Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти магістра

121 – Інженерія програмного забезпечення

На тему: Розробка системи моніторингу та логування RTSсервера для комп'ютерної стратегічної гри.

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ212

_____/ Яценко Р. О. /

Керівник кваліфікаційної роботи _____ / А. М. Стрюк /

Завідуючий кафедрою _____ / А. М. Стрюк /

Кривий Ріг

2025

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

_____ А. М. Стрюк

«____» _____ 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-21-2 Яценко Руслану Олександровичу

1. Тема: : Розробка системи моніторингу та логування RTSсервера для комп'ютерної стратегічної гри.
2. затверджена наказом по КНУ № 200 від 10 «квітня»2025 р.
3. Термін подання студентом закінченої роботи: 16 «червня» 2025 р.
4. Вихідні дані по роботі: Технічне завдання на розробку системи моніторингу та логування серверної частини комп’ютерної гри типу RTS, вимоги до обробки подій у реальному часі, вимоги до зберігання логів, навантаження на сервер, параметри надійності та масштабованості системи. .
5. Зміст пояснювальної записки (перелік питань, що їх треба розробити): виконано аналіз потреб моніторингу та логування RTS-серверів, обґрунтовано вибір технологій, спроєктовано архітектуру системи, реалізовано програмну частину, проведено тестування та економічне обґрунтування.
6. Перелік ілюстративного матеріалу: функціональна схема системи, блок-схема алгоритму логування, схема баз даних, діаграми класів і станів, макети веб-інтерфейсу, графіки навантаження, знімки екранних форм веб-додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Аналіз проблематики, постановка задачі, формування мети роботи	15.02.2025 – 05.03.2025
2	Аналіз архітектур і підходів до моніторингу й логування в RTS-іграх	06.03.2025 – 12.03.2025
3	Дослідження інструментів баз даних та порівняння рішень (PostgreSQL, MongoDB, InfluxDB)	13.03.2025 – 19.03.2025
4	Розробка технічних вимог і проектування системи моніторингу та логування	20.03.2025 – 27.03.2025
5	Створення UML-діаграм, схем БД, специфікацій API	28.03.2025 – 04.04.2025
6	Реалізація серверної частини (Node.js, Firebird, WebSocket)	05.04.2025 – 15.04.2025
7	Розробка клієнтської частини з візуалізацією метрик та логів (HTML, JS, Chart.js)	16.04.2025 – 25.04.2025
8	Налагодження системи, інтеграція компонентів, тестування під навантаженням	26.04.2025 – 03.05.2025
9	Підготовка матеріалів розділів роботи, оформлення ілюстративного матеріалу	04.05.2025 – 15.05.2025
10	Підготовка висновків, оформлення пояснювальної записки та подання на перевірку	16.05.2025 – 16.06.2025

Дата видачі завдання: «15» лютого 2025 р.

Студент _____ / Яценко Р. О./

Керівник роботи _____ / А. М. Стрюк/

РЕФЕРАТ

RTS, СЕРВЕР, СИСТЕМА МОНІТОРИНГУ, ЛОГУВАННЯ, WEBSOCKET, FIREBIRD, NODE.JS, POSTGRESQL, GRAFANA, MONITORING DASHBOARD.

Пояснювальна записка: 72 с., 32 рис., 2 дод., 15 джерел.

Метою кваліфікаційної роботи є розробка програмного забезпечення для моніторингу та логування серверної частини комп’ютерної стратегічної гри в реальному часі (RTS).

У роботі проаналізовано специфіку навантажень на RTS-сервери, вимоги до логування та моніторингу, існуючі інструменти (Prometheus, Zabbix, Grafana, Netdata) та особливості їх використання в ігровому середовищі. Запропоновано архітектуру системи з використанням Node.js, баз даних Firebird, PostgreSQL, MongoDB та InfluxDB для зберігання логів і метрик. Розроблено веб-інтерфейс із використанням Chart.js для візуалізації даних.

Система підтримує збір подій у реальному часі через WebSocket, фільтрацію логів, кольорове маркування подій за рівнем критичності, інформування адміністратора у разі перевантаження або збою. Проведено тестування працездатності під ігровим навантаженням і підтверджено ефективність обраної архітектури.

Основні результати роботи можуть бути впроваджені для забезпечення стабільної роботи багатокористувацьких ігрових RTS-серверів.

ABSTRACT

RTS, SERVER, MONITORING SYSTEM, LOGGING, WEBSOCKET, FIREBIRD, NODE.JS, POSTGRESQL, GRAFANA, MONITORING DASHBOARD.

Thesis: 72 pages, 32 figures, 2 appendices, 15 references.

The aim of the qualification work is to develop software for monitoring and logging the server side of a real-time strategy (RTS) computer game.

The work analyzes the specifics of RTS server load, requirements for logging and monitoring, existing tools (Prometheus, Zabbix, Grafana, Netdata), and their applicability in game environments. A system architecture was proposed based on Node.js, using Firebird, PostgreSQL, MongoDB, and InfluxDB databases to store logs and metrics. A web interface was developed using Chart.js for data visualization.

The system supports real-time event collection via WebSocket, log filtering, color-coded alerts by severity level, and automatic notifications to administrators in case of overloads or failures. The system was tested under simulated game load, confirming the effectiveness of the proposed architecture.

The main outcomes of this work can be applied to ensure the stable operation of multiplayer RTS game servers.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API (Application Programming Interface) – інтерфейс прикладного програмування, набір правил і протоколів для взаємодії між різними програмними компонентами.

CPU (Central Processing Unit) – центральний процесор; основний елемент обчислювальної системи, що виконує обчислення та обробку інструкцій.

RAM (Random Access Memory) – оперативна пам’ять; тип енергозалежної пам’яті, що використовується для тимчасового зберігання даних.

RTS (Real-Time Strategy) – стратегія в реальному часі; жанр комп’ютерних ігор, у якому події відбуваються без пауз.

UI (User Interface) – інтерфейс користувача; спосіб взаємодії користувача з програмою чи системою.

UX (User Experience) – користувацький досвід; враження, яке користувач отримує від взаємодії із системою.

DB (Database) – база даних; організована структура для зберігання інформації.

PostgreSQL – потужна об’єктно-реляційна система керування базами даних з відкритим кодом.

MongoDB – нереляційна база даних (NoSQL), яка зберігає інформацію у форматі документів JSON.

InfluxDB – база даних, орієнтована на зберігання часових рядів, часто використовується для моніторингу систем.

Grafana – інструмент для візуалізації та аналітики даних у вигляді графіків і дашбордів.

WebSocket – протокол двостороннього зв’язку між клієнтом і сервером у режимі реального часу.

Firebird – легка реляційна СКБД з відкритим кодом, оптимізована для вбудованих рішень.

SNMP (Simple Network Management Protocol) – простий протокол управління мережею, який використовується для моніторингу пристроїв.

TTL (Time To Live) – час життя; параметр, що вказує, скільки часу дані залишаються дійсними або активними.

JSON (JavaScript Object Notation) – текстовий формат обміну даними, що використовується для зберігання та передачі структурованої інформації.

HTTP (HyperText Transfer Protocol) – протокол передачі гіпертексту; використовується в Інтернеті для обміну даними між браузером і сервером.

UDP (User Datagram Protocol) – протокол транспортного рівня для передачі даних без встановлення з’єднання.

Node.js – середовище виконання JavaScript-коду на стороні сервера, що використовує неблокуючий ввід/вивід.

Express.js – мінімалістичний фреймворк для створення веб-додатків на Node.js.

ВСТУП

У обраній темі дипломної роботи,розкривається більш технологічний аспект роботи з базами даних. Спочатку були ідеї, пов’язані з розробкою гри або чимось із графікою — це виглядало цікавіше й яскравіше. Але згодом зрозумів: у більшості ігор головні проблеми виникають не в графіці, а саме на рівні серверної частини. Особливо в стратегіях реального часу (RTS), де кожна секунда може вирішити результат гри.

Мені завжди подобалися RTS-ігри, бо в них важливо не просто швидко натискати кнопки, а думати стратегічно. І коли під час гри сервер раптово починає гальмувати — це дратує не менше, ніж програш. У певний момент я подумав: а чому б не зробити так, щоб таких ситуацій було менше? Або хоча б зрозуміти, чому вони взагалі трапляються. Так і з’явилася ідея створити систему логування та моніторингу для серверної частини RTS-гри.

Так, можливо, ця тема звучить не так ефектно, як геймплей чи візуальні ефекти. Але без надійної серверної частини жодна онлайн-гра не зможе працювати стабільно. Мені хотілося зробити щось, що дійсно допомагає — щоб розробник або адміністратор могли побачити проблему ще до того, як вона призведе до збою.

Під час реалізації виникало багато труднощів. Наприклад, спроба організувати збирання логів через UDP виявилася невдалою — занадто багато пакетів губилось. Довелося перейти на WebSocket — технологію, яка мені не дуже зручна в роботі, але вона забезпечила стабільнішу роботу. Також була інша проблема: логів стало так багато, що я сам не міг у них розібратись. Тому довелося впровадити рівні важливості, фільтрацію, а для зручності — навіть кольорове підсвічування помилок.

Чи актуальна ця тема? Я переконаний, що так. Не лише тому, що онлайн-ігри зараз дуже популярні, а й тому, що коли на сервері щось іде не так — потрібен інструмент, який допоможе швидко з’ясувати причину.

Цей проєкт навчив мене багато чому. Були моменти, коли доводилося починати з нуля, переробляти рішення, шукати інші підходи. Але в результаті я зміг розібратися, як працює серверна частина RTS-гри, і зрозумів, що з нею може піти не так — і як цього уникнути.

РОЗДІЛ І. АНАЛІЗ ПРОБЛЕМИ І ПОСТАНОВКА ЗАВДАННЯ РОБОТИ

1.1 Аналіз потреб логування та моніторінгу

Індустрія комп’ютерних ігор останнім часом реально вибухнула в розвитку. Якщо ще років 10 тому ігри були просто як хобі для «геймерів у підвалі», то зараз — це серйозний бізнес і навіть кар'єра для багатьох. Особливо швидко зростає сегмент онлайн-ігор, де все тримається на якісній серверній частині. З мого боку, найбільше завжди цікавили саме стратегії в реальному часі (RTS), бо ці ігри вчать мислити логічно, працювати в команді і розивати свої анілитичні здібності.

Я сам не раз грав у подібні ігри — і як гравець, і як той, хто пробував щось розробляти. І скажу прямо: зробити RTS — це не те саме, що написати простеньку гру з ботами. Тут все тримається на сервері. По суті, сервер — це «мозок» всієї системи: він синхронізує дії гравців, зберігає стан гри, контролює все, що відбувається. Якщо щось іде не так — можна важати, що гра зламана. У людей або зникає бажання грати, або вони просто видаляють гру.

Я вже мав невдалий досвід. Наприклад, тестував свій прототип RTS-сервера — і на 20+ гравців він починав підвисати, іноді навіть повністю «вмирав». Декілька людей одразу покинули тест — просто через лаги. Тоді я зрозумів, що стабільність — це не просто побажання, це життєва необхідність для таких проєктів.

Ось кілька реальних проблем, з якими стикаються розробники RTS-серверів:

- Серйозне навантаження. Коли одночасно заходить багато гравців — сервер повинен витримати. Не раз стикався з ситуацією, коли код працює в тесті, а під навантаженням — розсипається.

- Реальний час. Будь-яка, навіть маленька, затримка — змушує гравців почати нервувати. А якщо ще й хтось програє через фріз — то скарги забезпечені.
- Надійність. Сервер має працювати стабільно, без раптових падінь. Я, наприклад, одного разу вручну витягував з резервних копій дані, бо логи не збереглись. Це було справжнє випробування нервової системи.
- Безпека. Не можна допустити, щоб хтось зламав сервер, вкрав дані чи, ще гірше, вплинув на ігровий процес. Тобто це не просто «прикрутити SSL», а ціла система перевірок і захистів.

Окремо увага приділяється моніторингу і логуванню — без цього взагалі не обійтись. Без перебільшення: це як очі й вуха для сервера. Моніторинг допомагає бачити, що відбувається тут і зараз — чи не навантажений процесор, чи вистачає пам’яті, чи все гаразд з мережею. А логування — це, так би мовити, історія всіх подій. Коли щось ламається, перш за все йдеш в логи. Не раз вони рятували, особливо коли помилка проявлялася не відразу, а після кількох годин гри.

Що цікаво — хоча є купа готових рішень типу Prometheus чи Grafana, вони не завжди підходять саме для ігрових серверів. RTS має свою специфіку: подій багато, швидкість важлива, і типові шаблони моніторингу іноді не дають потрібного контролю або просто перевантажують систему.

Тому, якщо підсумувати — створення спеціалізованої системи моніторингу та логування для RTS-серверів — це справді потрібна річ. Я сам побачив, як важко працювати «в темряві», коли немає нормального логування чи моніторингу, і вирішив: час це змінити. Саме тому обрав цю тему — бо хочу зробити щось, що реально буде корисним не тільки мені, а й іншим, хто займається розробкою подібних ігор.

1.2 Аналіз бази даних

Коли я тільки почав думати, як організувати зберігання даних для свого RTS-сервера, швидко стало ясно: просто писати все у текстові файли — не варіант. Таки варіат дісно процює, але коли гравців двоє, а подій небагато. Як тільки їх стає більше — починаються не поладки. Я сам пробував такий підхід — нажаль він дуже ускладнював ігровий процес і задоволення від нього. А ще й файли іноді злітали через неправильне завершення роботи — це призводило до втрати важливої інформації.

Так я дійшов до висновку: треба більш налагоджена база даних. Але яка саме? Щоб не гадати навмання, я для себе виписав кілька ключових вимог:

- Швидкий запис і читання. Події в RTS сипляться постійно, і зберігати їх потрібно швидко, без затримок.
- Масштабованість. Якщо гра сподобається, користувачів буде в рази більше. База має витримувати це навантаження.
- Надійність. Ні в якому разі не можна втрачати логи чи дані про гравців. Один такий інцидент — і про репутацію можна забути.
- Гнучкість. Формат подій змінюється в процесі розробки, і база має бути до цього готова — без постійних переробок схеми.

Мій досвід із різними базами

Почав я з того, що знав — реляційні БД. Встановив PostgreSQL, створив таблиці для гравців, сесій, подій. Було зручно — можна робити складні запити: скільки разів за день виникала помилка, хто з гравців найчастіше стикається з лагами. Але з часом база почала рости — і продуктивність просіла. Деякі запити стали гальмувати. Пробував оптимізовувати індекси, чистив старі записи — іноді допомагало, іноді ні.

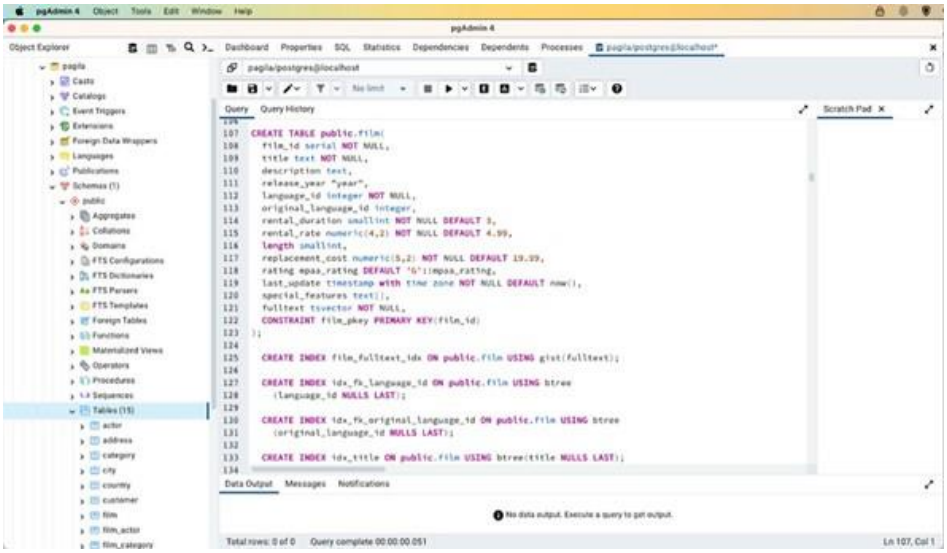

Рисунок 1.1 – Інтерфейс програми PostgreSQL

Потім узявся за MongoDB. Часто чув, що вона добре підходить для логів. І справді: зберігати події у вигляді JSON-документів — суперзручно. Не треба дуже довго працювати над схемою, додав нове поле — і все працює. Ідеально для гри, де формат подій постійно змінюється. Але коли дійшло до аналітики — почались танці з бубном. Агрегації в Mongo — це не завжди швидко і зрозуміло, особливо коли складна логіка.

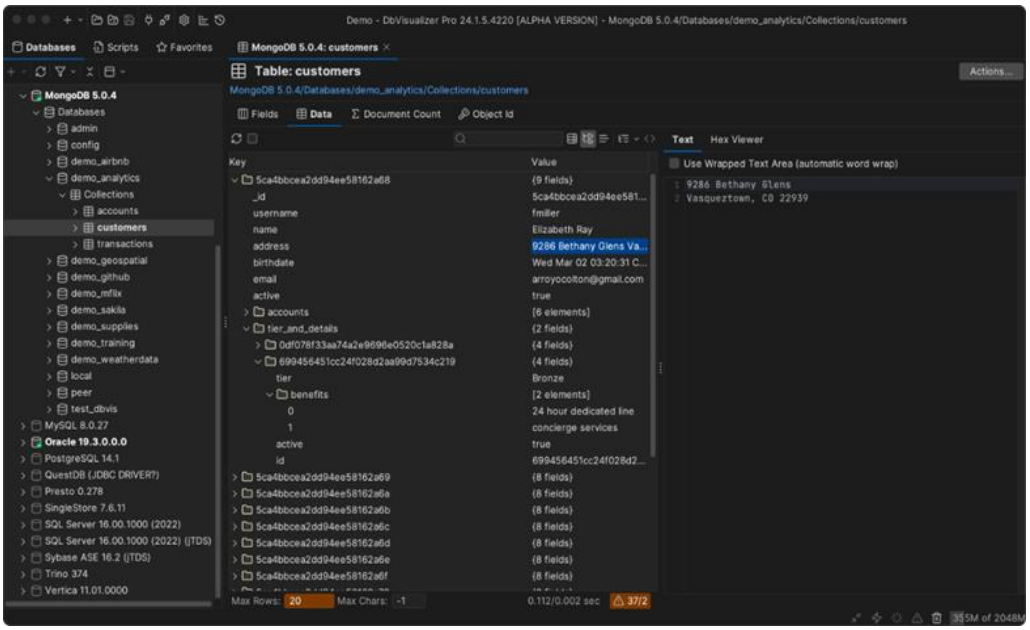

Рисунок 1.2 – Інтерфейс програми MongoDB

Пізніше, вже в процесі розробки, відкрив для себе InfluxDB. Це база, заточена під часові ряди. І тут вона прям влучила в ціль — метрики навантаження, статистика, системні показники — все це дуже зручно в неї писати. Підключив InfluxDB до сервера, почав знімати метрики по CPU, RAM, мережі — і побачив картину повністю. А з Grafana побудова графіків — дуже зручна: пара кліків, і вже бачиш, коли починається завантаження або щось іде не так.

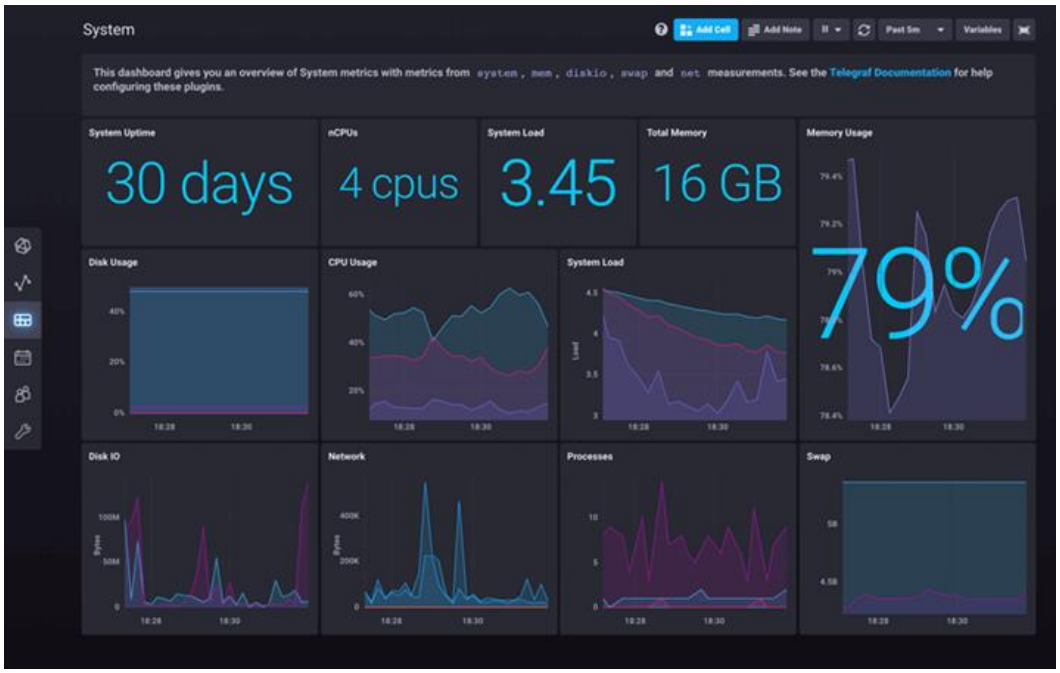

Рисунок 1.3 – Інтерфейс програми InfluxDB

Порівняння та висновки

Якщо коротко, то кожен тип БД має свої сильні й слабкі сторони:

- PostgreSQL / MySQL (реляційні): Добрі для структурованих даних і складних запитів. Але з логами — не завжди співпрацюють, особливо на великих об’ємах.
- MongoDB (NoSQL): Гнучка й зручна для логів, особливо там, де структура змінюється. Але якщо потрібно робити складну аналітику — вже важче.

- InfluxDB (часові ряди): Підходить для моніторингу й метрик. Мільйони записів — не проблема. Але для зберігання детальних логів подій — не найкращий вибір.

У підсумку я зібрав кобінацію ідельних складових:

- PostgreSQL — для гравців, сесій і основної структури гри.
- MongoDB — для логів і гнучких подій.
- InfluxDB + Grafana — для метрик і візуального моніторингу.

Так, це трохи ускладнює інфраструктуру, але на практиці — воно того варте. Сучасні інструменти мають конектори й готові бібліотеки, тому все це можна інтегрувати досить швидко.

Я спілкувався з кількома розробниками, які працюють над ігровими серверами. Усі як один казали: «Комбінуй бази, інакше будуть проблеми». Деякі навіть використовують Redis як кеш — щоб найактуальніші дані завжди були «під рукою». У літературі теж постійно повторюють одне й те саме: починай просто, але май на увазі, що доведеться масштабуватись.

Обрати базу даних для RTS-сервера — це завжди баланс: швидкість, гнучкість, надійність. Універсального рішення немає. Але якщо грамотно розділити задачі між системами — усе стає на свої місця. Саме такий підхід я і використав у своєму дипломному проєкті, і можу сказати, що це був правильний вибір.

1.3 Аналіз існуючих аналогів

Коли я тільки почав розбиратись із тим, як зробити нормальну систему моніторингу й логування для свого RTS-сервера, одразу стало ясно — без запозичення з існуючих рішень тут не обійтись. Тому я вирішив спершу подивитись, що вже є, і чи можна це якось використати або хоча б надихнутись.

Zabbix

Про Zabbix я дізнався ще на 3 курсі, але тоді навіть не уявляв, навіщо він мені. Тепер знання про нього мені знадобилися. Це потужний сервіс для

моніторингу, і він дісно вміє багато всього: і CPU, і RAM, і трафік, і сервіси, і ще купу всього. Я пробував його поставити — з першого разу нічого не зрозумів, інтерфейс трохи лякає, як для новачка. Але потім розібрався. Особливість в тому, що можна ставити тригери, і якщо щось пішло не так — тобі прилетить сповіщення. Дуже зручно.

Але от із логами в Zabbix все не так зручно. Можна, звісно, прикрутити, але це трохи збільшує об’єм роботи. Для моєї задачі — RTS-сервер із купою подій — не вистачало гнучкості. Тобто для базового стану “живий-мертвий” — нормально, але для логів і внутрішніх івентів — ні.

Рисунок 1.4 – Інтерфейс програми Zabbix

Prometheus і Grafana

Prometheus сам по собі цікавий — збирає метрики по pull-механізму, що трохи відрізняється від звичного push. Але воно працює, і працює стабільно. Особливо — PromQL. Спочатку було важко, але потім звик і вже міг витягувати складний фукціонал, як приклад “відсоток часу сервер був перевантажений”.

Що мені дійсно сподобалась в роботі — це Grafana. Можна зробити дуже зручні дашборди, все кольорове, з графіками, діаграмами і т.п. Я собі зробив

окремі панелі під CPU, RAM, online гравців і навіть кількість логів за хвилину. І це не просто цікаво і зручно, а ще й реально корисно.

Єдине «але»: Prometheus не вміє нормально зберігати логи, він же для метрик. Тому довелось паралельно використовувати щось інше для логування.

Рисунок 1.5 – Інтерфейс програми Grafana

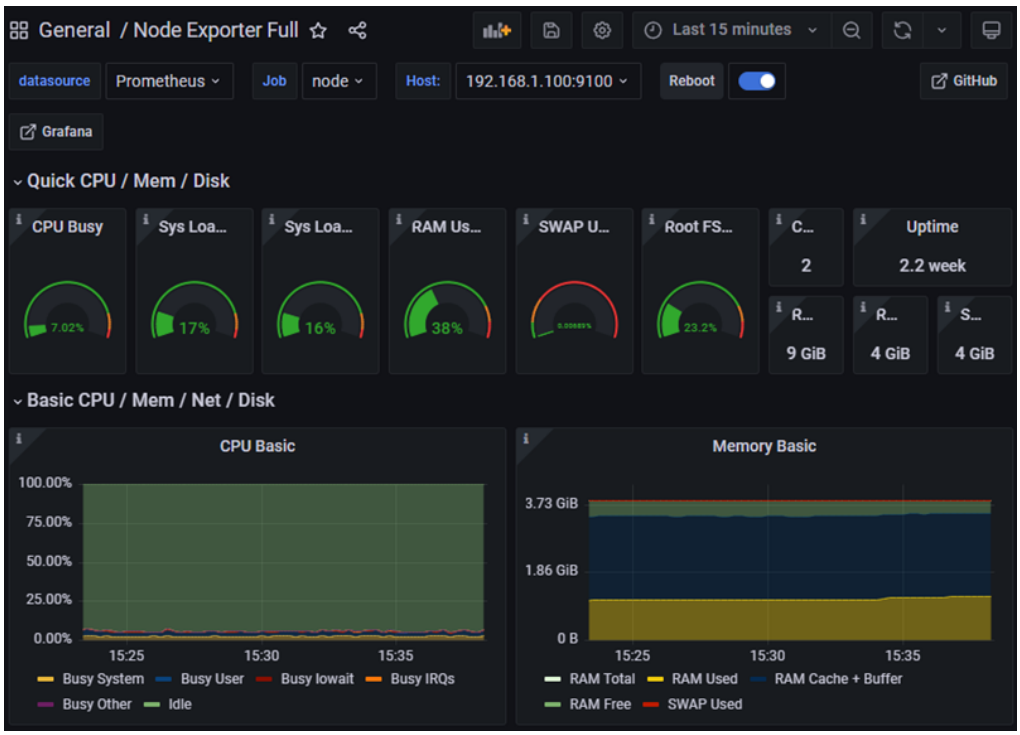

Рисунок 1.6 – Інтерфейс програми Prometheus

Nagios і Icinga

Nagios мені нагадує старі програми 2000-них. Інтерфейс — відповідний. Але зате він стабільний. Якщо треба просто пінгнути, чи живий сервер — це про нього. Налаштувань купа, але інтерфейс в нього все ж застарілий.

Icinga — це більш новіший Nagios. Там краще з інтерфейсом і масштабованістю. Але мені особисто не вистачало гнучкості з логами. Вони не дуже доречні, якщо треба обробляти багато швидких подій. Тому вирішив, що для RTS-сервера — вони не підходять.

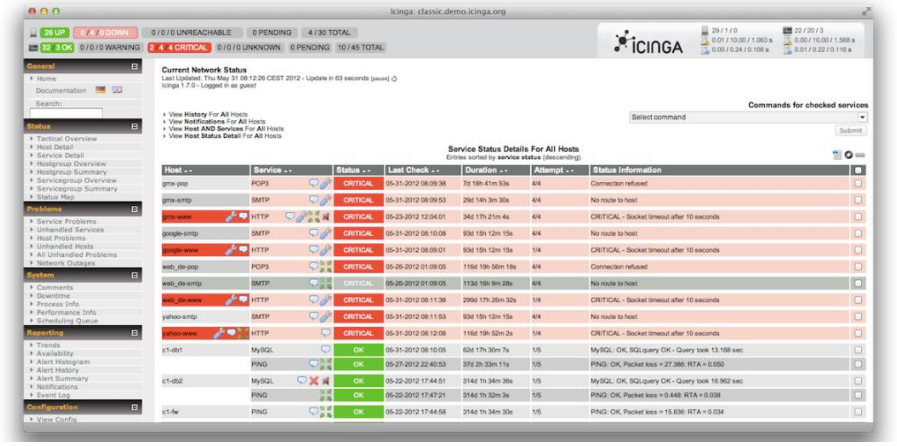

Рисунок 1.7 - Інтерфейс програми Nagios/Icinga

Рисуно1.8 – Інтерфейс програми PRTG Network Monitor

Netdata

Про Netdata дізнався випадково, коли просто гуглив “легкий моніторинг для Linux”. Встановлюється за 1 команду — і вже через хвилину бачиш купу графіків. Прям «з коробки». Дуже зручно, коли треба швидко подивитись, що твориться на сервері.

Але з масштабуванням там все трохи складно. Коли я пробував вішати Netdata на більше ніж один сервер, починалися затримками. Тому як основне рішення — не підходить, а от для діагностики локально — гарний вибір.

ManageEngine OpManager

Це вже комерційне рішення, і на нього я натрапив на форумі, де обговорювали моніторинг у великих ігрових студіях. Дуже іноваційна программа, але платна, що ускладнює роботу для починаючих розробників. Я трохи протестував дему версію — виглядає все сучасно та функціонально. Але ціни — теж серйозні. Як для студента, точно не варіант. Хоча, якщо працювати в команді з бюджетом — можна брати до уваги.

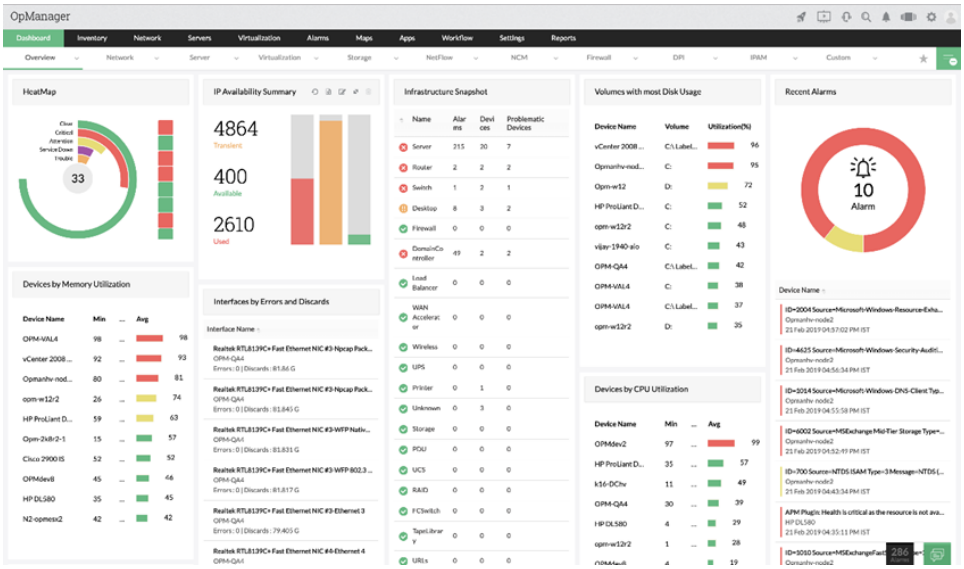

Рисунок 1.9 – Інтерфейс програми ManageEngine OpManage

1.4 Економічне обгрунтування

Коли я тільки почав займатися створенням цієї системи, взагалі не думав про гроші чи економію. Мені хотілося зробити так, щоб все працювало стабільно, без зависань і фризів. Але потім, коли пару разів сервер просто падав на рівному місці — і я по кілька годин сидів уночі, розбираючись, чого він «ляг» — почав замислюватися: скільки часу й нервів це коштує? А час, як відомо — це теж ресурс.

Було кілька ситуацій, коли без нормального логування знайти помилку було нереально. Наприклад, одного разу сервер «впав», бо оперативна пам'ять закінчилася. А я цього не побачив одразу, бо просто не було моніторингу RAM. Фізично він був, але збирав дані раз на 10 хвилин — а за цей час вже все злітало. Якби я тоді мав нормальну систему моніторингу — можна було б уникнути проблеми взагалі.

Потім я вирішив підрахувати — умовно, скільки я б «втратив», якби це був комерційний проєкт. Якщо гра — це онлайн-сервіс, то кожна хвилина, коли сервер лежить, дорівнює втрати. Гравці зляться, йдуть, пишуть гнівні відгуки, і навіть якщо потім усе виправиш — враження вже зіпсоване. Один знайомий розробник сказав мені: «Сервер не має права на паузу. А якщо щось ламається — ти маєш дізнатись про це швидше, ніж гравець».

Я прикинув, що навіть проста система моніторингу може зекономити купу часу. Наприклад, замість того, щоб 2-3 години шукати, чому сервер лагає — я можу просто подивитись графік, побачити пік навантаження, і одразу зрозуміти, де проблема. Тобто замість витрачання цінного ресурсу як час — я можу виконувати конкретні дії.

Логі — це окрема тема. Без них — як наосліп. Не раз було так, що баг проявлявся тільки після кількох годин гри. І без них —дуже важко щось зрозуміти. А з логами: відкрив, подивився — все по поличках. Навіть якщо помилка складна, все одно є шанс швидко знайти її причину.

Ще один момент — безпека. Якщо щось починає лізти не туди (наприклад, підозрілий трафік або спроби зламу) — краще дізнатись про це одразу. Бо потім, коли вже базу злили — пізно. Моніторинг і тут виручає.

З фінансової точки зору — більшість інструментів, які я використовував, безкоштовні або умовно безкоштовні. Prometheus, Grafana, навіть Netdata — все це open-source. Тобто треба тільки витратити трохи часу, щоб налаштувати. Але з досвідом налаштування вже займає годину-дві максимум. А зекономлений час на підтримці, діагностиці, пошуку проблем — реально величезний.

Я навіть у блокноті прикинув приблизну «вартість» таких речей, як затримка у виявленні проблеми, втрата користувачів, витрати на саппорт — і воно все сходиться: моніторинг і логування окупаються дуже швидко.

Отож, якщо дивитися на цю тему не просто як на «технічну аспект», а з точки зору економії — то користь очевидна. І якщо б я запускав гру в продакшн, я б починав саме з цього — із системи, яка допоможе мені спати спокійніше.

1.5 Аналіз вимог

Коли я почав більш-менш серйозно підходити до розробки своєї системи моніторингу для RTS-сервера, то зрозумів: без списку вимог — нікуди. Спочатку все здається простим: трохи графіків, трохи логів і все буде зроблено. Але надалі починаються складноші і проблеми з температурою процесора, зникшими логами і помилками які були до цього і ще багато незручностей.

Загальна робота системи

По перше — показ того, що відбувається на сервері прямо зараз. Тобто в реальному часі. І не просто загальні пкоазнки того, що сервер працює, а реально корисна інформація: навантаження на CPU, скільки пам’яті використовується, яка температура, якість з’єднання, чи не зростає пінг. І бажано щоб ці данні оновлювалися кожну секунду.

Також важливо логування. Бо без логів — як без пам’яті. Коли щось глючить, перше, куди ти дивишся — це логи. І це повинно буте не скудне сповіщення «помилка №123», а інформативне повідомлення з часом, рівнем важливості і бажано — з повним описом. Рівні логів – це дуже зручний функціонал який рятує коли іх стоновиться дуже багато.

А ще — сповіщення. Не зручно бути з відкритим моніторингомувесь час. Якщо щось ламається — треба, щоб система сама написала про це. Як система сповіщення з Telegram, або email. Наприклад: «На сервері, CPU вже 99%. Потрібно виправити». Ця функція допомогає і рятує від перегрузок.

Функціональні вимоги

- Реальний час. Моніторинг змін на сервері — без затримок, або з мінімальною (до кількох секунд).
- Логи подій. Збереження всього важливого — помилок, дій гравців, аварійних ситуацій, інфо про запуск/зупинку сервісів тощо.
- Архів логів. Щоб можна було подивитися, що було не лише зараз, а й тиждень тому.
- Фільтрація та пошук. Відображення всіх помилок за потрібний проміжок часу.
- Інтерфейс. Веб-інтерфейс, бажано простий. Для зручного використання як особистого, так і іншими користувачами.

Нефункціональні

- Надійність. Система має працювати постійно, без падінь.
- Продуктивність.
- Масштабованість. Витримка великої кіл-ті гравців у разі розвитка гри.
- Безпека. Закритий доступ який надає захищіність системі.
- Простота. Швадка налаштованність, для зручності і ефективності роботи.

Було кілька моментів, які я не врахував спочатку, а потім довелося терміново прикручувати. Наприклад, я не думав, що логів буде стільки. Буквально за день — десятки тисяч записів. Без нормальної індексації все почало гальмувати. Довелось ввести TTL (час життя логів), щоб старе автоматом чистилось.

А також адаптивність. Щоб і на мобільній версії було зручне використання функціоналу. Така уважність к дрібницям рятує і додає ефективність використання.

1.6 Встановлення системних вимог до розроблюваного програмного забезпечення.

Переходячи до етапу реалізації, навідь без чіткого плану, в мене було розуміння що має вміти система, як вона буде працювати і на чому взагалі запускатись. Спочатку хотілося просто,щоб працювало, але із збільшенням логів, графіки, кілька десятків користувачів — це вже справжній сервіс. І він має витримувати.

Основні можливості, які я заклав

По-перше — моніторинг у реальному часі. Відображення поломки одразу. Це не лише гарний інтерфейс, це важливий аспект роботи. Сервери RTS-ігор навантажуються моментально, і якщо щось глючить — гравці це відчувають ще до того, як адміністратор встигає помітити.

По-друге — логування. Записувати все, що має навідь невеличку важливість. Помилки, сповіщення, дії гравців, усе. Це допомогає зрозуміти коли зось йде не так в процессі.

По-трете — сповіщення. Вони повинні надходити як тільки, щось йде не так. Для швидкого усуненя проблеми без добового моніторінгу.

Платформа і технічні приладдя

Я вирішив, що підтримка як Windows Server, так і Linux — обов’язкова. Універсальність дуже потрібна в усіх аспектах які назараз торкаються IT сфери.

Системні характеристики які знадобляться :

- Процесор: Intel Xeon E3 або 4 ядра на 3+ ГГц.
- Оперативка: 8 ГБ
- Диск: від 100 ГБ і з RAID 1, бо були втрати даних через поганий HDD.
- Мережа: два інтерфейси по 1 Gbps — один на ігровий трафік, другий — на моніторинг. Для зручності використання.

Програмна частина

Інтерфейс я планував робити на .NET Framework (для мене це знайомий стек), дані зберігати в PostgreSQL і MongoDB, а також InfluxDB для метрик. Так це трохи складна структура, але воно того варте.

Для веб-серверу — або Apache, або Nginx.

Система повинна підтримувати SNMP, HTTP, TCP, для більш гарного результату також можно підключатись через SSH або VPN. Бо безпека понад усе.

Продуктивність

Якщо система не може обробити хоча б 10 000 подій на хвилину — вона не підходе для RTS. Навідь тестові ігри повинна витримати з великою кулькістю подій.

Інтерфейс має оновлюватись максимум раз на 5 секунд.

Також система має працювати без зупинок, цілодобово. Якщо щось падає — треба, щоб або автоматично перезапускалось, або принаймні не втрачало дані.

Надійність і захист

Окрема тема — резервування. Я передбачив RAID і можливість швидкої заміни дисків. Бо на моєму досвіді вже був випадок, коли довелось виймати жорсткий диск прямо під час гри.

Ще віддалене керування. Треба мати можливість зайти на сервер навіть якщо він на іншому кінці світу. І, що важливо — сповіщення про поломки

системи: кулер зупинився, температура вище норми, оперативна пам’ять підвисає — система має знати й передати дані.

Інтерфейс

Має бути простим і зрозумілим. Без складнощів в його розумінні. Веб-інтерфейс має бути зручним, адаптованим під телефон, працювати у Chrome/Firefox/Edge.

Додатково— кілька мов (укр/англ/нім), бо в команді бувають різні люди.

РОЗДІЛ ІІ. ПРОЕКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ ТА ЛОГУВАННЯ

2.1 Розробка проектної документації

При проектуванні системи моніторингу та логування для RTS-сервера я виділив кілька ключових компонентів, які мають працювати злагоджено. В основу лягла трирівнева архітектура, що складається з:

- Серверний збирач даних — агент, встановлений безпосередньо на ігровому сервері, який збирає метрики системи та логи.
- Центральний сервер обробки — компонент, що приймає, обробляє та зберігає всі дані.
- Веб-інтерфейс — для візуалізації, аналізу та керування всією системою.

Окремо розроблено документацію щодо інтеграційних протоколів, за допомогою яких ігровий сервер може передавати свої внутрішні події до системи моніторингу. Спроектовано API для передачі даних у двох режимах:

- Синхронний режим — для критичних подій, що вимагають негайної реакції.
- Асинхронний режим — для масових подій, де пріоритет на швидкості обробки.

Важливим аспектом стало розроблення схеми бази даних, що дозволяє ефективно зберігати різнотипні дані: від системних метрик до ігрових подій. У документації визначено стандарти форматування повідомлень логів, що включають такі поля:

- Часова мітка з мілісекундами
- Рівень важливості (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- Тип події (системна, гравець, ігрова логіка)

- Джерело повідомлення
- Унікальний ідентифікатор сесії
- Тіло повідомлення з контекстом

2.2 Діаграма варіантів використання

Діаграма варіантів використання (Use Case) відображає взаємодію користувачів з системою. У моєму проекті я виділив три основні ролі:

- Адміністратор — має повний доступ до всіх функцій системи
- Розробник — фокусується на логах ігрового процесу та помилках
- Менеджер — працює зі статистикою та загальним станом системи

Серед основних варіантів використання:

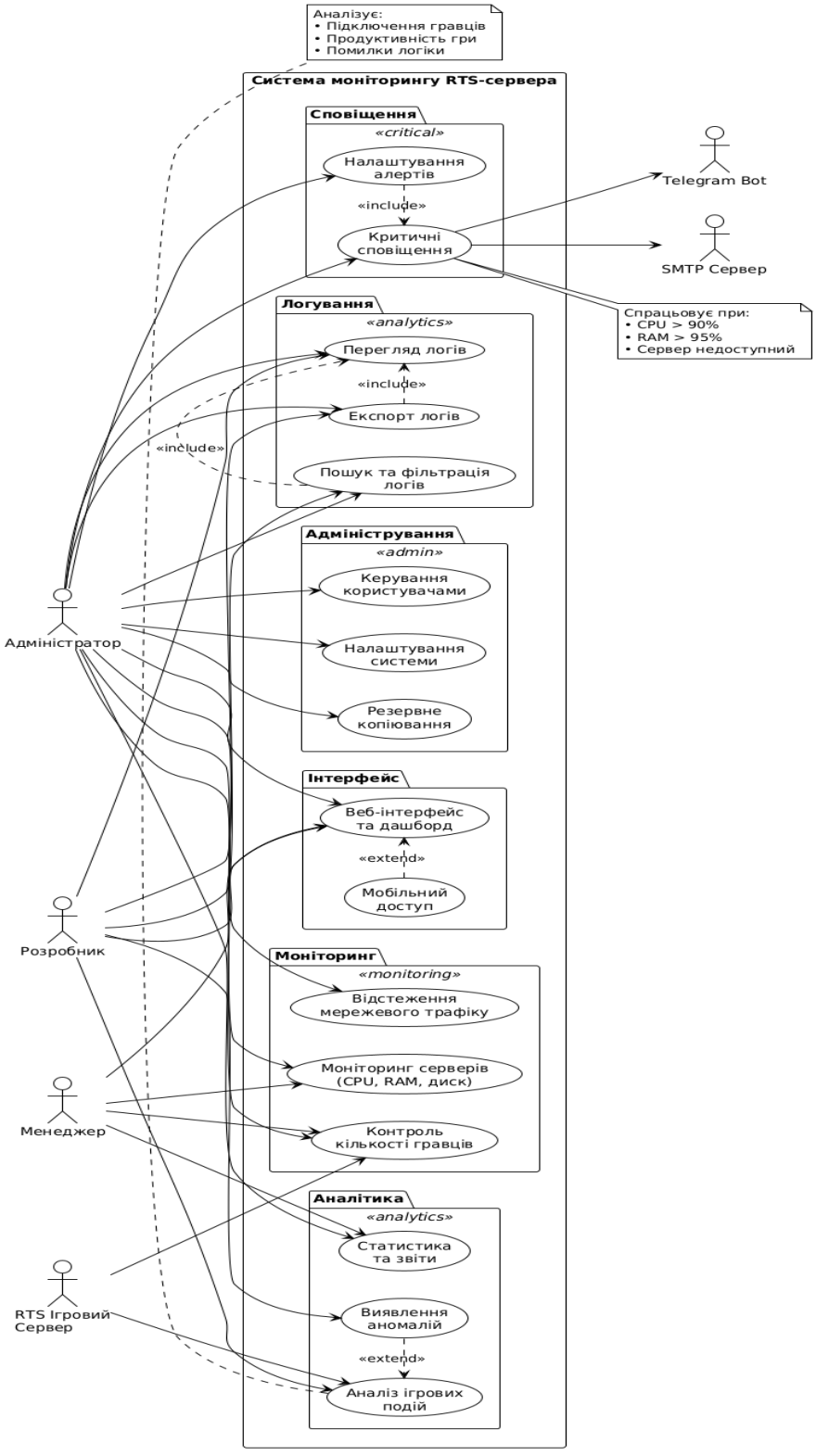

Рисунок 2.1 - Діаграма варіантів використання

1. Перегляд поточного стану серверів
    1.1. Моніторинг навантаження CPU, RAM, диску
    1.2. Відстеження мережевої активності
    1.3. Контроль кількості підключених гравців
2. Робота з логами
    2.1. Пошук за різними критеріями
    2.2. Фільтрація за рівнем важливості
    2.3. Експорт логів для подальшого аналізу
    2.4. Налаштування автоматичних повідомлень
3. Аналіз ігрових подій
    3.1. Перегляд статистики матчів
    3.2. Аналіз поведінки гравців
    3.3. Виявлення аномалій у грі
4. Налаштування системи
    4.1. Керування агентами моніторингу
    4.2. Налаштування порогових значень для сповіщень
    4.3. Керування користувачами та правами доступу

2.3 Діаграма класів сайту

Коли я дійшов до проектування архітектури своєї системи, зрозумів — просто створити декілька класів не робочий варіант. Треба продумати, працюючу схему для спільної роботи. Спочатку я намалював на папері схему того, що має бути, але швидко зрозумів: без UML-діаграми класів не обійтись. Бо коли в голові крутиться десяток компонентів, які мають між собою взаємодіяти — легко заплутатись.

Я почав з основни системи. По-перше, мені потрібен був LogManager — центральний клас, який займається всім, що пов'язано з логами. Це не просто «записати в файл», а ціла система: приймання логів від різних джерел, їх фільтрація за рівнем важливості, збереження в базу, і навіть архівування

старих записів. Спочатку треба було зробити це все в одному методі, але потім зрозумів — це буде жахливо підтримувати. Тому LogManager у мене розділений на кілька відповідальностей:

receiveLog() — приймає нові повідомлення від агентів

filterLogs() — відсіює сміття, залишає тільки важливе

storeLogs() — зберігає в базу з правильною індексацією

archiveLogs() — переносить старі логи в архів

Наступний важливий клас — MetricsCollector. Цей хлопець збирає всю системну інформацію: CPU, RAM, диск, мережу. Спочатку хотілось зробити один метод collectAll(), але потім стало зрозуміло, якщо мені треба збирати тільки метрики пам'яті, або тільки мережеві - треба розбити на окремі методи. Також додав можливість налаштувати інтервали збирання — бо не завжди потрібно опитувати систему кожну секунду.

AlertSystem — це мій захист. Він постійно моніторить все, що відбувається, і як тільки щось йде не так — одразу повідомляє про це. Але тут була цікава проблема: як зробити так, щоб система не спамила повідомленнями? Наприклад, якщо CPU завантажений на 95%, я не хочу отримувати повідомлення кожну секунду. Тому додав логіку «згладжування» — система чекає кілька вимірювань підряд, і тільки тоді відправляє алерт.

Окремо хочу розказати про DataStorage. Спочатку я думав просто використовувати одну базу для всього. Але швидко зрозумів — це не спрацює. Логи краще зберігати в MongoDB (гнучкість), метрики — в InfluxDB (швидкість для часових рядів), а конфігурацію — в PostgreSQL (надійність). Тому DataStorage у мене — це фактично адаптер, який знає, куди і що зберігати.

WebInterface — це те, що бачить користувач. Тут була змога зробити так, щоб один клас не займався всім підряд. Є окремі контролери для різних сторінок: один для дашборду, другий для логів, третій для налаштувань. Кожен контролер знає тільки про свою область відповідальності.

Ще один важливий момент — GameEventListener. Це клас, який слухає події від самого RTS-сервера. Спочатку я пробував через UDP отримувати дані, але пакети губились. Перейшов на WebSocket — і все стало набагато стабільніше. GameEventListener не просто приймає події, а ще й розуміє їх контекст: чи це подія гравця, чи системна помилка, чи щось пов'язане з ігровою логікою.

SessionTracker — це той, хто стежить за ігровими сесіями. Коли гравець заходить в гру — створюється сесія, коли виходить — закривається. А якщо зв'язок обривається несподівано — SessionTracker це теж відстежує. Дуже корисно для розуміння, скільки реально людей грає і як довго.

Щодо зв'язків між класами — я використовував кілька паттернів. Observer для системи сповіщень — дуже зручно, коли треба додати новий канал повідомлень (наприклад, Telegram бот). Strategy для різних способів збереження даних — залежно від типу інформації вибирається відповідна стратегія. Singleton для конфігурації — щоб уся система використовувала одні і ті ж налаштування.

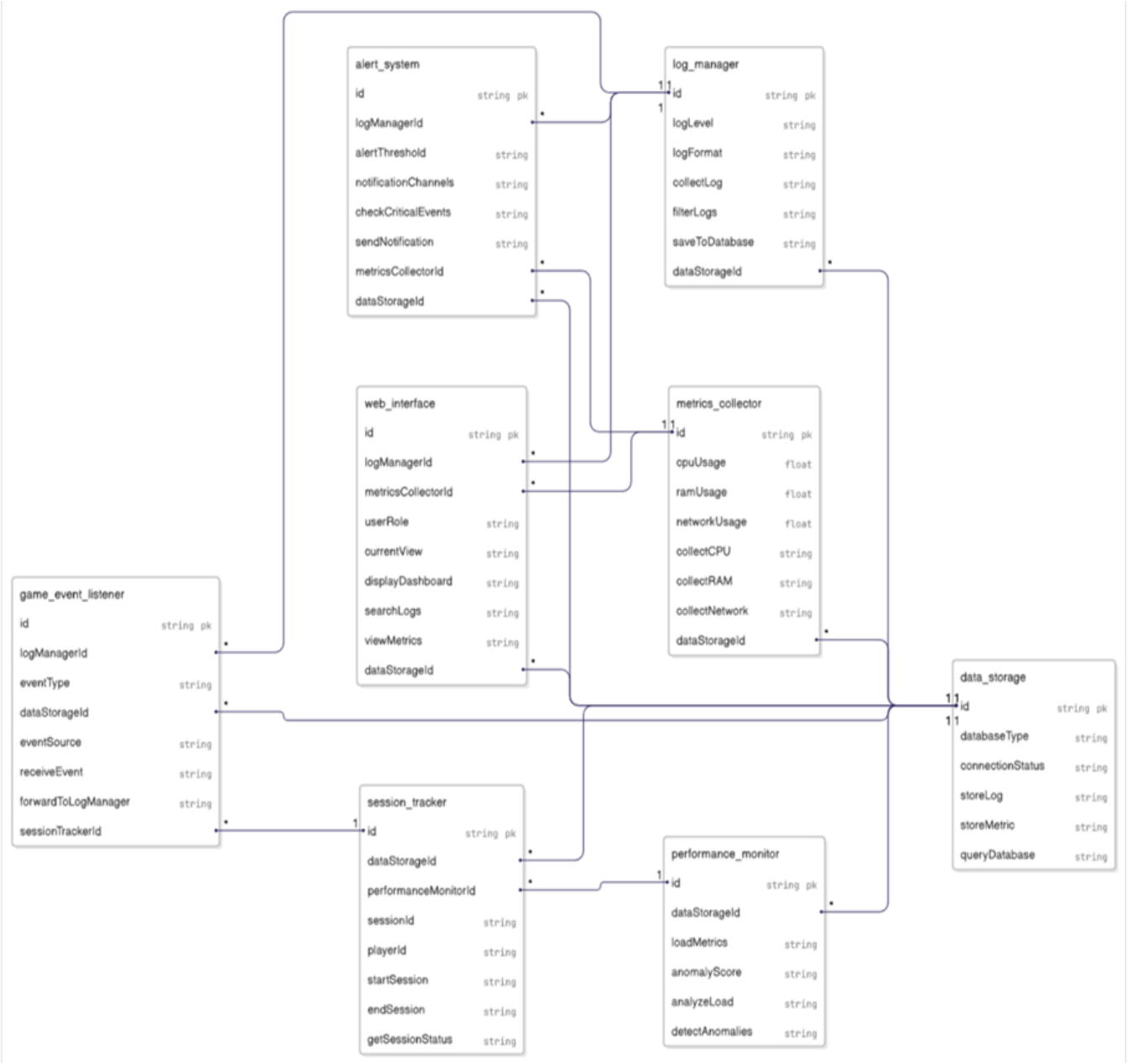

Рисунок 2.1 - Діаграма класів сайту додаток

Одна річ, яку я зрозумів в процесі — важливо не перестаратись з абстракцією. Спочатку я намагався зробити все максимально гнучко, але потім виявилось, що для моїх задач це зайве. Краще зробити просто, але щоб працювало стабільно.

Також додав інтерфейси для основних компонентів — це дуже допомагає при тестуванні. Можна легко створити mock-об'єкти і протестувати логіку, не підключаючись до реальних баз даних.

Загалом, діаграма класів вийшла не надто складна, але достатньо структурована. Кожен клас має свою чітку відповідальність, і змінити один компонент можна, не зламавши інші. Це дуже важливо, бо в процесі розробки доводилось не раз переробляти окремі частини.

2.4 Діаграма станів

Коли я почав детальніше розбиратись з тим, як має працювати вся система від початку до кінця, зрозумів — треба чітко уявляти, через які стани проходить кожне повідомлення або подія. Бо одна річ — намалювати красиві класи, і зовсім інша — зрозуміти, як вони взаємодіють у часі. Діаграма станів мені тут дуже допомогла — можна сказати, вона показала «життєвий цикл» моєї системи.

Життєвий цикл повідомлення

Починається все зі стану «Збір даних». Тут працюють мої агенти — вони постійно опитують систему, ігровий сервер, мережеві інтерфейси. Спочатку я зробив так, що всі агенти працювали з однаковою частотою — раз на секунду. Але швидко з'ясувалось, що це не дуже розумно. CPU можна опитувати частіше, а от температуру диска — досить і раз на хвилину. Тому додав можливість налаштувати інтервали для кожного типу даних.

Цікавий момент: якщо агент не може зібрати дані (наприклад, сервер не відповідає), система не зависає, а переходить у стан «Очікування повторної спроби». Через заданий інтервал спробує ще раз. Але якщо невдач підряд багато — перейде в стан “Помилка збирання” і відправить повідомлення адміністратору.

Наступний стан — «Фільтрація». Тут я зрозумів, що не всі події однаково важливі. Наприклад, якщо гравець просто рухає юніта — це не те саме, що критична помилка сервера. Тому створив систему пріоритетів:

- DEBUG — дрібниці для розробників
- INFO — загальна інформація
- WARNING — щось підозріле, але не критичне

- ERROR — помилка, яка впливає на роботу
- CRITICAL — все, система може впасти

Події рівня CRITICAL минають всі черги і одразу йдуть до обробки. А DEBUG-повідомлення можуть навіть відкидатись, якщо система перевантажена.

Стан «Обробка» — це найскладніший етап. Тут система не просто зберігає дані, а намагається їх зрозуміти. Наприклад, якщо CPU навантаження раптом підскочило до 90% — це може бути нормально (почався інтенсивний бій з багатьма гравцями) або проблема (зависла якась задача). Система дивиться на контекст: скільки гравців онлайн, чи були схожі ситуації раніше, чи збільшилось споживання пам'яті.

Дуже довго я працював над тим, щоб система розпізнавала аномалії. Спочатку просто ставив жорсткі порограмми: CPU > 80% = проблема. Але це давало багато хибних спрацьовувань. Тому додав машинне навчання — система аналізує історичні дані і вчиться розуміти, що нормально для конкретного сервера в конкретний час.

Критичні події

Також про стан «Обробка критичних подій». Це найвищий пріоритет, і коли система потрапляє в цей стан — все інше відходить на другий план. Наприклад, якщо сервер почав не відповідати на пінги, або пам'ять закінчилась — система одразу переходить до цього стану, минаючи звичайну обробку.

Тут працює така логіка: спочатку система намагається автоматично виправити проблему (перезапустити процес, очистити кеш), і тільки якщо це не допомагає — відправляє сповіщення людям. Це дозволяє уникнути ситуацій, коли адміністратор отримує повідомлення про проблему, яка вже вирішилась сама.

Зберігання та відображення

Стан «Зберігання» теж виявився не таким простим, як здавалось. Спочатку я просто писав все в одну базу, але швидко зрозумів — це неефективно. Логи краще зберігати окремо від метрик, а конфігурацію —

взагалі в іншому місці. Тому зараз система визначає тип даних і вибирає відповідне сховище.

Ще важливий момент — «Архівування». Логи накопичуються дуже швидко, і якщо їх не чистити — база швидко розпухне. Тому система автоматично переносить старі дані в архів і стискає їх. Налаштував так, що записи старше місяця автоматично архівуються, а старше року — видаляються зовсім (якщо це не критичні помилки).

Масштабування

Цікавий стан — «Масштабування». Коли навантаження на систему зростає, вона може автоматично запускати додаткові процеси збирання або обробки даних. Наприклад, якщо черга повідомлень стає занадто довгою — система запускає ще один обробник. А коли навантаження спадає — зайві процеси завершує.

Спочатку я не передбачав цього стану, але коли почав тестувати систему під навантаженням — швидко зрозумів, що без автоматичного масштабування не обійтись.

Взаємодія станів

Найскладніше було налаштувати переходи між станами. Наприклад, що робити, якщо система знаходиться в стані «Зберігання», а тут приходить критична подія? Вона має перервати зберігання і перейти до обробки критики, або дочекатись завершення?

Я вирішив так: критичні події завжди мають пріоритет. Але поточні операції не перериваються жорстко — система намагається завершити їх якомога швидше і тільки потім переключається.

Але є і нюанс — стан «Відновлення після помилки». Якщо щось пішло не так (наприклад, відключилась база даних), система не панікує, а переходить в режим збереження даних локально. Коли з'єднання відновиться — всі накопичені дані будуть синхронізовані.

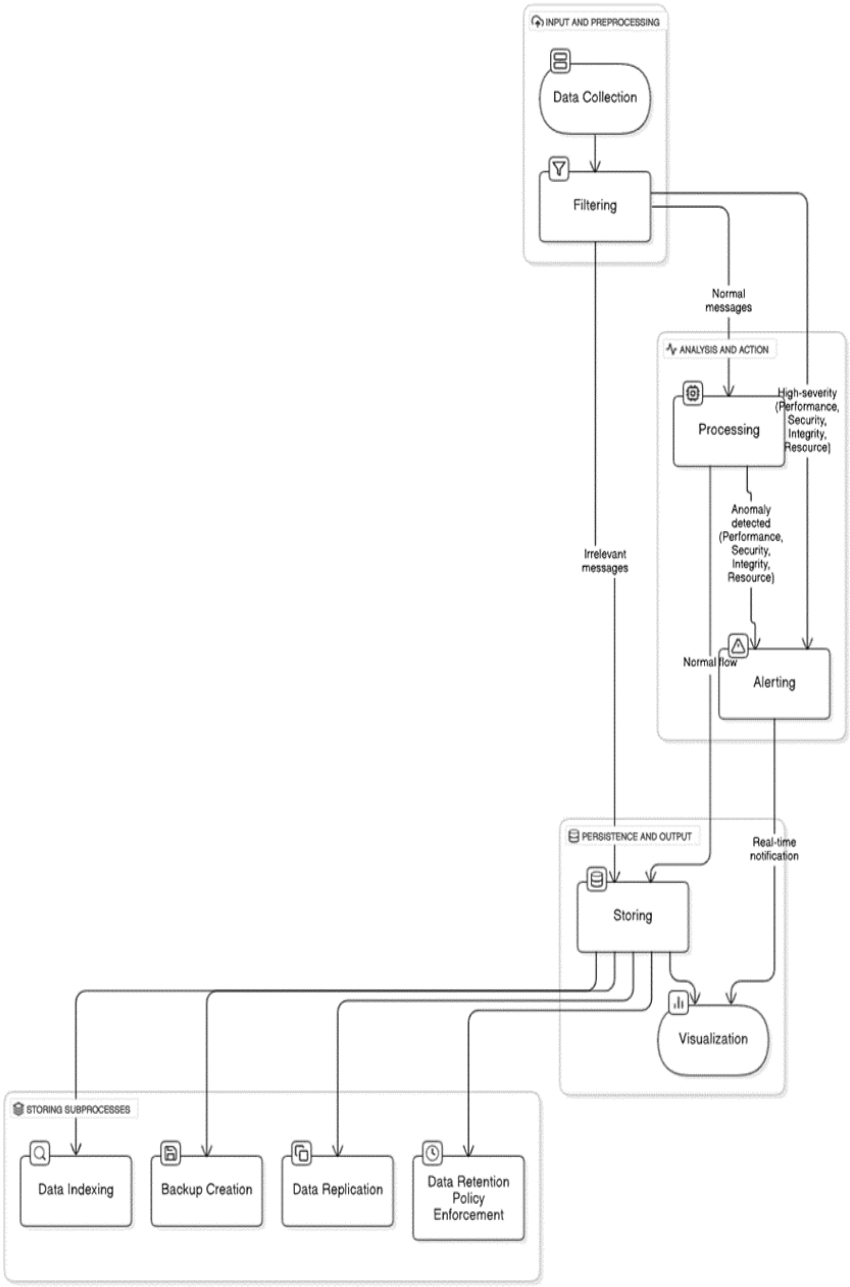

Рисунок 2.3 - Діаграма станів сайту програми

Загалом, діаграма станів допомогла мені зрозуміти, що система моніторингу — це не просто «збирай і показуй», а складний організм з купою

станів і переходів між ними. І найважливіше — передбачити всі можливі ситуації, особливо аварійні.

2.5 Розробка моделі інтерфейсу сайту

При розробці інтерфейсу системи моніторингу я керувався принципами User-Centered Design, фокусуючись на зручності використання та інформативності. Основою став адаптивний веб-інтерфейс, який добре виглядає як на великих моніторах адміністраторів, так і на мобільних пристроях.

Структура інтерфейсу:

- Головна панель (Dashboard) — загальний стан системи з ключовими показниками
- Моніторинг серверів — детальна інформація про кожен сервер
- Система логування — пошук та фільтрація логів
- Аналітика — графіки та діаграми для аналізу трендів

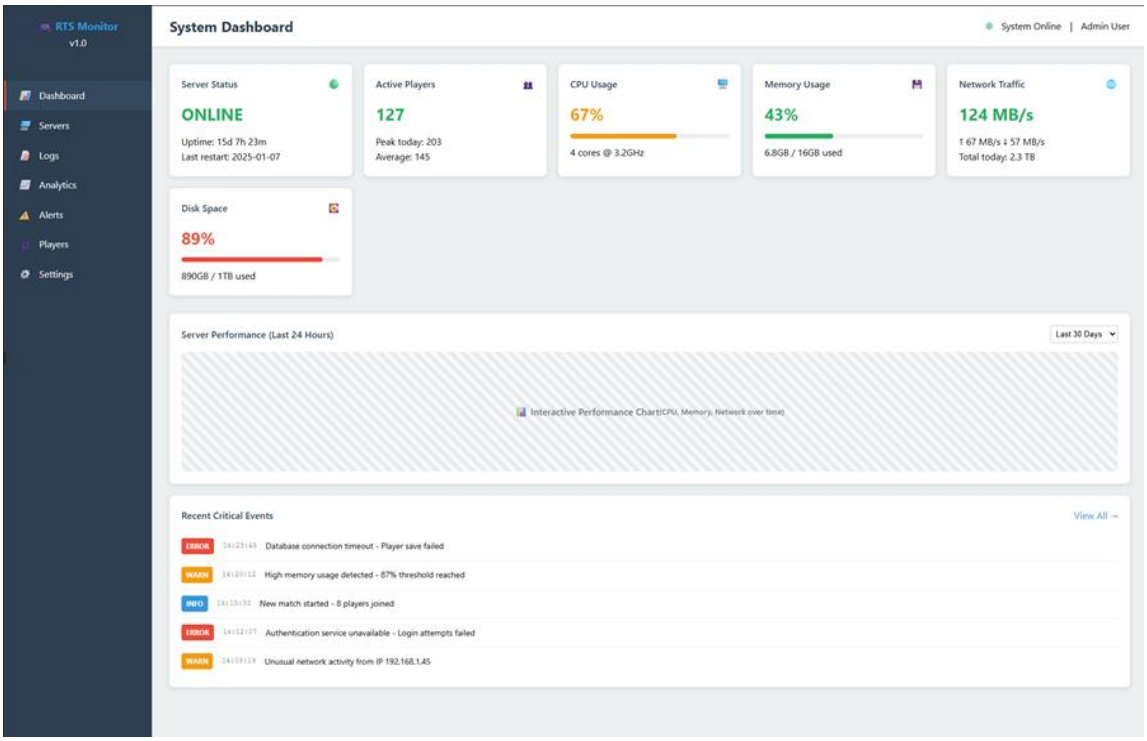

Рисунок 2.4 - Вигляд сайту

- Налаштування — конфігурація системи, керування користувачами

Особливу увагу приділено інформаційним панелям (віджетам), які користувач може налаштувати під свої потреби. Це дозволяє сфокусуватися на найважливіших метриках і швидко реагувати на проблеми.

Для підвищення ергономіки інтерфейсу розроблено систему кольорового кодування:

- Зелений — нормальний стан
- Жовтий — стан потребує уваги
- Помаранчевий — передкритичний стан
- Червоний — критична ситуація

Важливим елементом інтерфейсу стала система фільтрів для логів, що дозволяє швидко звузити область пошуку за різними параметрами: часовий проміжок, тип події, рівень важливості, ідентифікатор сесії тощо.

Для розробників ігрової логіки додано спеціальний режим, який фокусується на ігрових подіях та їх впливі на продуктивність сервера. Це дозволяє швидко виявляти та виправляти вузькі місця в коді гри.

Вибір технології для інтерфейсу:

Після аналізу різних фреймворків для веб-розробки я зупинився на HTML з використанням бібліотеки Material-UI для компонентів. Це забезпечує:

- Швидку розробку і модифікацію інтерфейсу
- Високу продуктивність завдяки віртуальному DOM
- Гарну підтримку для створення динамічних панелей даних
- Можливість гнучкого налаштування відображення під різні ролі користувачів

Для візуалізації даних обрано бібліотеку Chart.js, яка дозволяє створювати інтерактивні графіки та діаграми, що оновлюються в реальному часі.

Розроблений інтерфейс не лише відображає поточний стан системи, але й дозволяє прогнозувати потенційні проблеми на основі аналізу трендів, що особливо важливо для проактивного управління RTS-сервером.

РОЗДІЛ ІІІ. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Обгрунтування інструментів розробника

Коли я почав думати про те, якими інструментами реалізовувати свою систему моніторингу, зрозумів — вибір технологій тут критично важливий. Бо одна річ — красиво намалювати діаграми, і зовсім інша — зробити так, щоб воно дійсно працювало під навантаженням.

Вибір серверної платформи

Спочатку на розглядання були різні варіанти: PHP, Python, Java, C#. Але швидко зрозумів — для RTS-сервера потрібно щось, що може обробляти тисячі одночасних подій без блокувань. Ідеальним рішенням виявився Node.js. Асинхронна модель роботи означає, що коли один запит чекає відповіді від бази даних, сервер може обробляти інші запити. Це особливо важливо, коли система має обробляти логи від багатьох гравців одночасно.

Ще один плюс Node.js — це JavaScript і на сервері, і на клієнті. Менше переключення контексту, можна використовувати ті самі бібліотеки, та й команді розробників простіше підтримувати код.

База даних: чому Firebird 2.1

Тут була довга дискусія. Багато хто пропонував PostgreSQL або MySQL — мовляв, вони популярніші. Але для моїх завдань Firebird виявився оптимальним рішенням:

- По-перше, швидкість. Firebird дуже швидко обробляє SQL-запити, особливо коли справа доходить до вибірки логів за часовими діапазонами.
- По-друге, надійність. Firebird рідко падає, а якщо щось трапляється — швидко відновлюється.
- По-третє, розмір. Вся база займає мало місця на диску, що важливо для серверів з обмеженими ресурсами.

Також були добрацьовані деякі деталі — Firebird вимогає, правильне налаштування. Після цього система почала працювати як годинник.

WebSocket vs HTTP:

Спочатку для використовування планувався звичайні HTTP-запити для передачі логів. Але швидко сатло зрозуміло — це не буде працювати. Приклад: сервер генерує 100 логів на секунду, а система має опитувати HTTP-ендпоінт кожну секунду. Це величезне навантаження на мережу і сервер.

WebSocket вирішив цю проблему. Встановлюється одне з'єднання, і дані течуть в обох напрямках у реальному часі. Логи надходять миттєво, а клієнт може відправляти команди назад на сервер. Ідеально для моніторингу в реальному часі.

Chart.js для візуалізації

Для роботи на вибір було декілька варіантів: D3.js, Google Charts, ApexCharts. Але Chart.js став найкращім варіантом через простоту використання. Мені не потрібні складні 3D-графіки або інтерактивні карти — потрібні прості, зрозумілі діаграми, які швидко оновлюються.

Chart.js дозволяє легко створювати лінійні графіки для метрик CPU та пам'яті, кільцеві діаграми для розподілу типів логів, і все це оновлюється в реальному часі через WebSocket.

Чому не були використані готові рішення

Можно було б попрацювати з Grafana + Prometheus, але справа в тому, що готові рішення — це завжди компроміс. Вони можуть все, але не завжди добре. А для гарної роботи потрібна була система, яка ідеально підходить саме для RTS-серверів, розуміє специфіку ігрових подій і може інтегруватися з ігровою логікою.

До того ж, розробка власного рішення дала повний контроль над функціоналом і можливість додавати нові функціональні можливості, не обмежені сторонніми фреймворками.

3.2 Структура програми

Із початком розробки архітектури системи, одразу стало зрозуміло — без чіткої структури все перетвориться на безструктурований код. Тому було прийнято рішення побудувати модульну систему, де кожен компонент має свою чітко визначену роль і мінімальні залежності від інших.

Серверна частина (Node.js)

Серверна частина — це мозок всієї системи. Вона побудована навколо Express.js і розділена на логічні модулі:

Основний сервер (server.js)

```
const express = require('express');
const Firebird = require('node-firebird');
const cors = require('cors');
const fs = require('fs');
const path = require('path');

const app = express();
const port = 3000;

app.use(cors());
app.use(express.static('public'));

// Конфігурація Firebird
const dbOptions = {
  host: 'localhost',
  port: 3050,
  database: 'C:/Users/Ruslam/OneDrive/Desktop/diplom/OpenHV-main-
site/database.fdb',
  user: 'SYSDBA',
  password: 'masterkey',
  lowercase_keys: true,
  role: null,
  pageSize: 4096,
};
```

Тут налаштування основного серверу і підключення до бази даних. Firebird вимагає точної конфігурації — особливо важливі параметри pageSize і lowercase_keys, які впливають на продуктивність.

Менеджер бази даних (dbManager.js)

```
function queryDatabase(sql, params = []) {
  return new Promise((resolve, reject) => {
    Firebird.attach(dbOptions, (err, db) => {
      if (err) return reject(err);
      db.query(sql, params, (err, result) => {
```

```
        db.detach();
        if (err) return reject(err);
        resolve(result);
      });
    });
  });
}
```

Це основна функція для роботи з базою даних. Яка обернута в Promise, щоб зручніше працювати з async/await. Кожне з'єднання з базою відкривається і закривається для кожного запиту — це може здатися неефективним, але для Firebird це рекомендований підхід.

API ендпоінти для моніторингу

```
app.get('/api/monitoring', async (req, res) => {
  try {
    const results = await queryDatabase('SELECT * FROM players');
    res.json(results);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});

app.get('/api/online', async (req, res) => {
  try {
    const results = await queryDatabase(“SELECT COUNT(*) AS online_count
FROM players WHERE status = 'online'“);
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});
```

Кожен ендпоінт обробляє помилки і повертає структуровані JSON-відповіді. Це дозволяє клієнтській частині правильно реагувати на проблеми.

Збирач метрик (metricsCollector.js)

```
app.get('/api/cpu_chart', async (req, res) => {
  try {
    const results = await queryDatabase(`
      SELECT log_time, usage_percent FROM CPU_USAGE
      ORDER BY log_time
    `);
    const labels = results.map(r => new Date(r.log_time).toLocaleTimeString());
    const data = results.map(r => r.usage_percent);
    res.json({ labels, data });
  } catch (err) {
    console.error('CPU Chart Error:', err);
```

```
    res.status(500).json({ error: 'DB Error' });
  }
});
```

Цей процесс - підготовка даних для графіків. Я перетворюю час у зручний формат для Chart.js і групую дані так, щоб вони були готові для візуалізації.

Обробник логів (logReceiver.js)

javascript

```
const logPath = 'C:/Users/Ruslam/OneDrive/Desktop/diplom/OpenHV-main/OpenRA_Server_Log.txt';
let lastSize = 0;

setInterval(() => {
  fs.stat(logPath, (err, stats) => {
    if (err) {
      console.error('Не вдалося отримати лог-файл:', err);
      return;
    }

    if (stats.size > lastSize) {
      const stream = fs.createReadStream(logPath, {
        start: lastSize,
        end: stats.size
      });

      let buffer = '';

      stream.on('data', chunk => {
        buffer += chunk.toString();
      });

      stream.on('end', async () => {
        const lines = buffer.trim().split('\n');

        for (const line of lines) {
          const isConnect = line.includes('[Skirmish Game]');
          const isDisconnect = line.includes('[Disconnecte]');
          const match = line.match(/\[(?:Skirmish Game|Disconnecte)\] (\w+)/);

          if (!match) continue;

          const nickname = match[1];
          const newStatus = isConnect ? 'online' : isDisconnect ? 'offline' : null;

          if (nickname && newStatus) {
            await updatePlayerStatus(nickname, newStatus);
          }
        }
```

```
      });

      lastSize = stats.size;
    }
  });
}, 1000);
```

Це серце системи моніторингу в реальному часі. Кожну секунду я перевіряю, чи з'явилися нові записи в лог-файлі. Якщо так — читаю тільки нову частину файлу і обробляю її. Це дуже ефективно навіть для великих логів.

Функція оновлення статусу гравців

```
async function updatePlayerStatus(nickname, status) {
  const now = new Date();

  try {
    const existing = await queryDatabase('SELECT id FROM players WHERE
nickname = ?', [nickname]);

    if (existing.length === 0) {
      await queryDatabase(
        'INSERT INTO players (nickname, status, last_seen) VALUES (?, ?, ?)',
        [nickname, status, now]
      );
      console.log(`Додано нового гравця: ${nickname} → ${status}`);
    } else {
      await queryDatabase(
        'UPDATE players SET status = ?, last_seen = ? WHERE nickname = ?',
        [status, now, nickname]
      );
      console.log(`Статус оновлено: ${nickname} → ${status}`);
    }
  } catch (err) {
    console.error('Помилка оновлення:', err);
  }
}
```

Тут реалізована логіка «upsert» — якщо гравця немає в базі, додаю його, якщо є — оновлюю статус. Це гарантує, що база завжди містить актуальну інформацію.

Клієнтська частина (HTML/CSS/JavaScript)

Фронтенд побудований як SPA з чистим JavaScript. Ніяких фреймворків — тільки нативні API браузера і Chart.js для графіків.

Основна структура (index.html)

```
<!DOCTYPE html>
<html lang="uk">
<head>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <link rel="stylesheet" href="styles.css">
    <script src="Collector.js"></script>
    <script src="app.js"></script>
    <title>RTS Server Monitoring - Система моніторингу сервера</title>
  </head>
  <body>
    <div id="connectionStatus" class="connection-status connection-offline">
      Підключення втрачено
    </div>

    <div class="container">
      <div class="header">
        <h1>RTS Server Monitoring System</h1>
        <p>Система моніторингу та логування для RTS-сервера | Розроблено
згідно з дипломною роботою</p>
      </div>

      <div class="nav-tabs">
        <button class="nav-tab active"
onclick="showTab('dashboard')">Dashboard</button>
        <button class="nav-tab" onclick="showTab('monitoring')">Моніторинг
серверів</button>
        <button class="nav-tab" onclick="showTab('logs')">Система
логування</button>
        <button class="nav-tab" onclick="showTab('console')">Консоль</button>
        <button class="nav-tab"
onclick="showTab('analytics')">Аналітика</button>
        <button class="nav-tab"
onclick="showTab('settings')">Налаштування</button>
      </div>
```

Я вирішив використати табову навігацію — це зручно для користувачів і дозволяє логічно розділити функціонал. Кожна вкладка має свою роль: Dashboard для швидкого огляду, Моніторинг для детального аналізу, Логи для перегляду подій.

Віджети Dashboard

```
<div class="widget">
  <div class="widget-title">
    <div id="serverStatusIndicator" class="status-indicator status-offline"></div>
    Стан сервера
  </div>
  <div id="serverStatus" class="metric-value">OFFLINE</div>
  <div id="serverUptime" class="metric-label">Перевірка підключення...</div>
</div>
```

```
<div class="widget" id="ram-widget">
  <div class="widget-title">
    <div class="status-indicator status-normal"></div>
    Використання RAM
  </div>
  <div class="metric-value" id="ram-used">Завантаження...</div>
  <div class="metric-label" id="ram-label"></div>
  <div class="progress-bar">
    <div class="progress-fill progress-normal" id="ram-progress" style="width:
0%"></div>
  </div>
</div>
```

Кожен віджет має індикатор стану (зелений/жовтий/червоний), основне значення і додаткову інформацію. Прогрес-бари показують відсоткове використання ресурсів.

Система фільтрації логів

```
<div class="log-filters">
  <div class="filter-group">
    <label>Часовий проміжок</label>
    <select class="filter-select" onchange="filterLogs()">
      <option value="1h">Остання година</option>
      <option value="24h" selected>Останні 24 години</option>
      <option value="7d">Останній тиждень</option>
      <option value="30d">Останній місяць</option>
    </select>
  </div>
  <div class="filter-group">
    <label>Рівень важливості</label>
    <select class="filter-select" onchange="filterLogs()">
      <option value="all" selected>Всі рівні</option>
      <option value="debug">DEBUG</option>
      <option value="info">INFO</option>
      <option value="warning">WARNING</option>
      <option value="error">ERROR</option>
      <option value="critical">CRITICAL</option>
    </select>
  </div>
</div>
```

Фільтри дозволяють швидко знаходити потрібні логи. Це особливо важливо, коли в системі тисячі записів на день.

Структура бази даних

База даних побудована з урахуванням специфіки RTS-серверів. Основні таблиці:

Таблиця гравців (players)

- id (INTEGER, PRIMARY KEY)
- nickname (VARCHAR(50), UNIQUE)
- status (VARCHAR(20)) -- 'online', 'offline', 'away'
- last_seen (TIMESTAMP)
- total_playtime (INTEGER) -- в секундах

Таблиця логів (logs)

- id (INTEGER, PRIMARY KEY)
- event_time (TIMESTAMP)
- log_level (VARCHAR(20)) -- 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'
- event_type (VARCHAR(50)) -- 'system', 'game', 'player', 'network'
- message (BLOB SUB_TYPE TEXT)
- player_id (INTEGER, FOREIGN KEY)

Таблиці метрик

- CPU_USAGE (log_time, usage_percent)
- MEMORY_USAGE (log_time, used_mb, total_mb)
- ACTIVE_PLAYERS (log_time, player_count)
- NETWORK_TRAFFIC (log_time, incoming_mb, outgoing_mb)

Таблиця серверів (server)

- id (INTEGER, PRIMARY KEY)
- server_name (VARCHAR(100))
- status (VARCHAR(20))
- cpu_usage (FLOAT)
- ram_usage_mb (INTEGER)

- temperature_c (INTEGER)
- avg_ping_ms (INTEGER)
- last_update (TIMESTAMP)

Принципи організації коду

- Розділення відповідальності. Кожен модуль має одну чітко визначену функцію. Сервер не займається обробкою UI, клієнт не працює напряму з базою даних, збирач метрик не відповідає за відправку сповіщень.
- Обробка помилок. У кожному модулі є try-catch блоки і логування помилок. Це дозволяє швидко знаходити і виправляти проблеми.
- Масштабованість. Архітектура дозволяє легко додавати нові типи метрик, нові джерела логів, нові способи візуалізації без зміни основного коду.
- Конфігурація. Всі налаштування винесені в окремі файли або змінні оточення. Це спрощує розгортання на різних серверах.

Така структура виявилася дуже зручною в розробці можна було працювати над різними частинами системи незалежно, а потім легко інтегрувати їх разом.

3.3 Графічний інтерфейс програми

При починанні проектування інтерфейса системи моніторингу, одразу зрозумів — це не просто набір кнопок і графіків. Це інструмент, яким адміністратори будуть користуватися щодня, особливо в критичних ситуаціях, коли кожна секунда на рахунку. Тому інтерфейс має бути не тільки функціональним, а й інтуїтивно зрозумілим.

Принципи дизайну інтерфейсу

З самого початку вирішив дотримуватися кількох ключових принципів:

Перший принцип — «Важлива інформація завжди на виду». Статус сервера, критичні помилки, кількість онлайн-гравців — все це має бути видно одразу після завантаження сторінки. Ніхто не буде шукати по вкладках, коли сервер падає.

Другий принцип — «Мінімум кліків до результату». Якщо адміністратор хоче подивитися логи за останню годину з помилками — це має займати максимум 2-3 кліки. Не більше.

Третій принцип — «Зрозуміло навіть новачку». Навіть якщо хтось вперше відкрив систему, він має розуміти, що означають кольори, графіки та індикатори.

Рисунок 3.1 - Загальний вигляд головної сторінки системи

Весь інтерфейс побудований навколо табової навігації. Це рішення виявилося дуже вдалим — кожна вкладка відповідає конкретному завданню, і користувач завжди розуміє, де він знаходиться.

```
<div class="nav-tabs">
  <button class="nav-tab active"
onclick="showTab('dashboard')">Dashboard</button>
  <button class="nav-tab" onclick="showTab('monitoring')">Моніторинг
серверів</button>
  <button class="nav-tab" onclick="showTab('logs')">Система
логування</button>
  <button class="nav-tab" onclick="showTab('console')">Консоль</button>
  <button class="nav-tab" onclick="showTab('analytics')">Аналітика</button>
  <button class="nav-tab" onclick="showTab('settings')">Налаштування</button>
</div>
```

Кожна вкладка виділяється активним стилем, також був додані плавні переходи між вкладками. Це створює відчуття цілісності системи.

Dashboard — серце системи

Dashboard — це те, що бачить користувач першим. Тут зібрана вся критично важлива інформація в зручному для сприйняття вигляді.

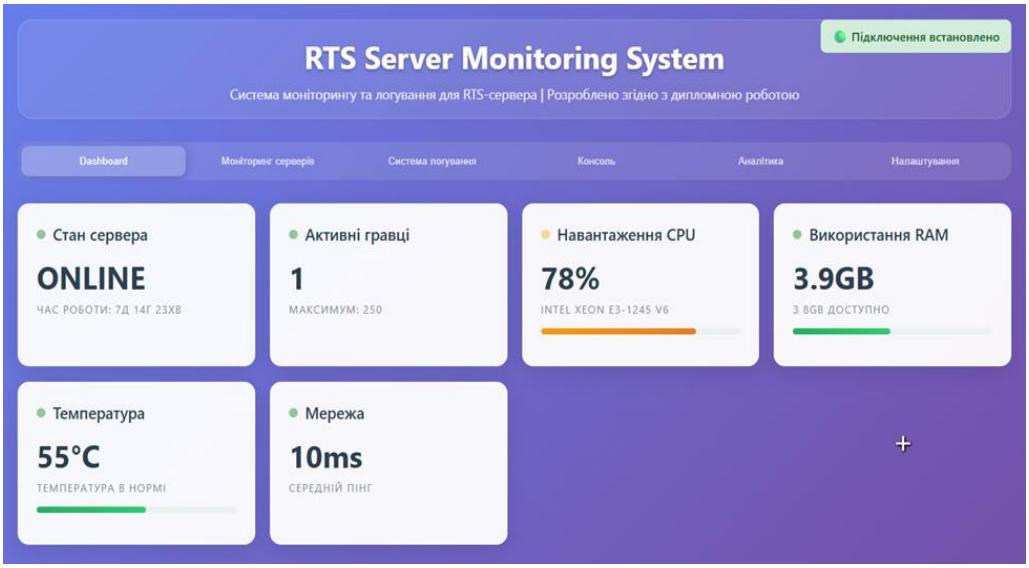

Рисунок 3.2 - Dashboard з усіма віджетами в робочому стані

Віджети розташовані за принципом важливості — зліва направо, зверху вниз. Найважливіше — статус сервера — завжди у верхньому лівому куті.

```
<div class="dashboard-grid">
  <div class="widget">
    <div class="widget-title">
      <div id="serverStatusIndicator" class="status-indicator status-
offline"></div>
      Стан сервера
    </div>
    <div id="serverStatus" class="metric-value">OFFLINE</div>
    <div id="serverUptime" class="metric-label">Перевірка підключення...</div>
  </div>

   <div class="widget" id="ram-widget">
  <div class="widget-title">
    <div class="status-indicator status-normal"></div>
    Використання RAM
  </div>
  <div class="metric-value" id="ram-used">Завантаження...</div>
  <div class="metric-label" id="ram-label"></div>
```

```
            <div class="progress-bar">
              <div class="progress-fill progress-normal" id="ram-progress" style="width: 0%"></div>
            </div>
          </div>
        </div>
```

Кожен віджет має три рівні інформації:

- Заголовок з індикатором стану — швидке розуміння, чи все в порядку
- Основне значення — ключова метрика великим шрифтом
- Додаткова інформація — деталі для тих, хто хоче глибше зрозуміти ситуацію

Система кольорового кодування

Розробив чітку систему кольорів, яка працює в усьому інтерфейсі:

```
.status-normal { background-color: #22c55e; }
.status-warning { background-color: #f59e0b; }
.status-critical { background-color: #ef4444; }
.status-offline { background-color: #6b7280; }
```

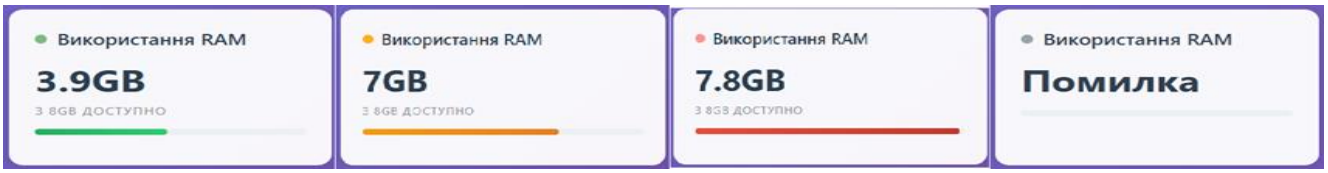

Рисунок 3.3 – Приклади віджетів у різних станах - нормальному, попереджувальному, критичному та недоступному

Ця система працює не тільки для індикаторів, а й для прогрес-барів, фону графіків та навіть тексту повідомлень. Користувач швидко звикає і на рівні інстинктів розуміє, що означає кожен колір.

Інтерактивні графіки та діаграми

Для візуалізації метрик використовую Chart.js — це дало можливість створити красиві та функціональні графіки без занурення в складності D3.js.

Рисунок 3.4 - Графік використання CPU з підписами осей

```
const cpuChart = new Chart(ctx, {
    type: 'line',
    data: {
        labels: timeLabels,
        datasets: [{
            label: 'CPU Usage (%)',
            data: cpuData,
            borderColor: 'rgb(59, 130, 246)',
            backgroundColor: 'rgba(59, 130, 246, 0.1)',
            borderWidth: 2,
            tension: 0.4,
            fill: true
        }]
    },
    options: {
        responsive: true,
        maintainAspectRatio: false,
        scales: {
            y: {
                beginAtZero: true,
                max: 100,
                ticks: {
```

```
                    callback: function(value) {
                        return value + '%';
                    }
                }
            }
        },
        plugins: {
            legend: {
                display: true,
                position: 'top'
            }
        }
    }
});
```

Особливу увагу приділив налаштуванню графіків — вони мають оновлюватися в реальному часі, але не «стрибати» при кожному оновленні. Для цього використовую плавні переходи та згладжування кривих.

Вкладка моніторингу серверів

Тут зібрана детальна інформація про стан сервера в системі. Якщо Dashboard показує загальну картину, то тут можна дізнатися більш детальну інформацію.

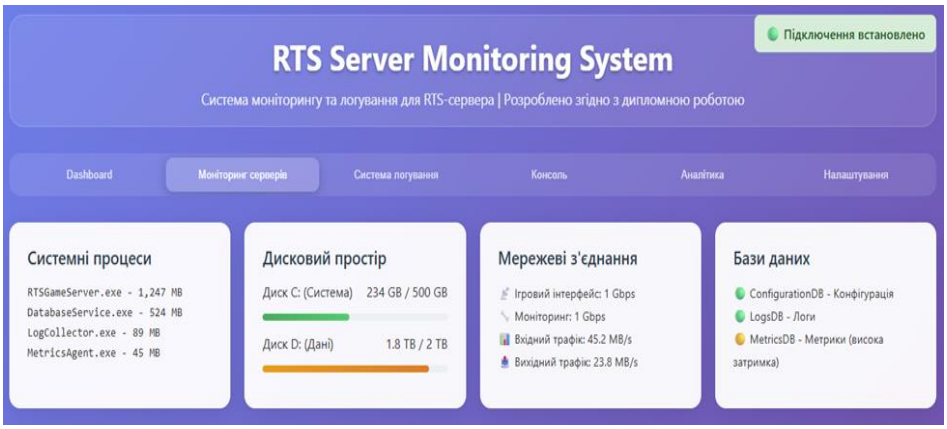

Рисунок 3.5 - Таблиця серверів з метриками та індикаторами стану

```
<div class="servers-grid">
    <div class="server-card" id="server-1">
        <div class="server-header">
            <div class="server-status status-normal"></div>
            <h3>Main Game Server</h3>
            <span class="server-ping">Ping: 12ms</span>
        </div>
        <div class="server-metrics">
```

```
            <div class="metric">
              <span class="metric-label">CPU:</span>
              <span class="metric-value">45%</span>
              <div class="metric-bar">
                <div class="metric-fill" style="width: 45%"></div>
              </div>
            </div>
            <div class="metric">
              <span class="metric-label">RAM:</span>
              <span class="metric-value">2.1 GB / 4 GB</span>
              <div class="metric-bar">
                <div class="metric-fill" style="width: 52%"></div>
              </div>
            </div>
          </div>
        </div>
      </div>
```

Кожна картка сервера — це мініатюрний Dashboard з найважливішими метриками. При натисканні на картку відкривається детальний вигляд з повною інформацією та історичними графіками.

Система логування з фільтрацією

Логи — це серце діагностики, тому їх інтерфейс має бути максимально зручним. Створив систему фільтрів, яка дозволяє швидко знаходити потрібну інформацію.

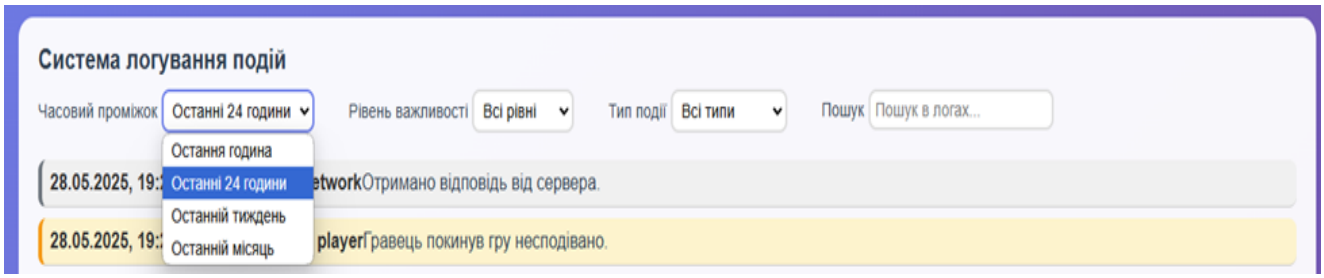

Рисунок 3.6 - Панель фільтрів логів з розкритими випадаючими списками

```
      <div class="log-filters">
        <div class="filter-group">
          <label>Часовий проміжок:</label>
          <select class="filter-select" id="timeFilter" onchange="filterLogs()">
            <option value="1h">Остання година</option>
            <option value="24h" selected>Останні 24 години</option>
            <option value="7d">Останній тиждень</option>
            <option value="30d">Останній місяць</option>
            <option value="custom">Власний діапазон</option>
          </select>
        </div>
```

```
<div class="filter-group">
  <label>Рівень важливості:</label>
  <select class="filter-select" id="levelFilter" onchange="filterLogs()">
    <option value="all" selected>Всі рівні</option>
    <option value="debug">DEBUG</option>
    <option value="info">INFO</option>
    <option value="warning">WARNING</option>
    <option value="error">ERROR</option>
    <option value="critical">CRITICAL</option>
  </select>
</div>

<div class="filter-group">
  <label>Пошук по тексту:</label>
  <input type="text" class="filter-input" id="searchFilter"
      placeholder="Введіть ключове слово..." onkeyup="filterLogs()">
</div>
</div>
```

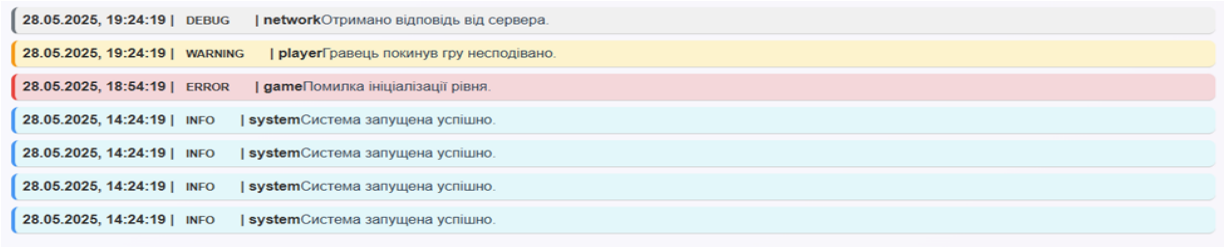

Рисунок 3.7 - Таблиця логів з різними рівнями важливості та кольоровим

Таблиця логів має розумну пагінацію — завантажую тільки ті записи, які користувач бачить на екрані. Це критично важливо, коли в системі сотні тисяч записів.

```
function renderLogTable(logs) {
  const tbody = document.getElementById('logs-tbody');
  tbody.innerHTML = '';

  logs.forEach(log => {
    const row = document.createElement('tr');
    row.className = `log-row log-${log.level.toLowerCase()}`;

    row.innerHTML = `
      <td class="log-time">${formatTime(log.timestamp)}</td>
      <td class="log-level">
        <span class="level-badge level-${log.level.toLowerCase()}">
          ${log.level}
        </span>
      </td>
      <td class="log-source">${log.source}</td>
      <td class="log-message">${escapeHtml(log.message)}</td>
```

```
            `;

            tbody.appendChild(row);
        });
    }
```

Консоль

Розробив мініатюрну консоль, яка дозволяє виконувати команди безпосередньо на сервері. Це дуже зручно для швидких дій — перезапуск сервісу, очищення кешу, зміна налаштувань.

Рисунок 3.8 - Консоль

```
<div class="console-container">
    <div class="console-output" id="console-output">
        <div class="console-line">
            <span class="console-prompt">system@monitoring:~$</span>
            <span class="console-text">Консоль готова до роботи</span>
        </div>
    </div>
    <div class="console-input-container">
        <span class="console-prompt">admin@server:~$</span>
        <input type="text" class="console-input" id="console-input"
               placeholder="Введіть команду..."
onkeydown="handleConsoleInput(event)">
    </div>
</div>
```

Консоль підтримує автодоповнення команд та зберігає історію — можна натиснути стрілку вгору і повторити попередню команду.

Аналітика та звіти

Вкладка аналітики дозволяє переглядати звіти за довільні періоди та експортувати дані в різних форматах.

Рисунок 3.9 - Сторінка аналітики з звітом та прев'ю графіка

```
<div class="analytics-controls">
  <div class="date-range-picker">
    <label>Період:</label>
    <input type="datetime-local" id="startDate" onchange="updateAnalytics()">
    <span>до</span>
    <input type="datetime-local" id="endDate" onchange="updateAnalytics()">
  </div>

  <div class="metric-selector">
    <label>Метрики:</label>
    <div class="checkbox-group">
      <label><input type="checkbox" value="cpu" checked> CPU Usage</label>
      <label><input type="checkbox" value="memory" checked> Memory Usage</label>
```

```
                <label><input type="checkbox" value="players" checked> Active Players</label>
                <label><input type="checkbox" value="network" checked> Network Traffic</label>
            </div>
        </div>

        <div class="export-controls">
            <button class="btn btn-primary" onclick="exportToCSV()">Експорт CSV</button>
            <button class="btn btn-secondary" onclick="exportToPDF()">Експорт PDF</button>
            <button class="btn btn-secondary" onclick="generateReport()">Створити звіт</button>
        </div>
    </div>
```

Налаштування системи

Останню вкладку присвятив налаштуванням — тут можна змінити пороги сповіщень, налаштувати інтеграції з зовнішніми сервісами, керувати користувачами.

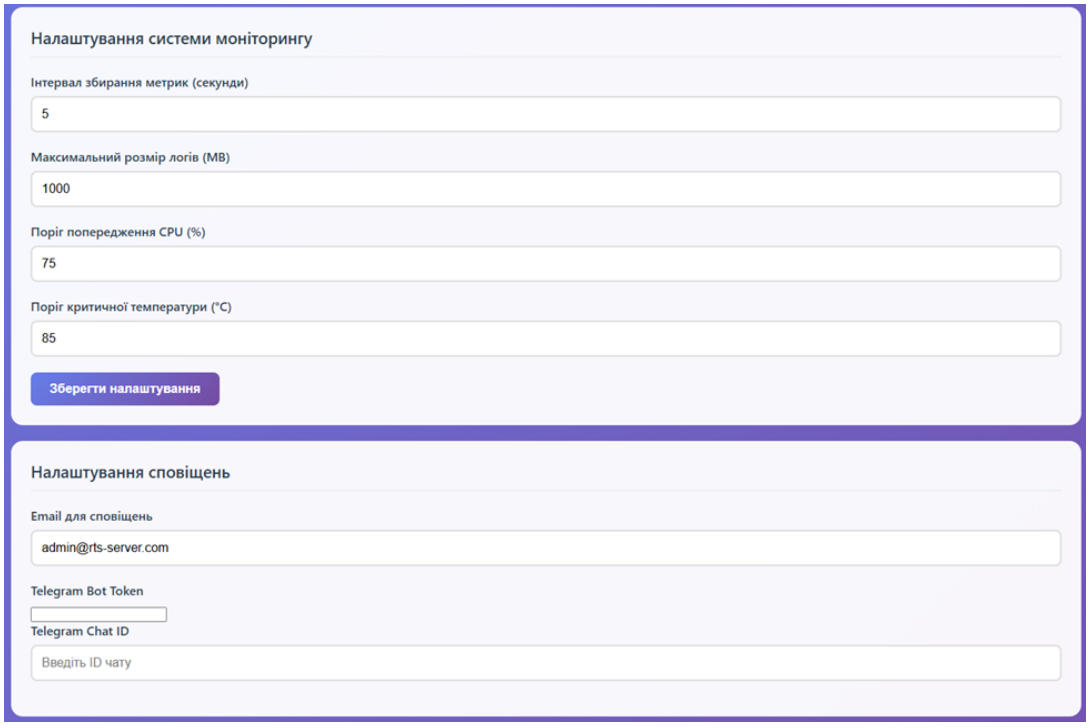

Рисунок 3.10 - Сторінка налаштувань з різними секціями

```
<div class="settings-sections">
    <div class="settings-section">
        <h3>Пороги сповіщень</h3>
```

```
                <div class="setting-item">
                    <label>CPU Warning (%):</label>
                    <input type="number" id="cpu-warning" value="70" min="0" max="100">
                </div>
                <div class="setting-item">
                    <label>CPU Critical (%):</label>
                    <input type="number" id="cpu-critical" value="90" min="0" max="100">
                </div>
                <div class="setting-item">
                    <label>Memory Warning (%):</label>
                    <input type="number" id="memory-warning" value="80" min="0"
max="100">
                </div>
            </div>

            <div class="settings-section">
                <h3>Інтеграції</h3>
                <div class="setting-item">
                    <label>Telegram Bot Token:</label>
                    <input type="text" id="telegram-token" placeholder="Введіть токен бота">
                </div>
                <div class="setting-item">
                    <label>Email сповіщення:</label>
                    <input type="email" id="notification-email"
placeholder="admin@example.com">
                </div>
            </div>
        </div>
```

Адаптивність та мобільна версія

Особливу увагу приділив адаптивності — часто потрібно перевіряти стан сервера з телефону або планшета.

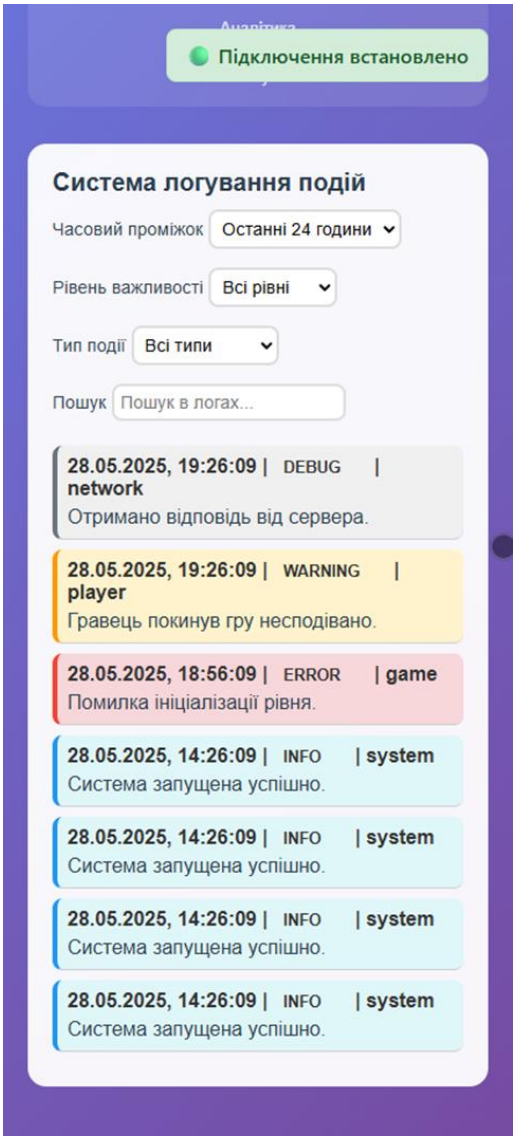

Рисунок 3.11 - Мобільна версія інтерфейсу на телефоні

```
@media (max-width: 768px) {
  .dashboard-grid {
    grid-template-columns: 1fr;
    gap: 1rem;
  }

  .widget {
    padding: 1rem;
  }

  .nav-tabs {
```

```
        overflow-x: auto;
        white-space: nowrap;
    }

    .console-container {
        font-size: 12px;
    }

    .servers-grid {
        grid-template-columns: 1fr;
    }
}
```

На мобільних пристроях віджети вишиковуються в одну колонку, шрифти стають трохи менші, а таблиці отримують горизонтальний скрол.

Стан підключення та індикатор роботи

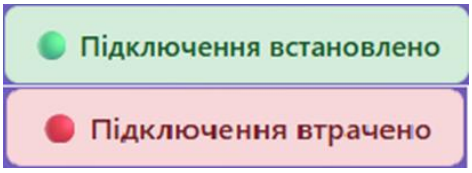

Рисунок 3.12 - Індикатор підключення в різних станах

В верхній частині інтерфейсу завжди видно статус підключення до сервера. Це критично важливо — користувач має розуміти, чи отримує він актуальні дані.

```
function updateConnectionStatus(connected) {
    const indicator = document.getElementById('connectionStatus');

    if (connected) {
        indicator.className = 'connection-status connection-online';
        indicator.textContent = 'Підключено';
    } else {
        indicator.className = 'connection-status connection-offline';
        indicator.textContent = 'Підключення втрачено';
    }
}
```

Оптимізація продуктивності інтерфейсу

Щоб інтерфейс працював швидко навіть з великими обсягами даних, застосував кілька технік оптимізації:

- Віртуалізація списків — для таблиць з тисячами записів рендерю тільки видимі рядки
- Дебаунсинг пошуку — фільтрація логів спрацьовує не на кожне натискання клавіші, а через 300мс після останнього
- Кешування запитів — результати API-викликів зберігаю в пам'яті на короткий час
- Ледаче завантаження графіків — Chart.js ініціалізується тільки коли вкладка стає активною

```
// Дебаунсинг для пошуку
let searchTimeout;
function handleSearchInput() {
    clearTimeout(searchTimeout);
    searchTimeout = setTimeout(() => {
        filterLogs();
    }, 300);
}

// Ледаче завантаження графіків
function showTab(tabName) {

    if (tabName === 'analytics' && !analyticsChartsInitialized) {
        initializeAnalyticsCharts();
        analyticsChartsInitialized = true;
    }
}
```

Доступність та зручність використання

Подбав про доступність інтерфейсу для користувачів з обмеженими можливостями:

- Всі інтерактивні елементи мають достатній контраст
- Можна керувати інтерфейсом за допомогою клавіатури
- Важлива інформація дублюється не тільки кольором, а й текстом
- Розміри шрифтів можна збільшувати за допомогою браузера

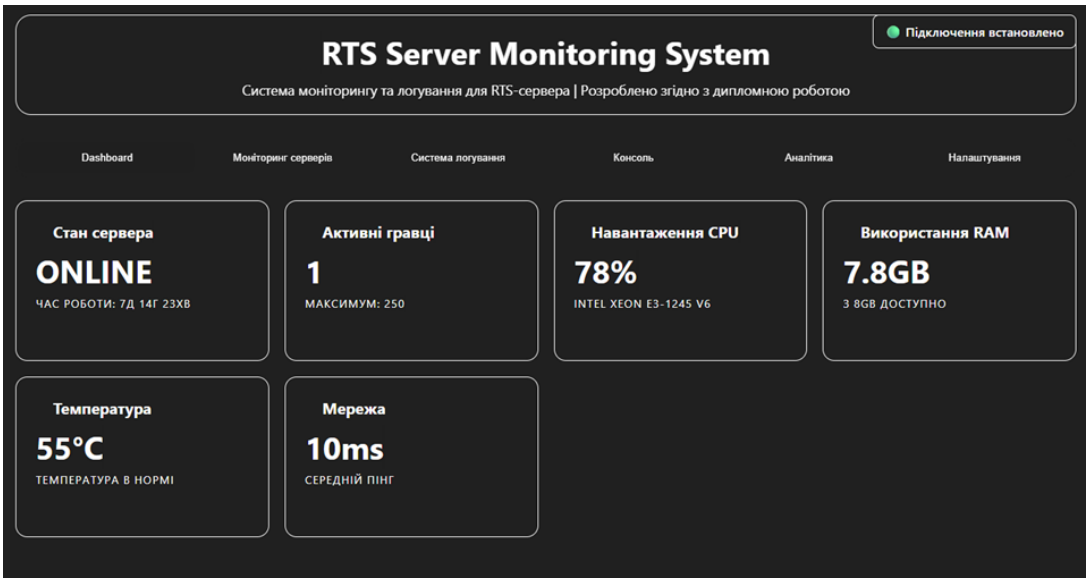

Рисунок 3.13 - Інтерфейс з увімкненим режимом високого контрасту

Результат — інтерфейс, яким приємно і зручно користуватися. Він не перевантажений деталями, але дає доступ до всієї необхідної інформації. А головне — він справді допомагає швидко знаходити та вирішувати проблеми з сервером.

3.4 Розробка інструкції з експлуатації

Цей розділ присвячено створенню інструкції користувача для експлуатації системи моніторингу та логування RTS-сервера. Інструкція розрахована на не технічного користувача, який не встановлює бази даних, не працює з кодом і не має досвіду адміністрування серверів. Основна мета — пояснити, як користуватися інтерфейсом, які вкладки для чого потрібні, і як зчитувати інформацію з системи.

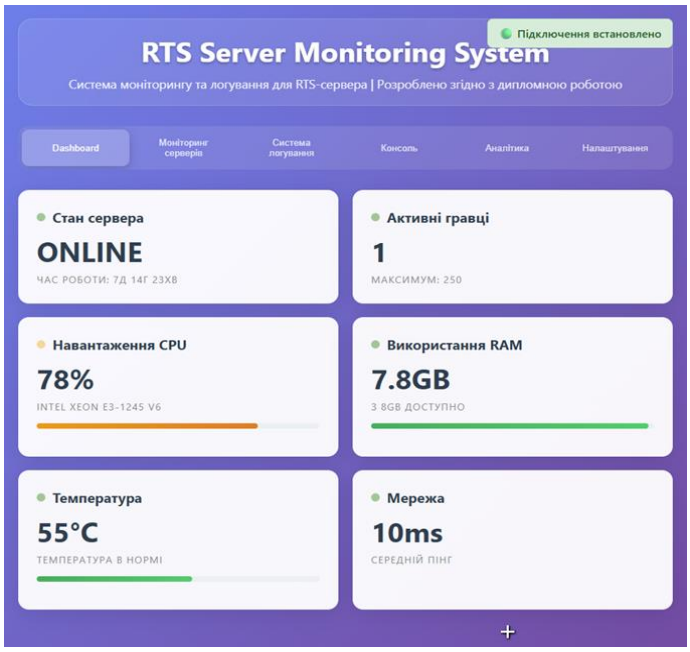

Рисунок 3.14 – Головна сторінка інтерфейсу моніторингу

Запуск системи

Після запуску (який здійснює технічний спеціаліст), користувачеві достатньо відкрити вебсторінку системи у браузері.

Основна структура інтерфейсу

Після відкриття сайту користувач бачить панель навігації з вкладками:

- Dashboard
- Моніторинг
- Система логування
- Консоль
- Аналітика
- Налаштування

Кожна з них має своє призначення.

Рисунок 3.15 – Панель навігації системи

Вкладка «Dashboard»

Це головна панель моніторингу. Вона містить ключові показники:

- Стан сервера (online / offline)
- Завантаження CPU
- Використання RAM
- Кількість гравців online
- Час роботи сервера

Кожен показник має кольорову індикацію:

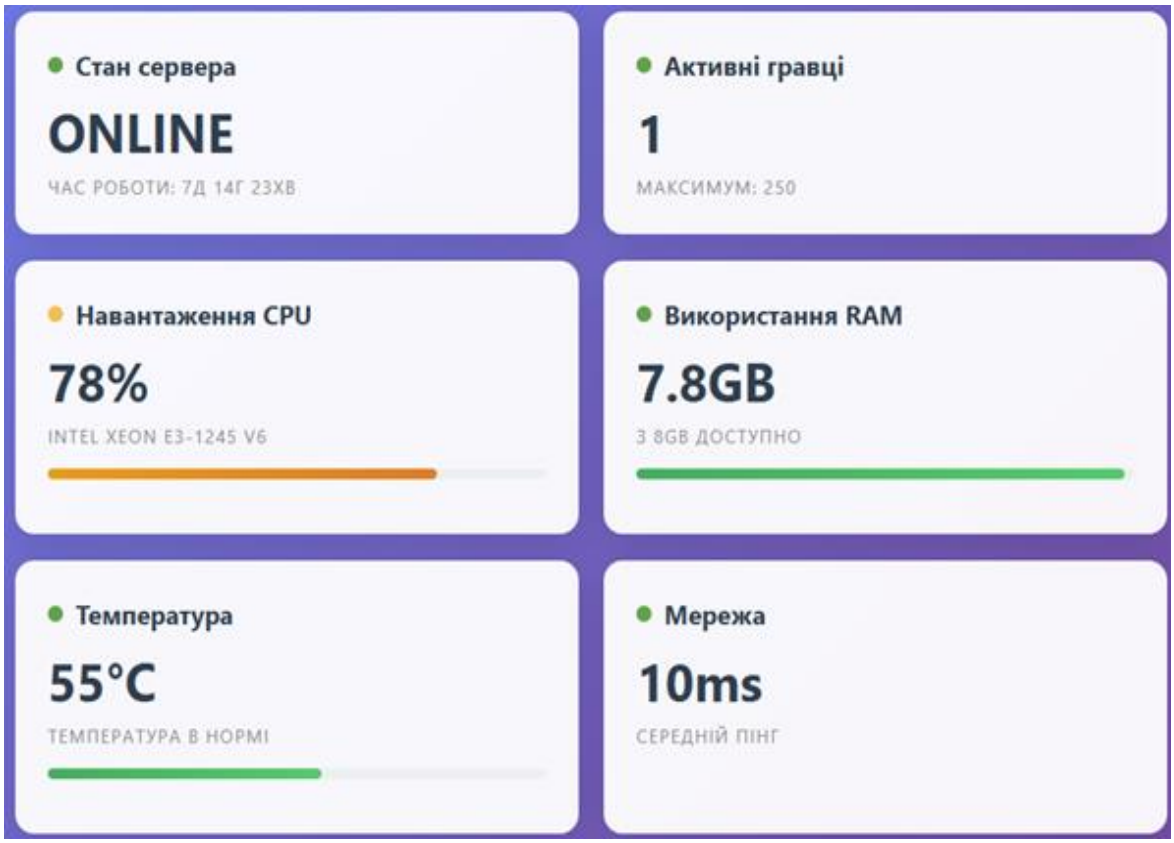

Рисунок 3.16 – Основні віджети системного моніторингу

- Зелений — норма

- Жовтий — підвищене навантаження
- Червоний — критичний стан

Вкладка «Моніторинг»

Ця секція показує детальну інформацію про навантаження:

- Графіки використання процесора
- Статистика пам’яті
- Мережевий трафік
- Динаміка підключень

Графіки оновлюються в реальному часі, і дозволяють помітити пік навантаження або проблеми на сервері.

Рисунок 3.17 – Графік завантаження процесора

Вкладка «Система логування»

Тут відображаються всі події, які були зафіксовані сервером:

- Підключення / відключення гравців
- Помилки у грі
- Аномалії в роботі сервера

Логи можна фільтрувати за:

- Рівнем важливості (INFO, WARNING, ERROR)
- Проміжком часу
- Ім’ям гравця або подією

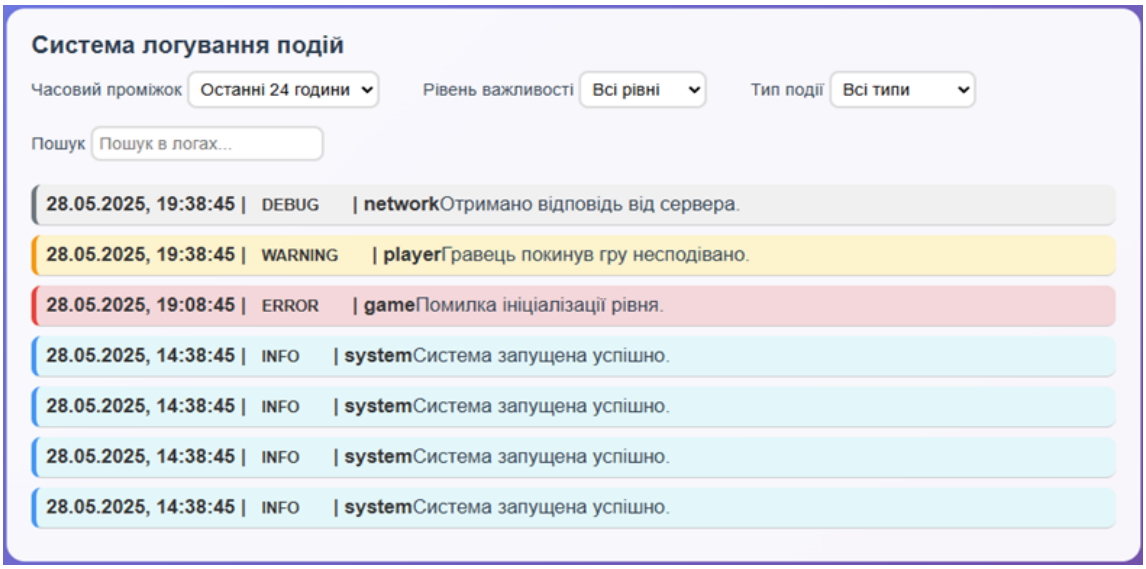

Рисунок 3.18 – Журнал подій

Вкладка «Консоль»

Консоль — це місце, де можна переглядати живі повідомлення, які надходять у реальному часі. Це корисно для тих, хто хоче бачити, що відбувається «прямо зараз» — під час гри, тестування чи навантаження.

Вкладка «Аналітика»

У цій вкладці можна побачити тренди і статистику за останні години, дні або тижні. Наприклад:

- Коли сервер мав найбільше гравців
- Які помилки з’являлись найчастіше
- Як змінювалося навантаження з часом

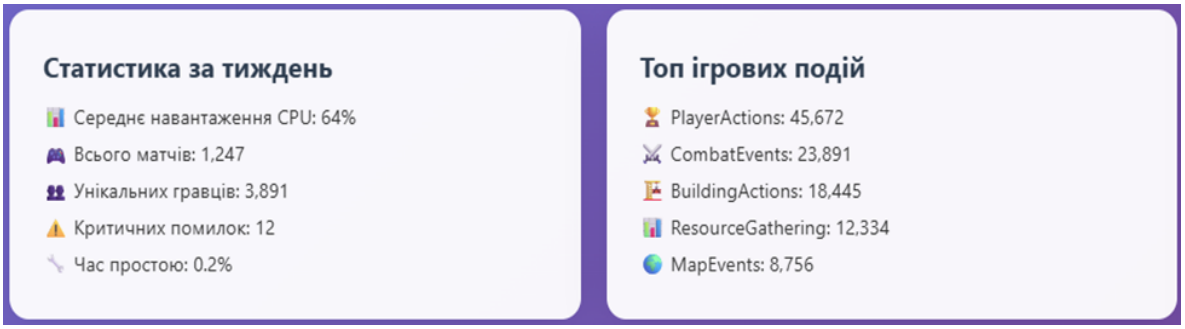

Рисунок 3.19 – Аналітика логів та навантаження

Вкладка «Налаштування»

Для звичайного користувача рекомендовано не змінювати налаштування, але тут можна:

- Змінити мову інтерфейсу
- Увімкнути / вимкнути певні сповіщення
- Обрати часовий пояс

Сповіщення

У разі виявлення критичної ситуації система може:

- Показати повідомлення на екрані
- Відправити сповіщення в Telegram або на email (залежно від налаштувань)

Як діяти у разі помилки

Якщо на екрані з’являється червона помилка (наприклад, «немає підключення до сервера»), варто:

1. Перевірити, чи є інтернет
2. Оновити сторінку (F5)
3. Якщо проблема не зникає — повідомити адміністратора

ВИСНОВКИ

У процесі розробки системи моніторингу та логування RTS-сервера було реалізовано повноцінний програмний продукт, здатний забезпечити стабільну та надійну роботу серверної частини комп’ютерної стратегічної гри в реальному часі.

Проаналізувавши проблеми, з якими стикаються розробники під час експлуатації ігрових серверів, було визначено основні вимоги до системи: моніторинг у реальному часі, гнучке логування, автоматичні сповіщення, зручний інтерфейс та масштабованість. На підставі цих вимог було спроектовано та реалізовано архітектуру з чітким розподілом обов’язків між компонентами системи.

Для реалізації проєкту було використано такі технології: Node.js — для побудови серверної логіки, HTML/CSS/JS — для клієнтської частини, Firebird 2.1 — як основна реляційна база даних. Також застосовано WebSocket для передачі даних у реальному часі та бібліотеку Chart.js для інтерактивної візуалізації метрик.

Під час розробки виникали технічні складнощі, пов’язані з високим обсягом даних та навантаженням на сервер, що вимагало впровадження рівнів логування, архівування, оптимізації зберігання та фільтрації подій. Усі ці виклики було успішно подолано.

Запропонована система дозволяє:

- у режимі реального часу контролювати стан ігрового сервера;
- швидко виявляти та аналізувати помилки;
- отримувати сповіщення про критичні події;
- прогнозувати можливі проблеми на основі історичних даних;
- забезпечити прозорість і контроль для розробників та адміністраторів.

Таким чином, результати дипломної роботи мають не лише теоретичну, а й практичну цінність. Розроблена система може бути використана в реальних

проєктах для підтримки як інді-ігор, так і більш масштабних серверних рішень. Отриманий досвід підтвердив доцільність впровадження власного програмного рішення для моніторингу і логування в сфері геймдеву.

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Бойко В. М. Системи моніторингу комп’ютерних мереж: навч. посіб. Київ: КНУ, 2021. 240 с.

2. Коваленко О. П., Іванова Т. С. Бази даних: проектування, реалізація, застосування. Харків: ХНУ, 2022. 312 с.

3. Глушков В. М. Основи програмної інженерії. Львів: Видавництво ЛНУ, 2020. 198 с.

4. Литвиненко Д. О. Моніторинг і логування в хмарних сервісах // Вісник КНУ. Серія: Інформатика. 2023. №2. С. 45–52.

5. Zabbix Documentation 6.0. URL: https://www.zabbix.com/documentation/6.0/manual (дата звернення: 27.05.2025).

6. Prometheus: Monitoring system & time series database. URL: https://prometheus.io/docs/introduction/overview/

7. Grafana Labs. Grafana Documentation. URL: https://grafana.com/docs/grafana/latest/

8. MongoDB Manual. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 27.05.2025).

9. InfluxData Documentation. URL: https://docs.influxdata.com/influxdb/

10. Netdata Documentation. URL: https://learn.netdata.cloud/docs/ (дата звернення: 27.05.2025).

11. A systematic review of gamification in software engineering education URL: https://elibrary.kdpu.edu.ua/bitstream/123456789/10951/1/paper04.pdf

12. Закон України «Про захист інформації в інформаційно-телекомунікаційних системах» від 05.07.1994 № 80/94-ВР.

13. Денисенко М. П. Архітектура серверних систем. Дніпро: ДНУ, 2021. 210 с.

14. Nagios Documentation. URL: https://www.nagios.org/documentation/

15. ManageEngine OpManager Documentation. URL: https://www.manageengine.com/network-monitoring/help/

ДОДАТОК А

А.1ПОВНИЙ ТЕКСТ ПРОГРАМНИХ МОДУЛІВ

```
const express = require('express');
const Firebird = require('node-firebird');
const cors = require('cors');
const fs = require('fs');
const path = require('path');

const app = express();
const port = 3000;

app.use(cors());
app.use(express.static('public'));

// === Firebird config ===
const dbOptions = {
  host: 'localhost',
  port: 3050,
  database: 'C:/Users/Ruslam/OneDrive/Desktop/diplom/OpenHV-main-
site/database.fdb',
  user: 'SYSDBA',
  password: 'masterkey',
  lowercase_keys: true,
  role: null,
  pageSize: 4096,
};

function queryDatabase(sql, params = []) {
  return new Promise((resolve, reject) => {
    Firebird.attach(dbOptions, (err, db) => {
      if (err) return reject(err);
      db.query(sql, params, (err, result) => {
        db.detach();
        if (err) return reject(err);
        resolve(result);
      });
    });
  });
}

// === API ===

app.get('/api/monitoring', async (req, res) => {
  try {
    const results = await queryDatabase('SELECT * FROM players');
    res.json(results);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
```

```
  }
});

app.get('/api/logs', async (req, res) => {
  try {
    const results = await queryDatabase('SELECT * FROM logs ORDER BY
event_time DESC');
    res.json(results);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});

app.get('/api/online', async (req, res) => {
  try {
    const results = await queryDatabase(“SELECT COUNT(*) AS online_count
FROM players WHERE status = 'online'“);
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});
app.get('/api/server_status', async (req, res) => {
  try {
    const results = await queryDatabase(“SELECT COUNT(*) AS online_count
FROM server WHERE status = 'online'“);
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});
app.get('/api/RAM', async (req, res) => {
  try {
    const results = await queryDatabase(“SELECT SUM(RAM_USAGE_MB) AS
total_ram_usage FROM server WHERE status = 'online'“);
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});
app.get('/api/PING', async (req, res) => {
  try {
    const results = await queryDatabase(“SELECT AVG(AVG_PING_MS) AS
average_ping FROM server WHERE status = 'online'“);
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
```

```
  }
});
app.get('/api/TEMPERATURE', async (req, res) => {
  try {
    const results = await queryDatabase("SELECT AVG(TEMPERATURE_C) AS
average_TEMPERATURE FROM server WHERE status = 'online'");
    res.json(results[0]);
  } catch (error) {
    console.error('Помилка БД:', error);
    res.status(500).json({ error: 'Помилка БД' });
  }
});
app.get('/api/cpu_chart', async (req, res) => {
  try {
    const results = await queryDatabase(`
      SELECT log_time, usage_percent FROM CPU_USAGE
      ORDER BY log_time
    `);
    const labels = results.map(r => new Date(r.log_time).toLocaleTimeString());
    const data = results.map(r => r.usage_percent);
    res.json({ labels, data });
  } catch (err) {
    console.error('CPU Chart Error:', err);
    res.status(500).json({ error: 'DB Error' });
  }
});
app.get('/api/memory_chart', async (req, res) => {
  try {
    const results = await queryDatabase(`
      SELECT log_time, used_mb FROM MEMORY_USAGE
      ORDER BY log_time
    `);
    const labels = results.map(r => new Date(r.log_time).toLocaleTimeString());
    const data = results.map(r => r.used_mb);
    res.json({ labels, data });
  } catch (err) {
    console.error('Memory Chart Error:', err);
    res.status(500).json({ error: 'DB Error' });
  }
});
app.get('/api/players_chart', async (req, res) => {
  try {
    const results = await queryDatabase(`
      SELECT log_time, player_count FROM ACTIVE_PLAYERS
      ORDER BY log_time
    `);
    const labels = results.map(r => new Date(r.log_time).toLocaleTimeString());
    const data = results.map(r => r.player_count);
    res.json({ labels, data });
  } catch (err) {
    console.error('Players Chart Error:', err);
    res.status(500).json({ error: 'DB Error' });
```

```
  }
});
app.get('/api/network_chart', async (req, res) => {
  try {
    const results = await queryDatabase(`
      SELECT log_time, incoming_mb, outgoing_mb FROM NETWORK_TRAFFIC
      ORDER BY log_time
    `);
    const labels = results.map(r => new Date(r.log_time).toLocaleTimeString());
    const incoming = results.map(r => r.incoming_mb);
    const outgoing = results.map(r => r.outgoing_mb);
    res.json({ labels, incoming, outgoing });
  } catch (err) {
    console.error('Network Chart Error:', err);
    res.status(500).json({ error: 'DB Error' });
  }
});

app.get('/api/console-logs', (req, res) => {
  fs.readFile('C:/Users/Ruslam/OneDrive/Desktop/diplom/OpenHV-
main/OpenRA_Server_Log.txt', 'utf8', (err, data) => {
    if (err) {
      console.error('Помилка читання логів:', err);
      return res.status(500).json({ error: 'Не вдалося прочитати логи' });
    }
    const lines = data.trim().split('\n');
    res.json(lines);
  });
});
async function updatePlayerStatus(nickname, status) {
  const now = new Date();

  try {
    const existing = await queryDatabase('SELECT id FROM players WHERE
nickname = ?', [nickname]);

    if (existing.length === 0) {
      await queryDatabase(
        'INSERT INTO players (nickname, status, last_seen) VALUES (?, ?, ?)',
        [nickname, status, now]
      );
      console.log(`🆕 Додано нового гравця: ${nickname} → ${status}`);
    } else {
      await queryDatabase(
        'UPDATE players SET status = ?, last_seen = ? WHERE nickname = ?',
        [status, now, nickname]
      );
      console.log(`✅ Статус оновлено: ${nickname} → ${status}`);
    }
  } catch (err) {
    console.error('❌ Помилка оновлення:', err);
  }
```

```
}
const logPath = 'C:/Users/Ruslam/OneDrive/Desktop/diplom/OpenHV-
main/OpenRA_Server_Log.txt';
let lastSize = 0;

setInterval(() => {
  fs.stat(logPath, (err, stats) => {
    if (err) {
      console.error('Не вдалося отримати лог-файл:', err);
      return;
    }

    if (stats.size > lastSize) {
      const stream = fs.createReadStream(logPath, {
        start: lastSize,
        end: stats.size
      });

      let buffer = '';

      stream.on('data', chunk => {
        buffer += chunk.toString();
      });

      stream.on('end', async () => {
        const lines = buffer.trim().split('\n');

        for (const line of lines) {
          const isConnect = line.includes('[Skirmish Game]');
          const isDisconnect = line.includes('[Disconnecte]');
          const match = line.match(/\[(?:Skirmish Game|Disconnecte)\] (\w+)/);

          if (!match) continue;

          const nickname = match[1];
          const newStatus = isConnect ? 'online' : isDisconnect ? 'offline' : null;

          if (nickname && newStatus) {
            await updatePlayerStatus(nickname, newStatus);
          }
        }
      });

      lastSize = stats.size;
    }
  });
}, 1000);

app.listen(port, () => {
  console.log(`Сервер запущено на http://localhost:${port}`);
});
<!DOCTYPE html>
```

```
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <link rel="stylesheet" href="styles.css">
  <script src="Collector.js"></script>
  <script src="app.js"></script>

  <title>RTS Server Monitoring - Система моніторингу сервера</title>
</head>
<body>
  <div id="connectionStatus" class="connection-status connection-offline">
    Підключення втрачено
  </div>

  <div class="container">
    <div class="header">
      <h1>RTS Server Monitoring System</h1>
      <p>Система моніторингу та логування для RTS-сервера | Розроблено
згідно з дипломною роботою</p>
    </div>

    <div class="nav-tabs">
      <button class="nav-tab active"
onclick="showTab('dashboard')">Dashboard</button>
      <button class="nav-tab" onclick="showTab('monitoring')">Моніторинг
серверів</button>
      <button class="nav-tab" onclick="showTab('logs')">Система
логування</button>
      <button class="nav-tab" onclick="showTab('console')">Консоль</button>
      <button class="nav-tab"
onclick="showTab('analytics')">Аналітика</button>
      <button class="nav-tab"
onclick="showTab('settings')">Налаштування</button>
    </div>

    <!-- Dashboard Tab -->
    <div id="dashboard" class="tab-content active">
      <div class="dashboard-grid">
        <div class="widget">
          <div class="widget-title">
            <div class="status-indicator status-normal"></div>
            Стан сервера
          </div>
          <div class="metric-value">ONLINE</div>
          <div class="metric-label">Час роботи: 7д 14г 23хв</div>
        </div>

        <div class="widget">
          <div class="widget-title">
```

```
                    <div class="status-indicator status-normal"></div>
                    Активні гравці
                </div>
                <div id="onlineCount" class="metric-value">Завантаження...</div>
                <div class="metric-label">Максимум: 250</div>
            </div>

            <div class="widget">
                <div class="widget-title">
                    <div class="status-indicator status-warning"></div>
                    Навантаження CPU
                </div>
                <div class="metric-value">78%</div>
                <div class="metric-label">Intel Xeon E3-1245 v6</div>
                <div class="progress-bar">
                    <div class="progress-fill progress-warning" style="width:
78%"></div>
                </div>
            </div>

            <div class="widget" id="ram-widget">
        <div class="widget-title">
            <div class="status-indicator status-normal"></div>
            Використання RAM
        </div>
        <div class="metric-value" id="ram-used">Завантаження...</div>
        <div class="metric-label" id="ram-label"></div>
        <div class="progress-bar">
            <div class="progress-fill progress-normal" id="ram-progress" style="width:
0%"></div>
        </div>
    </div>

            <div class="widget" id="temperature-widget">
        <div class="widget-title">
            <div class="status-indicator" id="temp-status-indicator"></div>
            Температура
        </div>
        <div class="metric-value" id="temp-value">Завантаження...</div>
        <div class="metric-label" id="temp-label"></div>
        <div class="progress-bar">
            <div class="progress-fill" id="temp-progress" style="width: 0%"></div>
        </div>
    </div>

            <div class="widget" id="ping-widget">
        <div class="widget-title">
            <div class="status-indicator status-normal"></div>
            Мережа
        </div>
        <div class="metric-value" id="ping-value">Завантаження...</div>
```

```
            <div class="metric-label">Середній пінг</div>
        </div>
            </div>

            <div id="alertsContainer">
                <!-- Alerts will be dynamically added here -->
            </div>
        </div>

        <!-- Server Monitoring Tab -->
        <div id="monitoring" class="tab-content">
            <div class="dashboard-grid">
                <div class="widget">
                    <div class="widget-title">Системні процеси</div>
                    <div style="font-family: monospace; font-size: 0.9rem; line-height:
1.6;">
                        <div>RTSGameServer.exe - 1,247 MB</div>
                        <div>DatabaseService.exe - 524 MB</div>
                        <div>LogCollector.exe - 89 MB</div>
                        <div>MetricsAgent.exe - 45 MB</div>
                    </div>
                </div>

                <div class="widget">
                    <div class="widget-title">Дисковий простір</div>
                    <div style="margin-bottom: 10px;">
                        <div style="display: flex; justify-content: space-between;">
                            <span>Диск C: (Система)</span>
                            <span>234 GB / 500 GB</span>
                        </div>
                        <div class="progress-bar">
                            <div class="progress-fill progress-normal" style="width:
47%"></div>
                        </div>
                    </div>
                    <div>
                        <div style="display: flex; justify-content: space-between;">
                            <span>Диск D: (Дані)</span>
                            <span>1.8 TB / 2 TB</span>
                        </div>
                        <div class="progress-bar">
                            <div class="progress-fill progress-warning" style="width:
90%"></div>
                        </div>
                    </div>
                </div>

                <div class="widget">
                    <div class="widget-title">Мережеві з'єднання</div>
                    <div style="font-size: 0.9rem; line-height: 1.8;">
                        <div>📡 Ігровий інтерфейс: 1 Gbps</div>
```

```
                <div>🔧 Моніторинг: 1 Gbps</div>
                <div>📊 Вхідний трафік: 45.2 MB/s</div>
                <div>📤 Вихідний трафік: 23.8 MB/s</div>
              </div>
            </div>

            <div class="widget">
              <div class="widget-title">Бази даних</div>
              <div style="font-size: 0.9rem; line-height: 1.8;">
                <div>🗄️ ConfigurationDB - Конфігурація</div>
                <div>🗄️ LogsDB - Логи</div>
                <div>🗄️ MetricsDB - Метрики (висока затримка)</div>
              </div>
            </div>
          </div>
        </div>

        <!-- Logs Tab -->
        <div id="logs" class="tab-content">
<div class="logs-container">
  <div class="widget-title">Система логування подій</div>

  <div class="log-filters">
    <div class="filter-group">
      <label>Часовий проміжок</label>
      <select class="filter-select" id="timeFilter" onchange="filterLogs()">
        <option value="1h">Остання година</option>
        <option value="24h" selected>Останні 24 години</option>
        <option value="7d">Останній тиждень</option>
        <option value="30d">Останній місяць</option>
      </select>
    </div>
    <div class="filter-group">
      <label>Рівень важливості</label>
      <select class="filter-select" id="levelFilter" onchange="filterLogs()">
        <option value="all" selected>Всі рівні</option>
        <option value="debug">DEBUG</option>
        <option value="info">INFO</option>
        <option value="warning">WARNING</option>
        <option value="error">ERROR</option>
        <option value="critical">CRITICAL</option>
      </select>
    </div>
    <div class="filter-group">
      <label>Тип події</label>
      <select class="filter-select" id="typeFilter" onchange="filterLogs()">
        <option value="all" selected>Всі типи</option>
        <option value="system">Системна</option>
        <option value="game">Ігрова логіка</option>
        <option value="player">Дії гравців</option>
```

```
                <option value="network">Мережа</option>
              </select>
            </div>
            <div class="filter-group">
              <label>Пошук</label>
              <input type="text" class="filter-input" id="searchInput" placeholder="Пошук в логах..." onkeyup="filterLogs()">
            </div>
          </div>

          <div id="log-entries">
            <!-- Logs will be dynamically loaded here -->
          </div>
        </div>
      </div>

          <!-- Console Tab -->
          <div id="console" class="tab-content">
            <div class="console-container">
              <div class="widget-title">Консоль логів сервера</div>
              <div id="logConsole" class="console"></div>
            </div>
          </div>

          <!-- Analytics Tab -->
          <div id="analytics" class="tab-content">
            <div class="analytics-grid">
              <div class="chart-container">
                <div class="chart-title">Навантаження CPU за останні 24 години</div>
                <canvas id="cpuChart" width="100%" height="200"></canvas>
              </div>

              <div class="chart-container">
                <div class="chart-title">Використання пам'яті</div>
                <canvas id="memoryChart" width="100%" height="200"></canvas>
              </div>

              <div class="chart-container">
                <div class="chart-title">Кількість активних гравців</div>
                <canvas id="playersChart" width="100%" height="200"></canvas>
              </div>

              <div class="chart-container">
                <div class="chart-title">Мережевий трафік</div>
                <canvas id="networkChart" width="100%" height="200"></canvas>
              </div>
            </div>

            <div class="dashboard-grid">
              <div class="widget">
```

```
                    <div class="widget-title">Статистика за тиждень</div>
                    <div style="font-size: 0.9rem; line-height: 1.8;">
                        <div>📊 Середнє навантаження CPU: 64%</div>
                        <div>🎮 Всього матчів: 1,247</div>
                        <div>👥 Унікальних гравців: 3,891</div>
                        <div>⚠️ Критичних помилок: 12</div>
                        <div>🔧 Час простою: 0.2%</div>
                    </div>
                </div>

                <div class="widget">
                    <div class="widget-title">Топ ігрових подій</div>
                    <div style="font-size: 0.9rem; line-height: 1.8;">
                        <div>🏆 PlayerActions: 45,672</div>
                        <div>⚔️ CombatEvents: 23,891</div>
                        <div>🏗️ BuildingActions: 18,445</div>
                        <div>📊 ResourceGathering: 12,334</div>
                        <div>🌍 MapEvents: 8,756</div>
                    </div>
                </div>
            </div>
        </div>

        <!-- Settings Tab -->
        <div id="settings" class="tab-content">
            <div class="settings-section">
                <div class="settings-title">Налаштування системи моніторингу</div>

                <div class="settings-group">
                    <label>Інтервал збирання метрик (секунди)</label>
                    <input type="number" class="settings-input" value="5" min="1"
max="60">
                </div>

                <div class="settings-group">
                    <label>Максимальний розмір логів (MB)</label>
                    <input type="number" class="settings-input" value="1000" min="100"
max="10000">
                </div>

                <div class="settings-group">
                    <label>Поріг попередження CPU (%)</label>
                    <input type="number" class="settings-input" value="75" min="50"
max="95">
                </div>

                <div class="settings-group">
                    <label>Поріг критичної температури (°C)</label>
                    <input type="number" class="settings-input" value="85" min="70"
max="100">
```

```
                </div>

                <button class="btn btn-primary" onclick="saveSettings()">Зберегти
налаштування</button>
            </div>

            <div class="settings-section">
                <div class="settings-title">Налаштування сповіщень</div>

                <div class="settings-group">
                    <label>Email для сповіщень</label>
                    <input type="email" class="settings-input" value="admin@rts-
server.com">
                </div>

                <div class="settings-group">
                    <label>Telegram Bot Token</label>
                    <input type="text"
                </div>

                <div class="settings-group">
                    <label>Telegram Chat ID</label>
                    <input type="text" class="settings-input" placeholder="Введіть ID
чату">
                </div>
            </div>
        </div>
    </div>
</body>
function showTab(tabId) {
    const tabs = document.querySelectorAll('.tab-content');
    const buttons = document.querySelectorAll('.nav-tab');

    tabs.forEach(tab => tab.classList.remove('active'));
    buttons.forEach(btn => btn.classList.remove('active'));

    document.getElementById(tabId).classList.add('active');
    event.target.classList.add('active');
}

function updateConnectionStatus() {
    const statusEl = document.getElementById('connectionStatus');
    fetch('/api/server_status')
        .then(res => res.json())
        .then(data => {
            if (data.online_count > 0) {
                statusEl.className = 'connection-status connection-online';
                statusEl.textContent = '□ Підключення встановлено';
            } else {
                statusEl.className = 'connection-status connection-offline';
                statusEl.textContent = '● Підключення втрачено';
```

```
            }
        })
        .catch(() => {
            statusEl.className = 'connection-status connection-offline';
            statusEl.textContent = '🔴 Підключення втрачено';
        });
}

function loadOnlineCount() {
    fetch('/api/online')
        .then(res => res.json())
        .then(data => {
            document.getElementById('onlineCount').textContent = data.online_count
|| '0';
        })
        .catch(() => {
            document.getElementById('onlineCount').textContent = 'N/A';
        });
}
const TOTAL_RAM_MB = 8192; // Общее доступное значение

async function updateRAMWidget() {
    try {
        const response = await fetch('/api/RAM');
        const data = await response.json();
        const usedRAM = data.total_ram_usage; // в МБ

        const usedGB = (usedRAM / 1024).toFixed(1);
        const totalGB = (TOTAL_RAM_MB / 1024).toFixed(0);
        const percent = Math.min(100, (usedRAM / TOTAL_RAM_MB *
100).toFixed(0));

        document.getElementById('ram-used').textContent = `${usedGB}GB`;
        document.getElementById('ram-label').textContent = `з ${totalGB}GB
доступно`;
        document.getElementById('ram-progress').style.width = `${percent}%`;

    } catch (error) {
        console.error('Помилка отримання RAM:', error);
        document.getElementById('ram-used').textContent = 'Помилка';
        document.getElementById('ram-label').textContent = '';
    }
}

updateRAMWidget();
setInterval(updateRAMWidget, 30000);
async function updatePingWidget() {
    try {
        const response = await fetch('/api/PING');
        const data = await response.json();
        const ping = parseFloat(data.average_ping).toFixed(0);
```

```
        document.getElementById('ping-value').textContent = `${ping}ms`;
    } catch (error) {
        console.error('Помилка отримання PING:', error);
        document.getElementById('ping-value').textContent = 'Помилка';
    }
}

updatePingWidget();
setInterval(updatePingWidget, 30000);
async function updateTemperatureWidget() {
    try {
        const response = await fetch('/api/TEMPERATURE');
        const data = await response.json();
        const temperature = parseFloat(data.average_temperature).toFixed(0);

        const progress = Math.min(100, temperature);
        const indicator = document.getElementById('temp-status-indicator');
        const fill = document.getElementById('temp-progress');
        const label = document.getElementById('temp-label');

        document.getElementById('temp-value').textContent = `${temperature}°C`;
        fill.style.width = `${progress}%`;

        if (temperature >= 80) {
            indicator.className = 'status-indicator status-critical pulse-critical';
            fill.className = 'progress-fill progress-critical';
            label.textContent = 'Критично! Перевірте охолодження';
        } else if (temperature >= 60) {
            indicator.className = 'status-indicator status-warning';
            fill.className = 'progress-fill progress-warning';
            label.textContent = 'Підвищена температура';
        } else {
            indicator.className = 'status-indicator status-normal';
            fill.className = 'progress-fill progress-normal';
            label.textContent = 'Температура в нормі';
        }

    } catch (error) {
        console.error('Помилка отримання температури:', error);
        document.getElementById('temp-value').textContent = 'Помилка';
        document.getElementById('temp-label').textContent = '';
    }
}

updateTemperatureWidget();
setInterval(updateTemperatureWidget, 30000);
function loadConsoleLogs() {
    fetch('/api/console-logs')
        .then(res => res.json())
        .then(lines => {
            const consoleEl = document.getElementById('logConsole');
            consoleEl.innerHTML = lines.map(line => {
```

```
                    if (line.includes('ERROR')) return `<div class="console-log-entry
console-log-error">${line}</div>`;
                    if (line.includes('WARN')) return `<div class="console-log-entry console-
log-warn">${line}</div>`;
                    if (line.includes('SUCCESS')) return `<div class="console-log-entry
console-log-success">${line}</div>`;
                    return `<div class="console-log-entry console-log-info">${line}</div>`;
                }).join('');
                consoleEl.scrollTop = consoleEl.scrollHeight;
            });
        }
        function addLogEntry(timestamp, level, type, message) {
            const logContainer = document.getElementById("log-entries");

            const logEntry = document.createElement("div");
            logEntry.className = `log-entry log-${level.toLowerCase()}`; // додає клас за
рівнем важливості (наприклад, log-error)

            logEntry.innerHTML = `
                <div class="log-header">
                    <span class="log-time">${timestamp}</span>
                    <span class="log-level">${level}</span>
                    <span class="log-type">${type}</span>
                </div>
                <div class="log-message">${message}</div>
            `;

            logContainer.prepend(logEntry); // додає новий лог у верхню частину списку
        }
        const logs = [
            {
                timestamp: new Date(Date.now() - 5 * 60 * 60 * 1000), // 5 годин тому
                level: "INFO",
                type: "system",
                message: "Система запущена успішно."
            },
            {
                timestamp: new Date(Date.now() - 5 * 60 * 60 * 1000), // 5 годин тому
                level: "INFO",
                type: "system",
                message: "Система запущена успішно."
            },
            {
                timestamp: new Date(Date.now() - 5 * 60 * 60 * 1000), // 5 годин тому
                level: "INFO",
                type: "system",
                message: "Система запущена успішно."
            },
            {
                timestamp: new Date(Date.now() - 5 * 60 * 60 * 1000), // 5 годин тому
                level: "INFO",
                type: "system",
```

```
            message: “Система запущена успішно.”
          },
          {
            timestamp: new Date(Date.now() - 30 * 60 * 1000), // 30 хв тому
            level: “ERROR”,
            type: “game”,
            message: “Помилка ініціалізації рівня.”
          },
          {
            timestamp: new Date(),
            level: “DEBUG”,
            type: “network”,
            message: “Отримано відповідь від сервера.”
          },
          {
            timestamp: new Date(),
            level: “WARNING”,
            type: “player”,
            message: “Гравець покинув гру несподівано.”
          }
        ];

        function filterLogs() {
          const timeFilter = document.getElementById(“timeFilter”).value;
          const levelFilter = document.getElementById(“levelFilter”).value;
          const typeFilter = document.getElementById(“typeFilter”).value;
          const searchQuery =
document.getElementById(“searchInput”).value.toLowerCase();

          const now = new Date();
          let timeThreshold;
          switch (timeFilter) {
            case “1h”: timeThreshold = new Date(now - 1 * 60 * 60 * 1000); break;
            case “24h”: timeThreshold = new Date(now - 24 * 60 * 60 * 1000); break;
            case “7d”: timeThreshold = new Date(now - 7 * 24 * 60 * 60 * 1000); break;
            case “30d”: timeThreshold = new Date(now - 30 * 24 * 60 * 60 * 1000); break;
            default: timeThreshold = new Date(0); break;
          }

          const filtered = logs.filter(log => {
            return (
              log.timestamp >= timeThreshold &&
              (levelFilter === “all” || log.level.toLowerCase() === levelFilter) &&
              (typeFilter === “all” || log.type === typeFilter) &&
              (log.message.toLowerCase().includes(searchQuery) ||
log.type.includes(searchQuery))
            );
          });

          displayLogs(filtered);
        }
```

```
  function displayLogs(logList) {
    const container = document.getElementById("log-entries");
    container.innerHTML = "";

    if (logList.length === 0) {
      container.innerHTML = "<p>Логів не знайдено.</p>";
      return;
    }

    logList.sort((a, b) => b.timestamp - a.timestamp); // від новіших до старіших

    for (const log of logList) {
      const entry = document.createElement("div");
      entry.className = `log-entry log-${log.level.toLowerCase()}`;
      entry.innerHTML = `
        <div class="log-header">
          <span class="log-time">${log.timestamp.toLocaleString()}</span> |
          <span class="log-level">${log.level}</span> |
          <span class="log-type">${log.type}</span>
        </div>
        <div class="log-message">${log.message}</div>
      `;
      container.appendChild(entry);
    }
  }

  window.addEventListener("DOMContentLoaded", function () {
  filterLogs(); // лише після того, як HTML завантажився
});

// Ініціалізація при завантаженні сторінки
window.onload = () => {
    updateConnectionStatus();
    loadOnlineCount();
    loadConsoleLogs();
    setInterval(updateConnectionStatus, 10000);
    setInterval(loadConsoleLogs, 15000);
};

async function fetchChartData(endpoint, callback) {
    try {
        const res = await fetch(endpoint);
        const data = await res.json();
        callback(data);
    } catch (err) {
        console.error('Помилка завантаження графіка:', endpoint, err);
    }
}

// CPU
fetchChartData('/api/cpu_chart', data => {
    new Chart(document.getElementById('cpuChart'), {
```

```
        type: 'line',
        data: {
            labels: data.labels,
            datasets: [{
                label: 'CPU (%)',
                data: data.data,
                borderColor: 'rgba(231, 76, 60, 1)',
                backgroundColor: 'rgba(231, 76, 60, 0.2)',
                fill: true
            }]
        }
    });
});

// Memory
fetchChartData('/api/memory_chart', data => {
    new Chart(document.getElementById('memoryChart'), {
        type: 'line',
        data: {
            labels: data.labels,
            datasets: [{
                label: 'Пам’ять (MB)',
                data: data.data,
                borderColor: 'rgba(41, 128, 185, 1)',
                backgroundColor: 'rgba(41, 128, 185, 0.2)',
                fill: true
            }]
        }
    });
});

// Players
fetchChartData('/api/players_chart', data => {
    new Chart(document.getElementById('playersChart'), {
        type: 'line',
        data: {
            labels: data.labels,
            datasets: [{
                label: 'Гравці',
                data: data.data,
                borderColor: 'rgba(46, 204, 113, 1)',
                backgroundColor: 'rgba(46, 204, 113, 0.2)',
                fill: true
            }]
        }
    });
});

// Network
fetchChartData('/api/network_chart', data => {
    new Chart(document.getElementById('networkChart'), {
        type: 'line',
```

```
        data: {
            labels: data.labels,
            datasets: [
                {
                    label: 'Вхідний трафік (MB/s)',
                    data: data.incoming,
                    borderColor: 'rgba(52, 152, 219, 1)',
                    backgroundColor: 'rgba(52, 152, 219, 0.2)',
                    fill: true
                },
                {
                    label: 'Вихідний трафік (MB/s)',
                    data: data.outgoing,
                    borderColor: 'rgba(155, 89, 182, 1)',
                    backgroundColor: 'rgba(155, 89, 182, 0.2)',
                    fill: true
                }
            ]
        }
    });
});
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            color: #333;
        }

        .container {
            max-width: 1400px;
            margin: 0 auto;
            padding: 20px;
        }

        .header {
            background: rgba(255, 255, 255, 0.1);
            backdrop-filter: blur(10px);
            border-radius: 20px;
            padding: 20px;
            margin-bottom: 30px;
            border: 1px solid rgba(255, 255, 255, 0.2);
        }

        .header h1 {
            color: white;
            font-size: 2.5rem;
```

```
    text-align: center;
    margin-bottom: 10px;
    text-shadow: 0 2px 4px rgba(0,0,0,0.3);
}

.header p {
    color: rgba(255, 255, 255, 0.9);
    text-align: center;
    font-size: 1.1rem;
}

.nav-tabs {
    display: flex;
    background: rgba(255, 255, 255, 0.1);
    backdrop-filter: blur(10px);
    border-radius: 15px;
    padding: 5px;
    margin-bottom: 30px;
    overflow-x: auto;
}

.nav-tab {
    flex: 1;
    padding: 12px 20px;
    background: transparent;
    border: none;
    color: rgba(255, 255, 255, 0.8);
    cursor: pointer;
    border-radius: 10px;
    transition: all 0.3s ease;
    font-weight: 500;
    min-width: 120px;
}

.nav-tab:hover {
    background: rgba(255, 255, 255, 0.1);
    color: white;
    transform: translateY(-2px);
}

.nav-tab.active {
    background: rgba(255, 255, 255, 0.2);
    color: white;
    box-shadow: 0 4px 15px rgba(0,0,0,0.2);
}

.tab-content {
    display: none;
}

.tab-content.active {
    display: block;
```

```
    animation: fadeIn 0.5s ease-in-out;
}

@keyframes fadeIn {
    from { opacity: 0; transform: translateY(10px); }
    to { opacity: 1; transform: translateY(0); }
}

.dashboard-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
    gap: 20px;
    margin-bottom: 30px;
}

.widget {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 25px;
    box-shadow: 0 8px 32px rgba(0,0,0,0.1);
    border: 1px solid rgba(255, 255, 255, 0.18);
    transition: transform 0.3s ease, box-shadow 0.3s ease;
}

.widget:hover {
    transform: translateY(-5px);
    box-shadow: 0 12px 40px rgba(0,0,0,0.15);
}

.widget-title {
    font-size: 1.3rem;
    font-weight: 600;
    margin-bottom: 15px;
    color: #2c3e50;
    display: flex;
    align-items: center;
    gap: 10px;
}

.status-indicator {
    width: 12px;
    height: 12px;
    border-radius: 50%;
    animation: pulse 2s infinite;
}

.status-normal { background: #27ae60; }
.status-warning { background: #f39c12; }
.status-critical { background: #e74c3c; }
.status-offline { background: #95a5a6; }

@keyframes pulse {
```

```
    0% { box-shadow: 0 0 0 0 rgba(39, 174, 96, 0.7); }
    70% { box-shadow: 0 0 0 10px rgba(39, 174, 96, 0); }
    100% { box-shadow: 0 0 0 0 rgba(39, 174, 96, 0); }
}

.pulse-warning {
    animation: pulse-warning 2s infinite;
}

@keyframes pulse-warning {
    0% { box-shadow: 0 0 0 0 rgba(243, 156, 18, 0.7); }
    70% { box-shadow: 0 0 0 10px rgba(243, 156, 18, 0); }
    100% { box-shadow: 0 0 0 0 rgba(243, 156, 18, 0); }
}

.pulse-critical {
    animation: pulse-critical 2s infinite;
}

@keyframes pulse-critical {
    0% { box-shadow: 0 0 0 0 rgba(231, 76, 60, 0.7); }
    70% { box-shadow: 0 0 0 10px rgba(231, 76, 60, 0); }
    100% { box-shadow: 0 0 0 0 rgba(231, 76, 60, 0); }
}

.metric-value {
    font-size: 2.5rem;
    font-weight: 700;
    color: #2c3e50;
    margin-bottom: 5px;
}

.metric-label {
    color: #7f8c8d;
    font-size: 0.9rem;
    text-transform: uppercase;
    letter-spacing: 1px;
}

.progress-bar {
    width: 100%;
    height: 8px;
    background: #ecf0f1;
    border-radius: 4px;
    margin: 15px 0;
    overflow: hidden;
}

.progress-fill {
    height: 100%;
    border-radius: 4px;
    transition: width 0.3s ease;
```

```
        }

        .progress-normal { background: linear-gradient(90deg, #27ae60, #2ecc71); }
        .progress-warning { background: linear-gradient(90deg, #f39c12, #e67e22);
}
        .progress-critical { background: linear-gradient(90deg, #e74c3c, #c0392b); }

        .logs-container {
            background: rgba(255, 255, 255, 0.95);
            border-radius: 15px;
            padding: 25px;
            margin-bottom: 20px;
        }

        .log-filters {
            display: flex;
            gap: 15px;
            margin-bottom: 20px;
            flex-wrap: wrap;
        }

        .filter-group {
            display: flex;
            flex-direction: column;
            gap: 5px;
        }

        .filter-group label {
            font-weight: 500;
            color: #2c3e50;
            font-size: 0.9rem;
        }

        .filter-input, .filter-select {
            padding: 8px 12px;
            border: 2px solid #ddd;
            border-radius: 8px;
            font-size: 0.9rem;
            transition: border-color 0.3s ease;
        }

        .filter-input:focus, .filter-select:focus {
            outline: none;
            border-color: #667eea;
        }

        .log-entry {
            display: flex;
            align-items: center;
            padding: 12px;
            margin-bottom: 8px;
            border-radius: 8px;
```

```
    border-left: 4px solid;
    transition: all 0.3s ease;
}

.log-entry:hover {
    transform: translateX(5px);
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
}

.log-entry.new-log {
    animation: slideIn 0.5s ease-out;
}

@keyframes slideIn {
    from {
        opacity: 0;
        transform: translateX(-20px);
    }
    to {
        opacity: 1;
        transform: translateX(0);
    }
}

.log-debug {
    background: #f8f9fa;
    border-left-color: #6c757d;
}

.log-info {
    background: #e3f2fd;
    border-left-color: #2196f3;
}

.log-warning {
    background: #fff3e0;
    border-left-color: #ff9800;
}

.log-error {
    background: #ffebee;
    border-left-color: #f44336;
}

.log-critical {
    background: #fce4ec;
    border-left-color: #e91e63;
}

.log-success {
    background: #e8f5e8;
    border-left-color: #4caf50;
```

```
}

.log-timestamp {
    font-family: 'Courier New', monospace;
    color: #666;
    margin-right: 15px;
    min-width: 180px;
}

.log-level {
    padding: 4px 8px;
    border-radius: 4px;
    font-size: 0.8rem;
    font-weight: 600;
    margin-right: 15px;
    min-width: 80px;
    text-align: center;
}

.level-debug { background: #6c757d; color: white; }
.level-info { background: #17a2b8; color: white; }
.level-warning { background: #ffc107; color: #333; }
.level-error { background: #dc3545; color: white; }
.level-critical { background: #e91e63; color: white; }
.level-success { background: #28a745; color: white; }

.log-message {
    flex: 1;
    color: #2c3e50;
}

.console-container {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 25px;
    margin-bottom: 20px;
}

.console {
    width: 100%;
    height: 400px;
    background-color: #1e1e1e;
    color: #dcdcdc;
    font-family: 'Courier New', monospace;
    overflow-y: auto;
    padding: 15px;
    border: 1px solid #444;
    border-radius: 8px;
    white-space: pre-wrap;
    font-size: 0.9rem;
    line-height: 1.4;
}
```

```
.console-log-entry {
    margin-bottom: 5px;
    padding: 2px 0;
}

.console-log-info {
    color: #9cdcfe;
}

.console-log-warn {
    color: #f9f1a5;
}

.console-log-error {
    color: #f44747;
}

.console-log-success {
    color: #6a9955;
}

.analytics-grid {
    display: grid;
    grid-template-columns: 1fr 1fr;
    gap: 20px;
    margin-bottom: 20px;
}

.chart-container {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 25px;
    height: 400px;
}

.chart-title {
    font-size: 1.2rem;
    font-weight: 600;
    margin-bottom: 15px;
    color: #2c3e50;
}

.settings-section {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 25px;
    margin-bottom: 20px;
}

.settings-title {
    font-size: 1.3rem;
```

```
    font-weight: 600;
    margin-bottom: 20px;
    color: #2c3e50;
    border-bottom: 2px solid #ecf0f1;
    padding-bottom: 10px;
}

.settings-group {
    margin-bottom: 20px;
}

.settings-group label {
    display: block;
    margin-bottom: 8px;
    font-weight: 500;
    color: #2c3e50;
}

.settings-input {
    width: 100%;
    padding: 12px;
    border: 2px solid #ddd;
    border-radius: 8px;
    font-size: 1rem;
    transition: border-color 0.3s ease;
}

.settings-input:focus {
    outline: none;
    border-color: #667eea;
}

.btn {
    padding: 12px 24px;
    border: none;
    border-radius: 8px;
    font-weight: 600;
    cursor: pointer;
    transition: all 0.3s ease;
    font-size: 1rem;
}

.btn-primary {
    background: linear-gradient(135deg, #667eea, #764ba2);
    color: white;
}

.btn-primary:hover {
    transform: translateY(-2px);
    box-shadow: 0 5px 15px rgba(102, 126, 234, 0.4);
}
```

```
.alert {
    padding: 15px;
    border-radius: 8px;
    margin-bottom: 20px;
    border-left: 4px solid;
}

.alert-info {
    background: #e3f2fd;
    border-left-color: #2196f3;
    color: #1976d2;
}

.alert-warning {
    background: #fff3e0;
    border-left-color: #ff9800;
    color: #f57c00;
}

.alert-error {
    background: #ffebee;
    border-left-color: #f44336;
    color: #d32f2f;
}

.connection-status {
    position: fixed;
    top: 20px;
    right: 20px;
    padding: 10px 15px;
    border-radius: 8px;
    font-weight: 500;
    z-index: 1000;
    transition: all 0.3s ease;
}

.connection-online {
    background: #d4edda;
    color: #155724;
    border: 1px solid #c3e6cb;
}

.connection-offline {
    background: #f8d7da;
    color: #721c24;
    border: 1px solid #f5c6cb;
}

@media (max-width: 768px) {
    .dashboard-grid {
        grid-template-columns: 1fr;
    }
```

```
            .analytics-grid {
                grid-template-columns: 1fr;
            }

            .log-filters {
                flex-direction: column;
            }

            .log-entry {
                flex-direction: column;
                align-items: flex-start;
                gap: 5px;
            }

            .log-timestamp, .log-level {
                min-width: auto;
            }

            .header h1 {
                font-size: 2rem;
            }

            .nav-tabs {
                flex-direction: column;
            }

            .nav-tab {
                min-width: auto;
            }
        }
        .status-indicator.status-critical { background-color: red; }
.status-indicator.status-warning { background-color: orange; }
.status-indicator.status-normal { background-color: green; }

.progress-fill.progress-critical { background-color: red; }
.progress-fill.progress-warning { background-color: orange; }
.progress-fill.progress-normal { background-color: green; }

.pulse-critical {
    animation: pulse 1s infinite;
}

@keyframes pulse {
    0% { opacity: 1; }
    50% { opacity: 0.4; }
    100% { opacity: 1; }
}
    .log-entry { border-bottom: 1px solid #ccc; padding: 8px; }
    .log-header { font-weight: bold; }
    .log-debug { background-color: #f0f0f0; }
    .log-info { background-color: #e0f7fa; }
```

```
.log-warning { background-color: #fff3cd; }
.log-error { background-color: #f8d7da; }
.log-critical { background-color: #f5c6cb; }
.filter-group { margin-right: 20px; display: inline-block; }
.filter-select, .filter-input { padding: 5px; }
.logs-container { padding: 20px; font-family: sans-serif; }
.widget-title { font-size: 20px; font-weight: bold; margin-bottom: 10px; }
```

ДОДАТОК Б

Б.1СКРІНШОТИ ІНТЕРФЕЙСУ

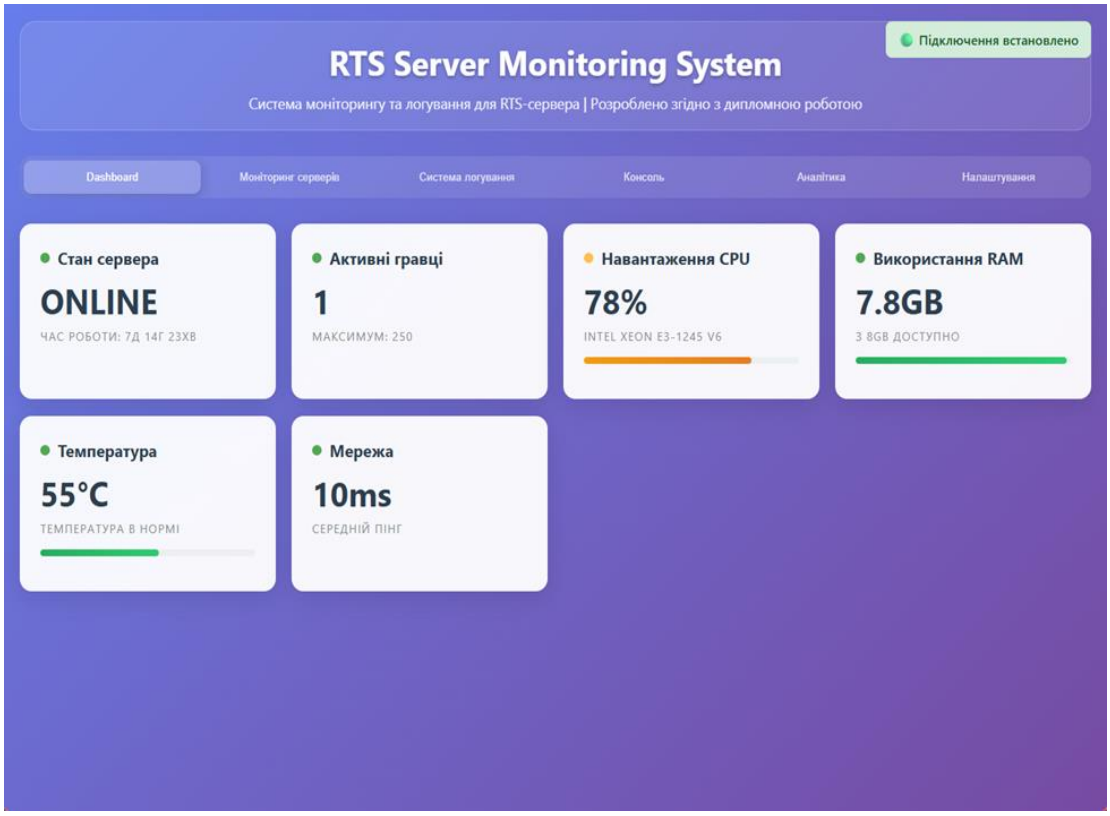

Рисунок Б.1. – Dashboard

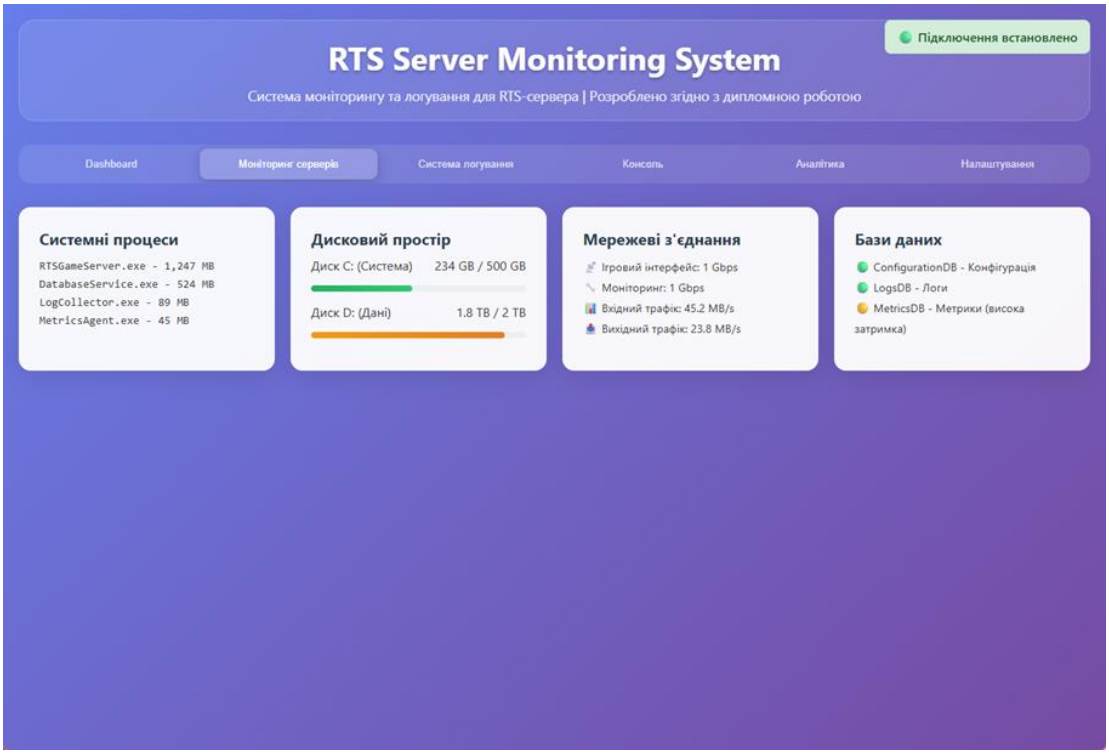

Рисунок Б.2. – Моніторинг серверів

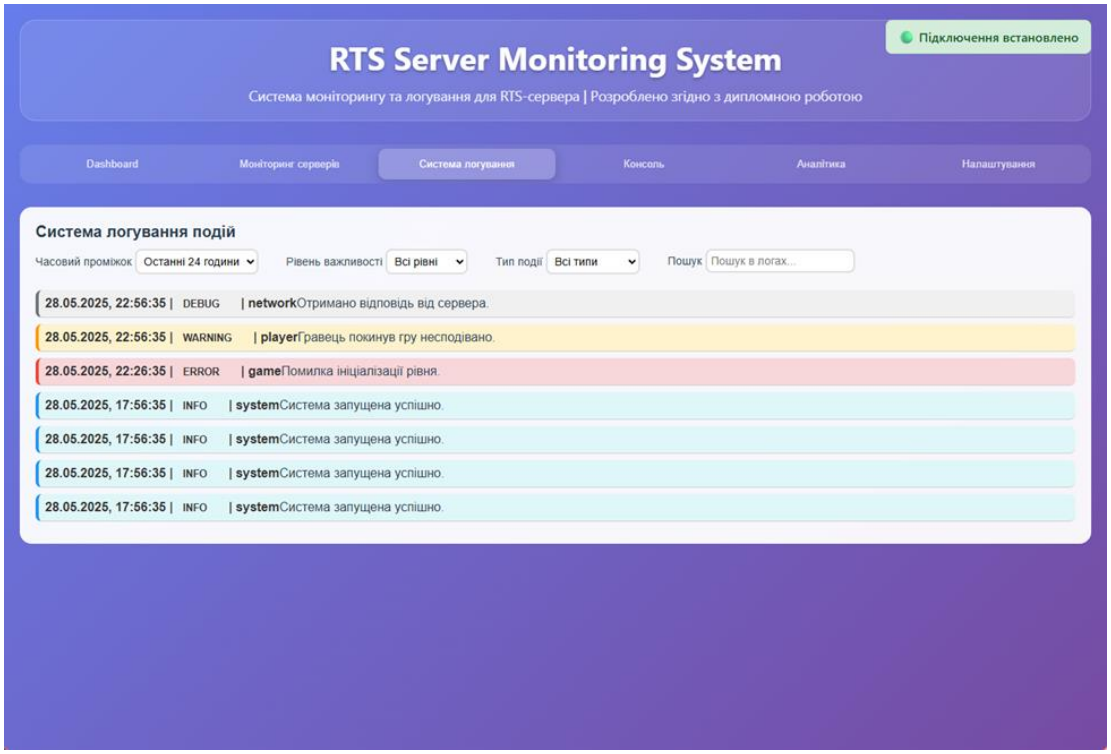

Рисунок Б.3. – Система логування

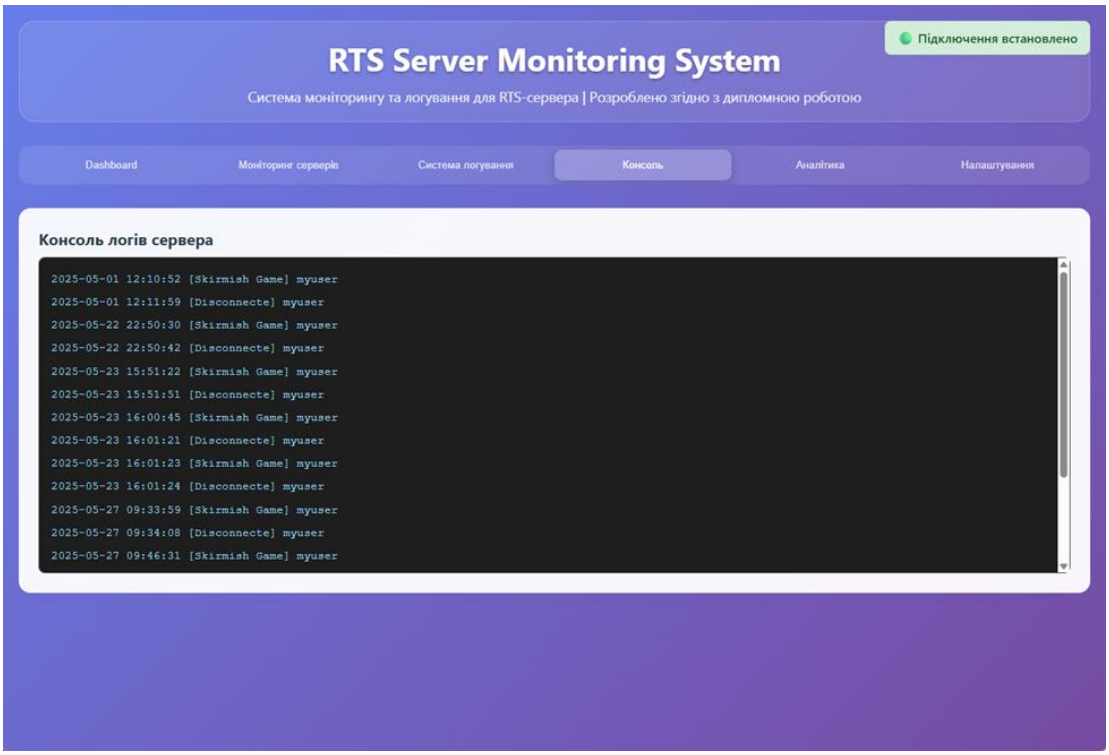

Рисунок Б.4. – Консоль

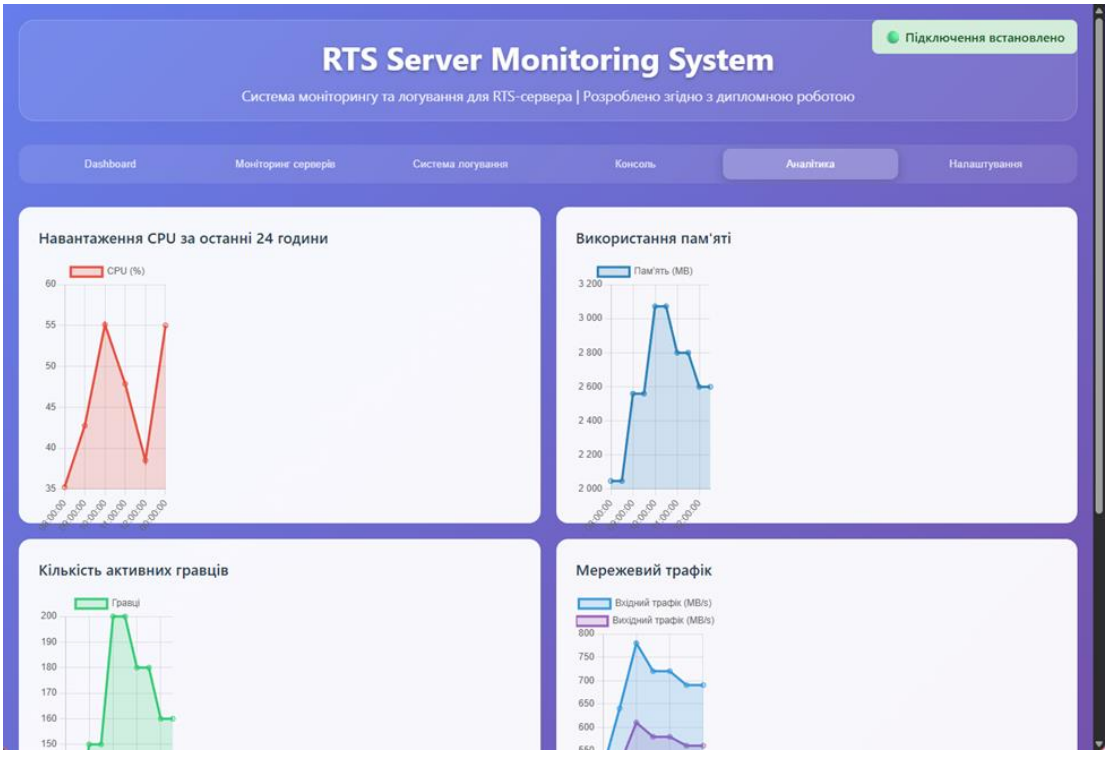

Рисунок Б.5. – Аналітика

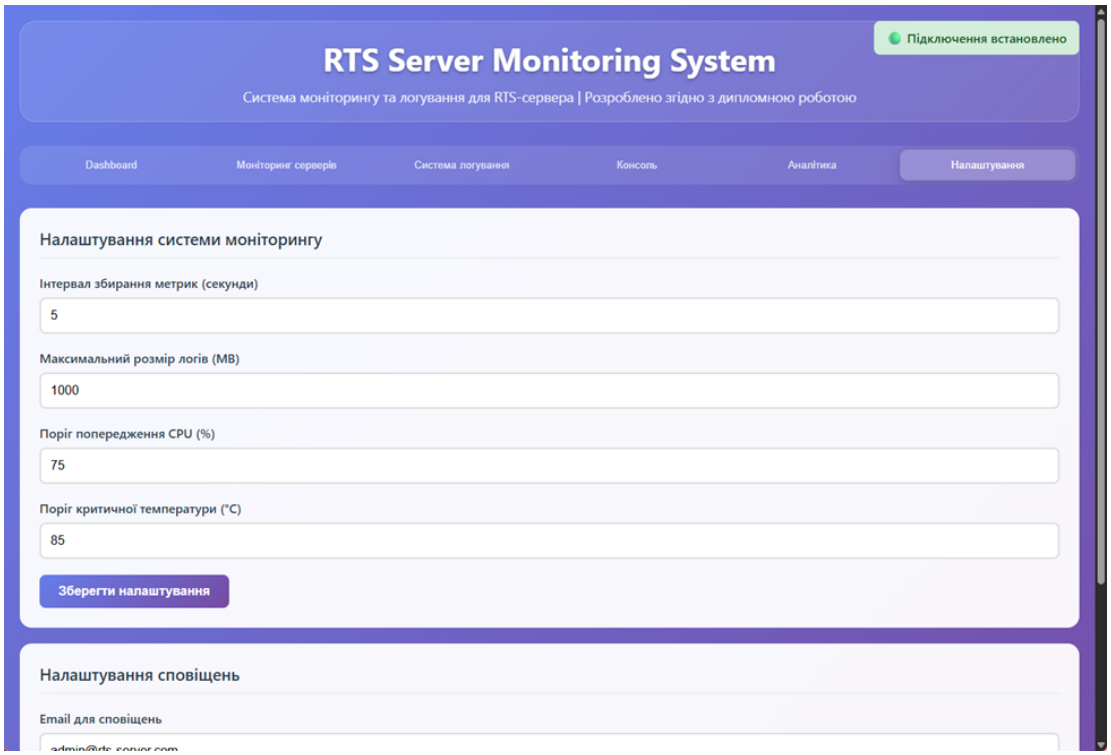

Рисунок Б.6. – Налаштування