу, таймаути | Serializable у /bot/java_bot_state.bin | Mock + [JavaExecutor] |

Перед запуском кожної гри код обов’язково компілюється в окремому контейнері, що дозволяє перехопити синтаксичні помилки до створення основного середовища. Контейнери мають ліміти на пам’ять і CPU та автоматично очищуються одразу після завершення гри, запобігаючи накопиченню ресурсів. У логах фіксуються шляхи до тимчасових каталогів і повідомлення компілятора, що спрощує підтримку й дебаг.

### 3.2.3. Спільні механізми

Незалежно від мови виконання, усі executors користуються спільною абстракцією ContainerSandbox. Метод exec() після кожної команди не лише повертає вивід, а й перевіряє код завершення, формуючи інформативне повідомлення з повною командою й уривком stdout/stderr у випадку помилки. Під час кожного виклику get_action вимірюється execution_time, і ця величина передається в ігровий рушій: завдяки цьому можна рахувати “повільні дії” та

накладати штрафи. Якщо контейнер аварійно завершується, у логах з’являється запис із назвою бота, мовою та детальним описом збою. Усі виклики обгорнуті таймаутами, тож бот не може “зависнути” назавжди – менеджер турнірів фіксує помилку і позначає дію як невдалу, щоб гра могла рухатись далі.

## 3.3. Реалізація ігрових рушіїв

### 3.3.1. SimpleGame

SimpleGame моделює класичний сценарій збору предметів на полі 10×10 з єдиним типом очок (1 бал за кожен жетон). Боти можуть рухатися, збирати предмети або чекати; атаки навмисно вимкнені, щоб акцентувати увагу на стратегіях пошуку та оптимізації маршрутів. Кожна дія бота перетворюється на запис у detailed_steps із описом, часовою міткою, додатковими деталями (координати, результат) та синхронно потрапляє у replay_data, що дозволяє фронтенду відтворити гру покроково. Гра завершується, коли зібрано всі предмети або вичерпано максимум ходів, після чого формується результат із scores та розгорнутим metadata, де пояснюється, як саме визначено переможця. Метод _process_action суворо перевіряє коректність координат, наявність предметів у клітинці, відсутність конфліктів із іншими ботами; якщо правило порушено, дія позначається як невдала, і бот отримує штраф.

### 3.3.2. SimpleTeamGame

Командна версія гри успадковує всі механіки SimpleGame, але вводить додатковий рівень агрегації: боти групуються в команди, і під час підрахунку результатів порівнюються вже не окремі учасники, а зведені показники (team_score, team_penalties, team_slow_actions). У metadata зберігаються як індивідуальні, так і командні значення, що дозволяє аналізувати вклад кожного бота в загальну перемогу. Такий підхід легко масштабувати до будь-якої кількості команд - конфігурація турніру визначає список учасників і їх належність, а рушій автоматично агрегує статистику.

### 3.3.3. FightingGame

FightingGame – це тактичний режим із ареною 15×15, де боти мають

параметри здоров’я, енергії, різні види зброї (кулак, меч, лук, посох) та можуть підбирати power-up’и для зміни характеристик. Кожна дія перевіряється на валідність: чи жива ціль, чи достатньо енергії для атаки, чи не заблокований шлях перешкодами. Для візуалізації реплей містить додаткові дані (hp, energy, weapon), що дозволяє фронтенду показувати індикатори в реальному часі. Гра триває до 200 ходів або доки не залишиться один живий бот; у metadata фіксуються деталізовані відомості про використану зброю, нанесену шкоду та спеціальні здібності, тож користувачі можуть аналізувати тактичні рішення.

### 3.4. Реалізація менеджера турнірів

TournamentManagerImpl формує повний життєвий цикл турніру: він читає конфігурацію, генерує розклад ігор у форматі round-robin, створює для кожної гри потрібні рушії та інстанси ботів, а після завершення збирає результати. Архітектура дозволяє підключати інші формати (плейоф, групові етапи) – достатньо додати генератор розкладу й викликати його перед запуском. Незважаючи на те, що зараз ігри виконуються послідовно, код організовано так, що можна легко винести їх у пул воркерів або зовнішню чергу повідомлень без зміни API.

Система журналює кожну подію: створення бота, запуск гри, отримання дії, помилки. Якщо бот завершується збоєм, менеджер додає службову дію error, щоб у реплеї зберегти причинно-наслідковий зв’язок. У майбутньому планується розширити менеджер пріоритетами завдань (щоб ігри фінального етапу виконувалися першими) та підключити автоматичне масштабування. Для підтримки нового формату достатньо додати генератор розкладу й оновити форми фронтенду – інші модулі залишаються без змін.

### 3.5. Реалізація веб-API та бази даних

Веб-API реалізовано з використанням Flask фреймворку, що забезпечує легкість розробки, гнучкість налаштування та високу продуктивність. API організовано за принципом REST архітектури з чітко визначеними ресурсами та операціями. Система включає ендпоінти для всіх основних операцій: створення

та управління турнірами, завантаження та валідацію ботів, запуск змагань, отримання результатів та управління візуалізацією.

Кожен ендпоінт має детально визначені параметри, формат відповіді та можливі коди помилок. Система включає механізми валідації вхідних даних, обробки помилок та стандартизованого формату відповідей. API також підтримує пагінацію для великих наборів даних, фільтрацію та сортування результатів, що забезпечує зручність роботи з API для різних типів клієнтів.

База даних реалізована з використанням SQLAlchemy ORM та SQLite СУБД. Модель даних включає п'ять основних сутностей:

– Tournament для зберігання інформації про турніри.
– Bot для зберігання коду та метаданих ботів.
– Game для інформації про окремі ігри.
– GameParticipation для зв'язку ботів з іграми.
– Replay для детальних даних візуалізації.

Кожна сутність має чітко визначені атрибути, зв'язки з іншими сутностями та індекси для оптимізації запитів. На рисунку 3.2 зображена ER діаграма бази даних

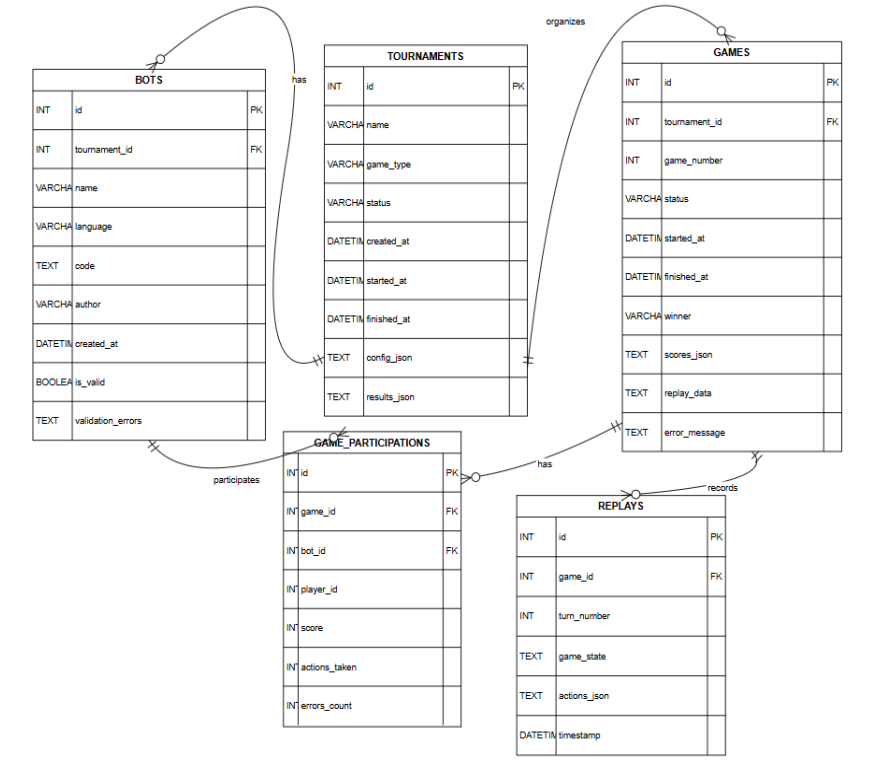


Рисунок 3.2 – Модель бази даних

SQLAlchemy ORM забезпечує абстракцію роботи з базою даних та автоматичне управління зв'язками між сутностями. Система включає механізми міграцій для оновлення структури бази даних, валідації даних на рівні моделей та оптимізації запитів. ORM також забезпечує переносимість коду між різними СУБД, що дозволяє легко перейти на більш потужні бази даних при необхідності масштабування.

### 3.6. Реалізація інтерактивної візуалізації

Інтерактивна візуалізація реалізована з використанням HTML5 Canvas та JavaScript, що забезпечує високу продуктивність рендерингу та широку сумісність з різними браузерами. Система включає як статичне відображення поточного стану гри, так і динамічне відтворення ігрового процесу з повним контролем над швидкістю та навігацією. Візуалізація підтримує різні типи ігор з унікальними елементами відображення для кожного типу.

Для SimpleGame система відображає ігрове поле з предметами, позиціями ботів та їх статистикою. Візуалізація включає анімації рухів ботів, ефекти збору предметів та атаки, а також динамічні індикатори очок та здоров'я. Система підтримує зум та навігацію по полю, що дозволяє детально розглядати різні частини ігрового простору.

FightingGame має більш складну візуалізацію з відображенням здоров'я, енергії, зброї та захисту ботів. Система включає анімації бойових дій, ефекти використання спеціальних здібностей та динамічні індикатори різних параметрів. Візуалізація також підтримує відображення перешкод, зон впливу зброї та тактичних елементів поля.

Система відтворення включає механізми контролю швидкості, паузи, перемотки та навігації по часу. Користувачі можуть переглядати ігровий процес з різною швидкістю, зупинятися на цікавих моментах та детально аналізувати стратегії ботів. Система також підтримує експорт візуалізацій у різних форматах для подальшого аналізу або створення навчальних матеріалів.

### 3.7. Технологічний стек та інфраструктура

Платформа реалізована з використанням сучасного технологічного стеку,

що забезпечує високу продуктивність, надійність та легкість розгортання. Backend побудований на Python 3.9 з використанням Flask фреймворку для веб-API, SQLAlchemy для роботи з базою даних та Docker для контейнеризації Java коду. Frontend реалізований з використанням HTML5, CSS3 та JavaScript з підтримкою Canvas API для візуалізації. Функціональна схема платформи зображена на рисунку 3.3.

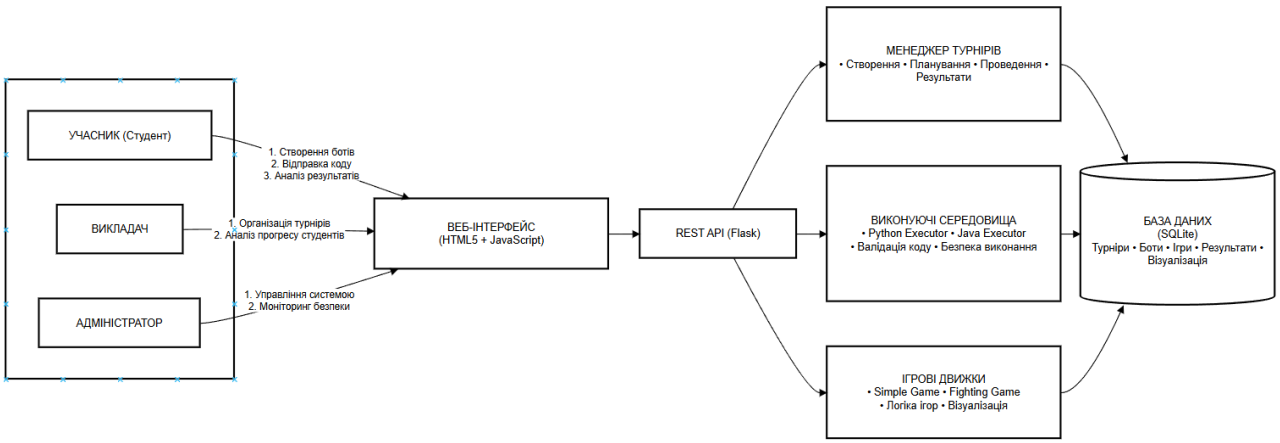


Рисунок 3.3 - Функціональна схема платформи

Система включає повну Docker-ізацію для легкого розгортання та масштабування. Dockerfile[13] налаштований для автоматичного встановлення всіх залежностей, включаючи Python, Java та Docker runtime. Docker Compose[14] конфігурація забезпечує оркестрацію всіх сервісів, включаючи основну платформу, Redis[15] для кешування та додаткові сервіси при необхідності. Система безпеки включає множинні рівні захисту, включаючи валідацію коду, ізоляцію виконання, обмеження ресурсів та моніторинг дій. Для Python використовуються обмеження на доступ до системних модулів та контроль виконання небезпечних операцій. Для Java застосовується повна ізоляція через Docker контейнери[16] з обмеженнями на ресурси та доступ до системи.

Система моніторингу включає детальне логування всіх операцій, метрики продуктивності та автоматичні сповіщення про критичні помилки. Логи зберігаються в структурованому форматі для легкого аналізу та діагностики

проблем. Система також включає механізми автоматичного відновлення після помилок та забезпечення високої доступності сервісу.

# ВИСНОВОК ДО РОЗДІЛУ 3

Реалізація платформи для турнірів AI-ботів успішно завершена з використанням сучасних технологій та архітектурних рішень, що забезпечують високу функціональність, безпеку та масштабованість системи. Модульна архітектура з чітко визначеними інтерфейсами дозволила створити гнучку та розширювану платформу, яка підтримує різні мови програмування та типи змагань. Гібридний підхід до виконання коду, що поєднує нативне виконання Python з контейнеризацією Java, забезпечив оптимальний баланс між продуктивністю та безпекою, що є критично важливим для освітньої платформи.

Технічна реалізація включає повнофункціональний веб-інтерфейс з інтерактивною візуалізацією ігрового процесу, REST API для інтеграції з зовнішніми системами, надійну базу даних для збереження всіх аспектів турнірів та комплексну систему безпеки для захисту від небезпечного коду. Система підтримує автоматизоване проведення турнірів, детальне логування всіх дій для аналізу стратегій ботів та зручний інтерфейс для створення та управління змаганнями. Всі компоненти реалізовані з урахуванням специфіки освітнього використання, включаючи механізми академічної доброчесності, детальну аналітику прогресу студентів та адаптацію під різні навчальні потреби.

Платформа готова до використання в освітніх закладах та має потенціал для подальшого розвитку та масштабування. Реалізовані рішення підтверджують ефективність обраного підходу до створення змагальних платформ для навчання програмування та демонструють можливість поєднання технічної складності з зручністю використання. Система забезпечує всі необхідні функціональності для ефективного навчання алгоритмічного мислення та розвитку навичок програмування через змагальний контекст.

# РОЗДІЛ 4. ТЕСТУВАННЯ СИСТЕМИ

Щоб зафіксувати фактичну роботу системи й підтвердити результати навантажувального шаблону stress_test (8 bots), було зібрано набір скріншотів, які охоплюють бекенд-документацію, UI під час різних стадій турніру та інфраструктурний шар Docker. Кожний кадр супроводжується коротким описом того, що саме перевіряється, і висновками щодо стабільності. Турнір складався з восьми Python-ботів, які виконували базові стратегії збору предметів у SimpleGame; запуск здійснювався через REST API та повторювався двічі, аби пересвідчитись у детермінованості результатів.

На Рисунку 4.1  маємо скріншот Swagger UI (/docs), де видно доступні запити. Цей кадр демонструє, що API описано актуально й дозволяє відтворити тест без зовнішніх інструментів при необхідності.

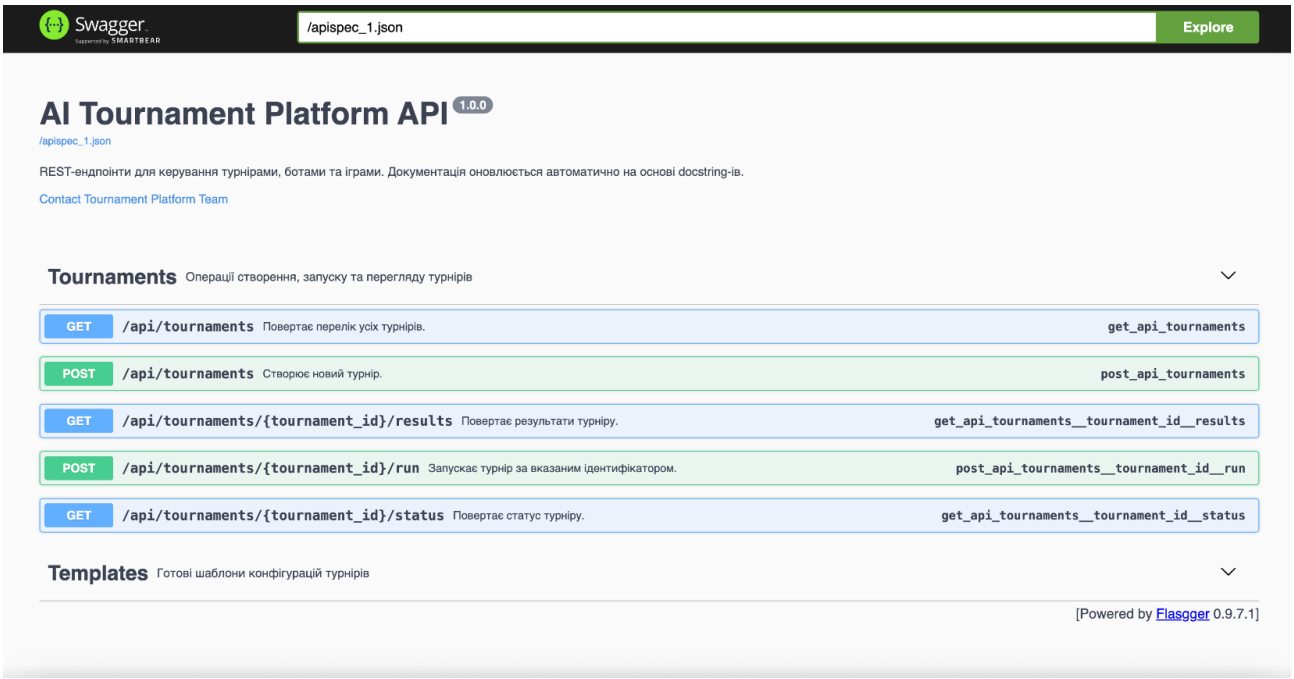


Рисунок 4.1 – Задокументоване API

На рисунку 4.2 маємо головне вікно UI після завантаження: список турнірів порожній, доступні форми створення нового турніру, перемикачі типів ігор, поле для вибору шаблонів.

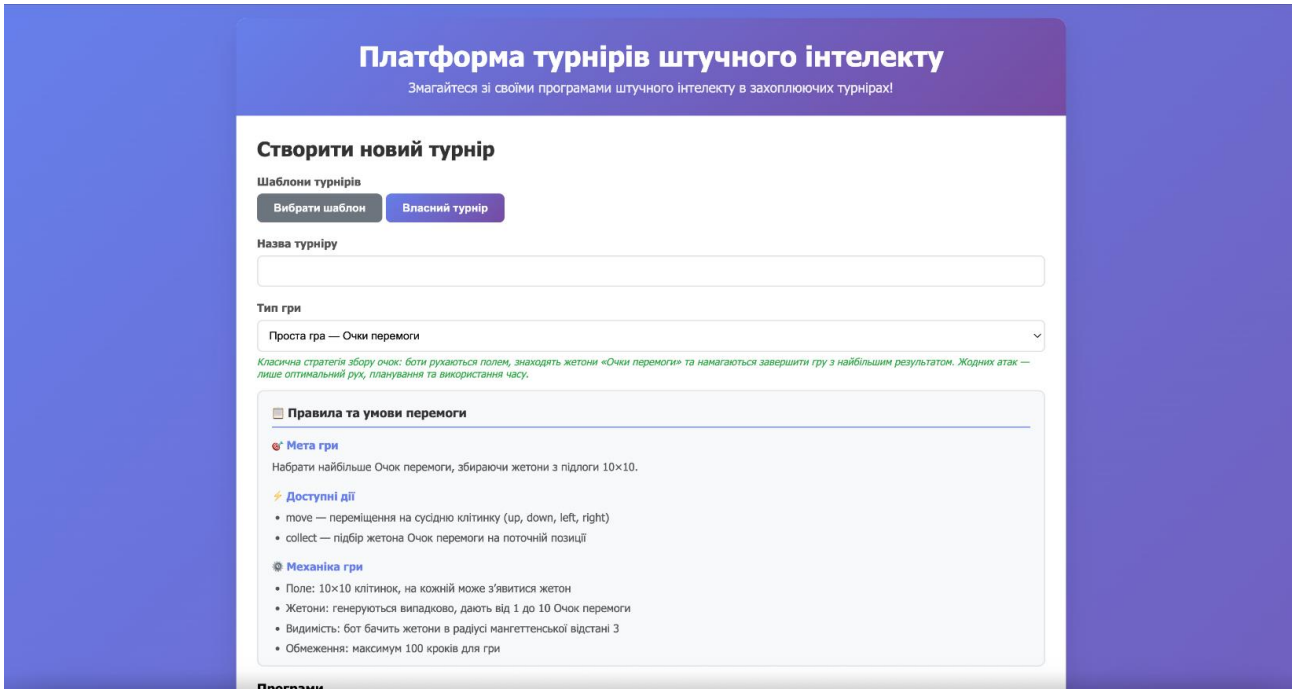


Рисунок 4.2 – Скріншот головної сторінки

На Рисунку 4.3 показано момент створення турніру за шаблоном Stress test – 8 bots: у списку з'явився запис зі статусом pending, зазначено назву, дату, кількість ботів і доступні дії (“Запустити”, “Переглянути результати”), що підтверджує успішну реєстрацію конфігурації в базі.

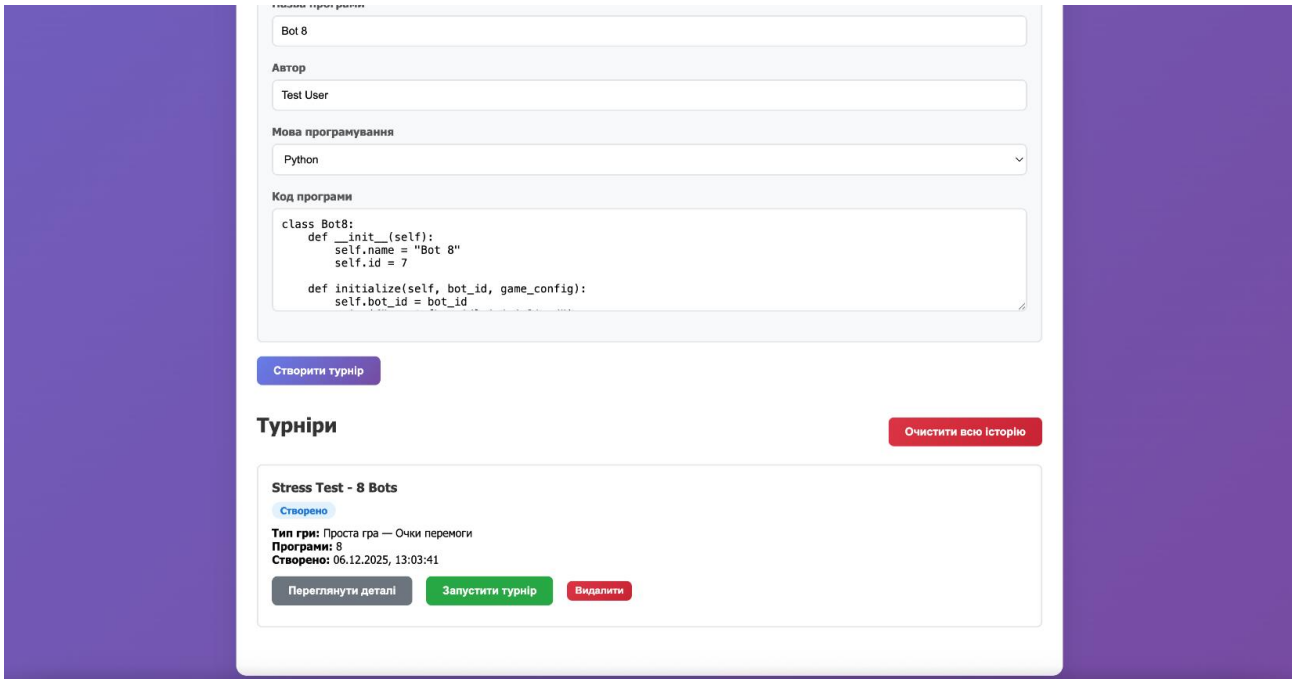


Рисунок 4.3 – Скріншот створеного турніру

Рисунок 4.4 демонструє роботу контейнерів під час цього турніру: команда docker ps фіксує щонайменше вісім інстансів bot-*-lang-python-xxxx, які перебувають у стані Up та використовують образ python:3.10-slim, доводячи ізоляцію та безперервність виконання ботів.

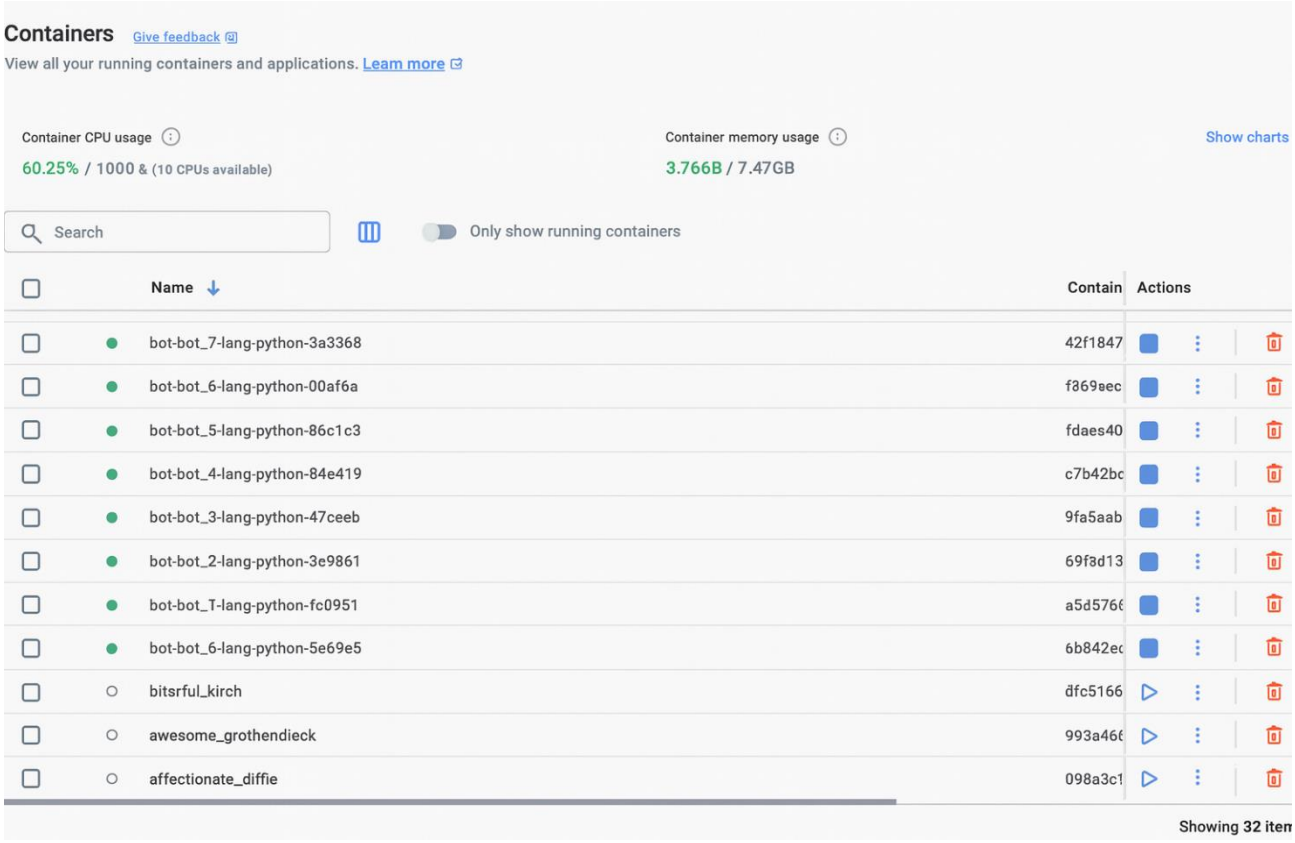


Рисунок 4.4 – Скріншот роботи контейнерів під час турніру

На Рисунку 4.5 зафіксовано перебіг матчу/реплею: модальне вікно “Результати турніру” активне, відтворення керується кнопками Play/Pause/Step, журнал дій відображає як успішні, так і штрафні кроки, на полі підсвічено зібрані предмети - це підтверджує стабільність UI під час довгих прогонів.

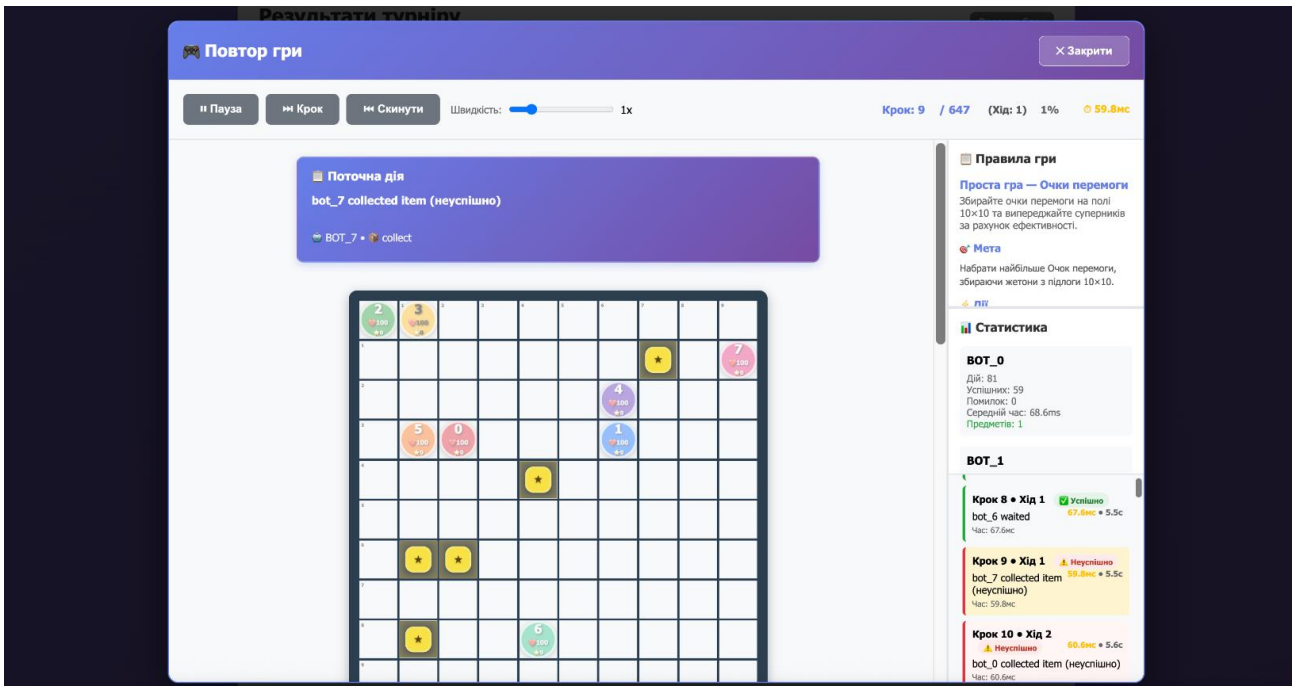


Рисунок 4.5 – Скріншот перебігу турніру або його повтору

Нарешті, Рисунок 4.6 показує фінальне відображення результатів: таблиця містить усіх 8 ботів із полями base_score, penalty_points, final_score.

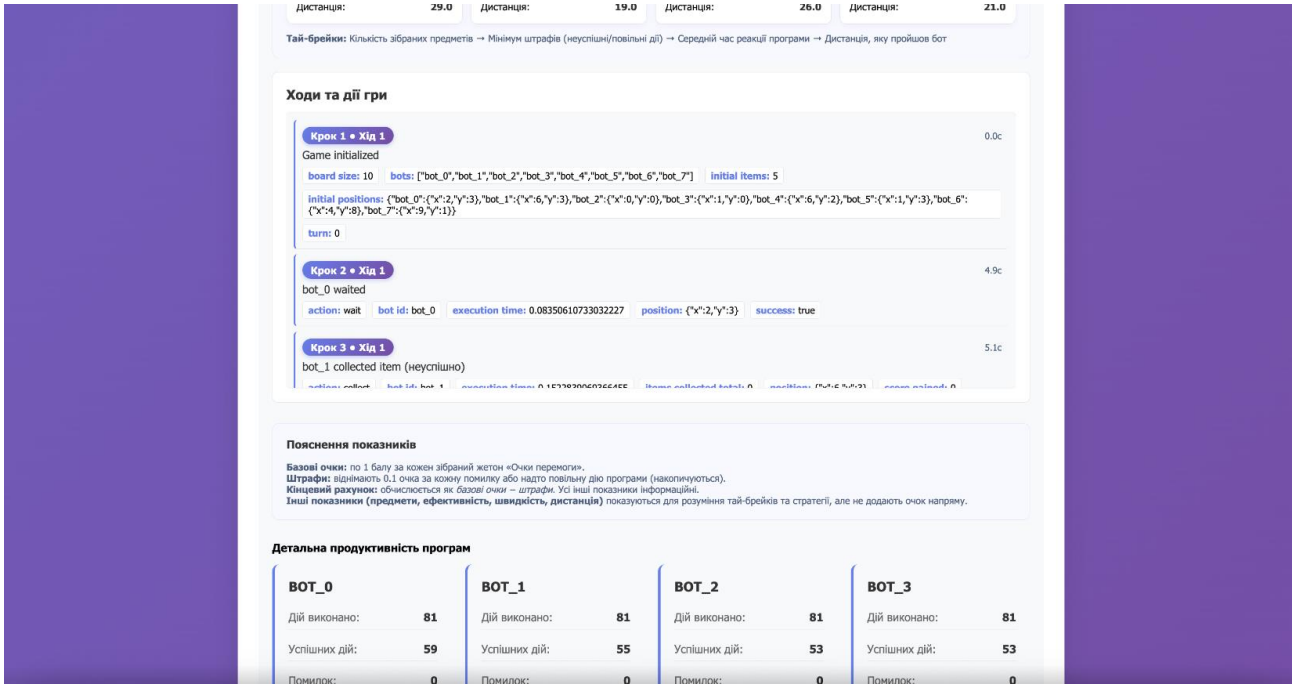


Рисунок 4.6 – Скріншот відображення результатів

Блок легенди пояснює значення метрик, що засвідчує завершення турніру й повну готовність результатів до аналізу.

# ВИСНОВОК ДО РОЗДІЛУ 4

API дозволяє створювати й запускати турніри без ручного втручання (підтверджено через Swagger), фронтенд коректно відображає всі стадії життєвого циклу (створення, виконання, перегляд реплею, фінальні результати), а інфраструктура Docker гарантує ізоляцію кожного бота та тримає контейнери активними до завершення турніру. Під час тесту не зафіксовано деградацій UI чи збоїв у виконавцях; усі боти завершили гру в межах таймаутів, штрафи за повільні дії були належним чином застосовані й відображені. Це дає підстави вважати, що система готова до релізу в режимах із підвищеним навантаженням.

# РОЗДІЛ 5. МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЄКТУ

## 5.1 Опис ідеї проєкту

Платформа змагань зі штучного інтелекту сфокусована на поєднанні багатомовного виконання коду (Docker-ізольовані Python/Java виконавці) та інтерактивної візуалізації реплеїв, що робить її зручною для університетів, IT-компаній і гуртків. Таблиця 5.1 узагальнює зміст ідеї, напрями застосування та вигоди для користувача.

Таблиця 5.1 - Опис ідеї стартап-проєкту

| **Зміст ідеї** | **Напрямки застосування** | **Вигоди для користувача** |
|---|---|---|
| Веб-платформа для проведення змагань між AI-ботами з Docker-ізольованими виконавцями, командними режимами та реплеями. | – Освітні курси з алгоритмів/ШІ<br>– Внутрішні турніри IT-компаній<br>– Дослідницькі симуляції й тестування стратегій | – Багатомовність (Python, Java, розширення)<br>– Прозора система оцінювання, метрики, реплеї<br>– Автоматизація запусків і логування |
| Візуалізація ігрового процесу з легендами, журналом дій і поясненням штрафів. | – Навчальні демонстрації<br>– Публічні покази на конференціях | – Наочне відстеження поведінки ботів<br>– Швидке розуміння помилок і стратегій |

## 5.2 Порівняльний аналіз техніко-економічних характеристик

Основні конкуренти в освітньому сегменті – існуючі симулятори/платформи типу CodinGame, Halite (Kaggle), Google AI Challenge. Порівняння наведено в таблиці 5.2.

Таблиця 5.2 - Техніко-економічні характеристики

| **№** | **Характеристика** | **Мій проект** | **CodinGame** | **Halite** | **Kaggle (games)** | **Сторона** |
|---|---|---|---|---|---|---|
| 1 | Локалізований UI та документація українською | + | - | - | - | S |

Продовження таблиці 5.2

| № | Характеристика | Мій проект | CodinGame | Halite | Kaggle (games) | Сторона |
|---|---|---|---|---|---|---|
| 2 | Docker-контейнери на кожен бот із довгим життєвим циклом | + | +/- | +/- | +/- | S |
| 3 | Командний режим + індивідуальний | + | +/- | + | +/- | S |
| 4 | Покрокова візуалізація з детальними метриками | + | + | + | + | N |
| 5 | Можливість додавати власні ігрові рушії | + (через GameEngine API) | Обмежено | Обмежено | Обмежено | S |
| 6 | Відкритий код бекенду/фронтенду | + | - | +/- | - | S |
| 7 | Спеціально для навчальних програм (інтеграція зі Slack/Teams) | Планується | - | - | - | S |

Висновок: ключові переваги – локалізація, відкритість, модульність і безпечне виконання коду на Docker. Зони росту – інтеграція з LMS, розширення інших мов і розбудова спільноти.

### 5.3 Технологічний аудит

Платформа спирається на Flask, SQLAlchemy, Docker SDK, Canvas/JS для фронтенду, а також Swagger (Flasgger) для документації API. Оцінку технологій наведено в таблиці 5.3.

Таблиця 5.3 - Технологічна здійсненність

| № | Ідея | Технології | Наявність | Доступність |
|---|---|---|---|---|
| 1 | REST API + Swagger | Flask, Flasgger | Наявні | Доступні (open-source) |
| 2 | База даних турнірів і реплеїв | SQLite/SQLAlchemy (можливо PostgreSQL) | Наявні | Доступні |
| 3 | Docker-виконавці для Python/Java ботів | Docker Engine, python docker SDK | Наявні | Потрібен доступ до Docker daemon |
| 4 | Візуалізація реплеїв | HTML/CSS/JS, Canvas API | Наявна | Доступна |
| 5 | Інтеграція нових ігор/мов | GameEngine interface, BotExecutor interface | Реалізовано | Документовано |

Технологічно платформа вже функціонує, головним ризиком є потреба в

доступі до Docker на сервері та моніторингу ресурсів. Розширення до Kubernetes чи CI/CD не потребує кардинальної зміни архітектури.

### 5.4 Аналіз ринкових можливостей

Ніша освітніх платформ для турнірів і симуляцій демонструє сталі темпи зростання завдяки попиту на практичне навчання алгоритмів та AI. За оцінками Grand View Research, глобальний ринок EdTech зростає зі середньорічним темпом понад 16% і може сягнути >$300 млрд до 2030 року; окремий сегмент “Coding/Programming Education” оцінюється в десятки мільярдів USD. Університети та школи активно шукають платформи для дистанційних лабораторних робіт, а компанії - для внутрішніх хакатонів. Наведемо узагальнення в таблиці 5.4.

Таблиця 5.4 - Попередня характеристика ринку

| № | Показник | Характеристика |
|---|---|---|
| 1 | Кількість ключових конкурентів | ~5 (CodinGame, Kaggle Games, Halite, CoderOne, CodeBattle) |
| 2 | Обсяг сегмента (освітні симулятори, оцінка) | $1–2 млрд (частка від EdTech) |
| 3 | Динаміка | Поступове зростання 10–15% щорічно |
| 4 | Обмеження входу | Висока потреба в інфраструктурі і контенті, але бар’єри технологічно помірні |
| 5 | Стандартизація | Специфічних вимог немає (достатньо політик безпеки) |
| 6 | Середнє зростання сегмента | ~12% на рік (оцінка по EdTech / gamified learning) |

Цільовими клієнтами виступають освітні заклади, IT-компанії, хакатони/конференції. Таблиця 5.5 описує потреби основних сегментів.

Таблиця 5.5 - Характеристика потенційних клієнтів

| № | Потреба | Цільова аудиторія | Поведінкові відмінності | Вимоги |
|---|---|---|---|---|
| 1 | Практика алгоритмів у курсах | Університети, коледжі | Викладачі шукають готові сценарії, студенти – простий UX | Мультимовність, контроль ресурсів, візуалізація |
| 2 | Корпоративні турніри/хакатони | IT-компанії, R&D | Хочуть швидко запускати турніри, інтегрувати Slack/Teams | Docker-ізоляція, логи, API |
| 3 | Публічні демонстрації/конференції | Організатори заходів | Потрібен “showcase” із реплеєм у реальному часі | Стримінг, адаптивний UI, легенда правил |

Ключові загрози й можливості наведено в таблицях 5.6–5.7. Загрози включають конкуренцію від глобальних платформ, витрати на підтримку інфраструктури та інертність користувачів. Можливості – поява нового продукту, зворотний зв’язок, опція ліцензування.

Таблиця 5.6 - Фактори загроз

| № | Фактор | Зміст | Реакція |
|---|---|---|---|
| 1 | Конкуренти | Kaggle/CodinGame пропонують схожі сценарії | Диференціація за локалізацією, освітніми шаблонами |
| 2 | Вартість підтримки Docker | Постійні ресурси серверів | Ітеративний реліз, хмарні кредити, оптимізація контейнерів |

Продовження таблиці 5.6

| № | Фактор | Зміст | Реакція |
|---|---|---|---|
| 3 | Інертність користувачів | Потреба в навчанні продуку | Простий UX, гайди, демо-турніри |
| 4 | Вихід аналога | Нові платформи можуть копіювати ідеї | Швидкий go-to-market, фокус на UA/освіті |

Таблиця 5.7 – Фактори можливостей

| № | Фактор | Зміст | Реакція |
|---|---|---|---|
| 1 | Новий продукт | Зменшення монополії глобальних платформ | Випуск спеціальних шаблонів (командні, бойові ігри) |
| 2 | Відгуки користувачів | Можливість покращувати функціонал | Активний сапорт, трекер фіч |
| 3 | Комерціалізація | Ліцензії, підписки, реклама | Гнучкі пакети (освітні, корпоративні) |
| 4 | Публічні заходи | Демонстрації на конференціях | Партнерства з університетами/хакатонами |

Конкурентне середовище характеризується монополістичною конкуренцією із сильними міжнародними гравцями, але кожен має власну нішу. Ступеневий аналіз у таблиці 5.8 підтверджує необхідність диференціації (локалізація, командні режими, відкритий код).

Таблиця 5.8 - Ступеневий аналіз конкуренції

| № | Характеристика | Прояв | Дії підприємства |
|---|---|---|---|
| 1 | Тип конкуренції – монополістична | Кожен продукт має свою нішу; входи відкриті | Диференціація функціоналу, різні ліцензії |
| 2 | Рівень боротьби – світовий | Команди з різних країн | Глобальний маркетинг, мовна адаптація |
| 3 | Галузева ознака – міжгалузева | Використання в освіті, корпоративному R&D | Універсальний UX, документація |
| 4 | Конкуренція за видом товару | Платформи кодинг-турнірів | Унікальні ігри, командні режими |
| 5 | Переваги – цінові/нецінові | Підписки, функціонал | Пакети ліцензій, нові можливості |
| 6 | Інтенсивність – марочна | Потрібен власний бренд | Назва, логотип, ком’юніті |

Таблиця 5.9 – Аналіз за М. Портером

| Складова | Прямі конкуренти | Потенційні | Постачальники | Клієнти | Замінники |
|---|---|---|---|---|---|
| Стан | Kaggle, CodinGame, Halite, CoderOne | Локальні платформи | Docker, хмарні провайдери | Університети, компанії | Внутрішні скрипти, МООС тести |

Продовження таблиці 5.9

| **Складова** | **Прямі конкуренти** | **Потенційні** | **Постачальники** | **Клієнти** | **Замінники** |
|---|---|---|---|---|---|
| Висновок | Висока конкуренція | Є ніші (локалізація) | Постачальники не диктують умови | Клієнтів треба приваблювати UX та сервісом | Потрібно пропонувати унікальні режими |

Таблиця 5.10 – SWOT аналіз стартап-проєкту

| Сильні сторони (S):<br>– локалізація;<br>– відкритий код;<br>– Docker-ізоляція;<br>– командний режим;<br>– інтерактивні реплеї;<br>– гнучкі інтерфейси; | Слабкі сторони (W):<br>– ресурсні вимоги Docker;<br>– зависимість від інфраструктури;<br>– молодий бренд; |
|---|---|
| Можливості (O):<br>– зростання EdTech;<br>– партнерства з університетами;<br>– ліцензійні пакети. | Загрози (T):<br>– конкуренти-глобалісти<br>– інертність користувачів<br>– обмежене фінансування. |

## 5.5 Розроблення ринкової стратегії

Цільовими групами обрано університети/коледжі, IT-компанії та організаторів публічних заходів (табл. 4.15). Враховуючи потребу в спеціалізованих функціях, обрана стратегія диференційованого маркетингу з позиціонуванням як “українська модульна платформа AI-турнірів”.

Таблиця 4.15 – Вибір цільових груп

| № | Цільова група | Готовність | Попит | Конкуренція | Простота входу |
|---|---|---|---|---|---|
| 1 | Освітні курси (університети) | Так | Високий | Помірна | Середня |
| 2 | ІТ-компанії (хакатони) | Так | Високий | Помірна | Складна |
| 3 | Публічні заходи | Так | Середній | Низька | Середня |

Базова стратегія розвитку – спеціалізація: надання інструменту, адаптованого під освітні сценарії з можливістю кастомізації. Конкурентна стратегія – заняття ніші (локалізована, відкрито-документована платформа з Docker-ізоляцією). Позиціонування підкреслює доступність, безпечність і аналітичність (“Прості турніри, зрозумілі реплеї, власні боти будь-якою мовою”).

## 5.6 Розроблення маркетингової програми

Ключові переваги: безпечне виконання коду, прозора візуалізація, гнучкі ліцензії. Модель товару відображає три рівні – задум (платформа AI-турнірів), реальне виконання (Docker, багатомовність, реплеї, UI), підкріплення (документація, сапорт, кастомні сценарії). Межі ціни варіюються від безкоштовної освітньої версії до корпоративних підписок (умовно $1–10 тис. на семестр/хакатон). Система збуту – прямі продажі/демо університетам та компаніям; комунікації – онлайн-конференції, блоги, присутність на освітніх подіях.

# ВИСНОВОК ДО РОЗДІЛУ 5

Проведений аналіз підтверджує життєздатність стартап-проєкту: ринок EdTech і корпоративних симуляцій зростає, конкуренти не закривають потреби локалізованої багатомовної платформи з Docker-ізоляцією та прозорими реплеями. Ідея має чіткі технічні переваги, відповідні технології доступні, а цільові аудиторії демонструють готовність впроваджувати рішення. Побудована ринкова стратегія (спеціалізація + нішева поведінка) і маркетингова програма (цінність, ціноутворення, збут, комунікації) дають підґрунтя для комерційного старту.

# ВИСНОВОК

Загалом проведена робота показала, що створена платформа турнірів ботів відповідає вимогам дипломного проекту: архітектура розділена на чіткі модулі (ядро, рушії, виконавці, турнірний менеджер, веб-шар), кожен із яких документовано та забезпечений контрактами для подальшого масштабування. Систему локалізовано, уся документація та коментарі переведені на українську, а ключові інструменти (документація, гайд зі створення ігор, архітектурний опис) доповнені актуальними діаграмами й таблицями, що спрощує адаптацію нових розробників. Реалізовано два режими змагань (індивідуальний і командний), деталізовану систему підрахунку очок із поясненнями на фронтенді, повноцінні індикатори метрик і тлумачення штрафів, що відповідає навчальним цілям платформи.

Особливу увагу приділено безпеці та відтворюваності експериментів: усі боти виконуються в Docker-контейнерах із перевіркою JSON-відповідей і таймаутами, передбачено логування в стандартному форматі, з’явився Swagger-доступ до API на /docs, а тестові сценарії документуються разом із скріншотами UI та станів інфраструктури. Проведений stress test (8 ботів) підтвердив, що і фронтенд, і backend витримують навантаження, контейнери не зупиняються до завершення турніру, а результати збігаються з очікуваними. Завдяки комплексним випробуванням, перевірці повторних запусків та ретельній документації можна стверджувати, що система готова до використання як навчальний майданчик і база для подальших досліджень у сфері змагального програмування та автономних агентів.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Rabin S. (ed.). Game AI Pro: Collected Wisdom of Game AI Professionals. A K Peters/CRC Press (Routledge), 2023. 626 с.
2. Sletten B. WebAssembly: The Definitive Guide: Safe, Fast, and Portable Code. O'Reilly, 2021. 400 с.
3. WebAssembly / WASI. [Електронний ресурс] – Режим доступу до ресурсу: https://wasi.dev/ (дата звернення 30.10.2025)
4. Creane B., Gupta A. Kubernetes Security and Observability: A Holistic Approach to Securing Containers and Cloud-Native Applications. O'Reilly, 2021. 182 с.
5. Majors C., Fong-Jones L., Miranda G. Observability Engineering: Achieving Production Excellence. O'Reilly, 2022. 318 с.
6. Firecracker microVM. [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/firecracker-microvm/firecracker (дата звернення 30.10.2025)
7. Firecracker aws docs. [Електронний ресурс] – Режим доступу до ресурсу: https://www.amazon.science/blog/how-awss-firecracker-virtual-machines-work/ (дата звернення 31.10.2025)
8. WebAssembly / WASI. [Електронний ресурс] – Режим доступу до ресурсу: https://wasi.dev/ (дата звернення 30.10.2025)
9. gRPC. [Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/docs/ (дата звернення 30.10.2025)
10. WebSockets API (MDN). [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/docs/Web/API/WebSockets_API (дата звернення 30.10.2025)
11. SQLAlchemy. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.sqlalchemy.org/en/20/intro.html (дата звернення 30.10.2025)
12. SQLite. [Електронний ресурс] – Режим доступу до ресурсу: https://www.sqlite.org/docs.html (дата звернення 30.10.2025)

13. Docker. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/ (дата звернення 30.10.2025)
14. Docker Compose. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.docker.com/compose/ (дата звернення 30.10.2025)
15. Redis. [Електронний ресурс] – Режим доступу до ресурсу: https://redis.io/docs/latest/ (дата звернення 30.10.2025)
16. Indrasiri K., Kuruppu D. gRPC: Up and Running: Building Cloud-Native Applications with Go and Java for Docker and Kubernetes. O'Reilly, 2020. 200 с.

# ДОДАТОК А: Лістинг коду

Далі наведені основні фрагменти коду.

## Файл main.py

```
#!/usr/bin/env python3
"""
Головна точка входу платформи турнірів ботів.
Запускає веб-інтерфейс для створення та проведення змагань між ботами.
"""

import os
import sys
from flask import Flask
from src.api.app import app

def main():
    """Основна точка входу застосунку."""
    print("Платформа турнірів ботів")
    print("=" * 40)
    print("Запускаємо платформу турнірів...")

    # Налаштовуємо дефолтні змінні середовища
    os.environ.setdefault('FLASK_ENV', 'development')
    os.environ.setdefault('DATABASE_URL', 'sqlite:///tournament.db')

    # Запускаємо Flask-додаток
    app.run(
        host='0.0.0.0',
        port=5253,
        debug=True
    )

if __name__ == '__main__':
    main()
```

## Файл config.py

```
"""
Налаштування платформи турнірів ботів.
"""

import os
from typing import Dict, Any

class Config:
```

```
    """Базовий клас конфігурації."""

    # Параметри БД
    DATABASE_URL = os.getenv('DATABASE_URL', 'sqlite:///tournament.db')

    # Налаштування Flask
    SECRET_KEY = os.getenv('SECRET_KEY', 'dev-secret-key-change-in-production')
    FLASK_ENV = os.getenv('FLASK_ENV', 'development')
    DEBUG = os.getenv('DEBUG', 'True').lower() == 'true'

    # Параметри сервера
    HOST = os.getenv('HOST', '0.0.0.0')
    PORT = int(os.getenv('PORT', 5000))

    # Налаштування ігор
    MAX_GAME_TURNS = int(os.getenv('MAX_GAME_TURNS', 100))
    GAME_TIMEOUT = int(os.getenv('GAME_TIMEOUT', 300))  # 5 minutes

    # Виконання ботів
    BOT_TIMEOUT = int(os.getenv('BOT_TIMEOUT', 30))  # 30 seconds per bot action
    MAX_BOTS_PER_TOURNAMENT = int(os.getenv('MAX_BOTS_PER_TOURNAMENT', 10))

    # Налаштування Docker (для Java-ботів)
    DOCKER_IMAGE_PREFIX = 'ai-bot'
    DOCKER_NETWORK = 'ai-tournament'

    # Підтримувані мови
    SUPPORTED_LANGUAGES = ['python', 'java']

    # Підтримувані типи ігор
    SUPPORTED_GAME_TYPES = ['simple_game']

    # Обмеження на завантаження файлів
    MAX_CONTENT_LENGTH = 16 * 1024 * 1024  # 16MB max file size

    # Параметри логування
    LOG_LEVEL = os.getenv('LOG_LEVEL', 'INFO')
    LOG_FILE = os.getenv('LOG_FILE', 'logs/tournament.log')

class DevelopmentConfig(Config):
    """Конфігурація для розробки."""
    DEBUG = True
    FLASK_ENV = 'development'

class ProductionConfig(Config):
    """Конфігурація для продакшену."""
    DEBUG = False
```

```
    FLASK_ENV = 'production'
    SECRET_KEY = os.getenv('SECRET_KEY')

    if not SECRET_KEY:
        raise ValueError("SECRET_KEY environment variable must be set in production")

class TestingConfig(Config):
    """Конфігурація для тестів."""
    TESTING = True
    DATABASE_URL = 'sqlite:///:memory:'
    WTF_CSRF_ENABLED = False

# Відповідність назв режимів → класам
config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'testing': TestingConfig,
    'default': DevelopmentConfig
}

def get_config(config_name: str = None) -> Config:
    """Повертає клас конфігурації за назвою."""
    if config_name is None:
        config_name = os.getenv('FLASK_ENV', 'default')

    return config.get(config_name, config['default'])
```

## Файл src/api/app.py

```
"""
Flask-додаток для платформи турнірів ботів.
"""

from flask import Flask, request, jsonify, render_template
from flask_cors import CORS
from flasgger import Swagger
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from src.core.models import Base, Tournament, Bot, Game, GameParticipation
from src.tournament.tournament_manager import TournamentManagerImpl
import os

# Ініціалізуємо Flask із правильними шляхами до шаблонів/статичних файлів
base_dir = os.path.dirname(__file__)
template_dir = os.path.abspath(os.path.join(base_dir, '../../templates'))
static_dir = os.path.abspath(os.path.join(base_dir, '../../static'))
app    =    Flask(__name__,    template_folder=template_dir,    static_folder=static_dir,
```

```
static_url_path='/static')
CORS(app)
app.config['SWAGGER'] = {
    'title': 'AI Tournament Platform API',
    'uiversion': 3,
    'specs_route': '/docs/'
}
swagger_template = {
    'info': {
        'title': 'AI Tournament Platform API',
        'description': 'REST-ендпоінти для керування турнірами, ботами та іграми. '
                       'Документація оновлюється автоматично на основі docstring-ів.',
        'version': '1.0.0',
        'contact': {
            'name': 'Tournament Platform Team',
            'email': 'support@example.com'
        }
    },
    'tags': [
        {'name': 'Tournaments', 'description': 'Операції створення, запуску та перегляду
турнірів'},
        {'name': 'Templates', 'description': 'Готові шаблони конфігурацій турнірів'}
    ]
}
swagger = Swagger(app, template=swagger_template)

# Налаштування бази даних
DATABASE_URL = os.getenv('DATABASE_URL', 'sqlite:///tournament.db')
engine = create_engine(DATABASE_URL)
Base.metadata.create_all(engine)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

# Фабрика менеджера турнірів (створює сесію на вимогу)
def get_tournament_manager():
    db_session = SessionLocal()
    return TournamentManagerImpl(db_session), db_session

@app.route('/')
def index():
    """Повертає основний веб-інтерфейс."""
    return render_template('index.html')

@app.route('/api/tournaments', methods=['GET'])
def get_tournaments():
    """
    Повертає перелік усіх турнірів.
    ---
```

```
    tags:
      - Tournaments
    responses:
      200:
        description: Список існуючих турнірів
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  id:
                    type: string
                  name:
                    type: string
                  game_type:
                    type: string
                  status:
                    type: string
    """
    try:
        tm, db_session = get_tournament_manager()
        tournaments = db_session.query(Tournament).all()

        result = []
        for tournament in tournaments:
            result.append({
                'id': tournament.id,
                'name': tournament.name,
                'game_type': tournament.game_type,
                'status': tournament.status,
                'created_at': tournament.created_at.isoformat() if tournament.created_at
else None,
                'bot_count': len(tournament.bots),
                'game_count': len(tournament.games)
            })

        db_session.close()
        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments', methods=['POST'])
def create_tournament():
    """
```

```
Створює новий турнір.
---
tags:
  - Tournaments
requestBody:
  required: true
  content:
    application/json:
      schema:
        type: object
        required: [name, bots]
        properties:
          name:
            type: string
            example: "Мій турнір"
          game_type:
            type: string
            example: "simple_game"
          bots:
            type: array
            items:
              type: object
              required: [name, language, code]
              properties:
                name:
                  type: string
                language:
                  type: string
                  example: "python"
                code:
                  type: string
responses:
  200:
    description: Ідентифікатор створеного турніру
    content:
      application/json:
        schema:
          type: object
          properties:
            tournament_id:
                type: string
  400:
    description: Некоректні дані
  500:
    description: Внутрішня помилка
"""
try:
```

```
        data = request.get_json()

        if not data or 'name' not in data or 'bots' not in data:
            return jsonify({'error': 'Missing required fields'}), 400

        name = data['name']
        game_type = data.get('game_type', 'simple_game')
        bots = data['bots']

        # Переконуємось, що кожен бот має обов'язкові поля
        for bot in bots:
            if 'name' not in bot or 'language' not in bot or 'code' not in bot:
                return jsonify({'error': 'Each bot must have name, language, and code'}),
400

        tm, db_session = get_tournament_manager()
        tournament_id = tm.create_tournament(name, game_type, bots)
        db_session.close()

        return jsonify({'tournament_id': tournament_id})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/run', methods=['POST'])
def run_tournament(tournament_id):
    """
    Запускає турнір за вказаним ідентифікатором.
    ---
    tags:
      - Tournaments
    parameters:
      - in: path
        name: tournament_id
        schema:
          type: string
        required: True
    responses:
      200:
        description: Результати запуску
      404:
        description: Турнір не знайдено
      500:
        description: Внутрішня помилка
    """
    try:
        tm, db_session = get_tournament_manager()
```

```
        results = tm.run_tournament(tournament_id)
        db_session.close()

        return jsonify(results)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/rerun', methods=['POST'])
def rerun_tournament(tournament_id):
    """Перезапускає турнір, що завершився з помилкою."""
    try:
        tm, db_session = get_tournament_manager()

        # Повертаємо турнір у стан created
        tournament = db_session.query(Tournament).filter_by(id=tournament_id).first()
        if not tournament:
            return jsonify({'error': 'Tournament not found'}), 404

        # Обнуляємо часові поля та результати
        tournament.status = 'created'
        tournament.started_at = None
        tournament.finished_at = None
        tournament.results = None

        # Обнуляємо пов'язані ігри
        games = db_session.query(Game).filter_by(tournament_id=tournament_id).all()
        for game in games:
            game.status = 'pending'
            game.started_at = None
            game.finished_at = None
            game.winner = None
            game.scores = None
            game.replay_data = None
            game.error_message = None

        db_session.commit()

        # Запускаємо повторно
        results = tm.run_tournament(tournament_id)
        db_session.close()

        return jsonify(results)

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

```
@app.route('/api/tournaments/<tournament_id>/status', methods=['GET'])
def get_tournament_status(tournament_id):
    """
    Повертає статус турніру.
    ---
    tags:
      - Tournaments
    parameters:
      - in: path
        name: tournament_id
        schema:
          type: string
        required: True
    responses:
      200:
        description: Поточний стан турніру
      404:
        description: Турнір не знайдено
    """
    try:
        tm, db_session = get_tournament_manager()
        status = tm.get_tournament_status(tournament_id)
        db_session.close()

        return jsonify(status)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/results', methods=['GET'])
def get_tournament_results(tournament_id):
    """
    Повертає результати турніру.
    ---
    tags:
      - Tournaments
    parameters:
      - in: path
        name: tournament_id
        schema:
          type: string
        required: True
    responses:
      200:
        description: Повні результати
      404:
        description: Турнір не знайдено
```

```
    """
    try:
        tm, db_session = get_tournament_manager()
        results = tm.get_tournament_results(tournament_id)
        db_session.close()

        return jsonify(results)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/bots', methods=['GET'])
def get_tournament_bots(tournament_id):
    """Повертає список ботів турніру."""
    try:
        tm, db_session = get_tournament_manager()
        bots = db_session.query(Bot).filter_by(tournament_id=tournament_id).all()

        result = []
        for bot in bots:
            result.append({
                'id': bot.id,
                'name': bot.name,
                'language': bot.language,
                'author': bot.author,
                'is_valid': bot.is_valid,
                'validation_errors': bot.validation_errors,
                'created_at': bot.created_at.isoformat() if bot.created_at else None
            })

        db_session.close()
        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/games', methods=['GET'])
def get_tournament_games(tournament_id):
    """Повертає всі зіграні ігри."""
    try:
        tm, db_session = get_tournament_manager()
        games = db_session.query(Game).filter_by(tournament_id=tournament_id).all()

        result = []
        for game in games:
            result.append({
                'id': game.id,
```

```
                'game_number': game.game_number,
                'status': game.status,
                'started_at': game.started_at.isoformat() if game.started_at else None,
                'finished_at': game.finished_at.isoformat() if game.finished_at else None,
                'winner': game.winner,
                'scores': game.scores,
                'error_message': game.error_message
            })

        db_session.close()
        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/<tournament_id>/details', methods=['GET'])
def get_tournament_details(tournament_id):
    """Повертає розширену інформацію про турнір."""
    try:
        tm, db_session = get_tournament_manager()
        tournament = db_session.query(Tournament).filter_by(id=tournament_id).first()

        if not tournament:
            return jsonify({'error': 'Tournament not found'}), 404

        # Зчитуємо ботів
        bots = db_session.query(Bot).filter_by(tournament_id=tournament_id).all()
        bot_list = []
        for bot in bots:
            bot_list.append({
                'id': bot.id,
                'name': bot.name,
                'language': bot.language,
                'author': bot.author,
                'is_valid': bot.is_valid,
                'validation_errors': bot.validation_errors,
                'created_at': bot.created_at.isoformat() if bot.created_at else None
            })

        # Зчитуємо ігри
        games = db_session.query(Game).filter_by(tournament_id=tournament_id).all()
        game_list = []
        for game in games:
            game_list.append({
                'id': game.id,
                'game_number': game.game_number,
                'status': game.status,
```

```
                'started_at': game.started_at.isoformat() if game.started_at else None,
                'finished_at': game.finished_at.isoformat() if game.finished_at else None,
                'winner': game.winner,
                'scores': game.scores,
                'error_message': game.error_message
            })

        # Для кожної гри підтягуємо учасників
        for game in game_list:
            participations                                                    =
db_session.query(GameParticipation).filter_by(game_id=game['id']).all()
            game['participations'] = []
            for participation in participations:
                bot = db_session.query(Bot).filter_by(id=participation.bot_id).first()
                game['participations'].append({
                    'bot_name': bot.name if bot else f"Bot {participation.bot_id}",
                    'player_id': participation.player_id,
                    'score': participation.score,
                    'actions_taken': participation.actions_taken,
                    'errors_count': participation.errors_count
                })

        result = {
            'tournament': {
                'id': tournament.id,
                'name': tournament.name,
                'game_type': tournament.game_type,
                'status': tournament.status,
                'created_at':  tournament.created_at.isoformat()  if  tournament.created_at
else None,
                'started_at':  tournament.started_at.isoformat()  if  tournament.started_at
else None,
                'finished_at': tournament.finished_at.isoformat() if tournament.finished_at
else None,
                'config': tournament.config,
                'results': tournament.results
            },
            'bots': bot_list,
            'games': game_list
        }

        db_session.close()
        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

```python
@app.route('/api/tournament-templates', methods=['GET'])
def get_tournament_templates():
    """Повертає доступні шаблони турнірів."""
    templates = [
        {
            'id': 'basic_python',
            'name': 'Базовий Python-турнір',
            'description': 'Просте змагання двох Python-ботів',
            'game_type': 'simple_game',
            'bot_count': 2,
            'languages': ['python'],
            'difficulty': 'beginner'
        },
        {
            'id': 'aggressive_defensive',
            'name': 'Агресивний проти Захисного',
            'description': 'Стратегічне змагання з різними поведінковими патернами',
            'game_type': 'simple_game',
            'bot_count': 2,
            'languages': ['python'],
            'difficulty': 'intermediate'
        },
        {
            'id': 'four_bot_championship',
            'name': 'Чемпіонат чотирьох ботів',
            'description': 'Турнір із 4 унікальними стратегіями',
            'game_type': 'simple_game',
            'bot_count': 4,
            'languages': ['python'],
            'difficulty': 'intermediate'
        },
        {
            'id': 'python_java',
            'name': 'Python vs Java Championship',
            'description': 'Міжмовне змагання ботів на Python і Java',
            'game_type': 'simple_game',
            'bot_count': 2,
            'languages': ['python', 'java'],
            'difficulty': 'advanced'
        },
        {
            'id': 'stress_test',
            'name': 'Стрес-тест (8 ботів)',
            'description': 'Перевірка продуктивності з 8 простими ботами',
            'game_type': 'simple_game',
            'bot_count': 8,
            'languages': ['python'],
```

```
            'difficulty': 'advanced'
        },
        {
            'id': 'fighting_championship',
            'name': 'Арена — бойовий чемпіонат',
            'description': 'Епічний турнір зі зброєю та підсилювачами',
            'game_type': 'fighting_game',
            'bot_count': 3,
            'languages': ['python'],
            'difficulty': 'intermediate'
        },
        {
            'id': 'logic_puzzle',
            'name': 'Чемпіонат логічних пазлів',
            'description': 'Змагання на стратегічне мислення й впізнавання патернів',
            'game_type': 'simple_game',
            'bot_count': 2,
            'languages': ['python'],
            'difficulty': 'intermediate'
        },
        {
            'id': 'team_collect',
            'name': 'Командний збір Очок перемоги',
            'description': '4 боти, дві команди. Перемагає команда з сумарним більшим
рахунком.',
            'game_type': 'simple_team_game',
            'bot_count': 4,
            'languages': ['python'],
            'difficulty': 'intermediate'
        }
    ]
    return jsonify(templates)

@app.route('/api/tournament-templates/<template_id>', methods=['GET'])
def get_tournament_template(template_id):
    """Повертає конкретний шаблон турніру разом із кодом ботів."""
    from examples.test_tournaments import TOURNAMENT_EXAMPLES

    if template_id in TOURNAMENT_EXAMPLES:
        template_data = TOURNAMENT_EXAMPLES[template_id]
        return jsonify(template_data)
    else:
        return jsonify({'error': 'Template not found'}), 404

@app.route('/api/tournaments/<tournament_id>', methods=['DELETE'])
def delete_tournament(tournament_id):
    """Видаляє конкретний турнір."""
```

```
    try:
        tm, db_session = get_tournament_manager()
        result = tm.delete_tournament(tournament_id)
        db_session.close()

        if 'error' in result:
            return jsonify(result), 400

        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/tournaments/clear-all', methods=['DELETE'])
def clear_all_tournaments():
    """Повністю очищає історію турнірів."""
    try:
        tm, db_session = get_tournament_manager()
        result = tm.clear_all_tournaments()
        db_session.close()

        if 'error' in result:
            return jsonify(result), 400

        return jsonify(result)

    except Exception as e:
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

## Файл src/core/interfaces.py

```
"""
Базові інтерфейси платформи турнірів ботів.
Описують контракти для рушіїв і реалізацій ботів.
"""

from abc import ABC, abstractmethod
from typing import Any, Dict, List, Optional, Union
from dataclasses import dataclass
from enum import Enum
```

```
class GameState(Enum):
    """Поточний стан гри."""
    WAITING = "waiting"
    RUNNING = "running"
    FINISHED = "finished"
    ERROR = "error"


@dataclass
class GameObservation:
    """Набір даних, які бот отримує перед ходом."""
    game_state: Dict[str, Any]
    turn_number: int
    player_id: str
    available_actions: List[str]
    metadata: Dict[str, Any] = None


@dataclass
class BotAction:
    """Дія, яку повернув бот."""
    action_type: str
    parameters: Dict[str, Any]
    bot_id: str
    turn_number: int
    timestamp: float


@dataclass
class GameResult:
    """Підсумок зіграної партії."""
    winner: Optional[str]
    scores: Dict[str, float]
    game_duration: float
    total_turns: int
    replay_data: List[Dict[str, Any]]
    metadata: Dict[str, Any] = None


class BotInterface(ABC):
    """
    Абстрактна база для всіх реалізацій ботів.
    Кожний виконавець мови має надати свою імплементацію.
    """

    @abstractmethod
    def initialize(self, bot_id: str, game_config: Dict[str, Any]) -> None:
```

```
        """Ініціалізувати бота конфігом гри."""
        pass

    @abstractmethod
    def get_action(self, observation: GameObservation) -> BotAction:
        """Повернути дію на основі спостереження."""
        pass

    @abstractmethod
    def cleanup(self) -> None:
        """Прибрати ресурси після завершення гри."""
        pass


class GameEngine(ABC):
    """
    Абстрактна база для всіх ігрових рушіїв.
    Кожен тип гри реалізує ці методи.
    """

    @abstractmethod
    def initialize_game(self, bot_configs: List[Dict[str, Any]]) -> str:
        """Створити та ініціалізувати гру з переданими конфігами ботів."""
        pass

    @abstractmethod
    def get_observation(self, bot_id: str) -> GameObservation:
        """Повернути спостереження для конкретного бота."""
        pass

    @abstractmethod
    def apply_action(self, action: BotAction) -> bool:
        """Застосувати дію бота до стану гри."""
        pass

    @abstractmethod
    def is_game_finished(self) -> bool:
        """Чи завершено гру."""
        pass

    @abstractmethod
    def get_game_result(self) -> GameResult:
        """Повернути фінальний результат."""
        pass

    @abstractmethod
    def get_game_state(self) -> GameState:
```

```
        """Поточний статус гри."""
        pass


class BotExecutor(ABC):
    """
    Абстракція для виконавців ботів різних мов.
    """

    @abstractmethod
    def validate_code(self, code: str) -> bool:
        """Перевірити синтаксис коду бота."""
        pass

    @abstractmethod
    def create_bot_instance(self, code: str, bot_id: str) -> BotInterface:
        """Створити інстанс бота з коду."""
        pass

    @abstractmethod
    def get_supported_language(self) -> str:
        """Повернути мову, яку підтримує виконавець."""
        pass


class TournamentManager(ABC):
    """
    Абстракція для менеджера турнірів.
    """

    @abstractmethod
    def create_tournament(self, name: str, game_type: str, bot_configs: List[Dict[str,
Any]]) -> str:
        """Створити новий турнір."""
        pass

    @abstractmethod
    def run_tournament(self, tournament_id: str) -> Dict[str, Any]:
        """Запустити турнір і повернути результати."""
        pass

    @abstractmethod
    def get_tournament_status(self, tournament_id: str) -> Dict[str, Any]:
        """Повернути поточний стан турніру."""
        pass

    @abstractmethod
```

```
    def get_tournament_results(self, tournament_id: str) -> Dict[str, Any]:
        """Повернути результати завершеного турніру."""
        pass
```

## Файл src/tournament/tournament_manager.py

```
"""Менеджер турнірів платформи: створення, запуск, історія та очищення даних."""

import time
import uuid
from typing import Dict, List, Any, Optional
from datetime import datetime
from src.core.interfaces import TournamentManager, GameEngine, BotExecutor
from src.core.models import Tournament, Bot, Game, GameParticipation, Replay
from src.games.simple_game import SimpleGame
from src.games.simple_team_game import SimpleTeamGame
from src.games.fighting_game import FightingGame
from src.executors.python_executor import PythonExecutor
from src.executors.java_executor import JavaExecutor


class TournamentManagerImpl(TournamentManager):
    """Практична реалізація TournamentManager з підтримкою кількох рушіїв та мов."""

    def __init__(self, db_session):
        self.db_session = db_session
        self.executors = {
            'python': PythonExecutor(),
            'java': JavaExecutor()
        }
        self.game_engines = {
            'simple_game': SimpleGame,
            'simple_team_game': SimpleTeamGame,
            'fighting_game': FightingGame
        }
        self.active_games = {}

    def create_tournament(self, name: str, game_type: str, bot_configs: List[Dict[str,
Any]]) -> str:
        """Створює новий запис турніру та всі пов’язані боти."""
        try:
            # Перевіряємо, що тип гри підтримується
            if game_type not in self.game_engines:
                raise ValueError(f"Unsupported game type: {game_type}")
```

```
            # Створюємо сутність турніру
            tournament = Tournament(
                name=name,
                game_type=game_type,
                status='created',
                config={'bot_count': len(bot_configs)}
            )
            self.db_session.add(tournament)
            self.db_session.flush()  # Потрібен ID для прив'язки ботів

            # Додаємо ботів із конфігурації
            for i, bot_config in enumerate(bot_configs):
                bot = Bot(
                    tournament_id=tournament.id,
                    name=bot_config.get('name', f'Bot {i+1}'),
                    language=bot_config.get('language', 'python'),
                    code=bot_config.get('code', ''),
                    author=bot_config.get('author', ''),
                    is_valid=False
                )
                self.db_session.add(bot)

            self.db_session.commit()

            return str(tournament.id)

        except Exception as e:
            self.db_session.rollback()
            raise RuntimeError(f"Failed to create tournament: {str(e)}")

    def run_tournament(self, tournament_id: str) -> Dict[str, Any]:
        """Запускає турнір та повертає підсумкові результати."""
        try:
            tournament_id = int(tournament_id)
            tournament                                                    =
self.db_session.query(Tournament).filter_by(id=tournament_id).first()

            if not tournament:
                raise ValueError(f"Tournament {tournament_id} not found")

            # Переводимо турнір у стан "running"
            tournament.status = 'running'
            tournament.started_at = datetime.utcnow()
            self.db_session.commit()

            # Підтягуємо всіх ботів для цього турніру
            bots = self.db_session.query(Bot).filter_by(tournament_id=tournament_id).all()
```

```
            # Перевіряємо кожного бота
            valid_bots = []
            for bot in bots:
                if self._validate_bot(bot):
                    valid_bots.append(bot)
                else:
                    bot.is_valid = False
                    self.db_session.commit()

            if len(valid_bots) < 2:
                tournament.status = 'error'
                tournament.results = {'error': 'Not enough valid bots to run tournament'}
                self.db_session.commit()
                return {'status': 'error', 'message': 'Not enough valid bots'}

            # Запускаємо гру
            results = self._run_tournament_games(tournament, valid_bots)

            # Зберігаємо результати
            tournament.status = 'finished'
            tournament.finished_at = datetime.utcnow()
            tournament.results = results
            self.db_session.commit()

            return results

        except Exception as e:
            self.db_session.rollback()
            raise RuntimeError(f"Failed to run tournament: {str(e)}")

    def get_tournament_status(self, tournament_id: str) -> Dict[str, Any]:
        """Повертає поточний стан турніру."""
        try:
            tournament_id = int(tournament_id)
            tournament                                                    =
self.db_session.query(Tournament).filter_by(id=tournament_id).first()

            if not tournament:
                return {'error': 'Tournament not found'}

            return {
                'id': tournament.id,
                'name': tournament.name,
                'game_type': tournament.game_type,
                'status': tournament.status,
                'created_at':  tournament.created_at.isoformat()  if  tournament.created_at
```

```
else None,
                'started_at': tournament.started_at.isoformat() if tournament.started_at
else None,
                'finished_at': tournament.finished_at.isoformat() if tournament.finished_at
else None,
                'bot_count': len(tournament.bots),
                'game_count': len(tournament.games)
            }

        except Exception as e:
            return {'error': str(e)}

    def get_tournament_results(self, tournament_id: str) -> Dict[str, Any]:
        """Повертає збережені результати завершеного турніру."""
        try:
            tournament_id = int(tournament_id)
            tournament                                                              =
self.db_session.query(Tournament).filter_by(id=tournament_id).first()

            if not tournament:
                return {'error': 'Tournament not found'}

            if tournament.status != 'finished':
                return {'error': 'Tournament not finished yet'}

            return tournament.results or {}

        except Exception as e:
            return {'error': str(e)}

    def delete_tournament(self, tournament_id: str) -> Dict[str, Any]:
        """Видаляє турнір разом із пов’язаними іграми та ботами."""
        try:
            tournament_id = int(tournament_id)
            tournament                                                              =
self.db_session.query(Tournament).filter_by(id=tournament_id).first()

            if not tournament:
                return {'error': 'Tournament not found'}

            # Видаляємо всі залежні дані
            # Спершу участі в іграх (через зовнішні ключі)
            for game in tournament.games:
                for participation in game.participations:
                    self.db_session.delete(participation)

            # Потім ігри
```

```
            for game in tournament.games:
                self.db_session.delete(game)

            # Далі ботів
            for bot in tournament.bots:
                self.db_session.delete(bot)

            # І нарешті сам турнір
            self.db_session.delete(tournament)
            self.db_session.commit()

            return {'success': True, 'message': f'Tournament {tournament_id} deleted
successfully'}

        except Exception as e:
            self.db_session.rollback()
            return {'error': str(e)}

    def clear_all_tournaments(self) -> Dict[str, Any]:
        """Повністю очищає історію турнірів."""
        try:
            # Видаляємо участі у всіх іграх
            self.db_session.query(GameParticipation).delete()

            # Видаляємо ігри
            self.db_session.query(Game).delete()

            # Видаляємо ботів
            self.db_session.query(Bot).delete()

            # Підраховуємо та видаляємо турніри
            deleted_count = self.db_session.query(Tournament).count()
            self.db_session.query(Tournament).delete()

            # Видаляємо реплеї
            self.db_session.query(Replay).delete()

            self.db_session.commit()

            return {
                'success': True,
                'message': f'All tournament history cleared successfully. Deleted
{deleted_count} tournaments.'
            }

        except Exception as e:
            self.db_session.rollback()
```

```
            return {'error': str(e)}

    def _validate_bot(self, bot: Bot) -> bool:
        """Перевіряє код бота та зберігає результат."""
        try:
            language = bot.language.lower()
            if language not in self.executors:
                bot.validation_errors = f"Unsupported language: {language}"
                return False

            executor = self.executors[language]
            is_valid = executor.validate_code(bot.code)

            if not is_valid:
                bot.validation_errors = "Code validation failed"
                return False

            bot.is_valid = True
            bot.validation_errors = None
            return True

        except Exception as e:
            bot.validation_errors = str(e)
            return False

    def _run_tournament_games(self, tournament: Tournament, bots: List[Bot]) -> Dict[str,
Any]:
        """Запускає всі ігри турніру (поки що одну)."""
        game_type = tournament.game_type
        game_engine_class = self.game_engines[game_type]

        # Поки що проводимо один матч за участю всіх ботів
        # У майбутньому можна буде додати сітки/раунди
        game = Game(
            tournament_id=tournament.id,
            game_number=1,
            status='running',
            started_at=datetime.utcnow()
        )
        self.db_session.add(game)
        self.db_session.flush()

        # Фіксуємо участі ботів у грі
        for i, bot in enumerate(bots):
            participation = GameParticipation(
                game_id=game.id,
                bot_id=bot.id,
```

```
                player_id=f"player_{i}"
            )
            self.db_session.add(participation)

        self.db_session.commit()

        try:
            # Запускаємо матч
            game_result = self._run_single_game(game, bots, game_engine_class)

            # Зберігаємо результати та реплей
            game.status = 'finished'
            game.finished_at = datetime.utcnow()
            game.winner = game_result.winner
            game.scores = game_result.scores
            game.replay_data = game_result.replay_data

            # Проставляємо очки для кожного участника
            for participation in game.participations:
                bot_id = f"bot_{participation.bot_id}"
                if bot_id in game_result.scores:
                    participation.score = game_result.scores[bot_id]

            self.db_session.commit()

            return {
                'winner': game_result.winner,
                'scores': game_result.scores,
                'game_duration': game_result.game_duration,
                'total_turns': game_result.total_turns,
                'replay_data': game_result.replay_data,
                'metadata': game_result.metadata,
                'replay_available': True
            }

        except Exception as e:
            game.status = 'error'
            game.error_message = str(e)
            self.db_session.commit()
            raise

    def _run_single_game(self, game: Game, bots: List[Bot], game_engine_class) -> Any:
        """Проводить окрему гру з переданими ботами."""
        # Створюємо екземпляр рушія
        game_engine = game_engine_class()

        # Визначаємо тип гри турніру
```

```
        tournament                                                              =
self.db_session.query(Tournament).filter_by(id=game.tournament_id).first()
        game_type = tournament.game_type if tournament else 'simple_game'

        # Готуємо конфіг та порядок ботів
        bot_configs = []
        bot_instances = {}

        for i, bot in enumerate(bots):
            bot_id = f"bot_{i}"
            bot_config = {
                'bot_id': bot_id,
                'name': bot.name,
                'language': bot.language,
                'code': bot.code
            }
            bot_configs.append(bot_config)

        # Запускаємо гру в рушії
        game_id = game_engine.initialize_game(bot_configs)

        # Створюємо інстанси ботів
        for i, bot in enumerate(bots):
            bot_id = f"bot_{i}"
            language = bot.language.lower()
            executor = self.executors[language]

            try:
                bot_instance = executor.create_bot_instance(bot.code, bot_id)
                bot_instance.initialize(bot_id, {'game_type': game_type})
                bot_instances[bot_id] = bot_instance
            except Exception as e:
                print(f"Failed to create bot instance for {bot.name}: {e}")
                # Для Java піднімаємо мок, якщо Docker недоступний
                if language == 'java':
                    print(f"Creating mock Java bot for {bot.name} (Docker not available)")
                    from src.executors.java_executor import MockJavaBot
                    mock_bot = MockJavaBot(bot_id)
                    mock_bot.initialize(bot_id, {'game_type': game_type})
                    bot_instances[bot_id] = mock_bot
                else:
                    continue

        # Основний цикл гри
        while not game_engine.is_game_finished():
            # Знімаємо спостереження та дії для кожного бота
            for bot_id, bot_instance in bot_instances.items():
```

```
            try:
                observation = game_engine.get_observation(bot_id)

                # Вимірюємо час відповіді бота
                import time
                start_time = time.time()
                action = bot_instance.get_action(observation)
                execution_time = time.time() - start_time

                # Передаємо дію разом з часом виконання
                game_engine.apply_action(action, execution_time)
            except Exception as e:
                print(f"Error with bot {bot_id}: {e}")
                # Все одно фіксуємо помилку як окрему дію
                try:
                    # Створюємо службову дію з описом помилки
                    from src.core.interfaces import BotAction
                    error_action = BotAction(
                        action_type='error',
                        parameters={'error': str(e)},
                        bot_id=bot_id,
                        turn_number=game_engine.turn_number,
                        timestamp=time.time()
                    )
                    game_engine.apply_action(error_action, 0.0)
                except:
                    pass
                continue

        # Переходимо до наступного ходу
        game_engine.advance_turn()

    # Прибираємо інстанси ботів
    for bot_instance in bot_instances.values():
        try:
            bot_instance.cleanup()
        except Exception:
            pass

    # Повертаємо фінальний результат
    return game_engine.get_game_result()
```

## Файл src/executors/python_executor.py

```
"""
```

```
Реалізація виконавця Python-ботів з обов'язковою ізоляцією у Docker-контейнерах.
"""

import ast
import json
import os
import shutil
import tempfile
from typing import Any, Dict, Optional

try:
    import docker
    from docker.errors import ContainerError
except ImportError:  # pragma: no cover - Docker може бути відсутнім на CI
    docker = None  # type: ignore
    ContainerError = Exception  # type: ignore

from src.core.interfaces import BotInterface, BotExecutor, GameObservation, BotAction
from src.executors.container_sandbox import ContainerSandbox


class PythonContainerBot(BotInterface):
    """
    Запускає користувацький Python-бот у виділеному контейнері та виконує всі команди
    через міст `bridge.py`, що відповідає за серіалізацію стану між ходами.
    """

    BRIDGE_SCRIPT = r"""
import importlib.util
import json
import os
import pickle
import sys
import traceback

CODE_PATH = "/bot/bot_code.py"
STATE_PATH = "/bot/bot_state.pkl"
ORIGINAL_STDOUT = sys.stdout
sys.stdout = sys.stderr


def load_module():
    spec = importlib.util.spec_from_file_location("bot_module", CODE_PATH)
    module = importlib.util.module_from_spec(spec)
    sys.modules["bot_module"] = module
    spec.loader.exec_module(module)
    return module
```

```
def resolve_bot_class(module):
    for attr in dir(module):
        obj = getattr(module, attr)
        if isinstance(obj, type) and hasattr(obj, "get_action"):
            return obj
    raise RuntimeError("Не знайдено класу з методом get_action")


def load_bot_instance(bot_cls):
    if os.path.exists(STATE_PATH):
        with open(STATE_PATH, "rb") as fh:
            return pickle.load(fh)
    return bot_cls()


def save_bot_instance(bot):
    with open(STATE_PATH, "wb") as fh:
        pickle.dump(bot, fh)


def main():
    if len(sys.argv) < 2:
        raise SystemExit("command is required")
    command = sys.argv[1]
    payload = json.loads(os.environ.get("BOT_PAYLOAD", "{}"))

    module = load_module()
    bot_cls = resolve_bot_class(module)
    bot = load_bot_instance(bot_cls)

    response = {}
    if command == "init":
        cfg = payload.get("game_config", {})
        bot_id = payload.get("bot_id")
        if hasattr(bot, "initialize"):
            bot.initialize(bot_id, cfg)
        response = {"status": "ok"}
    elif command == "action":
        observation = payload.get("observation", {})
        action = bot.get_action(observation)
        response = {"action": action}
    elif command == "cleanup":
        if hasattr(bot, "cleanup"):
            bot.cleanup()
        if os.path.exists(STATE_PATH):
```

```
            os.remove(STATE_PATH)
        response = {"status": "ok"}
    else:
        raise RuntimeError(f"Unsupported command {command}")

    if command != "cleanup":
        save_bot_instance(bot)

    print(json.dumps(response), file=ORIGINAL_STDOUT, flush=True)


if __name__ == "__main__":
    try:
        main()
    except Exception:
        traceback.print_exc()
        sys.exit(1)
"""

    def __init__(self, bot_id: str, code: str, docker_client, image: str = "python:3.10-
slim"):
        self.bot_id = bot_id
        self.docker_client = docker_client
        self.image = image
        self.workdir = tempfile.mkdtemp(prefix=f"pybot_{bot_id}_")
        self._write_file("bot_code.py", code)
        self._write_file("bridge.py", self.BRIDGE_SCRIPT.strip() + "\n")
        self.sandbox = ContainerSandbox(
            docker_client=self.docker_client,
            language="python",
            bot_id=bot_id,
            image=self.image,
            host_workdir=self.workdir,
            container_workdir="/bot",
            volumes={self.workdir: {"bind": "/bot", "mode": "rw"}},
            network_disabled=True,
        )

    def _write_file(self, filename: str, content: str) -> None:
        path = os.path.join(self.workdir, filename)
        with open(path, "w", encoding="utf-8") as fh:
            fh.write(content)

    def _run_bridge(
        self,
        command: str,
        payload: Optional[Dict[str, Any]] = None,
```

```
        expect_response: bool = True,
    ) -> Dict[str, Any]:
        env = {}
        if payload is not None:
            env["BOT_PAYLOAD"] = json.dumps(payload)
        try:
            decoded = (
                self.sandbox.exec(
                    ["python3", "/bot/bridge.py", command],
                    environment=env,
                )
                .decode()
                .strip()
            )
            if not expect_response:
                return {}
            if not decoded:
                raise RuntimeError("Python container returned empty response.")
            try:
                return json.loads(decoded)
            except json.JSONDecodeError as exc:
                snippet = decoded[:200]
                raise RuntimeError(
                    f"Invalid JSON від python-контейнера ({command}): {snippet}"
                ) from exc
        except ContainerError as exc:
            stderr = ""
            if hasattr(exc, "stderr") and exc.stderr:
                stderr = exc.stderr.decode(errors="ignore").strip()
            raise RuntimeError(f"Python container error ({command}): {stderr}") from exc
        except RuntimeError as exc:
            raise RuntimeError(f"Python container command failed ({command}): {exc}") from
exc

    def initialize(self, bot_id: str, game_config: Dict[str, Any]) -> None:
        self._run_bridge("init", {"bot_id": bot_id, "game_config": game_config or {}},
expect_response=False)

    def get_action(self, observation: GameObservation) -> BotAction:
        obs_dict = observation.game_state.copy()
        obs_dict.update(
            {
                "turn_number": observation.turn_number,
                "player_id": observation.player_id,
                "available_actions": observation.available_actions,
                "metadata": observation.metadata or {},
            }
```

```
        )
        response = self._run_bridge("action", {"observation": obs_dict})
        action_data = response.get("action")
        if isinstance(action_data, dict):
            action_type = action_data.get("action_type", "unknown")
            parameters = action_data.get("parameters", {})
        else:
            action_type = str(action_data)
            parameters = {}
        return BotAction(
            action_type=action_type,
            parameters=parameters,
            bot_id=self.bot_id,
            turn_number=observation.turn_number,
            timestamp=observation.turn_number,
        )

    def cleanup(self) -> None:
        try:
            self._run_bridge("cleanup", expect_response=False)
        finally:
            if self.sandbox:
                self.sandbox.cleanup()
            shutil.rmtree(self.workdir, ignore_errors=True)


class PythonExecutor(BotExecutor):
    """
    Виконавець Python-ботів, який завжди використовує Docker і відмовляється працювати без нього.
    """

    def __init__(self):
        if docker is None:
            raise RuntimeError("Python executor вимагає встановленого пакета 'docker'.")
        try:
            self.docker_client = docker.from_env()
            self.docker_client.ping()
        except Exception as exc:  # pragma: no cover - залежить від середовища
            raise RuntimeError(f"Docker недоступний для Python-ботів: {exc}") from exc

    def validate_code(self, code: str) -> bool:
        try:
            ast.parse(code)
            return True
        except SyntaxError:
            return False
```

```
        except Exception:
            return False

    def create_bot_instance(self, code: str, bot_id: str) -> BotInterface:
        if not self.docker_client:
            raise RuntimeError("Docker клієнт не ініціалізований, бот не може бути створений.")
        return PythonContainerBot(bot_id, code, self.docker_client)

    def get_supported_language(self) -> str:
        return "python"
```

## Файл src/executors/java_executor.py

```
"""
Реалізація Java-виконавця (поки в режимі mock) з окремими контейнерами на мову.
"""

import json
import os
import re
import shutil
import tempfile
from typing import Dict, Any, Optional

try:
    import docker
    from docker.errors import ContainerError
except ImportError:  # pragma: no cover - docker може бути відсутнім
    docker = None  # type: ignore
    ContainerError = Exception  # type: ignore

from src.core.interfaces import BotInterface, BotExecutor, GameObservation, BotAction
from src.executors.container_sandbox import ContainerSandbox


def _extract_public_class_name(code: str) -> str:
    match = re.search(r'public\s+class\s+(\w+)', code)
    if not match:
        raise ValueError("Java code must contain a public class declaration.")
    return match.group(1)


class MockJavaBot(BotInterface):
    """Заміна для Java-бота, коли Docker недоступний."""
```

```
    def __init__(self, bot_id: str):
        self.bot_id = bot_id
        print(f"[JavaExecutor] Використано mock-runner для бота {bot_id}")

    def initialize(self, bot_id: str, game_config: Dict[str, Any]) -> None:
        print(f"Mock Java Bot {bot_id} initialized (Docker not available)")

    def get_action(self, observation: GameObservation) -> BotAction:
        # Повертаємо просту випадкову дію
        import random
        actions = ['move', 'collect', 'wait']
        action_type = random.choice(actions)

        if action_type == 'move':
            direction = random.choice(['up', 'down', 'left', 'right'])
            return BotAction(
                action_type='move',
                parameters={'direction': direction},
                bot_id=self.bot_id,
                turn_number=observation.turn_number,
                timestamp=observation.turn_number
            )
        elif action_type == 'collect':
            return BotAction(
                action_type='collect',
                parameters={},
                bot_id=self.bot_id,
                turn_number=observation.turn_number,
                timestamp=observation.turn_number
            )
        else:
            return BotAction(
                action_type='wait',
                parameters={},
                bot_id=self.bot_id,
                turn_number=observation.turn_number,
                timestamp=observation.turn_number
            )

    def cleanup(self) -> None:
        print(f"Mock Java Bot {self.bot_id} cleaning up")


class JavaContainerBot(BotInterface):
    """Повноцінна контейнеризована реалізація Java-бота."""
```

```
    BRIDGE_TEMPLATE = r"""
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.SimpleBindings;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

public class JavaBotBridge {{
    private static final String STATE_PATH = "/bot/java_bot_state.bin";
    private static final ScriptEngine ENGINE = new
ScriptEngineManager().getEngineByName("javascript");
    private static final String BOT_CLASS = "{bot_class}";
    private static boolean STATE_WARNING_EMITTED = false;
    private Object botInstance;

    public static void main(String[] args) throws Exception {{
        if (args.length == 0) {{
            throw new IllegalArgumentException("Command is required");
        }}
        String payloadRaw = System.getenv("BOT_PAYLOAD");
        Map<String, Object> payload = parseJson(payloadRaw == null ? "{{}}" : payloadRaw);
        JavaBotBridge bridge = new JavaBotBridge();
        Map<String, Object> response = bridge.handle(args[0], payload);
        if (response != null) {{
            System.out.println(toJson(response));
        }} else {{
            System.out.println("");
        }}
    }}

    private Map<String, Object> handle(String command, Map<String, Object> payload) throws
Exception {{
        Object bot = loadBotInstance();
        Class<?> botClass = Class.forName(BOT_CLASS);
        switch (command) {{
            case "init":
                String botId = payload.get("bot_id") == null ? "" :
payload.get("bot_id").toString();
                Map<String, Object> cfg = asMap(payload.get("game_config"));
                botClass.getMethod("initialize", String.class, Map.class).invoke(bot,
botId, cfg);
```

```
                saveBotInstance(bot);
                Map<String, Object> initResp = new HashMap<>();
                initResp.put("status", "ok");
                return initResp;
            case "action":
                Map<String, Object> obs = asMap(payload.get("observation"));
                Object action = botClass.getMethod("getAction", Map.class).invoke(bot,
obs);
                saveBotInstance(bot);
                Map<String, Object> actionResp = new HashMap<>();
                actionResp.put("action", action);
                return actionResp;
            case "cleanup":
                try {{
                    botClass.getMethod("cleanup").invoke(bot);
                }} catch (NoSuchMethodException ignored) {{
                }}
                deleteState();
                Map<String, Object> cleanupResp = new HashMap<>();
                cleanupResp.put("status", "ok");
                return cleanupResp;
            default:
                throw new IllegalArgumentException("Unsupported command: " + command);
        }}
    }}

    private Object loadBotInstance() throws Exception {{
        if (botInstance != null) {{
            return botInstance;
        }}
        File stateFile = new File(STATE_PATH);
        if (stateFile.exists()) {{
            try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(stateFile))) {{
                botInstance = ois.readObject();
                return botInstance;
            }} catch (Exception ex) {{
                System.err.println("Failed to restore Java bot state: " + ex.getMessage());
                stateFile.delete();
            }}
        }}
        Class<?> botClass = Class.forName(BOT_CLASS);
        botInstance = botClass.getDeclaredConstructor().newInstance();
        return botInstance;
    }}

    private void saveBotInstance(Object bot) {{
```

```
        if (!(bot instanceof Serializable)) {{
            if (!STATE_WARNING_EMITTED) {{
                System.err.println("Java bot is not Serializable; стан не буде збережено.");
                STATE_WARNING_EMITTED = true;
            }}
            return;
        }}
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(STATE_PATH))) {{
            oos.writeObject(bot);
        }} catch (Exception ex) {{
            System.err.println("Failed to persist Java bot state: " + ex.getMessage());
        }}
    }}

    private void deleteState() {{
        File stateFile = new File(STATE_PATH);
        if (stateFile.exists()) {{
            stateFile.delete();
        }}
    }}

    @SuppressWarnings("unchecked")
    private static Map<String, Object> parseJson(String json) throws Exception {{
        if (ENGINE == null) {{
            throw new IllegalStateException("JavaScript engine is not available in this
JDK.");
        }}
        SimpleBindings bindings = new SimpleBindings();
        bindings.put("jsonString", json);
        Object raw = ENGINE.eval("Java.asJSONCompatible(JSON.parse(jsonString))",
bindings);
        if (raw instanceof Map) {{
            return (Map<String, Object>) raw;
        }}
        return new HashMap<>();
    }}

    @SuppressWarnings("unchecked")
    private static Map<String, Object> asMap(Object candidate) {{
        if (candidate instanceof Map) {{
            return (Map<String, Object>) candidate;
        }}
        return new HashMap<>();
    }}

    private static String toJson(Object value) {{
```

```
        if (value == null) {{
            return "null";
        }}
        if (value instanceof String) {{
            return "\"" + escape((String) value) + "\"";
        }}
        if (value instanceof Number || value instanceof Boolean) {{
            return String.valueOf(value);
        }}
        if (value instanceof Map) {{
            StringBuilder builder = new StringBuilder();
            builder.append("{{");
            boolean first = true;
            for (Map.Entry<?, ?> entry : ((Map<?, ?>) value).entrySet()) {{
                if (!first) {{
                    builder.append(",");
                }}
                first = false;

builder.append("\"").append(escape(String.valueOf(entry.getKey()))).append("\":");
                builder.append(toJson(entry.getValue()));
            }}
            builder.append("}}");
            return builder.toString();
        }}
        if (value instanceof Iterable) {{
            StringBuilder builder = new StringBuilder();
            builder.append("[");
            boolean first = true;
            for (Object item : (Iterable<?>) value) {{
                if (!first) {{
                    builder.append(",");
                }}
                first = false;
                builder.append(toJson(item));
            }}
            builder.append("]");
            return builder.toString();
        }}
        return "\"" + escape(String.valueOf(value)) + "\"";
    }}

    private static String escape(String input) {{
        return     input.replace("\\",     "\\\\").replace("\"",     "\\\"").replace("\n",
"\\n").replace("\r", "\\r");
    }}
}
```

```
"""

    def __init__(self, bot_id: str, code: str, docker_client, image: str = "openjdk:11-jdk-slim"):
        self.bot_id = bot_id
        self.docker_client = docker_client
        self.image = image
        self.class_name = _extract_public_class_name(code)
        self.workdir = tempfile.mkdtemp(prefix=f"javabot_{bot_id}_")
        self._write_file(f"{self.class_name}.java", code)
        bridge_source = self.BRIDGE_TEMPLATE.format(bot_class=self.class_name)
        self._write_file("JavaBotBridge.java", bridge_source)
        self.sandbox = ContainerSandbox(
            docker_client=self.docker_client,
            language="java",
            bot_id=bot_id,
            image=self.image,
            host_workdir=self.workdir,
            container_workdir="/bot",
            volumes={self.workdir: {'bind': '/bot', 'mode': 'rw'}},
            network_disabled=True,
        )
        self._compile_sources([f"{self.class_name}.java", "JavaBotBridge.java"])

    def _write_file(self, filename: str, content: str) -> None:
        path = os.path.join(self.workdir, filename)
        with open(path, "w", encoding="utf-8") as fh:
            fh.write(content)

    def _compile_sources(self, files) -> None:
        command = ["javac"] + list(files)
        self.sandbox.exec(command)

    def _run_bridge(self, command: str, payload: Optional[Dict[str, Any]] = None, expect_response: bool = True) -> Dict[str, Any]:
        env = {}
        if payload is not None:
            env["BOT_PAYLOAD"] = json.dumps(payload)
        decoded = self.sandbox.exec(
            ["java", "JavaBotBridge", command],
            environment=env,
        ).decode().strip()
        if not expect_response:
            return {}
        if not decoded:
            raise RuntimeError("Java bridge повернула порожню відповідь.")
        try:
```

```
            return json.loads(decoded)
        except json.JSONDecodeError as exc:
            snippet = decoded[:200]
            raise RuntimeError(f"Некоректний JSON від Java-бота ({command}): {snippet}")
from exc

    def initialize(self, bot_id: str, game_config: Dict[str, Any]) -> None:
        self._run_bridge("init", {"bot_id": bot_id, "game_config": game_config or {}},
expect_response=False)

    def get_action(self, observation: GameObservation) -> BotAction:
        payload = {
            "observation": {
                **observation.game_state,
                "turn_number": observation.turn_number,
                "player_id": observation.player_id,
                "available_actions": observation.available_actions,
                "metadata": observation.metadata or {},
            }
        }
        response = self._run_bridge("action", payload)
        action_payload = response.get("action", {})
        if isinstance(action_payload, dict):
            action_type = action_payload.get("action_type", "wait")
            parameters = action_payload.get("parameters", {})
        else:
            action_type = str(action_payload)
            parameters = {}
        return BotAction(
            action_type=action_type,
            parameters=parameters,
            bot_id=self.bot_id,
            turn_number=observation.turn_number,
            timestamp=observation.turn_number
        )

    def cleanup(self) -> None:
        try:
            self._run_bridge("cleanup", expect_response=False)
        finally:
            if self.sandbox:
                self.sandbox.cleanup()
            shutil.rmtree(self.workdir, ignore_errors=True)


class JavaExecutor(BotExecutor):
    """Виконавець Java-ботів на Docker (поки із мок-логікою)."""
```

```
    def __init__(self):
        self.java_image = "openjdk:11-jdk-slim"
        if docker is None:
            print("Warning: docker SDK not installed; Java bots will fall back to host
mock.")
            self.docker_client = None
            self.docker_available = False
            return
        try:
            self.docker_client = docker.from_env()
            self.docker_client.ping()
            self.docker_available = True
        except Exception as e:
            print(f"Warning: Docker not available for Java execution: {e}")
            self.docker_client = None
            self.docker_available = False

    def validate_code(self, code: str) -> bool:
        """Базово перевіряє код бота на синтаксичні помилки."""
        if not self.docker_available:
            print("Warning: Java validation skipped - Docker not available")
            return True  # Дозволяємо Java-ботів навіть без Docker

        try:
            class_name = _extract_public_class_name(code)
        except ValueError as exc:
            print(f"Java validation error: {exc}")
            return False

        # Мінімальна перевірка: чи існують потрібні методи
        required_methods = ['initialize', 'getAction', 'cleanup']
        if not all(method in code for method in required_methods):
            print("Java bot missing required methods")
            return False

        temp_dir = tempfile.mkdtemp()
        temp_file = os.path.join(temp_dir, f"{class_name}.java")
        with open(temp_file, 'w') as f:
            f.write(code)

        try:
            # Пробуємо скомпілювати код
            container = self.docker_client.containers.run(
                self.java_image,
                f"javac {class_name}.java",
                volumes={temp_dir: {'bind': '/tmp', 'mode': 'rw'}},
```

```
                working_dir='/tmp',
                detach=True
            )
            result = container.wait()
            container.remove()
            return result['StatusCode'] == 0
        except Exception as e:
            print(f"Java validation error: {e}")
            # Якщо компіляція у Docker не вдалася — все одно дозволяємо бот, але як мок
            return True
        finally:
            import shutil
            shutil.rmtree(temp_dir)

    def create_bot_instance(self, code: str, bot_id: str) -> BotInterface:
        """Створює інстанс бота з переданого коду."""
        if self.docker_available and self.docker_client:
            try:
                return JavaContainerBot(bot_id, code, self.docker_client, self.java_image)
            except Exception as exc:
                print(f"[JavaExecutor] Не вдалося створити контейнер для {bot_id}: {exc}.
Використовую мок-ранер.")
        print(f"[JavaExecutor] Docker недоступний для {bot_id}, запускаємо mock-runner.")
        return MockJavaBot(bot_id)

    def get_supported_language(self) -> str:
        """Повертає мову, яку підтримує цей виконавець."""
        return "java"
```

## Файл src/executors/container_sandbox.py

```
"""
Допоміжний шар для управління довгоживучими Docker-контейнерами під кожну мову/бота.
"""

from __future__ import annotations

import uuid
from typing import Dict, Optional, Sequence

try:
    import docker
    from docker.errors import NotFound
except ImportError:  # pragma: no cover - docker може бути відсутнім на середовищі запуску
    docker = None  # type: ignore
```

```
    NotFound = Exception  # type: ignore


class ContainerSandbox:
    """
    Невеликий враппер над контейнером, який живе стільки, скільки й бот.
    Надає зручні exec/cleanup-методи, щоб виконавці зосередилися на логіці мови.
    """

    def __init__(
        self,
        docker_client,
        *,
        language: str,
        bot_id: str,
        image: str,
        host_workdir: str,
        container_workdir: str = "/bot",
        volumes: Optional[Dict[str, Dict[str, str]]] = None,
        network_disabled: bool = True,
    ):
        if docker_client is None:
            raise RuntimeError("Docker client is not available for sandbox creation.")

        self.docker_client = docker_client
        self.language = language
        self.bot_id = bot_id
        self.image = image
        self.host_workdir = host_workdir
        self.container_workdir = container_workdir
        self.volumes = volumes or {}
        self.network_disabled = network_disabled
        suffix = uuid.uuid4().hex[:6]
        self.name = f"bot-{bot_id}-lang-{language}-{suffix}"
        self.container = self._start_container()

    def _start_container(self):
        try:
            existing = self.docker_client.containers.get(self.name)
            existing.remove(force=True)
        except NotFound:
            pass
        volumes = self.volumes or {}
        if not volumes and self.host_workdir:
            volumes = {
                self.host_workdir: {'bind': self.container_workdir, 'mode': 'rw'}
            }
```

```
        return self.docker_client.containers.run(
            self.image,
            ["sleep", "infinity"],
            name=self.name,
            volumes=volumes,
            network_disabled=self.network_disabled,
            detach=True,
            working_dir=self.container_workdir,
        )

    def exec(
        self,
        command: Sequence[str],
        *,
        environment: Optional[Dict[str, str]] = None,
        workdir: Optional[str] = None,
    ) -> bytes:
        exec_id = self.docker_client.api.exec_create(
            self.container.id,
            command,
            environment=environment,
            workdir=workdir or self.container_workdir,
        )
        output = self.docker_client.api.exec_start(exec_id)
        inspect = self.docker_client.api.exec_inspect(exec_id)
        exit_code = inspect.get("ExitCode", 0)
        if exit_code != 0:
            joined = " ".join(command)
            snippet = output.decode(errors="ignore").strip()
            raise RuntimeError(f"Команда '{joined}' завершилась з кодом {exit_code}:
{snippet}")
        return output

    def cleanup(self) -> None:
        if not self.container:
            return
        try:
            self.container.remove(force=True)
        except Exception:
            pass
        finally:
            self.container = None
```

## Файл src/games/simple_game.py

```
"""
Проста покрокова гра: боти пересуваються полем та збирають жетони «Очки перемоги».
```

```
Основна мета — зібрати найбільше жетонів або зробити це ефективніше за суперників.
"""

import copy
import random
import time
from typing import Any, Dict, List

from src.core.interfaces import (
    BotAction,
    GameEngine,
    GameObservation,
    GameResult,
    GameState,
)


class SimpleGame(GameEngine):
    """Найпростіший рушій збору предметів у форматі «кожен сам за себе»."""

    BOARD_SIZE = 10
    MAX_TURNS = 100
    FAILURE_PENALTY = 0.1
    SLOW_ACTION_THRESHOLD = 0.2  # секунди
    SLOW_ACTION_PENALTY = 0.1

    def __init__(self):
        self._reset_runtime_state()

    def _reset_runtime_state(self) -> None:
        """Скидає ігровий контекст між запусками."""
        self.game_id = None
        self.bots: Dict[str, Dict[str, Any]] = {}
        self.turn_number = 0
        self.max_turns = self.MAX_TURNS
        self.current_state = GameState.WAITING
        self.start_time = None
        self.replay_data: List[Dict[str, Any]] = []
        self.detailed_steps: List[Dict[str, Any]] = []
        self.bot_metrics: Dict[str, Dict[str, Any]] = {}
        self.last_collect_gain: Dict[str, int] = {}
        self.game_state = self._create_empty_game_state()

    def _create_empty_game_state(self) -> Dict[str, Any]:
        """Створює початковий стан поля без ботів та предметів."""
        return {
            'board_size': self.BOARD_SIZE,
```

```
            'items': [],
            'bot_positions': {},
            'bot_scores': {},
            'bot_health': {}
        }

    def _initial_bot_metrics(self) -> Dict[str, Any]:
        """Стандартизований набір показників для будь-якого бота."""
        return {
            'total_execution_time': 0.0,
            'actions_count': 0,
            'successful_actions': 0,
            'failed_actions': 0,
            'errors_count': 0,
            'average_action_time': 0.0,
            'items_collected': 0,
            'distance_traveled': 0,
            'slow_actions': 0,
            'penalty_points': 0.0
        }

    def _update_average_time(self, bot_id: str) -> None:
        """Перераховує середній час реакції для конкретного бота."""
        metrics = self.bot_metrics[bot_id]
        if metrics['actions_count'] > 0:
            metrics['average_action_time'] = (
                metrics['total_execution_time'] / metrics['actions_count']
            )

    def initialize_game(self, bot_configs: List[Dict[str, Any]]) -> str:
        """Готує гру: розставляє ботів, генерує предмети та обнуляє показники."""
        self._reset_runtime_state()
        self.game_id = f"game_{int(time.time())}"
        self.current_state = GameState.RUNNING
        self.start_time = time.time()

        # Ініціалізуємо ботів
        for i, bot_config in enumerate(bot_configs):
            bot_id = f"bot_{i}"
            self.bots[bot_id] = bot_config
            self.game_state['bot_positions'][bot_id] = {
                'x': random.randint(0, self.BOARD_SIZE - 1),
                'y': random.randint(0, self.BOARD_SIZE - 1)
            }
            self.game_state['bot_scores'][bot_id] = 0
            self.game_state['bot_health'][bot_id] = 100
```

```
            self.bot_metrics[bot_id] = self._initial_bot_metrics()
            self.last_collect_gain[bot_id] = 0

        # Генеруємо випадкові предмети на полі (щонайменше один)
        self._generate_items(min_items=5, max_items=12)

        # Фіксуємо початковий стан (до першого ходу)
        self._record_game_state()
        self._add_detailed_step("Game initialized", {
            'bots': list(self.bots.keys()),
            'board_size': self.game_state['board_size'],
            'initial_positions': self.game_state['bot_positions'].copy(),
            'initial_items': len(self.game_state['items']),
            'turn': 0
        })

        return self.game_id

    def get_observation(self, bot_id: str) -> GameObservation:
        """Формує спостереження для конкретного бота з урахуванням видимості."""
        if bot_id not in self.bots:
            raise ValueError(f"Bot {bot_id} not found in game")

        # Формуємо спостереження для бота
        observation = GameObservation(
            game_state=self._get_visible_state(bot_id),
            turn_number=self.turn_number,
            player_id=bot_id,
            available_actions=self._get_available_actions(),
            metadata={
                'board_size': self.game_state['board_size'],
                'max_turns': self.max_turns
            }
        )

        return observation

    def apply_action(self, action: BotAction, execution_time: float = 0.0) -> bool:
        """Обробляє дію бота та оновлює пов'язані метрики."""
        if self.current_state != GameState.RUNNING:
            return False

        bot_id = action.bot_id
        if bot_id not in self.bots:
            return False

        try:
```

```
            # Оновлюємо метрики
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['total_execution_time'] += execution_time
                self.bot_metrics[bot_id]['actions_count'] += 1
                if execution_time > 0:
                    self._update_average_time(bot_id)

            # Обробляємо дію
            success = self._process_action(bot_id, action)

            # Фіксуємо успіх / невдачу
            if bot_id in self.bot_metrics:
                if success:
                    self.bot_metrics[bot_id]['successful_actions'] += 1
                else:
                    self.bot_metrics[bot_id]['failed_actions'] += 1
                    self.bot_metrics[bot_id]['penalty_points'] += self.FAILURE_PENALTY
            if bot_id in self.bot_metrics and execution_time and execution_time >
self.SLOW_ACTION_THRESHOLD:
                self.bot_metrics[bot_id]['slow_actions'] += 1
                self.bot_metrics[bot_id]['penalty_points'] += self.SLOW_ACTION_PENALTY

            # Записуємо дію в реплей
            self._record_action(bot_id, action, success, execution_time)

            # Додаємо деталізований крок
            self._add_detailed_step_from_action(bot_id, action, success, execution_time)

            # Перевіряємо, чи треба завершити гру
            if self._should_end_game():
                self.current_state = GameState.FINISHED

            return success

        except Exception as e:
            print(f"Error processing action for {bot_id}: {e}")
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['errors_count'] += 1
            return False

    def is_game_finished(self) -> bool:
        """Перевіряє, чи досягла гра завершального стану."""
        return self.current_state == GameState.FINISHED

    def get_game_result(self) -> GameResult:
        """Повертає фінальний протокол, включно з розшифровкою тай-брейків."""
        if not self.is_game_finished():
```

```
            raise RuntimeError("Game is not finished yet")

        # Перевіряємо, що гра мала хоч якусь динаміку
        if len(self.detailed_steps) < 2:  # Мінімум ініціалізація + одна дія
            raise RuntimeError("Game finished with insufficient step data - bots may not have made any moves")

        # Упевнюємось, що боти справді діяли після старту
        action_steps = [step for step in self.detailed_steps if step['description'] != 'Game initialized']
        if len(action_steps) == 0:
            raise RuntimeError("Game finished with no bot actions - bots did not make any moves")

        # Визначаємо переможця (найбільший рахунок)
        scores = self.game_state['bot_scores']

        # Фінальні очки (база - штрафи)
        enhanced_scores = {}
        tiebreaker_breakdown = {}
        for bot_id in self.bots.keys():
            base_score = scores.get(bot_id, 0)
            metrics = self.bot_metrics.get(bot_id, {})

            penalty_points = metrics.get('penalty_points', 0.0)
            actions_count = metrics.get('actions_count', 0)
            success_rate = round(metrics.get('successful_actions', 0) / actions_count, 3) if actions_count > 0 else 0
            avg_time = metrics.get('average_action_time', 0)

            final_score = base_score - penalty_points
            enhanced_scores[bot_id] = final_score
            tiebreaker_breakdown[bot_id] = {
                'base_score': base_score,
                'items_collected': metrics.get('items_collected', 0),
                'success_rate': success_rate,
                'average_action_time': avg_time,
                'distance_traveled': metrics.get('distance_traveled', 0),
                'slow_actions': metrics.get('slow_actions', 0),
                'errors_count': metrics.get('errors_count', 0),
                'penalty_points': penalty_points
            }

        winner = max(enhanced_scores.keys(), key=lambda k: enhanced_scores[k]) if enhanced_scores else None

        # Підраховуємо тривалість гри
```

```
        duration = time.time() - self.start_time if self.start_time else 0

        return GameResult(
            winner=winner,
            scores=enhanced_scores,
            game_duration=duration,
            total_turns=self.turn_number,
            replay_data=self.replay_data.copy(),
            metadata={
                'game_type': 'simple_game',
                'max_turns': self.max_turns,
                'detailed_steps': self.detailed_steps.copy(),
                'final_scores': scores.copy(),
                'final_positions': self.game_state['bot_positions'].copy(),
                'bot_metrics': self.bot_metrics.copy(),
                'victory_point_label': 'Очки перемоги',
                'tiebreaker_rules': [
                    'Кількість зібраних предметів',
                    'Мінімум штрафів (неуспішні/повільні дії)',
                    'Середній час реакції програми',
                    'Дистанція, яку пройшов бот'
                ],
                'tiebreaker_breakdown': tiebreaker_breakdown
            }
        )

    def get_game_state(self) -> GameState:
        """Повертає поточний статус (WAITING/RUNNING/FINISHED)."""
        return self.current_state

    def _generate_items(self, min_items: int = 5, max_items: int = 15):
        """Випадково розкладає жетони по клітинках (не менше одного на гру)."""
        num_items = max(1, random.randint(min_items, max_items))
        self.game_state['items'] = []

        attempts = 0
        limit = num_items * 5
        while len(self.game_state['items']) < num_items and attempts < limit:
            attempts += 1
            item = {
                'x': random.randint(0, self.BOARD_SIZE - 1),
                'y': random.randint(0, self.BOARD_SIZE - 1),
                'value': 1,
                'type': 'score',
                'label': 'Очки перемоги'
            }
            if any(existing['x'] == item['x'] and existing['y'] == item['y'] for existing
```

```
in self.game_state['items']):
                continue
            self.game_state['items'].append(item)

    def _get_visible_state(self, bot_id: str) -> Dict[str, Any]:
        """Повертає частину стану, яку бот бачить у радіусі Манхеттена = 3."""
        bot_pos = self.game_state['bot_positions'][bot_id]

        # Видимість: бот бачить предмети в радіусі 3 клітин
        visible_items = []
        for item in self.game_state['items']:
            distance = abs(item['x'] - bot_pos['x']) + abs(item['y'] - bot_pos['y'])
            if distance <= 3:
                visible_items.append(item)

        return {
            'my_position': bot_pos,
            'my_score': self.game_state['bot_scores'][bot_id],
            'my_health': self.game_state['bot_health'][bot_id],
            'visible_items': visible_items,
            'turn_number': self.turn_number
        }

    def _get_available_actions(self) -> List[str]:
        """Список дозволених дій для нинішнього режиму."""
        # У цьому режимі доступні лише рух і підбір жетонів
        return ['move', 'collect']

    def _process_action(self, bot_id: str, action: BotAction) -> bool:
        """Маршрутизує дію до конкретного обробника."""
        action_type = action.action_type
        parameters = action.parameters

        if action_type == 'move':
            return self._handle_move(bot_id, parameters)
        elif action_type == 'collect':
            return self._handle_collect(bot_id, parameters)
        elif action_type == 'wait':
            return True
        else:
            return False

    def _handle_move(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Рухає бота на суміжну клітинку з перевіркою зіткнень."""
        direction = parameters.get('direction', 'up')
        pos = self.game_state['bot_positions'][bot_id]
```

```
        old_pos = pos.copy()
        new_pos = pos.copy()
        if direction == 'up' and pos['y'] > 0:
            new_pos['y'] -= 1
        elif direction == 'down' and pos['y'] < self.BOARD_SIZE - 1:
            new_pos['y'] += 1
        elif direction == 'left' and pos['x'] > 0:
            new_pos['x'] -= 1
        elif direction == 'right' and pos['x'] < self.BOARD_SIZE - 1:
            new_pos['x'] += 1

        # Перевіряємо, чи вільна клітинка
        occupied = any(
            other_pos['x'] == new_pos['x'] and other_pos['y'] == new_pos['y']
            for other_bot, other_pos in self.game_state['bot_positions'].items()
            if other_bot != bot_id
        )

        if not occupied:
            self.game_state['bot_positions'][bot_id] = new_pos
        # Оновлюємо пройдену відстань
            if bot_id in self.bot_metrics:
                distance = abs(new_pos['x'] - old_pos['x']) + abs(new_pos['y'] -
old_pos['y'])
                self.bot_metrics[bot_id]['distance_traveled'] += distance
            return True

        return False

    def _handle_collect(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Обробка підбору предметів у поточній клітинці."""
        bot_pos = self.game_state['bot_positions'][bot_id]
        collected_indexes = []
        total_gain = 0

        for idx, item in enumerate(self.game_state['items']):
            if item['x'] == bot_pos['x'] and item['y'] == bot_pos['y']:
                collected_indexes.append(idx)

        for idx in reversed(collected_indexes):
            item = self.game_state['items'].pop(idx)
            value = item.get('value', 1)
            total_gain += value
            self.game_state['bot_scores'][bot_id] += value
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['items_collected'] += 1
```

```
        self.last_collect_gain[bot_id] = total_gain
        return total_gain > 0

    def _should_end_game(self) -> bool:
        """Завершує гру після max_turns або коли жетонів не лишилось."""
        # Завершуємо, якщо досягли ліміту ходів
        if self.turn_number >= self.max_turns:
            return True

        # Або якщо всі жетони вже зібрані
        if not self.game_state.get('items'):
            return True

        return False

    def _record_game_state(self):
        """Фіксує знімок стану для подальшого повтору."""
        state_snapshot = {
            'turn_number': self.turn_number,
            'game_state': copy.deepcopy(self.game_state),
            'timestamp': time.time(),
            'bot_metrics': {k: v.copy() for k, v in self.bot_metrics.items()}
        }
        self.replay_data.append(state_snapshot)

    def _record_action(self, bot_id: str, action: BotAction, success: bool, execution_time:
float = 0.0):
        """Додає дію до останнього кроку реплею."""
        action_record = {
            'bot_id': bot_id,
            'action': {
                'action_type': action.action_type,
                'parameters': action.parameters
            },
            'success': success,
            'turn_number': self.turn_number,
            'timestamp': time.time(),
            'execution_time': execution_time
        }

        if self.replay_data:
            self.replay_data[-1]['actions'] = self.replay_data[-1].get('actions', [])
            self.replay_data[-1]['actions'].append(action_record)

    def advance_turn(self):
        """Зсуває лічильник ходів та зберігає стан."""
        if self.current_state == GameState.RUNNING:
```

```
            self.turn_number += 1
            self._record_game_state()

    def _add_detailed_step(self, description: str, details: Dict[str, Any]):
        """Зберігає людинозрозумілий опис дії для UI."""
        step = {
            'turn': self.turn_number,
            'description': description,
            'details': details,
            'timestamp': time.time() - self.start_time if self.start_time else 0
        }
        self.detailed_steps.append(step)

    def _add_detailed_step_from_action(self, bot_id: str, action: BotAction, success: bool,
execution_time: float = 0.0):
        """Генерує опис з огляду на тип дії бота."""
        action_type = action.action_type
        parameters = action.parameters

        # Формуємо опис залежно від типу дії
        if action_type == 'move':
            direction = parameters.get('direction', 'unknown')
            description = f"{bot_id} moved {direction}"
            details = {
                'bot_id': bot_id,
                'action': 'move',
                'direction': direction,
                'new_position': self.game_state['bot_positions'][bot_id]
            }
        elif action_type == 'collect':
            description = f"{bot_id} collected item"
            details = {
                'bot_id': bot_id,
                'action': 'collect',
                'position': self.game_state['bot_positions'][bot_id],
                'score_gained': self.last_collect_gain.get(bot_id, 0),
                'items_collected_total':                    self.bot_metrics.get(bot_id,
{}).get('items_collected', 0)
            }
        elif action_type == 'wait':
            description = f"{bot_id} waited"
            details = {
                'bot_id': bot_id,
                'action': 'wait',
                'position': self.game_state['bot_positions'][bot_id]
            }
        else:
```

```
            description = f"{bot_id} performed {action_type}"
            details = {
                'bot_id': bot_id,
                'action': action_type,
                'parameters': parameters
            }

        details['success'] = success
        if execution_time and execution_time > 0:
            details['execution_time'] = execution_time

        if not success:
            description += " (неуспішно)"

        self._add_detailed_step(description, details)
```

## Файл src/games/simple_team_game.py

```
"""Командний різновид простої гри на збирання жетонів «Очки перемоги»."""

import copy
import random
import time
from typing import Any, Dict, List

from src.core.interfaces import BotAction, GameEngine, GameObservation, GameResult,
GameState


class SimpleTeamGame(GameEngine):
    """Дві команди ботів кооперуються та змагаються за сумарні очки."""

    BOARD_SIZE = 10
    MAX_TURNS = 100
    FAILURE_PENALTY = 0.1
    SLOW_ACTION_THRESHOLD = 0.2  # секунди
    SLOW_ACTION_PENALTY = 0.1

    def __init__(self):
        self.teams = ['team_alpha', 'team_beta']
        self._reset_runtime_state()

    def _reset_runtime_state(self) -> None:
        """Скидає увесь робочий стан між запусками турніру."""
        self.game_id = None
        self.bots: Dict[str, Dict[str, Any]] = {}
        self.team_assignments: Dict[str, str] = {}
```

```
        self.turn_number = 0
        self.max_turns = self.MAX_TURNS
        self.current_state = GameState.WAITING
        self.start_time = None
        self.replay_data: List[Dict[str, Any]] = []
        self.detailed_steps: List[Dict[str, Any]] = []
        self.bot_metrics: Dict[str, Dict[str, Any]] = {}
        self.last_collect_gain: Dict[str, int] = {}
        self.game_state = self._create_empty_state()

    def _create_empty_state(self) -> Dict[str, Any]:
        """Формує початкову конфігурацію поля."""
        return {
            'board_size': self.BOARD_SIZE,
            'items': [],
            'bot_positions': {},
            'bot_scores': {},
            'bot_health': {},
            'team_scores': {team: 0 for team in self.teams},
            'team_assignments': {},
        }

    def _initial_bot_metrics(self) -> Dict[str, Any]:
        """Стандартизований набір метрик для учасника команди."""
        return {
            'total_execution_time': 0.0,
            'actions_count': 0,
            'successful_actions': 0,
            'failed_actions': 0,
            'errors_count': 0,
            'average_action_time': 0.0,
            'items_collected': 0,
            'distance_traveled': 0,
            'slow_actions': 0,
            'penalty_points': 0.0
        }

    def _update_average_time(self, bot_id: str) -> None:
        """Перерахування середньої затримки реакції бота."""
        metrics = self.bot_metrics[bot_id]
        if metrics['actions_count'] > 0:
            metrics['average_action_time'] = (
                metrics['total_execution_time'] / metrics['actions_count']
            )

    def initialize_game(self, bot_configs: List[Dict[str, Any]]) -> str:
        """Готує командну гру: розподіл ботів та генерація предметів."""
```

```
        self._reset_runtime_state()
        self.game_id = f"team_game_{int(time.time())}"
        self.current_state = GameState.RUNNING
        self.start_time = time.time()

        half = max(1, len(bot_configs) // 2)
        for i, bot_config in enumerate(bot_configs):
            bot_id = f"bot_{i}"
            team = self.teams[0] if i < half else self.teams[1]
            self.bots[bot_id] = bot_config
            self.team_assignments[bot_id] = team
            self.game_state['team_assignments'][bot_id] = team
            self.game_state['bot_positions'][bot_id] = {
                'x': random.randint(0, self.BOARD_SIZE - 1),
                'y': random.randint(0, self.BOARD_SIZE - 1)
            }
            self.game_state['bot_scores'][bot_id] = 0
            self.game_state['bot_health'][bot_id] = 100

            self.bot_metrics[bot_id] = self._initial_bot_metrics()
            self.last_collect_gain[bot_id] = 0

        self._generate_items()
        self._record_game_state()
        self._add_detailed_step("Game initialized", {
            'bots': list(self.bots.keys()),
            'board_size': self.game_state['board_size'],
            'initial_positions': copy.deepcopy(self.game_state['bot_positions']),
            'initial_items': len(self.game_state['items']),
            'team_assignments': self.team_assignments.copy(),
            'turn': 0
        })
        return self.game_id

    def get_observation(self, bot_id: str) -> GameObservation:
        """Повертає те, що бачить конкретний бот (позиція, предмети, статус команди)."""
        if bot_id not in self.bots:
            raise ValueError(f"Bot {bot_id} not found in game")

        bot_pos = self.game_state['bot_positions'][bot_id]
        visible_items = []
        for item in self.game_state['items']:
            distance = abs(item['x'] - bot_pos['x']) + abs(item['y'] - bot_pos['y'])
            if distance <= 3:
                visible_items.append(item)

        observation = GameObservation(
```

```
            game_state={
                'my_position': bot_pos,
                'my_score': self.game_state['bot_scores'][bot_id],
                'my_team': self.team_assignments.get(bot_id),
                'team_scores': self.game_state['team_scores'],
                'visible_items': visible_items
            },
            turn_number=self.turn_number,
            player_id=bot_id,
            available_actions=self._get_available_actions(),
            metadata={
                'board_size': self.game_state['board_size'],
                'max_turns': self.max_turns
            }
        )
        return observation

    def apply_action(self, action: BotAction, execution_time: float = 0.0) -> bool:
        """Обробляє дію бота, фіксує штрафи та статистику."""
        if self.current_state != GameState.RUNNING:
            return False

        bot_id = action.bot_id
        if bot_id not in self.bots:
            return False

        try:
            if bot_id in self.bot_metrics:
                metrics = self.bot_metrics[bot_id]
                metrics['total_execution_time'] += execution_time
                metrics['actions_count'] += 1
                self._update_average_time(bot_id)

            success = self._process_action(bot_id, action)

            if bot_id in self.bot_metrics:
                if success:
                    self.bot_metrics[bot_id]['successful_actions'] += 1
                else:
                    self.bot_metrics[bot_id]['failed_actions'] += 1
                    self.bot_metrics[bot_id]['penalty_points'] += self.FAILURE_PENALTY
            if bot_id in self.bot_metrics and execution_time and execution_time >
self.SLOW_ACTION_THRESHOLD:
                self.bot_metrics[bot_id]['slow_actions'] += 1
                self.bot_metrics[bot_id]['penalty_points'] += self.SLOW_ACTION_PENALTY

            self._record_action(bot_id, action, success, execution_time)
```

```
            self._add_detailed_step_from_action(bot_id, action, success, execution_time)

            if self._should_end_game():
                self.current_state = GameState.FINISHED

            return success
        except Exception as exc:
            print(f"Error processing action for {bot_id}: {exc}")
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['errors_count'] += 1
            return False

    def is_game_finished(self) -> bool:
        """Повертає True, коли матч завершено (ліміт ходів чи відсутність жетонів)."""
        return self.current_state == GameState.FINISHED

    def get_game_result(self) -> GameResult:
        """Формує фінальний протокол з акцентом на командні тай-брейки."""
        if not self.is_game_finished():
            raise RuntimeError("Game is not finished yet")

        if len(self.detailed_steps) < 2:
            raise RuntimeError("Game finished with insufficient step data")

        action_steps = [step for step in self.detailed_steps if step['description'] != 'Game
initialized']
        if not action_steps:
            raise RuntimeError("Game finished with no bot actions")

        team_scores = self.game_state['team_scores'].copy()
        team_metrics = {
            team: {
                'base_score': team_scores.get(team, 0),
                'items_collected': 0,
                'errors_count': 0,
                'successful_actions': 0,
                'actions_count': 0,
                'total_execution_time': 0.0,
                'distance_traveled': 0,
                'penalty_points': 0.0,
                'slow_actions': 0
            } for team in self.teams
        }

        for bot_id, metrics in self.bot_metrics.items():
            team = self.team_assignments.get(bot_id)
            if not team:
```

```
                continue
            team_entry = team_metrics[team]
            team_entry['items_collected'] += metrics.get('items_collected', 0)
            team_entry['errors_count'] += metrics.get('errors_count', 0)
            team_entry['successful_actions'] += metrics.get('successful_actions', 0)
            team_entry['actions_count'] += metrics.get('actions_count', 0)
            team_entry['total_execution_time'] += metrics.get('total_execution_time', 0.0)
            team_entry['distance_traveled'] += metrics.get('distance_traveled', 0)
            team_entry['penalty_points'] += metrics.get('penalty_points', 0.0)
            team_entry['slow_actions'] += metrics.get('slow_actions', 0)

        enhanced_scores = {}
        tiebreaker_breakdown = {}
        team_metrics_for_metadata: Dict[str, Dict[str, Any]] = {}
        for team in self.teams:
            stats = team_metrics[team]
            actions = stats['actions_count']
            success_rate = round((stats['successful_actions'] / actions), 3) if actions >
0 else 0
            avg_time = (stats['total_execution_time'] / actions) if actions > 0 else 0
            penalties = stats['penalty_points']
            final_score = stats['base_score'] - penalties
            enhanced_scores[team] = final_score
            tiebreaker_breakdown[team] = {
                'base_score': stats['base_score'],
                'items_collected': stats['items_collected'],
                'success_rate': success_rate,
                'average_action_time': avg_time,
                'distance_traveled': stats['distance_traveled'],
                'slow_actions': stats['slow_actions'],
                'errors_count': stats['errors_count'],
                'penalty_points': penalties
            }
            team_metrics_for_metadata[team] = {
                'actions_count': stats['actions_count'],
                'successful_actions': stats['successful_actions'],
                'items_collected': stats['items_collected'],
                'distance_traveled': stats['distance_traveled'],
                'penalty_points': penalties,
                'slow_actions': stats['slow_actions'],
                'average_action_time': avg_time
            }

        winner = max(enhanced_scores.keys(), key=lambda k: enhanced_scores[k]) if
enhanced_scores else None
        duration = time.time() - self.start_time if self.start_time else 0
```

```
        return GameResult(
            winner=winner,
            scores=enhanced_scores,
            game_duration=duration,
            total_turns=self.turn_number,
            replay_data=self.replay_data.copy(),
            metadata={
                'game_type': 'simple_team_game',
                'max_turns': self.max_turns,
                'detailed_steps': self.detailed_steps.copy(),
                'final_positions': self.game_state['bot_positions'].copy(),
                'team_scores': self.game_state['team_scores'].copy(),
                'team_assignments': self.team_assignments.copy(),
                'bot_metrics': self.bot_metrics.copy(),
                'team_metrics': team_metrics_for_metadata,
                'victory_point_label': 'Очки перемоги (команди)',
                'tiebreaker_rules': [
                    'Кількість зібраних предметів командою',
                    'Мінімум штрафів (неуспішні/повільні дії)',
                    'Середній час реакції команди',
                    'Дистанція, яку пройшла команда'
                ],
                'tiebreaker_breakdown': tiebreaker_breakdown
            }
        )

    def get_game_state(self) -> GameState:
        """Поточний статус життєвого циклу гри."""
        return self.current_state

    # Допоміжні методи повторно використовують логіку простої гри
    def _get_available_actions(self) -> List[str]:
        """Єдиний перелік дій для всіх ботів режиму."""
        return ['move', 'collect']

    def _process_action(self, bot_id: str, action: BotAction) -> bool:
        """Направляє запитану дію до відповідного обробника."""
        action_type = action.action_type
        parameters = action.parameters
        if action_type == 'move':
            return self._handle_move(bot_id, parameters)
        if action_type == 'collect':
            return self._handle_collect(bot_id, parameters)
        if action_type == 'wait':
            return True
        return False
```

```
    def _handle_move(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Контролює переміщення з перевіркою меж і зіткнень."""
        direction = parameters.get('direction', 'up')
        pos = self.game_state['bot_positions'][bot_id]
        old_pos = pos.copy()
        new_pos = pos.copy()
        if direction == 'up' and pos['y'] > 0:
            new_pos['y'] -= 1
        elif direction == 'down' and pos['y'] < 9:
            new_pos['y'] += 1
        elif direction == 'left' and pos['x'] > 0:
            new_pos['x'] -= 1
        elif direction == 'right' and pos['x'] < 9:
            new_pos['x'] += 1

        occupied = any(
            other_pos['x'] == new_pos['x'] and other_pos['y'] == new_pos['y']
            for other_bot, other_pos in self.game_state['bot_positions'].items()
            if other_bot != bot_id
        )
        if not occupied:
            self.game_state['bot_positions'][bot_id] = new_pos
            if bot_id in self.bot_metrics:
                distance = abs(new_pos['x'] - old_pos['x']) + abs(new_pos['y'] -
old_pos['y'])
                self.bot_metrics[bot_id]['distance_traveled'] += distance
            return True
        return False

    def _handle_collect(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Збирання жетонів з клітинки та оновлення командних очок."""
        bot_pos = self.game_state['bot_positions'][bot_id]
        collected = False
        total_gain = 0
        for i in reversed(range(len(self.game_state['items']))):
            item = self.game_state['items'][i]
            if item['x'] == bot_pos['x'] and item['y'] == bot_pos['y']:
                collected = True
                self.game_state['items'].pop(i)
                value = item.get('value', 1)
                total_gain += value
                self.game_state['bot_scores'][bot_id] += value
                team = self.team_assignments.get(bot_id)
                if team:
                    self.game_state['team_scores'][team] += value
                if bot_id in self.bot_metrics:
                    self.bot_metrics[bot_id]['items_collected'] += 1
```

```
        self.last_collect_gain[bot_id] = total_gain
        return collected

    def _should_end_game(self) -> bool:
        """Кінець гри: досягли MAX_TURNS або більше немає жетонів."""
        if self.turn_number >= self.max_turns:
            return True
        if not self.game_state.get('items'):
            return True
        return False

    def _record_game_state(self):
        """Фіксує стан для повтору та аналітики."""
        state_snapshot = {
            'turn_number': self.turn_number,
            'game_state': copy.deepcopy(self.game_state),
            'timestamp': time.time(),
            'bot_metrics': {k: v.copy() for k, v in self.bot_metrics.items()}
        }
        self.replay_data.append(state_snapshot)

    def _record_action(self, bot_id: str, action: BotAction, success: bool, execution_time:
float = 0.0):
        """Зберігає дію в історію та додає командний тег."""
        action_record = {
            'bot_id': bot_id,
            'team': self.team_assignments.get(bot_id),
            'action': {
                'action_type': action.action_type,
                'parameters': action.parameters
            },
            'success': success,
            'turn_number': self.turn_number,
            'timestamp': time.time(),
            'execution_time': execution_time
        }
        if self.replay_data:
            self.replay_data[-1].setdefault('actions', []).append(action_record)

    def advance_turn(self):
        """Просуває гру на крок і знімає стан."""
        if self.current_state == GameState.RUNNING:
            self.turn_number += 1
            self._record_game_state()

    def _add_detailed_step(self, description: str, details: Dict[str, Any]):
        """Додає людинозрозумілу подію в лог (для UI)."""
```

```
        step = {
            'turn': self.turn_number,
            'description': description,
            'details': details,
            'timestamp': time.time() - self.start_time if self.start_time else 0
        }
        self.detailed_steps.append(step)

    def _add_detailed_step_from_action(self, bot_id: str, action: BotAction, success: bool,
execution_time: float = 0.0):
        """Автоматично формує детальну подію за результатами кроку."""
        action_type = action.action_type
        parameters = action.parameters
        team = self.team_assignments.get(bot_id)

        if action_type == 'move':
            direction = parameters.get('direction', 'unknown')
            description = f"{bot_id} moved {direction}"
            details = {
                'bot_id': bot_id,
                'team': team,
                'action': 'move',
                'direction': direction,
                'new_position': self.game_state['bot_positions'][bot_id]
            }
        elif action_type == 'collect':
            description = f"{bot_id} collected item"
            details = {
                'bot_id': bot_id,
                'team': team,
                'action': 'collect',
                'position': self.game_state['bot_positions'][bot_id],
                'score_gained': self.last_collect_gain.get(bot_id, 0),
                'items_collected_total':                    self.bot_metrics.get(bot_id,
{}).get('items_collected', 0)
            }
        elif action_type == 'wait':
            description = f"{bot_id} waited"
            details = {
                'bot_id': bot_id,
                'team': team,
                'action': 'wait',
                'position': self.game_state['bot_positions'][bot_id]
            }
        else:
            description = f"{bot_id} performed {action_type}"
            details = {
```

```
                'bot_id': bot_id,
                'team': team,
                'action': action_type,
                'parameters': parameters
            }

        details['success'] = success
        if execution_time and execution_time > 0:
            details['execution_time'] = execution_time
        if not success:
            description += " (неуспішно)"

        self._add_detailed_step(description, details)

    def _generate_items(self, min_items: int = 10, max_items: int = 18):
        """Рівномірно розкладає жетони по дошці."""
        num_items = max(2, random.randint(min_items, max_items))
        self.game_state['items'] = []
        attempts = 0
        limit = num_items * 5
        while len(self.game_state['items']) < num_items and attempts < limit:
            attempts += 1
            item = {
                'x': random.randint(0, self.BOARD_SIZE - 1),
                'y': random.randint(0, self.BOARD_SIZE - 1),
                'value': 1,
                'type': 'score',
                'label': 'Очки перемоги'
            }
            if any(existing['x'] == item['x'] and existing['y'] == item['y'] for existing
in self.game_state['items']):
                continue
            self.game_state['items'].append(item)
```

## Файл src/games/fighting_game.py

```
"""
Рушій бою у стилі арени: боти рухаються, атакують, захищаються й застосовують здібності.
"""

import random
import time
import math
from typing import Dict, List, Any, Optional, Tuple
```

```
from src.core.interfaces import GameEngine, GameState, GameObservation, BotAction,
GameResult


class FightingGame(GameEngine):
    """Повноцінна бойова арена з набором атакувальних та підтримувальних дій."""

    def __init__(self):
        self.game_id = None
        self.bots = {}
        self.game_state = {
            'arena_size': 15,
            'bot_positions': {},
            'bot_health': {},
            'bot_energy': {},
            'bot_armor': {},
            'bot_weapons': {},
            'power_ups': [],
            'obstacles': []
        }
        self.turn_number = 0
        self.max_turns = 200
        self.current_state = GameState.WAITING
        self.start_time = None
        self.replay_data = []
        self.detailed_steps = []  # Зберігаємо покроковий журнал подій
        # Моніторинг продуктивності
        self.bot_metrics = {}  # Метрики бота: час реакції, кількість дій, помилки тощо

    def initialize_game(self, bot_configs: List[Dict[str, Any]]) -> str:
        """Створює нову гру на арені та розставляє ботів."""
        self.game_id = f"fighting_game_{int(time.time())}"
        self.turn_number = 0
        self.current_state = GameState.RUNNING
        self.start_time = time.time()
        self.replay_data = []
        self.detailed_steps = []

        # Оновлюємо глобальний стан арени
        self.game_state = {
            'arena_size': 15,
            'bot_positions': {},
            'bot_health': {},
            'bot_energy': {},
            'bot_armor': {},
            'bot_weapons': {},
            'power_ups': [],
```

```
            'obstacles': []
        }

        # Ініціалізуємо ботів
        for i, bot_config in enumerate(bot_configs):
            bot_id = f"bot_{i}"
            self.bots[bot_id] = bot_config

            # На старті розкладаємо ботів по кутах
            positions = [(2, 2), (12, 12), (2, 12), (12, 2)]
            pos = positions[i % len(positions)]

            self.game_state['bot_positions'][bot_id] = {'x': pos[0], 'y': pos[1]}
            self.game_state['bot_health'][bot_id] = 100
            self.game_state['bot_energy'][bot_id] = 50
            self.game_state['bot_armor'][bot_id] = 0
            self.game_state['bot_weapons'][bot_id] = 'fist'  # Базова зброя на старті

            # Початкові метрики продуктивності
            self.bot_metrics[bot_id] = {
                'total_execution_time': 0.0,
                'actions_count': 0,
                'successful_actions': 0,
                'failed_actions': 0,
                'errors_count': 0,
                'average_action_time': 0.0,
                'damage_dealt': 0,
                'damage_taken': 0,
                'items_collected': 0,
                'distance_traveled': 0
            }

        # Генеруємо перепони та підсилювачі
        self._generate_obstacles()
        self._generate_power_ups()

        # Фіксуємо стартовий стан для повтору
        self._record_game_state()
        self._add_detailed_step("Game initialized", {
            'bots': list(self.bots.keys()),
            'arena_size': self.game_state['arena_size'],
            'initial_positions': self.game_state['bot_positions'].copy()
        })

        return self.game_id

    def get_observation(self, bot_id: str) -> GameObservation:
```

```
        """Повертає видимий стан арени для конкретного бота."""
        if bot_id not in self.bots:
            raise ValueError(f"Bot {bot_id} not found in game")

        # Формуємо спостереження для бота
        observation = GameObservation(
            game_state=self._get_visible_state(bot_id),
            turn_number=self.turn_number,
            player_id=bot_id,
            available_actions=self._get_available_actions(),
            metadata={
                'arena_size': self.game_state['arena_size'],
                'max_turns': self.max_turns,
                'game_type': 'fighting_game'
            }
        )

        return observation

    def apply_action(self, action: BotAction, execution_time: float = 0.0) -> bool:
        """Застосовує дію бота до стану гри."""
        if self.current_state != GameState.RUNNING:
            return False

        bot_id = action.bot_id
        if bot_id not in self.bots:
            return False

        try:
            # Оновлюємо метрики
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['total_execution_time'] += execution_time
                self.bot_metrics[bot_id]['actions_count'] += 1
                if execution_time > 0:
                    # Перерахунок середнього часу ухвалення рішень
                    count = self.bot_metrics[bot_id]['actions_count']
                    total_time = self.bot_metrics[bot_id]['total_execution_time']
                    self.bot_metrics[bot_id]['average_action_time'] = total_time / count

            # Виконуємо запитанну дію
            success = self._process_action(bot_id, action)

            # Фіксуємо успіх чи невдачу
            if bot_id in self.bot_metrics:
                if success:
                    self.bot_metrics[bot_id]['successful_actions'] += 1
                else:
```

```
                    self.bot_metrics[bot_id]['failed_actions'] += 1

            # Зберігаємо дію для повтору
            self._record_action(bot_id, action, success, execution_time)

            # Перевіряємо завершення гри
            if self._should_end_game():
                self.current_state = GameState.FINISHED

            return success

        except Exception as e:
            print(f"Error processing action for {bot_id}: {e}")
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['errors_count'] += 1
            return False

    def is_game_finished(self) -> bool:
        """Повертає True, якщо матч завершився."""
        return self.current_state == GameState.FINISHED

    def get_game_result(self) -> GameResult:
        """Формує фінальний протокол бою."""
        if not self.is_game_finished():
            raise RuntimeError("Game is not finished yet")

        # Переконуємось, що бій має хоч якусь історію подій
        if len(self.detailed_steps) < 2:  # Щонайменше ініціалізація + 1 дія
            raise RuntimeError("Гра завершилась без даних по діям — боти не рухались")

        # Перевіряємо, чи хтось узагалі ходив після ініціалізації
        action_steps = [step for step in self.detailed_steps if step['description'] != 'Game
initialized']
        if len(action_steps) == 0:
            raise RuntimeError("Гра завершилась без жодної дії — боти не реагували")

        # Визначаємо переможця (останній живий або з найбільшим запасом здоров'я)
        alive_bots = [bot_id for bot_id, health in self.game_state['bot_health'].items() if
health > 0]

        if len(alive_bots) == 1:
            winner = alive_bots[0]
        else:
            # Якщо живих декілька — перемагає з найбільшим HP
            winner = max(self.game_state['bot_health'].keys(),
                         key=lambda k: self.game_state['bot_health'][k])
```

```
        # Підраховуємо очки за сукупністю факторів
        scores = {}
        for bot_id in self.bots.keys():
            metrics = self.bot_metrics.get(bot_id, {})

            # Базові показники
            health_score = self.game_state['bot_health'][bot_id]
            energy_score = self.game_state['bot_energy'][bot_id] * 0.5
            survival_bonus = 50 if self.game_state['bot_health'][bot_id] > 0 else 0

            # Результативність
            damage_score = metrics.get('damage_dealt', 0) * 0.3
            efficiency_score = 0
            if metrics.get('actions_count', 0) > 0:
                success_rate = metrics.get('successful_actions', 0) /
metrics.get('actions_count', 1)
                efficiency_score = success_rate * 20

            # Швидкодія (надто швидкі боти отримують невеликий бонус)
            speed_score = 0
            avg_time = metrics.get('average_action_time', 0)
            if avg_time > 0 and avg_time < 0.1:  # Дуже швидко (<100 мс)
                speed_score = 10
            elif avg_time < 0.5:  # Швидко (<500 мс)
                speed_score = 5

            # Досліджена відстань
            distance_score = min(metrics.get('distance_traveled', 0) * 0.1, 15)

            # Підібрані бонуси
            items_score = metrics.get('items_collected', 0) * 5

            # Штраф за помилки (винятки)
            error_penalty = metrics.get('errors_count', 0) * 2

            total_score = (health_score + energy_score + survival_bonus +
                           damage_score + efficiency_score + speed_score +
                           distance_score + items_score - error_penalty)

            scores[bot_id] = max(0, total_score)  # Не допускаємо від'ємних значень

        # Довжина матчу
        duration = time.time() - self.start_time if self.start_time else 0

        return GameResult(
            winner=winner,
            scores=scores,
```

```
            game_duration=duration,
            total_turns=self.turn_number,
            replay_data=self.replay_data.copy(),
            metadata={
                'game_type': 'fighting_game',
                'max_turns': self.max_turns,
                'detailed_steps': self.detailed_steps.copy(),
                'final_health': self.game_state['bot_health'].copy(),
                'final_energy': self.game_state['bot_energy'].copy(),
                'bot_metrics': self.bot_metrics.copy()
            }
        )

    def get_game_state(self) -> GameState:
        """Повертає статус гри (WAITING/RUNNING/FINISHED)."""
        return self.current_state

    def advance_turn(self):
        """Переходить до наступного ходу."""
        self.turn_number += 1

        # Броня поступово спадає
        for bot_id in self.game_state['bot_armor']:
            if self.game_state['bot_armor'][bot_id] > 0:
                self.game_state['bot_armor'][bot_id]                =                max(0,
self.game_state['bot_armor'][bot_id] - 1)

        # Зберігаємо стан для повтору
        self._record_game_state()

    def _get_visible_state(self, bot_id: str) -> Dict[str, Any]:
        """Повертає частину арени, яку бачить конкретний бот."""
        bot_pos = self.game_state['bot_positions'][bot_id]

        # Видимість інших ботів у радіусі 5 клітин
        visible_bots = []
        for other_bot_id, other_pos in self.game_state['bot_positions'].items():
            if other_bot_id != bot_id and self.game_state['bot_health'].get(other_bot_id,
0) > 0:
                distance = self._calculate_distance(bot_pos, other_pos)
                if distance <= 5:
                    visible_bots.append({
                        'id': other_bot_id,
                        'position': other_pos,
                        'health': self.game_state['bot_health'][other_bot_id],
                        'energy': self.game_state['bot_energy'][other_bot_id],
                        'weapon': self.game_state['bot_weapons'][other_bot_id],
```

```
                    'distance': distance
                })

        # Підсилювачі видно в радіусі 3 клітин
        visible_power_ups = []
        for power_up in self.game_state['power_ups']:
            distance = self._calculate_distance(bot_pos, power_up['position'])
            if distance <= 3:
                visible_power_ups.append({
                    'position': power_up['position'],
                    'type': power_up['type'],
                    'value': power_up['value'],
                    'distance': distance
                })

        # Перешкоди помітні в радіусі 2 клітин
        visible_obstacles = []
        for obstacle in self.game_state['obstacles']:
            distance = self._calculate_distance(bot_pos, obstacle['position'])
            if distance <= 2:
                visible_obstacles.append({
                    'position': obstacle['position'],
                    'type': obstacle['type']
                })

        return {
            'my_position': bot_pos,
            'my_health': self.game_state['bot_health'][bot_id],
            'my_energy': self.game_state['bot_energy'][bot_id],
            'my_armor': self.game_state['bot_armor'][bot_id],
            'my_weapon': self.game_state['bot_weapons'][bot_id],
            'visible_bots': visible_bots,
            'visible_power_ups': visible_power_ups,
            'visible_obstacles': visible_obstacles,
            'turn_number': self.turn_number,
            'arena_size': self.game_state['arena_size']
        }

    def _get_available_actions(self) -> List[str]:
        """Перелік дозволених дій для режиму бою."""
        return ['move', 'attack', 'defend', 'use_power_up', 'special_attack', 'wait']

    def _process_action(self, bot_id: str, action: BotAction) -> bool:
        """Маршрутизує дію до відповідного хендлера."""
        action_type = action.action_type
        parameters = action.parameters
```

```
        # Мертві боти не можуть виконувати дії
        if self.game_state['bot_health'].get(bot_id, 0) <= 0:
            return False

        # Щоходу трохи відновлюємо енергію
        self.game_state['bot_energy'][bot_id] = min(100,
self.game_state['bot_energy'][bot_id] + 5)

        if action_type == 'move':
            return self._handle_move(bot_id, parameters)
        elif action_type == 'attack':
            return self._handle_attack(bot_id, parameters)
        elif action_type == 'defend':
            return self._handle_defend(bot_id, parameters)
        elif action_type == 'use_power_up':
            return self._handle_use_power_up(bot_id, parameters)
        elif action_type == 'special_attack':
            return self._handle_special_attack(bot_id, parameters)
        elif action_type == 'wait':
            return self._handle_wait(bot_id, parameters)
        else:
            return False

    def _handle_move(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Опрацьовує переміщення бота."""
        direction = parameters.get('direction', 'up')
        bot_pos = self.game_state['bot_positions'][bot_id]

        # Обчислюємо нову позицію
        new_pos = self._calculate_new_position(bot_pos, direction)

        # Перевіряємо, чи можна зайти в клітинку
        if self._is_valid_position(new_pos):
            old_pos = bot_pos.copy()
            self.game_state['bot_positions'][bot_id] = new_pos

            # Оновлюємо подолану відстань
            if bot_id in self.bot_metrics:
                distance = self._calculate_distance(old_pos, new_pos)
                self.bot_metrics[bot_id]['distance_traveled'] += distance

            self._add_detailed_step(f"{bot_id} moved {direction}", {
                'bot_id': bot_id,
                'action': 'move',
                'direction': direction,
                'old_position': old_pos,
                'new_position': new_pos,
```

```
                'energy_cost': 2
            })

            # Витрачаємо енергію
            self.game_state['bot_energy'][bot_id] = max(0,
self.game_state['bot_energy'][bot_id] - 2)
            return True

        return False

    def _handle_attack(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Опрацьовує звичайну атаку."""
        target_id = parameters.get('target')
        if not target_id or target_id not in self.bots:
            return False
        if self.game_state['bot_health'].get(bot_id, 0) <= 0:
            return False
        if self.game_state['bot_health'].get(target_id, 0) <= 0:
            return False

        attacker_pos = self.game_state['bot_positions'][bot_id]
        target_pos = self.game_state['bot_positions'][target_id]

        # Перевіряємо дистанцію до цілі
        distance = self._calculate_distance(attacker_pos, target_pos)
        weapon_range = self._get_weapon_range(self.game_state['bot_weapons'][bot_id])

        if distance <= weapon_range:
            # Визначаємо шкоду
            base_damage = self._get_weapon_damage(self.game_state['bot_weapons'][bot_id])
            target_armor = self.game_state['bot_armor'][target_id]
            actual_damage = max(1, base_damage - target_armor)

            # Застосовуємо шкоду
            self.game_state['bot_health'][target_id] = max(0,
                self.game_state['bot_health'][target_id] - actual_damage)

            # Оновлюємо статистику
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['damage_dealt'] += actual_damage
            if target_id in self.bot_metrics:
                self.bot_metrics[target_id]['damage_taken'] += actual_damage

            # Витрачаємо енергію
            energy_cost = 10
            self.game_state['bot_energy'][bot_id] = max(0,
                self.game_state['bot_energy'][bot_id] - energy_cost)
```

```
            self._add_detailed_step(f"{bot_id} attacked {target_id}", {
                'bot_id': bot_id,
                'action': 'attack',
                'target': target_id,
                'damage_dealt': actual_damage,
                'target_health_after': self.game_state['bot_health'][target_id],
                'energy_cost': energy_cost,
                'weapon': self.game_state['bot_weapons'][bot_id]
            })

            return True

        return False

    def _handle_defend(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Вмикає тимчасовий захист."""
        # Тимчасово підсилюємо броню
        self.game_state['bot_armor'][bot_id] += 5

        # Витрачаємо енергію
        energy_cost = 5
        self.game_state['bot_energy'][bot_id] = max(0,
            self.game_state['bot_energy'][bot_id] - energy_cost)

        self._add_detailed_step(f"{bot_id} defended", {
            'bot_id': bot_id,
            'action': 'defend',
            'armor_increase': 5,
            'new_armor': self.game_state['bot_armor'][bot_id],
            'energy_cost': energy_cost
        })

        return True

    def _handle_use_power_up(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Застосовує підібраний підсилювач."""
        bot_pos = self.game_state['bot_positions'][bot_id]

        # Шукаємо підсилювач у поточній клітинці
        for i, power_up in enumerate(self.game_state['power_ups']):
            if power_up['position'] == bot_pos:
                power_up_type = power_up['type']
                value = power_up['value']

                if power_up_type == 'health':
                    self.game_state['bot_health'][bot_id] = min(100,
```

```
                    self.game_state['bot_health'][bot_id] + value)
            elif power_up_type == 'energy':
                self.game_state['bot_energy'][bot_id] = min(100,
                    self.game_state['bot_energy'][bot_id] + value)
            elif power_up_type == 'weapon':
                self.game_state['bot_weapons'][bot_id] = value

            # Видаляємо використаний бонус
            self.game_state['power_ups'].pop(i)

            # Оновлюємо статистику
            if bot_id in self.bot_metrics:
                self.bot_metrics[bot_id]['items_collected'] += 1

            self._add_detailed_step(f"{bot_id} used {power_up_type} power-up", {
                'bot_id': bot_id,
                'action': 'use_power_up',
                'power_up_type': power_up_type,
                'value': value,
                'new_health': self.game_state['bot_health'][bot_id],
                'new_energy': self.game_state['bot_energy'][bot_id],
                'new_weapon': self.game_state['bot_weapons'][bot_id]
            })

            return True

    return False

def _handle_special_attack(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
    """Опрацьовує особливу (дальню) атаку."""
    if self.game_state['bot_health'].get(bot_id, 0) <= 0:
        return False
    if self.game_state['bot_energy'][bot_id] < 30:
        return False  # Недостатньо енергії

    target_id = parameters.get('target')
    if not target_id or target_id not in self.bots:
        return False
    if self.game_state['bot_health'].get(target_id, 0) <= 0:
        return False

    attacker_pos = self.game_state['bot_positions'][bot_id]
    target_pos = self.game_state['bot_positions'][target_id]

    # Перевіряємо радіус (спецатака б'є далі)
    distance = self._calculate_distance(attacker_pos, target_pos)
    if distance <= 3:
```

```
            # Обчислюємо посилену шкоду
            base_damage = 25
            target_armor = self.game_state['bot_armor'][target_id]
            actual_damage = max(5, base_damage - target_armor)

            # Застосовуємо шкоду
            self.game_state['bot_health'][target_id] = max(0,
                self.game_state['bot_health'][target_id] - actual_damage)

            # Віднімаємо енергію за прийом
            energy_cost = 30
            self.game_state['bot_energy'][bot_id] = max(0,
                self.game_state['bot_energy'][bot_id] - energy_cost)

            self._add_detailed_step(f"{bot_id} used special attack on {target_id}", {
                'bot_id': bot_id,
                'action': 'special_attack',
                'target': target_id,
                'damage_dealt': actual_damage,
                'target_health_after': self.game_state['bot_health'][target_id],
                'energy_cost': energy_cost
            })

            return True

        return False

    def _handle_wait(self, bot_id: str, parameters: Dict[str, Any]) -> bool:
        """Діє паузи: бот відновлює енергію."""
        # Відновлюємо частину енергії
        self.game_state['bot_energy'][bot_id] = min(100,
            self.game_state['bot_energy'][bot_id] + 10)

        self._add_detailed_step(f"{bot_id} waited", {
            'bot_id': bot_id,
            'action': 'wait',
            'energy_gained': 10,
            'new_energy': self.game_state['bot_energy'][bot_id]
        })

        return True

    def _generate_obstacles(self):
        """Генерує перепони на полі."""
        num_obstacles = random.randint(5, 10)
        self.game_state['obstacles'] = []
```

```
        for _ in range(num_obstacles):
            obstacle = {
                'position': {
                    'x': random.randint(0, self.game_state['arena_size'] - 1),
                    'y': random.randint(0, self.game_state['arena_size'] - 1)
                },
                'type': random.choice(['wall', 'rock', 'tree'])
            }
            self.game_state['obstacles'].append(obstacle)

    def _generate_power_ups(self):
        """Розкладає підсилювачі на полі."""
        num_power_ups = random.randint(3, 6)
        self.game_state['power_ups'] = []

        for _ in range(num_power_ups):
            power_up = {
                'position': {
                    'x': random.randint(0, self.game_state['arena_size'] - 1),
                    'y': random.randint(0, self.game_state['arena_size'] - 1)
                },
                'type': random.choice(['health', 'energy', 'weapon']),
                'value': random.choice([20, 30, 40]) if random.choice(['health', 'energy'])
else random.choice(['sword', 'bow', 'staff'])
            }
            self.game_state['power_ups'].append(power_up)

    def _calculate_distance(self, pos1: Dict[str, int], pos2: Dict[str, int]) -> float:
        """Розраховує евклідову відстань між двома позиціями."""
        dx = pos1['x'] - pos2['x']
        dy = pos1['y'] - pos2['y']
        return math.sqrt(dx * dx + dy * dy)

    def _calculate_new_position(self, pos: Dict[str, int], direction: str) -> Dict[str,
int]:
        """Повертає нову координату залежно від напрямку."""
        new_pos = pos.copy()

        if direction == 'up':
            new_pos['y'] -= 1
        elif direction == 'down':
            new_pos['y'] += 1
        elif direction == 'left':
            new_pos['x'] -= 1
        elif direction == 'right':
            new_pos['x'] += 1
```

```
        return new_pos

    def _is_valid_position(self, pos: Dict[str, int]) -> bool:
        """Перевіряє, що клітинка в межах арени й не зайнята."""
        # Контролюємо, щоб координати були в межах арени
        if (pos['x'] < 0 or pos['x'] >= self.game_state['arena_size'] or
            pos['y'] < 0 or pos['y'] >= self.game_state['arena_size']):
            return False

        # Перевіряємо перепони
        for obstacle in self.game_state['obstacles']:
            if obstacle['position'] == pos:
                return False

        # Перевіряємо, чи не зайнято іншими ботами
        for bot_pos in self.game_state['bot_positions'].values():
            if bot_pos == pos:
                return False

        return True

    def _get_weapon_range(self, weapon: str) -> int:
        """Повертає дальність дії зброї."""
        ranges = {
            'fist': 1,
            'sword': 1,
            'bow': 3,
            'staff': 2
        }
        return ranges.get(weapon, 1)

    def _get_weapon_damage(self, weapon: str) -> int:
        """Повертає базову шкоду зброї."""
        damages = {
            'fist': 10,
            'sword': 20,
            'bow': 15,
            'staff': 18
        }
        return damages.get(weapon, 10)

    def _should_end_game(self) -> bool:
        """Визначає, чи потрібно завершити матч."""
        # Завершуємо, якщо вичерпали ліміт ходів
        if self.turn_number >= self.max_turns:
            return True
```

```
        # Або якщо в живих залишився ≤1 бот
        alive_bots = [bot_id for bot_id, health in self.game_state['bot_health'].items() if
health > 0]
        if len(alive_bots) <= 1:
            return True

        return False

    def _record_game_state(self):
        """Зберігає знімок стану для повтору."""
        state = {
            'turn': self.turn_number,
            'bot_positions': self.game_state['bot_positions'].copy(),
            'bot_health': self.game_state['bot_health'].copy(),
            'bot_energy': self.game_state['bot_energy'].copy(),
            'bot_armor': self.game_state['bot_armor'].copy(),
            'bot_weapons': self.game_state['bot_weapons'].copy(),
            'power_ups': self.game_state['power_ups'].copy(),
            'obstacles': self.game_state['obstacles'].copy(),
            'bot_metrics': {k: v.copy() for k, v in self.bot_metrics.items()}
        }
        self.replay_data.append(state)

    def _record_action(self, bot_id: str, action: BotAction, success: bool, execution_time:
float = 0.0):
        """Фіксує дію бота для повтору."""
        action_record = {
            'turn': self.turn_number,
            'bot_id': bot_id,
            'action_type': action.action_type,
            'parameters': action.parameters,
            'success': success,
            'execution_time': execution_time
        }
        self.replay_data.append(action_record)

    def _add_detailed_step(self, description: str, details: Dict[str, Any]):
        """Додає людинозрозумілий запис до журналу бою."""
        step = {
            'turn': self.turn_number,
            'description': description,
            'details': details,
            'timestamp': time.time() - self.start_time if self.start_time else 0
        }
        self.detailed_steps.append(step)
```