ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка SaaS-платформи для моніторингу продуктивності додатків з Windows-агентом»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Микола ПОПСУЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Микола ПОПСУЙ

Керівник: Оксана ЗОЛОТУХІНА
канд. техн. наук, доц.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Попсую Миколі Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка SaaS-платформи для моніторингу продуктивності додатків з Windows-агентом»
керівник кваліфікаційної роботи канд. техн. наук, доц. Оксана ЗОЛОТУХІНА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про архітектуру APM-систем та методи моніторингу продуктивності додатків, специфікації системних викликів Windows для збору метрик, технічна документація до інструментів профілювання та бібліотек візуалізації даних.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної галузі та порівняльний огляд існуючих рішень для моніторингу продуктивності додатків (APM-систем).
2. Визначення вимог до системи, обґрунтування вибору технологічного стеку та проектування архітектури трирівневої APM-системи.
3. Програмна реалізація системи: розробка C++ агента для Windows, серверної частини та веб-інтерфейсу користувача.

4. Тестування та оцінка функціональності й ефективності розробленої системи в реальних умовах.

5. Перелік графічного матеріалу: *презентація*
   1. Аналіз аналогів АРМ-систем.
   2. Вимоги до програмного забезпечення.
   3. Програмні засоби реалізації.
   4. Архітектурна схема системи.
   5. Діаграма компонентів та взаємодії.
   6. Схема бази даних.
   7. Екранні форми застосунку.
   8. Апробація результатів дослідження.
6. Дата видачі завдання «23» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02 – 28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03 – 07.03.2026	
3	Огляд існуючих архітектурних підходів та технологій побудови систем моніторингу продуктивності серверів	08.03 – 21.03.2026	
4	Проектування SaaS-платформи моніторингу продуктивності Windows-серверів	22.03 – 04.04.2026	
5	Програмна реалізація платформи	05.04 – 25.04.2026	
6	Тестування платформи	26.04 – 02.05.2026	
7	Оформлення роботи: вступ, висновки, реферат	03.05 – 08.05.2026	
8	Розробка демонстраційних матеріалів	09.05 – 10.05.2026	
9	Попередній захист роботи	11.05 – 22.05.2026	
10	Подання кваліфікаційної роботи	23.05 – 30.05.2026	

Здобувач вищої освіти _____ Микола ПОПСУЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 86 стор., 7 табл., 17 рис., 30 джерел.

Мета роботи – спрощення процесу моніторингу продуктивності серверів та додатків у розподілених IT-інфраструктурах на базі операційної системи Windows шляхом розробки модульної SaaS-платформи.

Об'єкт дослідження – процес моніторингу продуктивності серверів та додатків у розподілених IT-інфраструктурах на базі операційної системи Windows.

Предмет дослідження – архітектура та методи реалізації SaaS-платформи для моніторингу продуктивності додатків і серверів у Windows-орієнтованих середовищах.

Короткий зміст роботи: У роботі проаналізовано предметну галузь APM-систем та здійснено порівняльний огляд існуючих рішень: Zabbix, PRTG Network Monitor, Datadog, Prometheus+Grafana. Визначено функціональні та нефункціональні вимоги до системи. Спроектовано та реалізовано трирівневу розподілену APM-систему у складі: C++-агента на базі WinAPI/WinHTTP для збору метрик CPU, RAM, диску та мережі; серверної частини на ASP.NET Core .NET 9 з REST API, SignalR-хабом, PostgreSQL та рольовою моделлю RBAC; React 19 SPA з дашбордом у реальному часі, Remote Shell (ConPTY + xterm.js), Remote Desktop, File Manager, Task Manager та Services Manager. Проведено функціональне та навантажувальне тестування розробленої системи.

Сферою використання розробленої системи є адміністрування та моніторинг Windows-серверів в корпоративних середовищах, дата-центрах та хмарних інфраструктурах.

КЛЮЧОВІ СЛОВА: APM-СИСТЕМА, МОНІТОРИНГ ПРОДУКТИВНОСТІ, SAAS, WINDOWS-СЕРВЕР, ДИСТАНЦІЙНЕ УПРАВЛІННЯ, WEBSOCKET, SIGNALR, REACT, ASP.NET CORE, C++, RBAC, POSTGRESQL.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ACID – Atomicity, Consistency, Isolation, Durability.

AOT – Ahead-of-Time compilation.

API – Application Programming Interface.

APM – Application Performance Monitoring.

CNCF – Cloud Native Computing Foundation.

ConPTY – Windows Console Pseudo-Terminal API.

CORS – Cross-Origin Resource Sharing.

CRUD – Create, Read, Update, Delete.

CSS – Cascading Style Sheets.

DevOps – Development and Operations.

DLL – Dynamic-Link Library.

DOM – Document Object Model.

DXGI – DirectX Graphics Infrastructure.

EF Core – Entity Framework Core.

FPS – Frames Per Second.

GDI – Graphics Device Interface.

HMAC – Hash-based Message Authentication Code.

HTTP – HyperText Transfer Protocol.

HTTPS – HyperText Transfer Protocol Secure.

ICMP – Internet Control Message Protocol.

IOPS – Input/Output Operations Per Second.

IPHLPAPI – IP Helper API.

JIT – Just-In-Time compilation.

JSON – JavaScript Object Notation.

JWT – JSON Web Token.

MTTR – Mean Time To Repair.

NAT – Network Address Translation.

ORM – Object-Relational Mapping.

PDH – Performance Data Helper.

PID – Process Identifier.

RBAC – Role-Based Access Control.

RDP – Remote Desktop Protocol.

REST – Representational State Transfer.

RFC – Request for Comments.

RPC – Remote Procedure Call.

SaaS – Software as a Service.

SLA – Service Level Agreement.

SNMP – Simple Network Management Protocol.

SPA – Single Page Application.

SSH – Secure Shell.

TLS – Transport Layer Security.

TSDB – Time-Series Database.

WinAPI – Windows Application Programming Interface.

WMI – Windows Management Instrumentation.

WSS – WebSocket Secure.

СУБД – система управління базами даних.

ВСТУП

Актуальність: Сучасна корпоративна IT-інфраструктура характеризується постійним зростанням кількості серверних вузлів та підвищенням вимог до безперервності їх роботи. Угоди про рівень обслуговування (SLA) у більшості організацій передбачають доступність критичних сервісів на рівні 99,9 % і вище, що залишає адміністраторам лічені години на усунення збоїв протягом року.

За таких умов ручний контроль стану серверів є неприйнятним: адміністратор не може фізично перебувати біля кожного вузла інфраструктури та водночас реагувати на інциденти в режимі реального часу. Виникає потреба в інструменті, який забезпечує централізований збір системних метрик та миттєву доставку даних до адміністратора — без затримок, притаманних класичним poll-моделям збору даних.

Окремою проблемою є відсутність інтеграції моніторингу з інструментами дистанційного управління. Виявивши аномалію на сервері, адміністратор змушений перемикатися між різними програмними засобами - окремим SSH-клієнтом, RDP-клієнтом, файловим менеджером - що збільшує час реакції на інцидент. Ситуацію ускладнює необхідність командної роботи: у сучасних організаціях управлінням серверною інфраструктурою займається не один адміністратор, а команда з різними рівнями відповідальності та повноважень.

Таким чином, актуальною є розробка комплексної SaaS-платформи, яка поєднує моніторинг метрик у реальному часі, інструменти дистанційного управління серверами та підтримку командної роботи з розмежуванням прав доступу. Це дозволить мінімізувати показник MTTR та забезпечити гнучке масштабування засобів адміністрування відповідно до потреб бізнесу.

Об'єктом дослідження є процес моніторингу продуктивності серверів та додатків у розподілених IT-інфраструктурах на базі операційної системи Windows.

Предметом дослідження є архітектура та методи реалізації SaaS-платформи для моніторингу продуктивності додатків і серверів у Windows-орієнтованих середовищах.

Метою роботи є спрощення процесу моніторингу продуктивності серверів та додатків у розподілених IT-інфраструктурах на базі операційної системи Windows шляхом розробки модульної SaaS-платформи класу APM.

Для досягнення поставленої мети в дипломній роботі вирішуються наступні завдання*:* провести аналіз предметної галузі та порівняльний аналіз існуючих APM-систем для виявлення їх переваг та недоліків; сформулювати функціональні та нефункціональні вимоги до розроблюваної SaaS-платформи й обґрунтувати вибір технологічного стеку для кожного з її компонентів; спроектувати архітектуру платформи з розподілом на агентський, серверний та клієнтський компоненти, а також модель бази даних з оптимізацією для часових рядів метрик; реалізувати агентський компонент для збору системних метрик і виконання команд дистанційного управління; реалізувати серверну частину; реалізувати веб-інтерфейс з відображенням метрик у реальному часі, інструментами дистанційного управління та системою проектів із рольовою моделлю доступу; провести функціональне тестування, тестування продуктивності та безпеки розробленої системи для перевірки її відповідності визначеним вимогам.

Методи дослідження: для розв'язання поставлених задач у роботі використано такі методи: методи системного та порівняльного аналізу - для дослідження предметної галузі та оцінки існуючих APM-рішень за визначеними критеріями; методи структурного аналізу та формалізації вимог - для визначення функціональних і нефункціональних характеристик системи; методи об'єктно-орієнтованого проектування та UML-моделювання - для розробки архітектури платформи та її компонентів; методи проектування реляційних баз даних - для побудови схеми зберігання даних з урахуванням специфіки часових рядів; методи модульного програмування та проектних шаблонів - для реалізації агентського, серверного та клієнтського компонентів; методи тестування

програмного забезпечення (функціональне, навантажувальне та тестування безпеки) - для верифікації відповідності розробленої системи визначеним вимогам.

Практична значущість результатів полягає у можливості використання розробленої платформи для централізованого моніторингу та оперативного адміністрування Windows-серверів в організаціях різного масштабу без необхідності розгортання власної інфраструктури. Підписна модель із трьома тарифними планами дозволяє адаптувати рішення як для індивідуальних адміністраторів, так і для корпоративних команд з розподіленою відповідальністю.

Структура роботи: кваліфікаційна робота бакалавра складається зі вступу, п'яти розділів, висновків, переліку посилань та додатків. У першому розділі проведено аналіз предметної галузі систем моніторингу продуктивності серверів, розглянуто класифікацію АРМ-систем та виконано порівняльний аналіз чотирьох найпоширеніших рішень. У другому розділі сформульовано функціональні та нефункціональні вимоги до системи й обґрунтовано вибір технологічного стеку для кожного з її компонентів. У третьому розділі описано проектування трирівневої архітектури платформи, внутрішню структуру кожного компонента та схему бази даних. Четвертий розділ присвячено програмній реалізації агента, серверної частини та веб-інтерфейсу із наведенням ключових фрагментів коду. У п'ятому розділі викладено результати функціонального тестування, тестування продуктивності та безпеки розробленої системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Поняття та призначення APM-систем

Поняття моніторингу продуктивності виникло в середині 2000-х років у відповідь на зростання складності корпоративних програмних систем. На початковому етапі APM-інструменти зосереджувалися переважно на відстеженні продуктивності застосунків на рівні програмного коду: вимірюванні часу відгуку, виявленні вузьких місць у запитах до бази даних та аналізі трасувань викликів [17]. Однак зі збільшенням масштабу IT-інфраструктур стало очевидним, що продуктивність застосунку нерозривно пов'язана зі станом серверів, на яких він виконується.

Сучасне розуміння APM охоплює ширший спектр задач і включає два взаємодоповнюючих напрями: моніторинг на рівні застосунку (трасування транзакцій, метрики часу відгуку, частота помилок) та моніторинг на рівні інфраструктури (завантаженість процесора, стан оперативної пам'яті, дискова та мережева активність) [1]. У контексті даної роботи розглядається другий напрям - моніторинг продуктивності серверів на рівні операційної системи Windows.

Ключовим завданням APM-системи є безперервний збір, зберігання та відображення системних метрик - кількісних показників, що характеризують поточний стан та продуктивність серверного вузла. Своєчасний аналіз метрик дозволяє адміністратору виявляти деградацію продуктивності до настання критичного збою, обґрунтовувати рішення щодо масштабування інфраструктури та скорочувати MTTR [17].

Залежно від характеру вимірюваного ресурсу системні метрики поділяються на чотири основні категорії, наведені в таблиці 1.1.

Таблиця 1.1

Категорії системних метрик APM-систем

Категорія	Основні показники	Метод збору (Windows)
CPU	Загальне завантаження, частота, черга процесів	PDH (Performance Data Helper), WMI
RAM	Загальний обсяг, використана/вільна пам'ять	GlobalMemoryStatusEx, PDH
Дискова підсистема	Читання/запис, IOPS, вільний простір, час очікування	GetDiskFreeSpaceEx, PDH
Мережеві інтерфейси	Вхідний/вихідний трафік, кількість пакетів	GetIfTable, GetIfEntry (IPHLPAPI)

Збір метрик у режимі реального часу є принциповою вимогою до сучасних APM-систем. Затримка між виникненням аномалії та її відображенням на панелі моніторингу безпосередньо впливає на час реакції адміністратора. З цієї причини сучасні рішення відходять від класичної моделі опитування (pull), за якої сервер моніторингу сам ініціює запити до агентів через фіксовані інтервали, на користь push-моделі, де агент самостійно надсилає дані щойно вони зібрані [17].

Сфери застосування APM-систем охоплюють широкий спектр галузей та ролей. У практиці DevOps-команд моніторинг є невід'ємною частиною конвеєра безперервного розгортання: автоматичне відстеження метрик після кожного релізу дозволяє оперативно виявляти регресії продуктивності. Фахівці з надійності сайтів (SRE - Site Reliability Engineering) використовують APM-системи для контролю виконання SLA та прогнозування потреб у масштабуванні. Системні адміністратори покладаються на інструменти моніторингу для підтримки безперервної роботи серверної інфраструктури та реагування на інциденти [17].

1.2 Класифікація систем моніторингу

Різноманіття сучасних рішень у сфері моніторингу IT-інфраструктури зумовлює необхідність їх систематизації за ключовими ознаками. Класифікація систем моніторингу дозволяє чітко окреслити характеристики розроблюваного рішення та обґрунтувати прийняті архітектурні рішення.

За моделлю збору даних системи моніторингу поділяються на два типи. При pull-моделі (опитування) центральний сервер моніторингу через визначені інтервали часу сам ініціює запити до підконтрольних вузлів для отримання поточних значень метрик. Перевагою цього підходу є простота управління конфігурацією з єдиної точки; недоліком - наявність затримки між виникненням події та її фіксацією, а також зростання навантаження на мережу при збільшенні кількості вузлів [14]. При push-моделі агент, встановлений на підконтрольному сервері, самостійно збирає метрики та надсилає їх на центральний сервер відразу після формування. Це забезпечує мінімальну затримку доставки даних та рівномірний розподіл навантаження між вузлами, що робить push-модель більш придатною для моніторингу у реальному часі [17].

За моделлю розгортання виділяють три типи систем. Хмарні рішення розміщуються на інфраструктурі постачальника та надаються за підписною моделлю; користувач отримує доступ через веб-браузер і не несе відповідальності за обслуговування серверної частини. On-premise рішення розгортаються безпосередньо на інфраструктурі замовника, що забезпечує повний контроль над даними, але вимагає власних ресурсів для підтримки. Гібридні рішення поєднують обидва підходи: агенти встановлюються локально, тоді як сервер обробки та зберігання даних може знаходитися як на стороні замовника, так і в хмарі [1].

За архітектурою агентського компонента системи поділяються на агентні та безагентні. Агентні системи передбачають встановлення на кожен підконтрольний вузол спеціалізованого програмного компонента (агента), який збирає метрики безпосередньо засобами операційної системи. Це забезпечує

доступ до низькорівневих показників та можливість виконання команд на вузлі. Безагентні системи отримують дані через стандартні мережеві протоколи (SNMP, WMI, SSH) без встановлення додаткового програмного забезпечення; вони простіші у розгортанні, однак обмежені у глибині збору метрик та не підтримують функції дистанційного управління [18].

Таблиця 1.2

Класифікація систем моніторингу IT-інфраструктури

Ознака класифікації	Тип	Характеристика
Модель збору даних	Pull	Сервер ініціює запити до агентів з фіксованим інтервалом
	Push	Агент самостійно надсилає дані на сервер
Модель розгортання	Хмарна	Сервіс надається постачальником за підпискою
	On-premise	Розгортається на інфраструктурі замовника
	Гібридна	Поєднання локального агента з хмарним сервером
Архітектура агента	Агентна	Окремий процес на кожному вузлі
	Безагентна	SNMP, WMI або SSH

Розроблювана в даній роботі платформа належить до наступних класів за кожною ознакою: push-модель збору даних, SaaS-розгортання та агентна архітектура. Такий збіг характеристик обумовлений вимогами до мінімальної затримки відображення метрик, відсутності необхідності розгортання власної серверної інфраструктури з боку користувача та доступу до низькорівневих системних показників і функцій дистанційного управління засобами операційної системи Windows.

1.3 Архітектурні підходи до побудови APM-систем

Архітектура системи моніторингу визначає спосіб організації її компонентів, протоколи їх взаємодії та характеристики масштабованості. Вибір архітектурного підходу безпосередньо впливає на затримку доставки даних, надійність системи та складність її розгортання і супроводу.

Найбільш поширеним підходом до побудови APM-систем є трирівнева архітектура, що включає три логічні рівні: рівень збору даних, рівень обробки та зберігання та рівень відображення[1].

Рівень збору даних представлений агентом - легковаговим програмним компонентом, що встановлюється на кожен підконтрольний сервер. Агент відповідає за збір системних метрик засобами операційної системи, первинну обробку даних та їх передачу на центральний сервер. Ключовою вимогою до агента є мінімальне споживання ресурсів підконтрольного вузла, оскільки будь-яке помітне навантаження з боку інструменту моніторингу спотворює відображувані показники.

Рівень обробки та зберігання реалізується центральним сервером, який приймає дані від агентів, агрегує їх, зберігає в базі даних та забезпечує доступ до них через програмний інтерфейс. Сервер також реалізує бізнес-логіку системи: автентифікацію, авторизацію, управління проектами та підписками.

Рівень відображення представлений клієнтським застосунком - як правило, веб-інтерфейсом, що отримує дані від сервера та відображає їх у зручному для адміністратора вигляді: графіки метрик, таблиці процесів, журнали подій.

Вибір між push- та pull-моделлю є одним із ключових архітектурних рішень при проектуванні APM-системи. При pull-моделі, реалізованій у таких рішеннях як Prometheus, сервер моніторингу через визначені інтервали (scrape interval) надсилає HTTP-запити до агентів і отримує поточний знімок метрик. Перевагою є централізований контроль над частотою опитування та простота виявлення недоступних вузлів. Однак затримка між подією та її реєстрацією

дорівнює щонайменше половині інтервалу опитування, що є суттєвим обмеженням для систем реального часу [14].

При push-моделі агент самостійно ініціює передачу даних одразу після їх збору. Це дозволяє досягти затримки доставки метрик у межах сотень мілісекунд незалежно від кількості підконтрольних вузлів. Крім того, push-модель краще підходить для вузлів за мережевим екраном (NAT), оскільки не вимагає вхідних з'єднань до агента з боку сервера [17]. Порівняння підходів наведено в таблиці 1.3.

Таблиця 1.3

Порівняння pull- та push-моделей збору метрик

Критерій	Pull-модель	Push-модель
Ініціатор з'єднання	Сервер	Агент
Затримка доставки даних	Залежить від інтервалу опитування	Мінімальна
Робота за NAT/firewall	Ускладнена	Без обмежень
Навантаження на мережу	Рівномірне, прогнозоване	Нерівномірне при піках активності
Виявлення недоступних вузлів	Просте	Потребує механізму heartbeat
Типові реалізації	Prometheus, Zabbix (SNMP)	Datadog Agent, Telegraf

WebSocket як транспортний протокол: Для реалізації push-моделі в умовах веб-орієнтованої архітектури оптимальним транспортним протоколом є WebSocket - стандарт RFC 6455, що забезпечує повнодуплексний канал зв'язку між клієнтом і сервером поверх єдиного TCP-з'єднання [3]. На відміну від класичного HTTP, який передбачає модель «запит–відповідь», WebSocket дозволяє серверу надсилати дані клієнту у будь-який момент без очікування

запиту. Це усуває необхідність у техніках компенсації (long polling, server-sent events), які збільшують затримку та навантаження на сервер.

З точки зору APM-системи WebSocket використовується у двох напрямах взаємодії. По-перше, агент підтримує постійне WebSocket-з'єднання з сервером і надсилає пакети метрик з визначеною частотою. По-друге, веб-клієнт підписується на оновлення метрик через WebSocket-з'єднання з сервером і отримує дані в реальному часі без необхідності перезавантаження сторінки [9]. Такий підхід забезпечує наскрізну затримку відображення метрик на рівні, що не перевищує кількох секунд від моменту їх фіксації агентом.

1.4 Порівняльний аналіз існуючих рішень

Для обґрунтування доцільності розробки власної платформи проведено аналіз чотирьох найпоширеніших рішень у сфері моніторингу IT-інфраструктури. Кожне рішення розглянуто з точки зору архітектури, функціональних можливостей, моделі розгортання та обмежень.

1.4.1 Zabbix

Zabbix - відкрита платформа корпоративного класу для моніторингу мережевої інфраструктури та серверів, розроблена Олексієм Владишевим у 2001 році та поширювана за ліцензією GPL v2. Станом на 2024 рік платформа налічує понад 70 000 активних інсталяцій у понад 100 країнах світу [18].

Архітектура Zabbix базується на трьох компонентах: сервер, агент та веб-інтерфейс. Агент встановлюється на кожен підконтрольний вузол і підтримує два режими роботи: пасивний - за замовчуванням, при якому сервер ініціює опитування агента через визначені інтервали, та активний, при якому агент самостійно надсилає дані серверу. Окрім агентного збору, Zabbix підтримує безагентний моніторинг через протоколи SNMP, ICMP, SSH та WMI [18].

Система збору метрик реалізована через механізм елементів даних та тригерів. Адміністратор визначає перелік метрик для кожного вузла, встановлює

порогові значення та налаштовує умови спрацювання сповіщень. Zabbix надає розгалужену систему шаблонів, що дозволяє стандартизувати конфігурацію моніторингу для типових вузлів інфраструктури. Для зберігання даних підтримуються MySQL, PostgreSQL та Oracle. На рисунку 1.1 наведено інтерфейс платформи Zabbix.

Рис. 1.1 Інтерфейс Zabbix

Серед ключових переваг Zabbix слід відзначити відкритий вихідний код та відсутність ліцензійних обмежень, широкий спектр протоколів збору даних, потужну систему шаблонів і тригерів, а також наявність комерційної підтримки від розробника. Підтримка розподіленого моніторингу через проксі-сервери дозволяє масштабувати систему на великі інфраструктури.

Разом з тим, Zabbix має суттєві обмеження для сценаріїв, що розглядаються в даній роботі. Платформа не надає жодних вбудованих інструментів дистанційного управління: виявивши аномалію, адміністратор змушений перемикатися до окремого SSH- або RDP-клієнта для усунення проблеми. Процес початкового налаштування є трудомістким і вимагає глибоких технічних знань. Крім того, Zabbix розповсюджується виключно як on-premise рішення - розгортання та підтримка власної серверної інфраструктури

покладається на замовника. Відсутня також вбудована підтримка командної роботи з розмежуванням ролей на рівні проектів.

1.4.2 PRTG Network Monitor

PRTG Network Monitor - комерційна система моніторингу мережевої інфраструктури та серверів, розроблена німецькою компанією Paessler AG. Перша версія продукту вийшла у 2003 році, і на сьогодні він є одним із найпоширеніших рішень корпоративного класу для Windows-середовищ [12].

Ліцензійна модель PRTG базується на кількості сенсорів - окремих одиниць моніторингу, кожна з яких відстежує один конкретний показник (наприклад, завантаженість CPU або трафік мережевого інтерфейсу). Безкоштовна версія обмежена 100 сенсорами, комерційні ліцензії починаються від 500 сенсорів. Така модель є інтуїтивно зрозумілою, проте при масштабуванні інфраструктури вартість ліцензії зростає пропорційно кількості відстежуваних показників.

Архітектурно PRTG складається з двох основних компонентів: центрального сервера (PRTG Core Server), що виконується на Windows, та зондів (probes) - агентів, які можуть розгортатися на віддалених вузлах мережі для розподіленого збору даних. Збір метрик реалізується переважно за pull-моделлю з використанням широкого спектру протоколів: WMI та Performance Counters для Windows-серверів, SNMP для мережевого обладнання, SSH для Unix-систем, а також REST API для збору даних із вебзастосунків [12].

Суттєвою перевагою PRTG є орієнтація на Windows-інфраструктуру: нативна інтеграція з WMI забезпечує детальний моніторинг Windows-серверів без додаткового налаштування. Система підтримує автоматичне виявлення пристроїв у мережі, наочні карти інфраструктури та гнучке налаштування сповіщень через електронну пошту, SMS та сторонні сервіси. На рисунку 1.2 наведено інтерфейс системи PRTG Network Monitor.

Рис. 1.2 Інтерфейс PRTG Network Monitor

До сильних сторін PRTG належать зручний веб-інтерфейс з низьким порогом входу, глибока інтеграція з Windows-інфраструктурою через WMI та Performance Counters, підтримка широкого спектру протоколів, автоматичне виявлення пристроїв і безкоштовна версія для невеликих інфраструктур.

Однак платформа має низку обмежень. Комерційна ліцензія прив'язана до кількості сенсорів, і її вартість суттєво зростає при масштабуванні інфраструктури. Як і Zabbix, PRTG не надає засобів дистанційного управління серверами та підтримує виключно on-premise розгортання. Серверний компонент функціонує лише під управлінням Windows, що обмежує гнучкість розгортання. Відсутня підтримка командної роботи з рольовою моделлю доступу.

1.4.3 Datadog

Datadog - хмарна SaaS-платформа для моніторингу інфраструктури, застосунків та безпеки, заснована у 2010 році в Нью-Йорку. Компанія вийшла на біржу NASDAQ у 2019 році та станом на 2024 рік обслуговує понад 29 000

корпоративних клієнтів по всьому світу, що робить її одним із лідерів ринку хмарного моніторингу [2].

На відміну від розглянутих вище on-premise рішень, Datadog реалізує повноцінну SaaS-модель: уся серверна інфраструктура знаходиться на стороні постачальника, а клієнт отримує доступ до платформи через веб-браузер. На підконтрольні сервери встановлюється легковаговий агент (Datadog Agent), написаний мовою Go, який збирає системні метрики, трасування запитів та журнали і надсилає їх до хмарної інфраструктури Datadog за push-моделлю [2]. Агент підтримує широкий спектр операційних систем, включаючи Windows, Linux та macOS, а також середовища контейнеризації Docker і Kubernetes.

Функціональність платформи організована у вигляді модулів, що підключаються окремо: Infrastructure Monitoring, APM & Distributed Tracing, Log Management, Security Monitoring та Synthetic Monitoring. Така модульність дозволяє замовнику формувати набір функцій відповідно до потреб, проте кожен додатковий модуль збільшує вартість підписки. Базовий тариф на моніторинг інфраструктури становить від 15 до 23 доларів США за хост на місяць залежно від обраного плану [2].

Окремою конкурентною перевагою Datadog є система штучного інтелекту Watchdog, яка автоматично виявляє аномалії в метриках та журналах без необхідності налаштування порогових значень. Платформа також надає конструктор інтерактивних панелей моніторингу з понад 700 готовими інтеграціями з популярними сервісами та інструментами DevOps. Широкі можливості кореляції метрик, трасувань і журналів в єдиному інтерфейсі роблять Datadog одним із найбільш комплексних рішень на ринку для команд, що працюють із хмарною та мікросервісною інфраструктурою.

Платформа вирізняється зрілістю, широким набором інтеграцій та потужними аналітичними можливостями, орієнтованими на великі DevOps-команди та хмарну інфраструктуру. Механізм автоматичного виявлення аномалій Watchdog суттєво знижує операційне навантаження на адміністраторів у середовищах із великою кількістю сервісів.

Разом з тим, для сценаріїв моніторингу Windows-серверів малого та середнього бізнесу Datadog має ряд суттєвих обмежень. Вартість підписки є значною перешкодою: при інфраструктурі у 20–50 серверів щомісячні витрати можуть сягати кількох тисяч доларів США, що є невиправданим для організацій, яким не потрібні можливості APM-трасування та аналізу мікросервісів. Платформа не надає жодних інструментів дистанційного управління серверами: виявлення проблеми в Datadog і її усунення потребують переходу до окремих інструментів. Усі зібрані метрики та журнали передаються до хмарної інфраструктури постачальника, що може бути неприйнятним для організацій з обмеженнями щодо трансскордонної передачі даних. Нарешті, орієнтація платформи на хмарні та контейнеризовані середовища означає, що частина її функціональності є надлишковою для адміністраторів, які працюють виключно з Windows-серверами. На рисунку 1.3 наведено інтерфейс платформи Datadog.

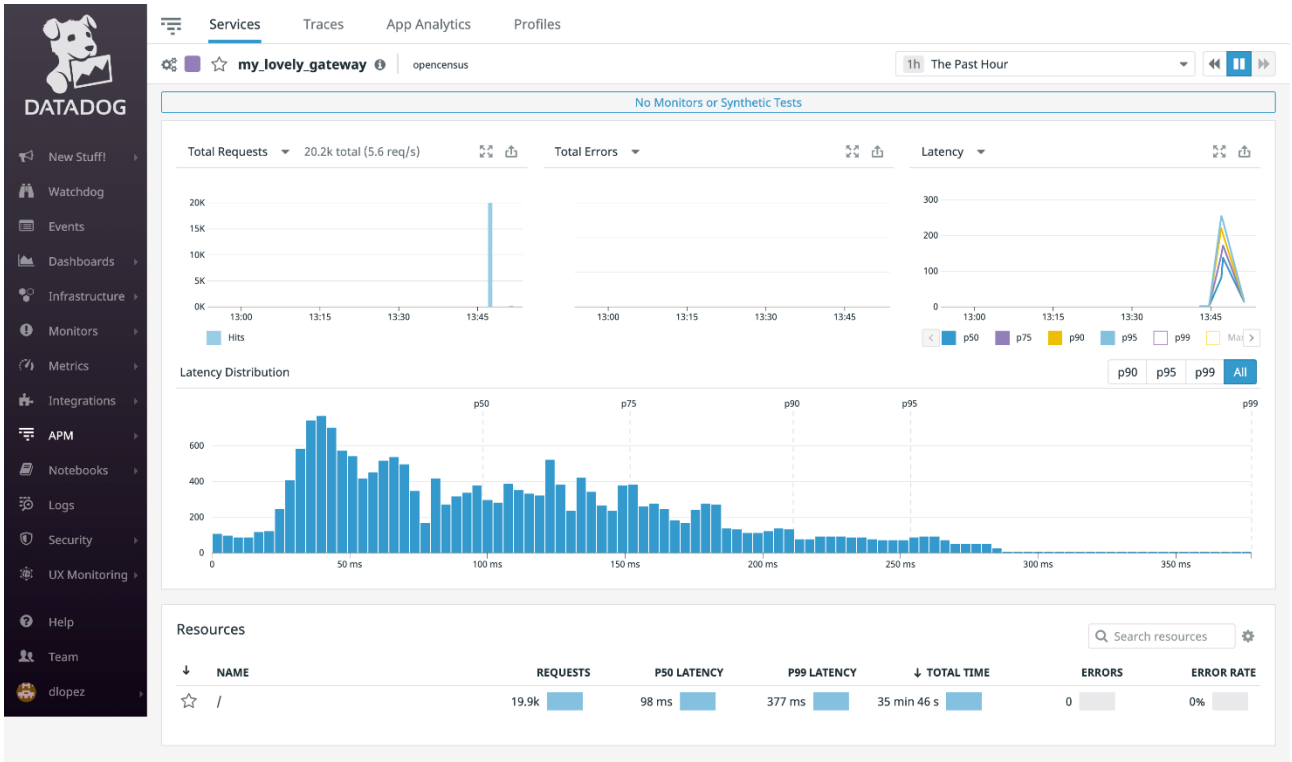


Рис. 1.3 Інтерфейс Datadog

1.4.4 Prometheus + Grafana

Prometheus - відкрита система моніторингу та сповіщень, розроблена компанією SoundCloud у 2012 році та передана під управління Cloud Native Computing Foundation (CNCF) у 2016 році. Разом із Grafana - платформою візуалізації даних, заснованою у 2014 році Торкелем Еддегором, - вони утворюють один із найпопулярніших стеків моніторингу у DevOps-спільноті, особливо у середовищах Kubernetes та мікросервісних архітектурах [14, 5].

Prometheus реалізує pull-модель збору даних: центральний сервер через визначені інтервали (scrape interval) надсилає HTTP-запити до так званих exporters - легковагових агентів, що перетворюють метрики операційної системи або застосунку у формат, придатний для читання Prometheus. Для збору метрик Windows-серверів використовується Windows Exporter (раніше відомий як wmi_exporter), який надає дані про завантаженість процесора, пам'яті, дискової підсистеми та мережевих інтерфейсів через WMI [14]. Зібрані метрики зберігаються у вбудованій часовій базі даних (TSDB) з ефективним стисненням даних, оптимізованою для зберігання числових рядів з часовими мітками.

Потужність Prometheus значною мірою визначається мовою запитів PromQL (Prometheus Query Language), яка дозволяє виконувати складні агрегації, обчислення похідних та прогнозування на основі зібраних метрик. Alertmanager - окремий компонент стеку - відповідає за маршрутизацію та дедублікацію сповіщень, відправляючи їх до електронної пошти, Slack, PagerDuty та інших систем [14].

Grafana виступає рівнем візуалізації: вона підключається до Prometheus як джерела даних і надає конструктор інтерактивних панелей моніторингу з широким набором типів графіків, таблиць та індикаторів. Grafana підтримує понад 150 джерел даних, що дозволяє об'єднувати метрики з Prometheus, журнали з Loki, трасування з Tempo та дані з реляційних баз даних в єдиному інтерфейсі [5]. Бібліотека готових панелей моніторингу (Grafana Dashboards) налічує тисячі шаблонів, опублікованих спільнотою. На рисунку 1.4 наведено інтерфейс системи Grafana.

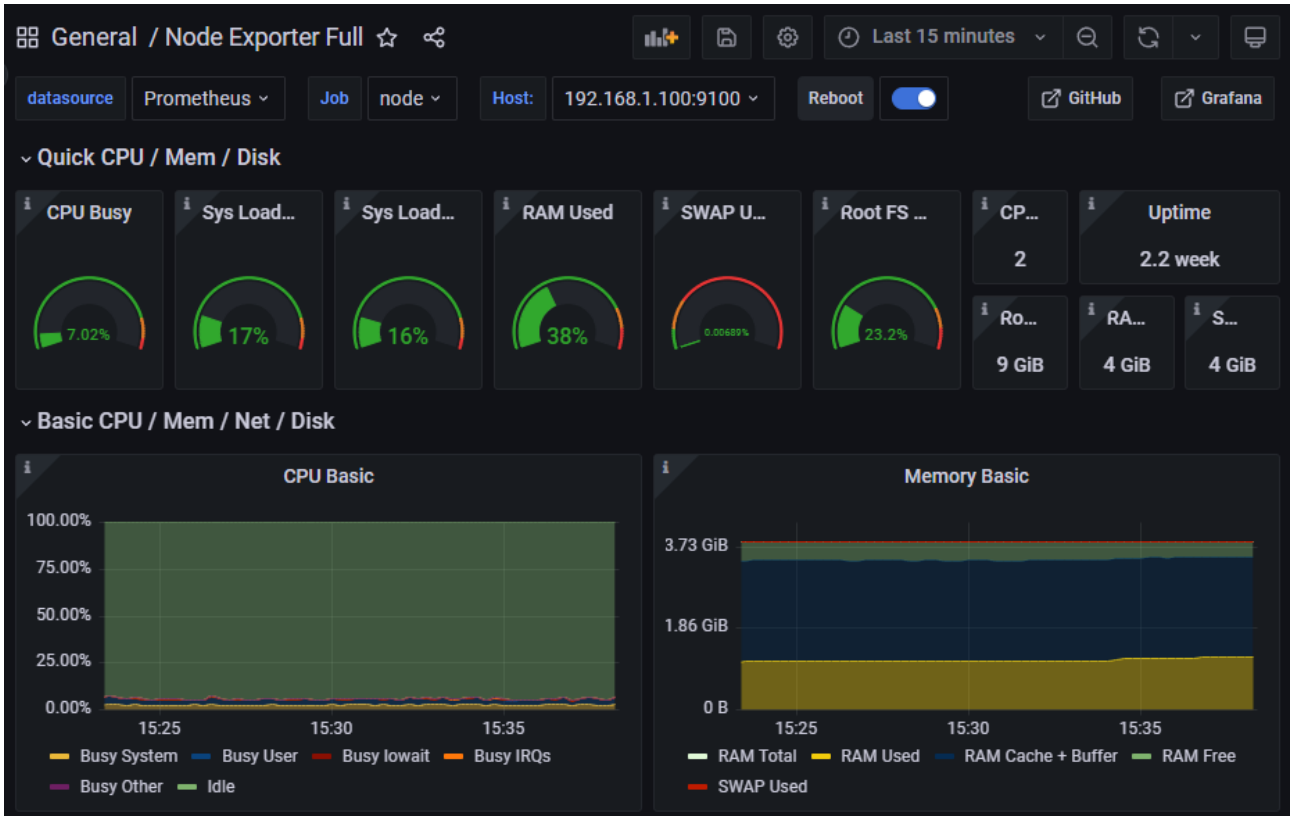


Рис. 1.4 Інтерфейс Grafana

Стек Prometheus + Grafana вирізняється винятковою гнучкістю та розширюваністю: завдяки відкритому коду, великій екосистемі exporters та підтримці численних джерел даних він адаптується до практично будь-якого середовища моніторингу. Висока зрілість проектів, активна спільнота та статус проектів CNCF гарантують довгострокову підтримку та розвиток.

Водночас для сценаріїв, що розглядаються в даній роботі, стек має суттєві обмеження. Розгортання та налаштування виробничого середовища вимагає значних зусиль: окремо необхідно встановити та сконфігурувати Prometheus, Alertmanager, Windows Exporter на кожному сервері та Grafana, забезпечити їх відмовостійкість і резервне копіювання. Pull-модель збору даних створює затримку між виникненням події та її відображенням, рівну щонайменше половині інтервалу опитування. Як і інші розглянуті рішення, стек Prometheus + Grafana не надає жодних засобів дистанційного управління серверами. Базова версія Grafana має обмежені можливості розмежування доступу, а повноцінна

рольова модель доступна лише у комерційній версії Grafana Enterprise. Нарешті, відсутність єдиного агента, який поєднував би збір метрик з можливостями дистанційного управління, унеможливлює реалізацію сценарію «виявив проблему - відразу усунув» в рамках одного інструменту.

1.5 Підсумки порівняльного аналізу

На основі проведеного огляду чотирьох APM-систем складено порівняльну характеристику, яка дозволяє системно оцінити їх відповідність потребам розроблюваної платформи.

Для зіставлення обрано п'ять ключових критеріїв, що безпосередньо випливають із виявлених на ринку запитів: можливість збору метрик у реальному часі, спосіб розгортання серверної частини, наявність агентного збору даних, підтримка рольової моделі доступу в межах командних проектів та наявність інструментів дистанційного виконання команд на підконтрольних серверах. Узагальнені результати наведено в таблиці 1.4.

Таблиця 1.4

Порівняльний аналіз існуючих рішень для моніторингу IT-інфраструктури

Критерій	Datadog	PRTG	Zabbix	Prometheus+Grafana
Збір метрик у реальному часі	+	+	+	+
Без налаштування сервера	+	-	-	-
Агентний збір даних	+	+	+	+

Продовження таблиці 2.4

Порівняльний аналіз існуючих рішень для моніторингу IT-інфраструктури

Рольова модель доступу в проектах	+	-	-	-
Віддалене виконання команд	-	-	-	-

Аналіз таблиці дозволяє виокремити кілька важливих закономірностей. По-перше, всі чотири системи підтримують збір метрик у реальному часі та використовують агентний підхід. По-друге, рішення чітко поділяються на два класи: хмарні комерційні платформи; відкриті on-premise, але вимагають розгортання та супроводження власної інфраструктури. По-третє, жодне з розглянутих рішень не надає вбудованих інструментів дистанційного виконання команд на підконтрольних.

2 ВИМОГИ ТА ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Функціональні вимоги

Функціональні вимоги визначають перелік операцій та поведінку системи з точки зору кінцевого користувача. На основі висновків порівняльного аналізу, проведеного в першому розділі, сформульовано перелік таких вимог до розроблюваної платформи. Перед детальним описом вимог наведемо склад акторів, які взаємодіють із системою, оскільки повноваження кожного з них напряму впливають на доступний набір функцій.

Платформа підтримує два контексти роботи користувача: особистий контекст у якому користувач взаємодіє лише з власними агентами без участі інших осіб, та проектний контекст, у якому декілька користувачів спільно адмініструють підконтрольні сервери в межах проекту з розмежуванням повноважень за ролями. Залежно від контексту користувач виступає в одній із описаних нижче ролей. На рисунку 2.1 наведено діаграму варіантів використання системи.

Користувач - власник облікового запису, який працює з агентами, зареєстрованими безпосередньо на нього, без передавання їх до проекту. У цьому контексті він має повний доступ до всіх функцій системи, обмежений лише рівнем своєї підписки: може реєструвати агенти, переглядати їх метрики, виконувати команди дистанційного управління та керувати власною підпискою. Поняття “роль” в особистому контексті не застосовується, оскільки користувач є одночасно й власником, і єдиним адміністратором своїх агентів.

У проектному контексті виділено чотири ролі з ієрархічним успадкуванням повноважень. Viewer - учасник проекту з мінімальним рівнем доступу. Operator успадковує всі повноваження Viewer і додатково отримує можливість використовувати інструменти дистанційного.

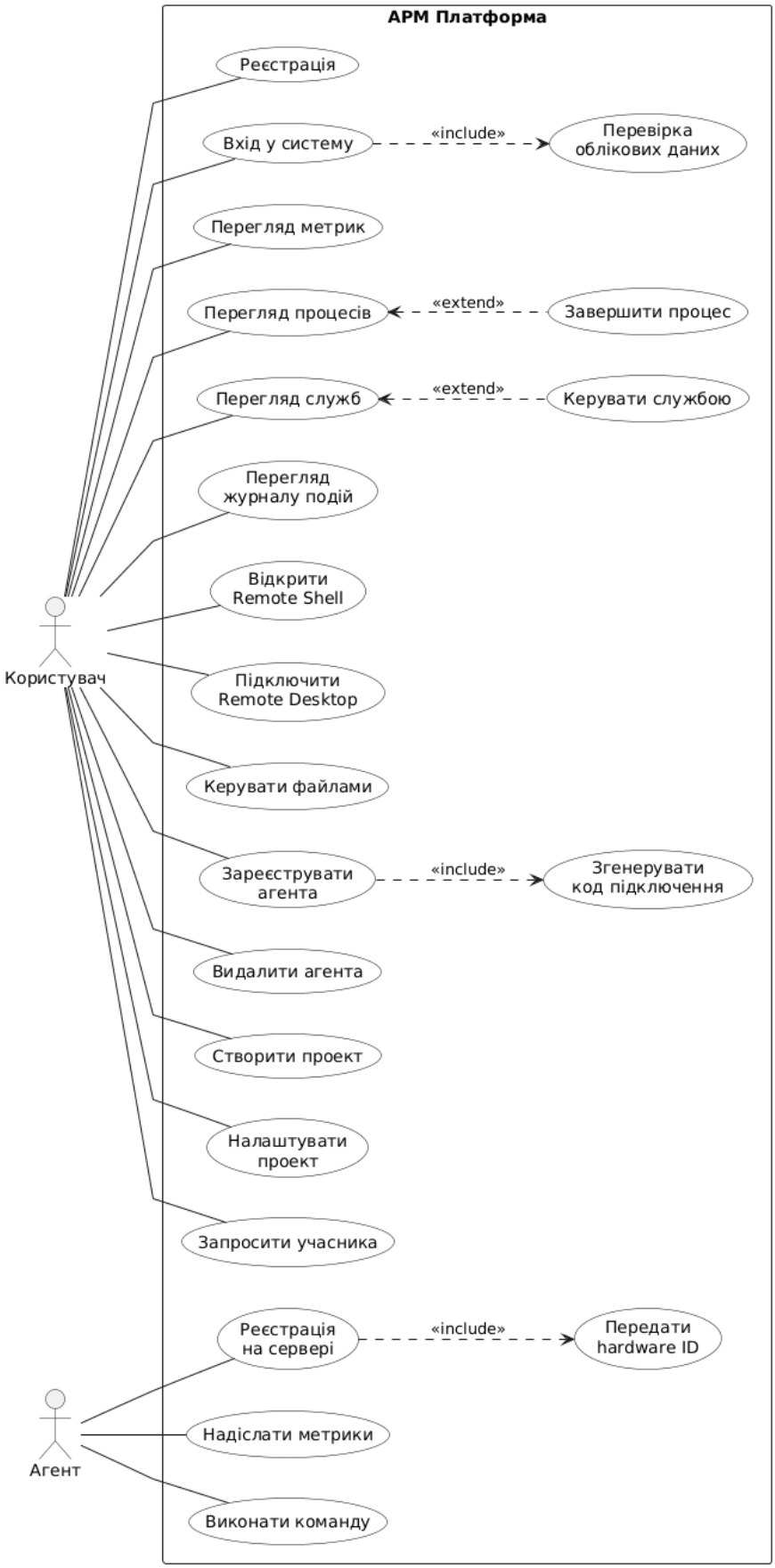


Рис. 2.1 Діаграма варіантів використання,
що відображає взаємодію акторів із системою

Admin успадковує всі повноваження Operator і відповідає за організаційне управління проектом. Owner успадковує всі повноваження Admin і має право на налаштування параметрів проекту, керування підпискою проекту та його видалення; ним автоматично стає користувач, який створив проект.

Агент - системний актор, що представляє програмний компонент, встановлений на підконтрольному Windows-сервері. Виконує реєстрацію в системі за кодом підключення, надсилає зібрані системні метрики на сервер у режимі реального часу та виконує команди дистанційного управління, отримані від нього.

На основі описаного складу акторів сформульовано такі функціональні вимоги до системи.

Управління обліковими записами - система повинна забезпечувати реєстрацію користувачів із верифікацією електронної пошти, автентифікацію за логіном та паролем з видачею JWT-токена, а також оновлення токена доступу без повторної автентифікації. Зберігання паролів має здійснюватися виключно у вигляді хешу з використанням алгоритму bcrypt.

Управління проектами та командою - користувач повинен мати можливість створювати проекти для групування підконтрольних серверів, запрошувати інших користувачів до проекту електронною поштою та призначати їм одну з чотирьох ролей. Система повинна підтримувати зміну ролей учасників та їх видалення з проекту. Рівень доступу до функцій у проекті визначається підпискою його власника.

Підключення та управління агентами - система повинна забезпечувати реєстрацію агента на сервері за унікальним кодом підключення та апаратним ідентифікатором. Зареєстрований агент має відображатися у списку агентів проекту зі статусом онлайн/офлайн. Користувач повинен мати можливість перейменовувати агенти та видаляти їх із проекту.

Моніторинг метрик у реальному часі - агент повинен збирати та надсилати на сервер системні метрики з інтервалом не більше однієї секунди:

завантаженість процесора (загальна та по ядрах), використання оперативної пам'яті, завантаженість дискової підсистеми (читання/запис), мережевий трафік (вхідний/вихідний) та перелік активних процесів із показниками споживання ресурсів. Веб-інтерфейс повинен відображати метрики у реальному часі без перезавантаження сторінки.

Дистанційне управління серверами - система повинна надавати такі інструменти дистанційного управління: командний інтерпретатор (Remote Shell) з підтримкою повноцінного псевдотермінала, захоплення екрану сервера (Remote Desktop) у режимі перегляду та управління, файловий менеджер із можливістю перегляду структури каталогів та завантаження файлів, менеджер служб Windows із можливістю запуску, зупинки та перезапуску служб, а також менеджер процесів із можливістю завершення процесів.

Підписна модель - Система повинна підтримувати підписки з різним набором доступних функцій. Перехід між рівнями підписки має відбуватися через інтерфейс налаштувань.

2.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи - безпеку, продуктивність, надійність та умови розгортання, - які не описують конкретну поведінку, але суттєво впливають на придатність системи до реального використання.

Безпека - уся взаємодія між компонентами системи має здійснюватися виключно по захищених каналах: HTTPS для REST API та WSS для з'єднань агента і клієнта з сервером. Автентифікація користувачів реалізується за схемою JWT (JSON Web Token) відповідно до стандарту RFC 7519 [6]: токен доступу має обмежений термін дії та передається у заголовку Authorization кожного запиту. Паролі користувачів зберігаються виключно у вигляді хешу, обчисленого алгоритмом bcrypt [15], що унеможливлює їх відновлення у разі компрометації

бази даних. Для агентів використовується окрема схема автентифікації на основі токенів, що видаються під час реєстрації.

Продуктивність - затримка між збором метрики агентом та її відображенням у веб-інтерфейсі не повинна перевищувати 5 секунд за нормальних умов мережі. Час відповіді REST API на типові запити не повинен перевищувати 500 мілісекунд. Агент розроблен без зовнішніх залежностей, завантаження центрального процесора підконтрольного сервера агентом не повинно перевищувати 2% у штатному режимі роботи, споживання оперативної пам'яті агентом не повинно перевищувати 50 МБ, розмір виконуваного файлу агента не повинен перевищувати 2 МБ для забезпечення швидкого розгортання. Для забезпечення стабільної передачі даних агент використовує буферизацію метрик: у разі тимчасової недоступності сервера накопичені дані надсилаються після відновлення з'єднання.

Надійність - агент повинен автоматично відновлювати WebSocket-з'єднання з сервером після його переривання без втрати зібраних метрик. Сервер повинен коректно обробляти одночасні підключення від множини агентів та клієнтів без деградації продуктивності.

Розгортання - серверна частина та веб-інтерфейс розгортаються як SaaS-сервіс і не вимагають від користувача жодних дій з налаштування серверної інфраструктури. Агент розповсюджується у вигляді єдиного виконуваного файлу без зовнішніх залежностей і встановлюється на Windows-сервер з мінімальними привілеями, необхідними для збору системних метрик.

Зручність використання - веб-інтерфейс повинен коректно відображатися у сучасних браузерах без встановлення додаткових розширень. Час первинного завантаження інтерфейсу не повинен перевищувати 3 секунди за умови широкосмугового підключення. Реєстрація нового агента на сервері має виконуватися адміністратором не більш ніж за 3 кроки через веб-інтерфейс.

2.3 Огляд та обґрунтування вибору засобів реалізації

Вибір технологічного стеку для розробки SaaS-платформи моніторингу базується на сформульованих у попередніх підрозділах функціональних та нефункціональних вимогах. Розроблювана система складається з трьох незалежних компонентів - агента для Windows-серверів, серверної частини та веб-інтерфейсу, - кожен з яких має різну операційну середу та різні вимоги до продуктивності, що зумовлює необхідність обґрунтованого вибору засобів реалізації для кожного з них.

2.3.1 Засоби реалізації агентського компонента

Агент - це програмний компонент, що встановлюється на кожен підконтрольний сервер і відповідає за збір системних метрик та виконання команд дистанційного управління. Згідно з нефункціональними вимогами, агент повинен мати мінімальний вплив на ресурси сервера, не потребувати додаткових залежностей і забезпечувати прямий доступ до низькорівневих API операційної системи Windows.

Серед потенційних мов програмування для реалізації агентського компонента розглядалися такі варіанти:

1. C# з фреймворком .NET у режимі AOT-компіляції. Перевагою є зручний доступ до WMI та Performance Counters через стандартну бібліотеку, проте навіть AOT-збірка потребує наявності частини .NET-середовища і має більший розмір виконуваного файлу порівняно з нативним кодом.
2. Go забезпечує крос-платформну компіляцію в один виконуваний файл та вбудовану підтримку конкурентності, але виконання Go-програм передбачає роботу збирача сміття, що періодично спричиняє паузи виконання та ускладнює прогнозування навантаження. Крім того, інтеграція з нативними WinAPI-функціями потребує обгорток (cgo або сторонніх бібліотек), що знижує продуктивність.

3. Rust як сучасна альтернатива C++ з гарантіями безпеки пам'яті. Має зрілу підтримку WinAPI через crate-и (windows-rs), однак екосистема для роботи з PDH та ConPTY поки менш зріла, що ускладнює реалізацію функцій моніторингу та дистанційного управління.
4. Python, Node.js та інші скриптові середовища відхилено через високе споживання пам'яті, повільність виконання та необхідність розгортання інтерпретатора на підконтрольному сервері, що суперечить вимогам до автономності агента.

З урахуванням наведених міркувань для реалізації агентського компонента обрано мову **C++** зі стандартом C++17. Цей вибір забезпечує: нативну компіляцію в єдиний виконуваний файл без зовнішніх залежностей; прямий доступ до повного спектру WinAPI-функцій без обгорток; повний контроль над використанням пам'яті без накладних витрат на роботу збирача сміття; мінімальний розмір виконуваного файлу та низьке споживання ресурсів підконтрольного сервера.

Для взаємодії з операційною системою Windows використовується набір функцій WinAPI [10], зокрема такі підсистеми:

1. PDH - стандартний інтерфейс отримання значень лічильників продуктивності Windows. Через PDH здійснюється збір метрик завантаженості процесора (\Processor(_Total)\% Processor Time), активності дискової підсистеми (\PhysicalDisk(_Total)\Disk Read Bytes/sec, \PhysicalDisk(_Total)\Disk Write Bytes/sec) та мережевих інтерфейсів (\Network Interface(*)\Bytes Sent/sec, \Network Interface(*)\Bytes Received/sec).
2. GlobalMemoryStatusEx - функція отримання інформації про стан оперативної пам'яті: загальний обсяг, використана пам'ять, доступна пам'ять, рівень завантаженості.
3. Toolhelp32 - інтерфейс перерахування активних процесів операційної системи. OpenProcess, GetProcessTimes та OpenProcessToken забезпечує

отримання повної інформації про процеси: ідентифікатор, ім'я виконуваного файлу, ім'я користувача-власника, споживання процесорного часу та обсягу пам'яті.

4. IPHLPAPI (GetAdaptersAddresses) - функція отримання інформації про мережеві адаптери системи, включаючи їх імена, IP- та MAC-адреси.

Для мережевої взаємодії з серверною частиною використовується бібліотека WinHTTP - вбудований у Windows стек HTTP-клієнта, що підтримує як звичайні HTTP/HTTPS-запити, так і протокол WebSocket згідно з RFC 6455 [3]. Перевагою WinHTTP перед сторонніми бібліотеками (libcurl, IXWebSocket) є відсутність необхідності лінкування зовнішніх залежностей: бібліотека winhttp.lib входить до складу стандартного SDK Windows. Це дозволяє отримати єдиний виконуваний файл агента без додаткових DLL.

Для протоколювання роботи агента на стороні підконтрольного сервера використовується бібліотека spdlog - швидкий та потокобезпечний логер з відкритим вихідним кодом, що забезпечує запис діагностичної інформації у локальні файли з ротацією за розміром.

2.3.2 Засоби реалізації серверної частини

Серверна частина платформи відповідає за прийом метрик від агентів, обробку запитів від веб-клієнтів, автентифікацію та авторизацію користувачів, управління підписками, проектами і ролями, а також за маршрутизацію команд дистанційного управління між клієнтами та агентами. До серверної частини висуваються підвищені вимоги щодо продуктивності, асинхронності обробки запитів та підтримки великої кількості одночасних з'єднань.

При виборі технологічного стеку для серверної частини розглядалися такі альтернативи:

1. Node.js з фреймворком Express або Fastify забезпечує високу швидкість розробки та зручну роботу з JSON, проте однопотокова модель JavaScript-

середовища не оптимальна для CPU-інтенсивних операцій, а слабка типізація мови ускладнює супроводження кодової бази великого обсягу.

2. Python з фреймворком Django або FastAPI має багату екосистему та зручний синтаксис, але поступається у продуктивності компільованим мовам та потребує спеціальних механізмів (uvicorn з gunicorn, ASGI) для забезпечення повноцінної асинхронності.
3. Java з фреймворком Spring Boot — зрілий промисловий стек з потужною екосистемою, проте має значне споживання пам'яті JVM-середовищем та більший “boilerplate” коду порівняно з сучасними альтернативами.
4. Go з фреймворками Gin або Echo демонструє високу продуктивність та просту модель конкурентності через goroutines, проте обмежені можливості об'єктно-орієнтованого програмування ускладнюють організацію складної бізнес-логіки з багаторівневою авторизацією та політиками доступу.

Для реалізації серверної частини обрано фреймворк ASP.NET Core .NET 9 [8] - кросплатформну платформу від Microsoft з відкритим вихідним кодом. Цей вибір обґрунтовується наступними перевагами:

1. Висока продуктивність. За даними тестів TechEmpower, ASP.NET Core стабільно входить до топ-10 найшвидших веб-фреймворків. Версія .NET 9 містить додаткові оптимізації JIT-компіляції, AOT-збірки та обробки HTTP/2 і HTTP/3.
2. Вбудований сервер Kestrel - кросплатформний асинхронний веб-сервер на основі бібліотеки libuv, що ефективно обробляє велику кількість одночасних з'єднань без необхідності розгортання окремого reverse-proxy для базових сценаріїв.
3. Dependency Injection забезпечує чистий поділ компонентів за принципом інверсії контролю: контролери, сервіси, репозиторії, middleware-компоненти реєструються в контейнері та автоматично надаються при потребі. Це спрощує тестування та підтримку коду.

4. Конвеєр middleware-компонентів дозволяє декларативно налаштовувати обробку запитів: автентифікацію, авторизацію, кешування, обробку помилок. У розроблюваній системі такий підхід застосовано для реалізації власних компонентів ProjectContextMiddleware та AgentAuthMiddleware.
5. Атрибутна авторизація через декоратори дозволяє лаконічно описувати правила доступу безпосередньо на рівні endpoint-ів контролерів.

REST API серверної частини спроектовано згідно з архітектурним стилем REST [4]: кожен ресурс має власний URI, операції над ресурсами виконуються через стандартні HTTP-методи, стан між запитами не зберігається на сервері (stateless), а автентифікація здійснюється через JWT-токен у заголовку Authorization.

2.3.3 Засоби зберігання даних

База даних повинна забезпечувати надійне зберігання структурованих даних - користувачів, проектів, агентів, підписок, а також високопродуктивний запис великих обсягів метрик у режимі реального часу. Усі дані повинні бути узгодженими, а БД має підтримувати транзакційну цілісність ACID при одночасних операціях запису від множини агентів. Окремою вимогою є ефективне зберігання часових рядів метрик з можливістю автоматичного видалення застарілих даних.

Серед розглянутих варіантів:

1. MySQL - поширена реляційна СУБД з відкритим кодом, проте поступається PostgreSQL за функціональністю (відсутність повноцінної підтримки JSONB-індексів у старіших версіях, обмежені можливості віконних функцій) та не має зрілих розширень для оптимізації роботи з часовими рядами.
2. MongoDB як документоорієнтована БД підходить для зберігання неструктурованих даних, проте відсутність повноцінних транзакцій між

колекціями та слабка узгодженість моделі ускладнює реалізацію бізнес-логіки з багатокомпонентними операціями.

3. nfluxDB це спеціалізована СУБД для часових рядів, оптимізована для зберігання метрик. Однак вона не підтримує реляційну модель і не може зберігати основні бізнес-дані, що змушує використовувати дві різні бази одночасно та ускладнює архітектуру системи.

Для розроблюваної платформи обрано PostgreSQL 17 [13] з розширенням TimescaleDB. Таке поєднання забезпечує переваги обох підходів - повноцінну реляційну модель для основних даних та оптимізації для часових рядів - у рамках однієї БД. Ключовими чинниками вибору стали:

1. Реляційна модель PostgreSQL забезпечує транзакційну цілісність ACID, посилальну цілісність через зовнішні ключі та потужні можливості SQL-запитів з агрегаціями і віконними функціями, що використовуються при формуванні аналітики метрик.
2. Підтримка типу JSONB - бінарного формату зберігання JSON-документів з можливістю індексації окремих полів. У розроблюваній системі цей тип використовується для зберігання метаданих метрик та параметрів команд, що дозволяє зберігати неструктуровані дані без втрати реляційної моделі.
3. Розширення TimescaleDB перетворює звичайні реляційні таблиці на hypertables - логічну абстракцію, що автоматично розбиває дані на chunks за часовою колонкою. У розроблюваній системі гіпертаблицями є таблиці метрик system_metrics та process_metrics, для яких визначено складений первинний ключ (id, timestamp) як того вимагає механізм партиціонування TimescaleDB. Розбиття на партиції за часом дозволяє зберігати запис нових метрик швидким незалежно від загального обсягу історичних даних та оптимізує запити за часовими діапазонами.
4. Політика ретенції даних реалізується через стандартні засоби TimescaleDB та параметр конфігурації RetentionDays серверного застосунку: партиції

метрик старші за 30 днів автоматично видаляються, що запобігає неконтрольованому зростанню обсягу БД.

5. Sequences у PostgreSQL це механізм генерації унікальних ідентифікаторів, що забезпечує атомарну та потокобезпечну видачу нових значень навіть при одночасних операціях запису від множини клієнтів.

Для роботи з PostgreSQL з боку серверного коду використовується ORM-фреймворк Entity Framework Core 9 - об'єктно-реляційний маппер від Microsoft, інтегрований з .NET. Перевагами EF Core є: типобезпечні запити мовою LINQ замість текстових SQL-запитів, що знижує ризик SQL-ін'єкцій; автоматичне відстеження змін у сутностях та формування пакетних SQL-операцій; система міграцій, що забезпечує версіонування схеми БД та її синхронізоване розгортання у різних середовищах. Для безпосередньої взаємодії з PostgreSQL EF Core використовує драйвер Npgsql.EntityFrameworkCore.PostgreSQL, який підтримує усі специфічні типи PostgreSQL та інтегрується з можливостями TimescaleDB.

2.3.4 Засоби двосторонньої комунікації у реальному часі

Однією з ключових архітектурних вимог до системи є мінімальна затримка передачі метрик від агента до користувацького інтерфейсу. Класична модель HTTP непридатна для цього сценарію, оскільки потребує постійного опитування сервера з боку клієнта, що збільшує навантаження на мережу та збільшує середню затримку відображення даних. Натомість оптимальним рішенням є технології, що забезпечують перманентний двонаправлений канал зв'язку.

Базовим стандартом такого зв'язку у веб-середовищі є WebSocket протокол, описаний у RFC 6455 [3], що забезпечує повнодуплексний канал між клієнтом і сервером поверх єдиного TCP-з'єднання після початкового HTTP-handshake. На відміну від HTTP, WebSocket дозволяє серверу ініціювати передачу даних клієнту в будь-який момент без очікування запиту, що принципово важливо для систем моніторингу реального часу.

Для розв'язання задачі двосторонньої комунікації у розроблюваній системі застосовано гібридний підхід, що передбачає використання двох різних механізмів для двох категорій клієнтів - агентів та веб-інтерфейсу. Це обумовлено принципово різними технічними обмеженнями та потребами цих компонентів.

WebSocket для агентів. Зв'язок між C++-агентом та сервером організовано через “сирий” WebSocket-ендпоінт /ws/agent без додаткових абстракцій. Такий вибір обґрунтовується тим, що: на стороні агента використовується вбудована у Windows бібліотека WinHTTP, яка нативно підтримує WebSocket-протокол згідно з RFC 6455 без необхідності лінкування зовнішніх клієнтських бібліотек; повідомлення між агентом та сервером мають фіксовану структуру JSON-пакетів метрик і команд, що не потребує абстракцій вищого рівня; мінімізація накладних витрат на стороні агента відповідає вимозі легковагового агентського компонента.

SignalR для веб-інтерфейсу. Для зв'язку між веб-клієнтом та сервером застосовано фреймворк ASP.NET Core SignalR [9] - бібліотеку від Microsoft, що надає високорівневу абстракцію над WebSocket-протоколом. SignalR значно спрощує розробку клієнт-серверних застосунків з реальним часом, надаючи такі можливості:

1. Hubs це серверні класи, методи яких можуть викликатися з клієнта в стилі RPC, а серверний код може звертатися до клієнтських методів симетрично. У розроблюваній системі реалізовано хаб MonitoringHub за маршрутом /hubs/monitoring, що надає клієнтам методи для підписки на метрики конкретного агента (SubscribeToAgent), приєднання до груп проектів (JoinProjectGroup), а серверу - можливості широкомовлення подій (MetricsUpdate, AgentStatusChanged, AgentListChanged).
2. Групи з'єднань це механізм логічного об'єднання клієнтів для широкомовної розсилки повідомлень. У системі застосовано три типи груп: user_{userId}, project_{projectId} та agent_{agentId}. Це дозволяє

ефективно маршрутизувати повідомлення без перевірки прав доступу для кожного з'єднання окремо.

3. Автоматичний fallback на альтернативні транспорти - якщо WebSocket-з'єднання не може бути встановлене через обмеження мережі, SignalR автоматично переходить на Server-Sent Events або Long Polling, забезпечуючи стабільну роботу інтерфейсу у будь-якому мережевому середовищі.
4. Автоматичне перепідключення це стандартний механізм відновлення з'єднання після короткочасних обривів мережі, що звільняє розробника від реалізації цієї логіки вручну.
5. Інтеграція з ASP.NET Core - SignalR-хаби автоматично використовують механізм Dependency Injection та підтримують ту саму JWT-автентифікацію, що й REST API. Передача JWT-токена для встановлення WebSocket-з'єднання здійснюється через query-параметр ?access_token=, оскільки браузерний WebSocket API не дозволяє додавати власні HTTP-заголовки.

Така комбінація забезпечує оптимальний баланс між простотою реалізації та функціональною повнотою: агент залишається легковаговим без додаткових залежностей, а на стороні веб-інтерфейсу розробник отримує усі переваги високорівневої абстракції з вбудованими механізмами стійкості з'єднання.

2.3.5 Засоби реалізації веб-інтерфейсу

Веб-інтерфейс розроблюваної платформи є основним засобом взаємодії користувача з системою та повинен забезпечувати: відображення метрик, інтерактивну роботу з інструментами дистанційного, а також зручне управління проектами та учасниками команди. Складність та обсяг функціональності обумовлюють використання сучасних архітектурних підходів - побудови інтерфейсу як односторінкового застосунку (SPA).

При виборі засобів реалізації клієнтської частини розглядалися такі альтернативи:

1. Angular це повнофункціональний фреймворк від Google з вбудованими модулями маршрутизації, форм та HTTP-клієнта. Перевагою є строга архітектура та інтегрована TypeScript-підтримка, проте крутіша крива навчання та значно більший розмір зібраного бандлу роблять його надлишковим для проектів середнього розміру.
2. Vue.js це прогресивний фреймворк зі спрощеним синтаксисом шаблонів. Має менший бандл та меншу складність, проте екосистема UI-компонентних бібліотек менша порівняно з React, що ускладнює пошук готових рішень для специфічних задач.
3. Svelte це фреймворк з компіляцією шаблонів у нативний JavaScript-код, що забезпечує мінімальний розмір бандлу та високу продуктивність. Однак менша зрілість екосистеми та обмежена кількість UI-бібліотек є суттєвим обмеженням для застосунку з широкою функціональністю.

Для реалізації клієнтської частини обрано бібліотеку React 19 [7] у поєднанні з TypeScript 5 як мовою програмування та Material UI v7 як компонентною бібліотекою. Такий стек обґрунтовується наступними чинниками:

1. React 19 це найновіша версія найпоширенішої у світі бібліотеки для побудови користувацьких інтерфейсів. Компонентна модель та віртуальний DOM забезпечують ефективне оновлення інтерфейсу при зміні даних - критично важливу властивість для дашбордів реального часу. Версія 19 містить нові можливості, що дозволяють пріоритезувати рендеринг високочастотних оновлень метрик без блокування користувацької взаємодії.
2. Hooks це функціональний підхід до управління станом і побічними ефектами компонентів. У розроблюваній системі активно

використовуються вбудовані хуки та власні, що забезпечують чисту інкапсуляцію логіки доступу.

3. TypeScript це мова, що додає статичну типізацію до JavaScript. Перевагами є виявлення помилок на етапі компіляції до запуску застосунку, покращена підтримка з боку інтегрованих середовищ розробки, а також самодокументування коду через декларації типів. Для системи з великою кількістю модулів та сутностей типізація суттєво знижує кількість помилок інтеграції між компонентами.
4. Material UI v7 це найпопулярніша компонентна бібліотека для React, що реалізує специфікацію Material Design. Бібліотека надає понад 50 готових компонентів, що мають вбудовану підтримку доступності, темізації та адаптивної верстки. Версія 7 містить оновлений Grid API на основі CSS Grid, оптимізації пакування CSS-стилів та покращену типізацію з TypeScript.
5. Zustand це легковагова бібліотека управління станом застосунку. Порівняно з Redux вона не потребує зайвого “boilerplate” та інтегрується з React Hooks через простий API. У розроблюваній системі Zustand використовується для зберігання глобального стану з автоматичною персистентністю критичних полів у localStorage.

Як інструмент збірки клієнтського застосунку обрано Vite - сучасний bundler з відкритим вихідним кодом, що базується на нативних ES-модулях у режимі розробки та використовує Rollup для виробничої збірки. Vite забезпечує миттєвий запуск середовища розробки, оптимізовану підтримку Hot Module Replacement та значно швидшу збірку порівняно зі застарілими рішеннями.

Для специфічних функцій інтерфейсу використано спеціалізовані бібліотеки: xterm.js - повноцінний термінал-емулятор у браузері, що відображає вивід Remote Shell з підтримкою ANSI escape-послідовностей; Recharts - бібліотека графіків на основі React і D3 для візуалізації метрик; Microsoft SignalR

JavaScript Client - офіційна клієнтська бібліотека для встановлення SignalR-з'єднання з сервером.

Взаємодія клієнта з REST API сервера реалізована через бібліотеку Axios з налаштованим інтерсептором, що автоматично додає JWT-токен до заголовка Authorization кожного запиту та обробляє відповіді з кодом 401, перенаправляючи користувача на сторінку входу.

2.3.6 Засоби автентифікації та захисту даних

Безпека є однією з ключових нефункціональних вимог до системи, що працює з даними множини користувачів та забезпечує дистанційне управління серверами. Захист платформи будується на трьох рівнях: захищені канали зв'язку, автентифікація користувачів і агентів, авторизація доступу до операцій.

Уся взаємодія між компонентами здійснюється виключно по захищених каналах: HTTPS та WSS. Шифрування забезпечується протоколом TLS 1.2 та вище, що унеможливлює перехоплення або модифікацію даних на шляху між клієнтом та сервером. Конфігурація TLS-сертифікатів виноситься на рівень reverse-proxy або зворотнього сервісу хмарного провайдера, що відповідає типовій практиці розгортання SaaS-платформ.

Автентифікація користувачів реалізована за схемою JSON Web Token згідно зі стандартом RFC 7519 [6]. JWT це компактний формат токена, що складається з трьох частин: заголовка, тіла та підпису. У тілі токена містяться claims - пари “ключ–значення” з інформацією про користувача та термін дії. Підпис обчислюється алгоритмом HMAC-SHA256 з використанням секретного ключа сервера, що унеможливлює підробку токена на стороні клієнта.

Перевагами JWT є: stateless-природа - серверу не потрібно зберігати стан сесії у БД або кеші, оскільки усі необхідні дані містяться в самому токені; масштабованість - будь-який екземпляр сервера може автентифікувати запит без звернення до спільного сховища сесій; простота передачі - токен передається у заголовку Authorization: Bearer <token> для REST-запитів та у query-

параметрі ?access_token=<token> для встановлення SignalR-з'єднання, що оминає обмеження браузерного WebSocket API.

Хешування паролів виконується алгоритмом bcrypt [15] - адаптивною функцією на основі шифру Blowfish, спеціально розробленою для безпечного зберігання паролів. Особливістю bcrypt є параметр work factor, що визначає обчислювальну складність хешування: при кожному збільшенні параметра на одиницю час обчислення подвоюється. Це дозволяє з часом збільшувати складність відповідно до зростання обчислювальних потужностей атакувальника, не змінюючи алгоритм. Використання bcrypt унеможливлює відновлення паролів навіть у випадку компрометації бази даних: атакувальнику довелося б витратити значні обчислювальні ресурси на перебір кожного хешу окремо.

Автентифікація агентів реалізована окремо від користувацької схеми. Кожен агент при реєстрації отримує власний токен, хеш якого зберігається у таблиці agent_tokens. Цей токен використовується агентом у заголовку запитів до спеціалізованих ендпоінтів (/api/metrics, /ws/agent), які обробляються власним middleware-компонентом AgentAuthMiddleware, не пов'язаним зі схемою JWT для користувачів.

Авторизація реалізована багаторівнево: атрибутом [Authorize] на рівні контролерів забезпечується перевірка валідності JWT-токена; перевірка ролі учасника проекту виконується безпосередньо в коді контролерів через сервіс IProjectContext та запит до таблиці project_members.

3. ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Загальна архітектура системи

Архітектурна побудова системи визначає взаємодію її компонентів, протоколи зв'язку між ними та принципи розмежування відповідальності. Для розроблюваної платформи обрано трирівневу архітектуру - класичний архітектурний шаблон, що передбачає поділ системи на три логічні рівні: рівень презентації, рівень бізнес-логіки та рівень даних. Такий підхід забезпечує слабку зв'язаність між компонентами, можливість незалежного розгортання та масштабування кожного рівня, а також узгоджується з загальноприйнятою практикою побудови SaaS-платформ.

Система складається з чотирьох взаємодіючих рівнів: рівня презентації, рівня бізнес-логіки, рівня даних та рівня операційного середовища. На рисунку 3.1 наведено загальну архітектурну структуру платформи з розподілом за цими рівнями.

Рівень презентації - веб-інтерфейс реалізований як односторінковий застосунок на стеку React 19, TypeScript та Material UI v7. Виконує функції відображення метрик, надає графічний інтерфейс інструментів дистанційного управління, забезпечує управління проектами та учасниками команди. Розгортається у вигляді статичного контенту, що завантажується браузером користувача та виконується на стороні клієнта. Інтерфейс не містить бізнес-логіки безпеки - усі перевірки прав доступу дублюються на стороні сервера.

Рівень бізнес-логіки - серверна частина реалізована як веб-сервіс на ASP.NET Core .NET 9 і виступає єдиною точкою входу для всіх клієнтів. Містить контролери REST API, SignalR-хаб, WebSocket-ендпоінт, middleware-компоненти автентифікації та авторизації, сервіси бізнес-логіки та репозиторії доступу до даних. Сервер маршрутизує команди дистанційного управління між

клієнтами та агентами, валідує усі вхідні запити та забезпечує дотримання обмежень.

Рівень даних - реляційна СУБД PostgreSQL з розширенням TimescaleDB забезпечує постійне зберігання основних бізнес-даних у звичайних реляційних таблицях, а часові ряди метрик у гіпертаблицях TimescaleDB. Доступ до даних здійснюється виключно через шар ORM серверної частини; прямий доступ з боку клієнтів до бази не передбачений.

Рівень операційного середовища – Windows/Windows Server не входить безпосередньо до архітектури платформи, але є середовищем виконання агента. Агент взаємодіє з підсистемами Windows як з джерелом даних та об'єктом дистанційного управління.

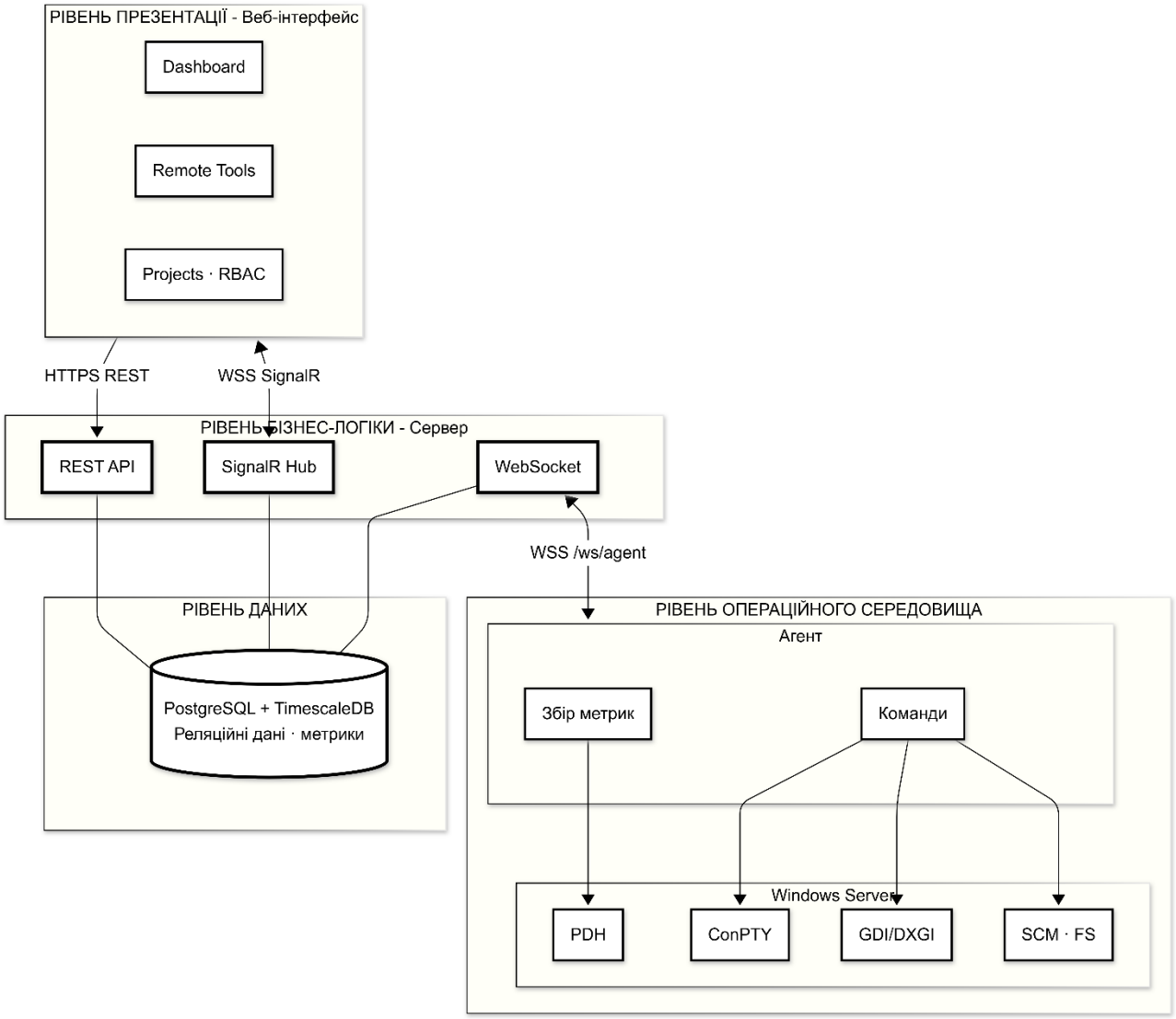


Рис. 3.1 Архітектура системи

Протоколи взаємодії між компонентами визначено трьома каналами зв'язку, кожен з яких відповідає певному типу взаємодії:

1. HTTPS REST використовується для запитів від веб-клієнта до серверної частини. Відповіді у форматі JSON, автентифікація через JWT-токен у заголовку Authorization.
2. WSS SignalR (/hubs/monitoring) - двосторонній канал між веб-клієнтом та сервером для подій реального часу: надходження нових метрик, зміни статусу агентів, оновлення складу учасників проекту, повідомлення про завершення команд. Маршрутизація повідомлень здійснюється через групи user_{userId}, project_{projectId}, agent_{agentId}.
3. WSS (/ws/agent) це “сирий” WebSocket-канал між агентом та сервером для передачі метрик у режимі push, а також для отримання агентом команд дистанційного управління та повернення результатів їх виконання.

При побудові архітектури системи дотримано таких принципів:

1. Separation of Concerns - кожен компонент має чітко визначену функцію та не дублює задачі інших. Агент відповідає виключно за збір метрик і виконання локальних команд; сервер - за бізнес-логіку та маршрутизацію; клієнт - за відображення та користувацьку взаємодію.
2. Stateless-сервер - серверна частина не зберігає стан клієнтських сесій між запитами. Уся необхідна інформація про автентифікацію передається у JWT-токені кожного запиту, що дозволяє масштабувати сервер горизонтально без необхідності спільного сховища сесій.
3. Push-модель замість pull-моделі - агент самостійно ініціює передачу метрик при їх збиранні замість того, щоб очікувати на запити від сервера. Це мінімізує затримку доставки даних та забезпечує рівномірний розподіл навантаження.
4. Багаторівневий захист - перевірки прав доступу здійснюються на кожному рівні: TLS на транспортному, JWT на рівні автентифікації, рольова авторизація та обмеження за підпискою на рівні бізнес-логіки, валідація політик на рівні агента.

5. Multi-tenancy - система розрахована на одночасну роботу множини незалежних користувачів та команд. Розмежування даних між тенантами забезпечується на рівні запитів до БД через перевірку приналежності до проекту та власника.

Описані архітектурні рішення створюють основу для деталізованого проектування кожного компонента системи, що розглядається у наступних підрозділах.

3.2 Проектування агентського компонента

Агент є складовою частиною системи, що встановлюється на кожен підконтрольний сервер. До нього висуваються вимоги легковагості, автономності та мінімального впливу на ресурси сервера, тому його архітектура побудована за модульним принципом - окремі компоненти відповідають за чітко окреслені задачі та взаємодіють через визначені інтерфейси. До складу агентського компонента входять такі основні модулі:

1. HardwareId Generator - модуль генерації унікального апаратного ідентифікатора підконтрольного сервера на основі серійних номерів комплектуючих та параметрів операційної системи. Ідентифікатор стабільний при перезавантаженні системи та дозволяє серверу однозначно ідентифікувати агента незалежно від його імені.
2. Registration Manager - модуль управління реєстрацією агента. При першому запуску агент очікує на введення 6-символьного коду підключення, отриманого через веб-інтерфейс, та надсилає його разом з апаратним ідентифікатором на сервер. У відповідь сервер повертає персональний токен автентифікації, який зберігається у системному реєстрі за шляхом HKLM\SOFTWARE\ApmMonitoring\AuthToken. При наступних запусках агент завантажує токен з реєстру та валідує його з сервером перед підключенням; у разі недійсного токена реєстрація скидається і агент повертається до очікування нового коду.

3. Колектори метрик реалізовані через інтерфейс IMetricCollector і об'єднані в межах спільного циклу збору. Включають п'ять незалежних колекторів: CpuCollector,MemoryCollector, DiskCollector, NetworkCollector , ProcessCollector.
4. WebSocket Client - модуль підтримки постійного з'єднання через бібліотеку WinHTTP. Реалізує handshake згідно з RFC 6455, серіалізацію метрик у JSON-пакети, надсилання даних та приймання вхідних команд. Працює у власному потоці прийому повідомлень.
5. HttpClient - резервний транспортний модуль на основі WinHTTP. Використовується для запитів реєстрації, передачі стартової інформації про систему та як fallback-канал передачі метрик у разі недоступності WebSocket-з'єднання.
6. Metric Buffer - два потокобезпечні буфери, що тимчасово накопичують зібрані метрики перед відправкою: systemBuffer_ місткістю до 500 системних метрик та processBuffer_ місткістю до 10 000 метрик процесів. Якщо відправка на сервер не вдається, метрики повертаються до буфера за умови що його розмір не перевищує визначений поріг - це запобігає неконтрольованому зростанню споживання пам'яті при тривалій недоступності сервера.
7. CommandExecutor - диспетчер команд дистанційного управління.
8. RemoteDesktopHandler - модуль захоплення екрану через GDI або DXGI з симуляцією введення миші та клавіатури через InputSimulator.
9. ShellSession - модуль інтерактивного псевдотермінала, що використовується для виконання команд.

Агент має дворівневу циклічну структуру виконання: зовнішній цикл управляє реєстрацією і повторним підключенням після скидання, внутрішній цикл виконує безперервне збирання та надсилання метрик.

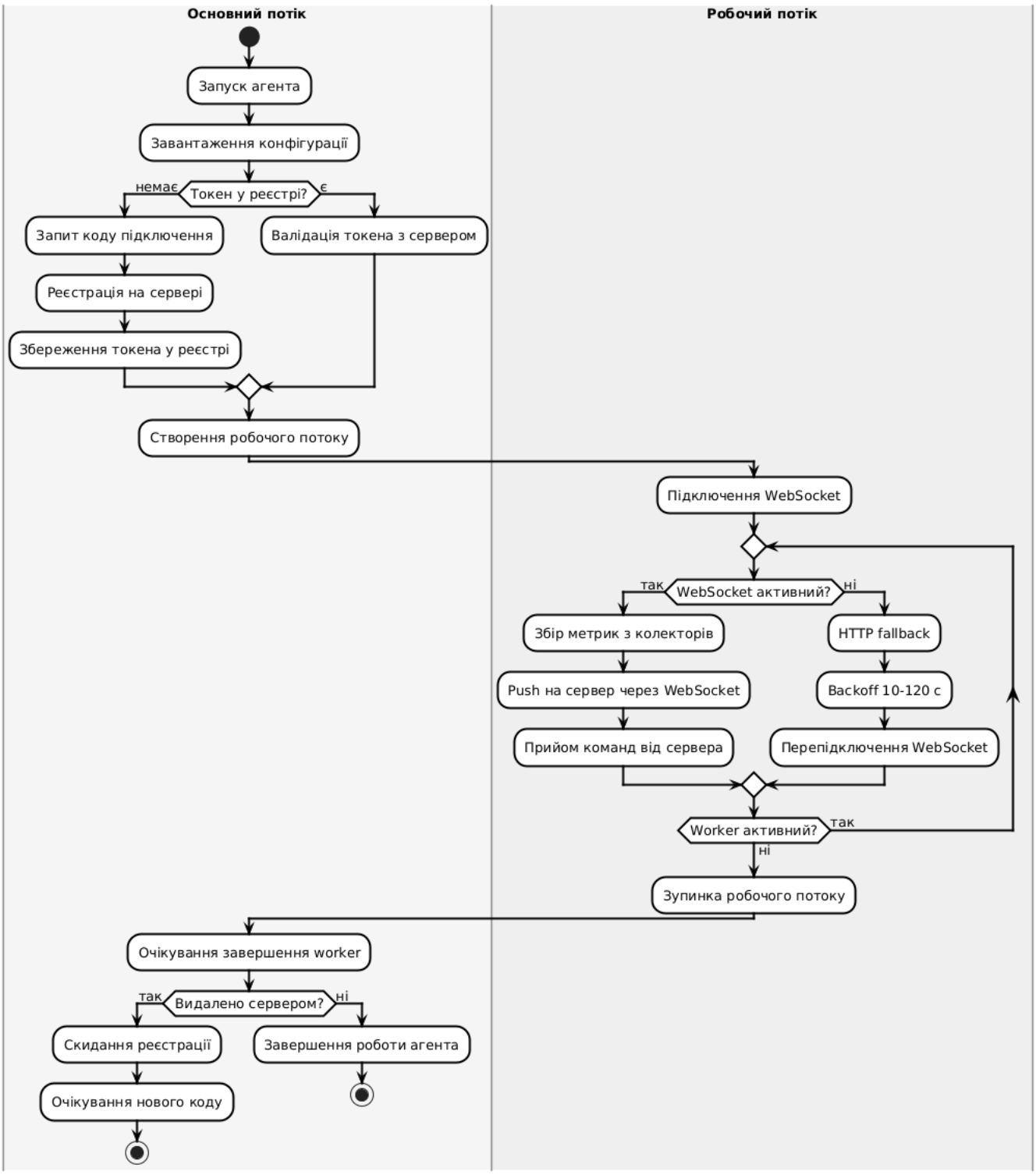


Рис. 3.2 Блок-схема життєвого циклу агента

При проектуванні агентського компонента дотримано таких принципів:

1. Розділення основного потоку та робочого потоку - реєстрація і взаємодія з користувачем виконуються в основному потоці, а збір метрик і обмін з сервером - в окремому робочому потоці. Тривалі операції делегуються

асинхронним задачам через std::future, що запобігає блокуванню основного циклу збору.

2. Багаторівнева стратегія передачі даних - основним каналом передачі метрик є WebSocket-з'єднання; у разі його недоступності агент автоматично переходить на HTTP-fallback (POST /api/metrics). Це гарантує доставку даних у будь-яких мережевих умовах.
3. Стійкість до короткочасних збоїв - при обриві WebSocket-з'єднання агент виконує повторні спроби підключення. Метрики, не доставлені через жоден з каналів, повертаються до буфера для повторної відправки після відновлення зв'язку.
4. Можливість віддаленого скидання - якщо адміністратор видаляє агента через веб-інтерфейс, сервер передає агенту сигнал про скидання. Агент автоматично очищує збережений токен з реєстру і повертається у режим очікування нового коду підключення без необхідності ручного переустановлення.

3.3 Проектування серверної частини

Серверна частина платформи реалізована у вигляді багатопроектного рішення на ASP.NET Core .NET 9 з чітким розмежуванням відповідальності між проектами. Така структура спрощує супроводження коду, дозволяє повторно використовувати компоненти у різних контекстах та полегшує модульне тестування. Кодова база організована у чотири проекти:

1. ApmBackend.Core - ядро доменної моделі. Містить класи сутностей, інтерфейси сервісів та константи бізнес-логіки. Цей проект не залежить від жодного з інших та може бути використаний як спільне ядро.
2. ApmBackend.Infrastructure - рівень інфраструктури. Містить контекст бази даних ApmDbContext, конфігурації сутностей для Entity Framework Core та міграції. Цей проект інкапсулює усі деталі взаємодії з PostgreSQL.

3. ApmBackend.Application - рівень бізнес-логіки. Містить сервіси, що реалізують ключові операції системи: AuthService, TokenService, ProjectService, ConnectionCodeService, AnalyticsService.
4. ApmBackend.API - рівень презентації серверної частини. Містить контролери REST API, middleware-компоненти, фільтри авторизації, SignalR-хаб, WebSocket-обробники для агентів та допоміжні служби. Це єдиний проект, який запускається безпосередньо як веб-сервер.

Кожен HTTP-запит проходить через послідовність middleware-компонентів, що виконують перевірки та збагачення контексту до того, як запит досягне контролера. Цю послідовність наведено на рисунку 3.3.

Спочатку відбувається перевірка походження запиту через CORS-політику, що дозволяє взаємодію з веб-клієнтом, розгорнутим на іншому домені. Далі вмикається підтримка WebSocket-протоколу з періодом keep-alive у 30 секунд. Після цього запит обробляється власним компонентом AgentAuthMiddleware, що відповідає за автентифікацію агентів: для запитів на ендпоінти /api/metrics, /api/processes та /api/agents/sysinfo він перевіряє наявність токена агента у заголовку Authorization, валідує його та зберігає ідентифікатор агента, його власника та проекту в контексті запиту для подальшого використання.

Наступний крок конвеєра - стандартна автентифікація користувачів за JWT-токеном. Окремо налаштовано підтримку передачі токена через query-параметр ?access_token= для SignalR-з'єднань та потокового завантаження файлів, оскільки браузерний WebSocket API не дозволяє додавати власні заголовки. Після автентифікації виконується авторизація на основі атрибутів контролерів.

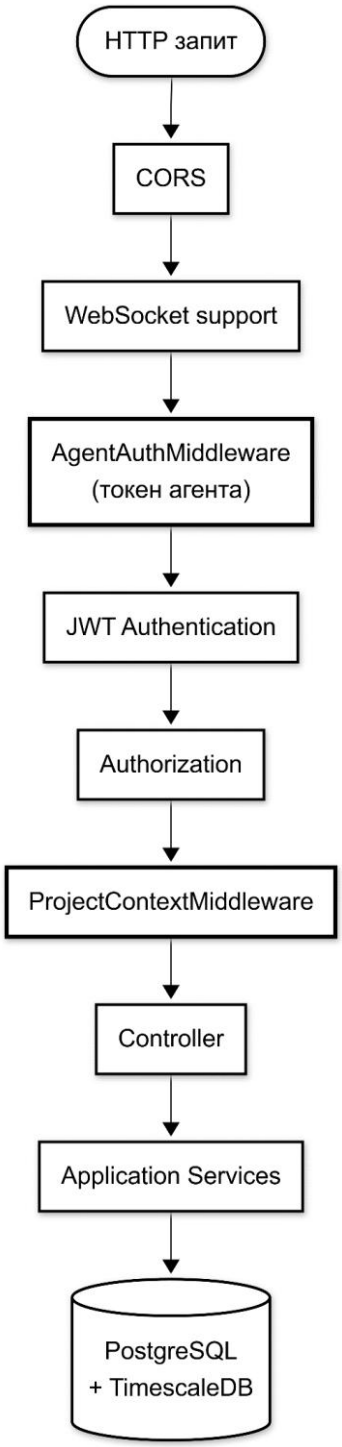


Рис. 3.3 Конвеєр обробки HTTP-запитів у серверній частині

Останнім з власних компонентів є ProjectContextMiddleware, що резолвить контекст поточного проекту: визначає його ідентифікатор з маршруту, query-параметра або заголовка X-Project-Id, валідує членство користувача в проекті та визначає рівень підписки, що застосовується. Особливістю реалізації є те, що для запитів у контексті проекту використовується підписка творця проекту, а не

самого користувача - це дозволяє учасникам команди користуватися функціями плану власника, не маючи власної підписки відповідного рівня. Лише після проходження усіх middleware-компонентів запит передається до відповідного контролера. Окремо від HTTP-конвеєра реєструються два спеціалізованих ендпоінти: SignalR-хаб MonitoringHub за маршрутом /hubs/monitoring та WebSocket-обробник для агентів за маршрутом /ws/agent.

Серверна частина надає вісім контролерів, що покривають усі функції системи. Огляд основних груп ендпоінтів наведено в таблиці 3.1.

Таблиця 3.1

Огляд REST API серверної частини

Контролер	Префікс	Призначення
AuthController	/api/auth	Авторизація
ProjectsController	/api/projects	CRUD проектів, учасники, запрошення, налаштування ролей
AgentsController	/api/agents	Перелік, перейменування, видалення, переведення між проектами
RegistrationController	/api/registration	Створення коду підключення, активація агента
MetricsController	/api/metrics	Прийом метрик від агента, отримання історії метрик
ProcessesController	/api/processes	Прийом снапшотів процесів, отримання поточного знімка
AnalyticsController	/api/analytics	Дашборди, часові ряди CPU/RAM, топ-процеси
RemoteControlController	/api/agents/{id}/remote	Сесії, команди дистанційного управління, передавання файлів

Авторизація доступу до ендпоінтів реалізована багаторівнево через систему атрибутів. Стандартний атрибут [Authorize] ASP.NET Core забезпечує перевірку наявності валідного JWT-токена і застосовується до всіх ендпоінтів, що потребують автентифікованого користувача. Для перевірки прав на конкретні функції системи розроблено власний фільтр [SubscriptionGuard("FeatureName")], що перевіряє, чи активна підписка користувача (або підписка творця проекту в контексті проектного запиту) містить запитувану функцію; у разі відсутності - повертає відповідь зі статусом 403. Перевірка спирається на матрицю функцій, визначену в константах PlanLimits ядра доменної моделі. Для операцій, що залежать від ролі учасника проекту, перевірка виконується безпосередньо в коді контролера через сервіс IProjectContext або запит до таблиці ProjectMembers. Окремо передбачено можливість гнучкого налаштування мінімальної ролі для специфічних дій - відповідні значення зберігаються у ProjectSettings і можуть змінюватися власником проекту через веб-інтерфейс.

Реєстрація сервісів у контейнері залежностей - при запуску застосунку у контейнері Dependency Injection реєструються три категорії сервісів за різним часом життя. Більшість прикладних сервісів - AnalyticsService, TokenService, ConnectionCodeService, AuthService, ProjectService, ProjectContext, WebSocketMessageRouter - реєструються як scoped, тобто кожен HTTP-запит отримує власний екземпляр, який автоматично знищується після завершення обробки. Контролери отримують ці сервіси через конструктор у класичній манері впровадження залежностей. Як singleton-сервіси, тобто екземпляри, спільні для всього часу життя застосунку, зареєстровано AgentConnectionManager - реєстр активних WebSocket-з'єднань з агентами, та CommandAwaiter - механізм очікування результатів асинхронних команд. Окремо як фонова служба запускається CleanupService, що періодично виконує очищення прострочених кодів підключення, завершених сесій та інших тимчасових даних. Контекст бази даних ApmDbContext реєструється з підключенням до PostgreSQL через драйвер Npgsql із вмиканням механізму

повторних спроб при тимчасових збоях, що підвищує стійкість до короткочасних мережевих проблем.

Маршрутизація команд між клієнтом та агентом - команди дистанційного управління, що ініціюються веб-клієнтом, надходять до серверної частини через REST-запит POST /api/agents/{id}/remote/command. Сервер зберігає команду у базі даних зі статусом pending, після чого передає її агенту через активне WebSocket-з'єднання за допомогою AgentConnectionManager. Коли агент завершує виконання команди, він надсилає результат назад через WebSocket; компонент WebSocketMessageRouter оновлює статус команди в базі даних та сигналізує очікуваним викликам REST API через CommandAwaiter. Веб-клієнт може отримати результат двома способами: через polling-запит GET /api/agents/{id}/remote/commands/{commandId} або через підписку на SignalR-подію CommandCompleted для отримання результату в реальному часі.

3.4 Проектування веб-інтерфейсу

Веб-інтерфейс розроблюваної платформи реалізовано як односторінковий застосунок на основі бібліотеки React 19 з мовою TypeScript та компонентною бібліотекою Material UI v7. Архітектура клієнтської частини побудована за принципом чіткого розмежування відповідальності: маршрутизація, керування глобальним станом, взаємодія з сервером та власне UI-компоненти інкапсульовані у відокремлені модулі.

Структура клієнтського коду - кодова база організована у функціональні директорії, що відображають архітектурні шари. Директорія pages містить компоненти-сторінки, кожен з яких відповідає окремому маршруту застосунку: дашборд, перелік агентів, керування проектами, інструменти дистанційного управління та системні розділи. У директорії components зосереджено допоміжні компоненти багаторазового використання - від великих структурних блоків до спеціалізованих елементів інтерфейсу. Логіка взаємодії з сервером винесена у

директорію services, де представлено два модулі: api.ts як обгортка над HTTP-клієнтом Axios для викликів REST API та signalr.ts для керування SignalR-з'єднанням реального часу. Власні React-хуки знаходяться у директорії hooks - useProjectRole для перевірки прав на основі ролі учасника проекту з урахуванням ProjectSettings та useSubscription для перевірки доступності функцій згідно з активним рівнем підписки. Глобальний стан застосунку реалізовано через бібліотеку Zustand у файлі store/useStore.ts. Допоміжні утиліти форматування чисел, обсягів пам'яті та часових інтервалів зосереджено в utils/format.ts, а декларації спільних типів даних у types/index.ts.

Маршрутизація та контроль доступу - маршрутизація застосунку реалізована за допомогою бібліотеки react-router-dom v7. Усі маршрути поділено на дві групи: публічні та захищені. Захищена частина обгорнута трьома компонентами-обгортками. Першим є ProtectedRoute, що перевіряє наявність токена в локальному сховищі та перенаправляє неавтентифікованого користувача на сторінку входу. Другим є ErrorBoundary, що перехоплює помилки React-компонентів та відображає коректне повідомлення замість “білого екрану”. Третім є Layout, що рендерить спільний каркас інтерфейсу: бічну навігаційну панель, верхню панель зі сповіщеннями та селектором поточного проекту, а також область вмісту сторінки. Усередині цього каркасу маршрутизатор відображає компонент-сторінку, що відповідає поточному URL: дашборд за коренем застосунку, сторінки управління проектами, агентами, процесами, дистанційним управлінням, файловою системою, службами, журналом подій, плановими завданнями та налаштуваннями. Структуру навігації зображено на рисунку 3.4.

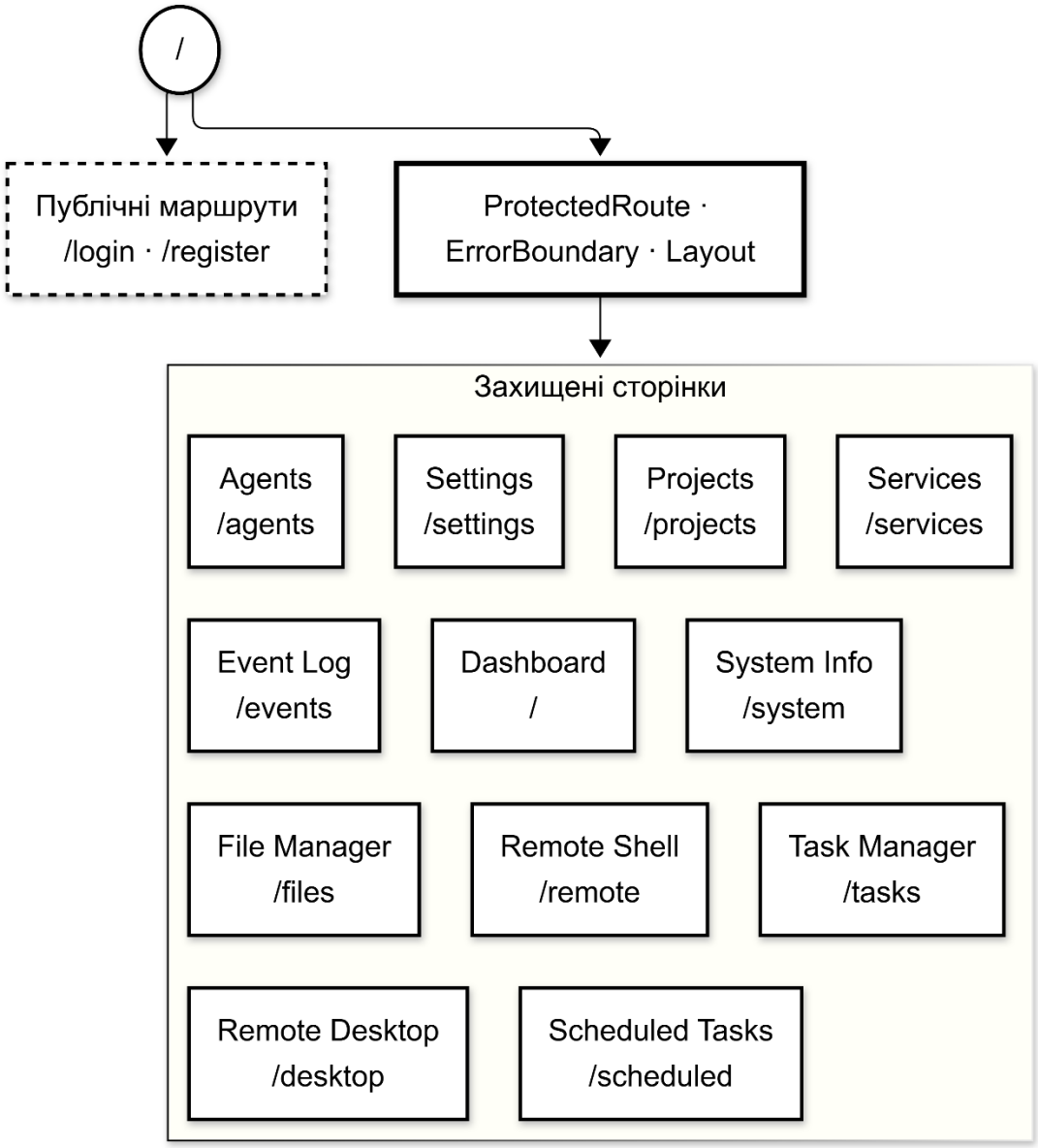


Рис. 3.4 Структура маршрутизації веб-інтерфейсу

Управління глобальним станом - глобальний стан застосунку зберігається у Zustand-сторі, що містить дані про автентифікованого користувача, список доступних проектів та поточний обраний проект, перелік агентів у поточному контексті, активну підписку, налаштування проекту, отримані запрошення, нотифікації та проміжні дані для дашборду. Окремою особливістю реалізації є збереження критичних полів стану в localStorage браузера: токен автентифікації, профіль користувача, список проектів та обраний проект автоматично зберігаються при оновленні і завантажуються при першому виклику стору, що дозволяє зберігати сесію між перезавантаженнями сторінки. Для запобігання

несанкціонованому доступу при зміні контексту проекту автоматично очищуються залежні від проекту дані - список агентів, дані дашборду, налаштування проекту та підписка - і виконується їх повторне завантаження.

Взаємодія з сервером - усі звернення до REST API серверної частини здійснюються через єдиний екземпляр Axios-клієнта, налаштований з базовим URL та автоматичним додаванням заголовка Authorization з JWT-токеном через перехоплювач запитів. Окремий перехоплювач відповідей централізовано обробляє ситуації закінчення сесії: при отриманні відповіді зі статусом 401 автоматично видаляються токен і дані користувача з локального сховища, а застосунок перенаправляється на сторінку входу з відповідним повідомленням. Така централізація логіки автентифікації позбавляє розробника необхідності повторювати її в кожному окремому виклику API. Для отримання подій реального часу використовується клієнтська бібліотека SignalR, обгорнута у власний сервіс signalrService, який встановлює з'єднання з хабом /hubs/monitoring після успішного входу, автоматично передає JWT-токен через query-параметр та надає компонентам зручний API підписки на події через систему обробників.

Контроль доступу на стороні клієнта - хоча основна перевірка прав виконується на сервері, веб-інтерфейс також реалізує клієнтську перевірку для покращення користувацького досвіду - приховування недоступних функцій, блокування кнопок і відображення повідомлень про відсутні можливості замість невдалих запитів. Хук useProjectRole повертає набір прапорців доступу залежно від поточної ролі користувача в проекті та налаштувань ProjectSettings: чи може користувач керувати учасниками, керувати агентами, відкривати Remote Shell, керувати службами, видаляти файли тощо. Хук useSubscription повертає прапорці доступності функцій відповідно до активного плану. Компонент FeatureGate обгортає блоки інтерфейсу і відображає або саму функцію, або заглушку із пропозицією оновити підписку, що забезпечує єдиний підхід до обробки функцій з обмеженим доступом по всьому застосунку.

3.5 Проектування бази даних

База даних платформи спроектована у вигляді реляційної моделі на PostgreSQL із вибірковим використанням розширення TimescaleDB для оптимізації зберігання часових рядів метрик. Схема містить чотирнадцять таблиць, об'єднаних у п'ять функціональних груп: облікові записи користувачів і підписки, проекти та командна робота, реєстрація і конфігурація агентів, часові ряди метрик, дистанційне управління та безпека. Загальну структуру схеми наведено на рисунку 3.5.

Облікові записи та підписки - центральною таблицею першої групи є users, що зберігає базову інформацію про зареєстрованих користувачів - унікальний ідентифікатор, ім'я входу, електронну адресу та хеш пароля, обчислений за алгоритмом bcrypt. Додатково таблиця містить службові поля для відстеження часу створення облікового запису, останнього входу та статусу активності. У відношенні “один до одного” з користувачем перебуває таблиця user_subscriptions, що описує активний рівень підписки. Ця таблиця містить тип плану, максимально дозволену кількість агентів, термін зберігання метрик у днях та часові поля початку і завершення підписки. Така окрема структура дозволяє в майбутньому відстежувати історію змін підписок без зміни структури основної таблиці користувачів.

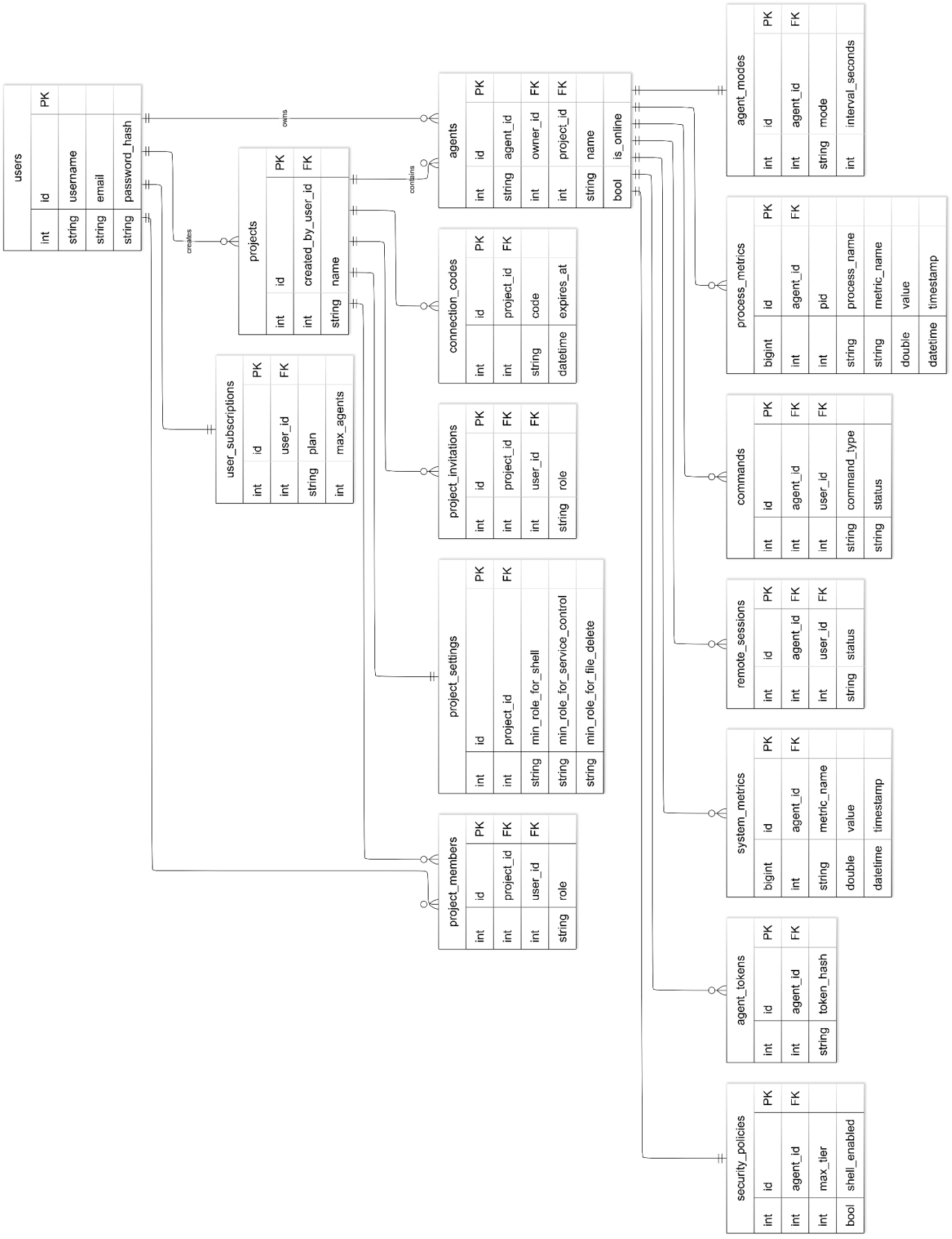


Рис. 3.5 Реляційна модель бази даних платформи

Проекти та командна робота - друга група таблиць забезпечує функціональність багатокористувацької роботи з проектами. Таблиця projects зберігає основні дані про створені проекти - назву, людиночитний ідентифікатор-slug, опис та посилання на користувача-творця через поле created_by_user_id. Учасники проекту зберігаються у таблиці project_members, що пов'язує користувача з проектом і визначає його роль через текстове поле role; додатково зберігається часова мітка приєднання до проекту і опціональне посилання на запрошувача. Запрошення нових учасників оформлюються в таблиці project_invitations, що містить посилання на проект, запрошеного користувача, призначену роль, ідентифікатор того, хто запрошує, та часову мітку створення запрошення. Окремо для кожного проекту створюється запис у таблиці project_settings - це дозволяє власнику проекту гнучко налаштовувати мінімальні ролі, необхідні для виконання різних дій.

Реєстрація і конфігурація агентів - третя група таблиць описує життєвий цикл агентів - від генерації коду підключення до повноцінної роботи. Таблиця connection_codes тимчасово зберігає 6-символьні коди реєстрації разом із попередньо заданим іменем агента, посиланням на користувача-творця коду, опціональним посиланням на проект, часом створення і завершення дії, а також прапорцем is_used для запобігання повторному використанню. При успішній реєстрації створюється запис у таблиці agents, що містить службовий цілочисловий ідентифікатор, зовнішній рядковий ідентифікатор agent_id, назву агента, інформацію про сервер, часові мітки першого та останнього з'єднання, статус “онлайн”, ідентифікатор власника та опціональне посилання на проект. У відношенні “один до багатьох” з агентом перебуває таблиця agent_tokens, де зберігаються хеші токенів автентифікації агента; зберігання саме хешів, а не самих токенів, забезпечує їх захист на випадок витоку бази даних. У відношенні “один до одного” з агентом перебуває таблиця agent_modes, що містить параметри режиму збору метрик - поточний режим, інтервал збору в секундах та обмеження на кількість процесів, які надсилаються в окремому пакеті.

Часові ряди метрик - четверта група таблиць є найбільш навантаженою з точки зору обсягу даних і реалізована як гіпертаблиці TimescaleDB. Таблиця system_metrics зберігає системні метрики у форматі довгого рядка: кожен запис містить посилання на агента, назву метрики, категорію метрики, числове значення, часову мітку та опціональні метадані у форматі JSONB для збереження додаткових атрибутів (наприклад, ідентифікатора конкретного диска чи мережевого інтерфейсу). Аналогічно влаштована таблиця process_metrics, що відрізняється наявністю додаткових полів для ідентифікації процесу - PID, ім'я виконуваного файлу, ім'я користувача-власника. Особливістю обох таблиць є складений первинний ключ (id, timestamp), що зумовлений вимогою TimescaleDB до гіпертаблиць - стовпець партиціонування за часом обов'язково повинен входити до унікальних ключів. Дані автоматично поділяються на партиції за часом і видаляються через 30 днів відповідно до політики ретенції, заданої параметром RetentionDays у конфігурації серверного застосунку.

Дистанційне управління - п'ята група таблиць описує функціональність дистанційного управління серверами та політики безпеки агентів. Таблиця remote_sessions фіксує сесії дистанційного управління, відкриті користувачами на конкретних агентах: вона містить посилання на агента і користувача, статус сесії, тривалість у хвилинах, часові мітки початку, завершення та запланованого закінчення, а також лічильник виконаних команд. Таблиця commands зберігає повну історію команд дистанційного управління і містить посилання на агента, користувача та опціональне посилання на сесію, тип команди, параметри у форматі JSONB, поточний статус, результат виконання у вигляді текстового виводу та коду завершення, а також детальний опис помилки за наявності. Таблиця security_policies визначає для кожного агента обмеження на функціонал дистанційного управління: максимально дозволений рівень операцій, прапорці увімкненості shell-доступу та операцій з файловою системою, граничну тривалість сесії та обмеження на частоту команд. Це дозволяє адміністратору на рівні окремого агента обмежувати потенційно небезпечні операції незалежно від ролі користувача в проекті чи рівня підписки.

Цілісність та продуктивність - між таблицями встановлено зовнішні ключі з відповідними діями каскадного видалення - наприклад, при видаленні користувача автоматично видаляються його підписка, членство в проектах та запрошення; при видаленні проекту видаляються його учасники, налаштування та запрошення. Для оптимізації типових запитів створено індекси на полях, що часто використовуються у фільтрах, а часовим рядам забезпечується автоматичне індексування за часом завдяки гіпертаблицям TimescaleDB. Послідовності system_metrics_id_seq і process_metrics_id_seq забезпечують атомарну генерацію унікальних ідентифікаторів метрик при одночасному запису від множини агентів.

4. ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1 Реалізація агентського компонента

Архітектура агентського компонента побудована за модульним принципом. Структуру класів агента з ключовими методами та зв'язками між ними наведено на рисунку 4.1.

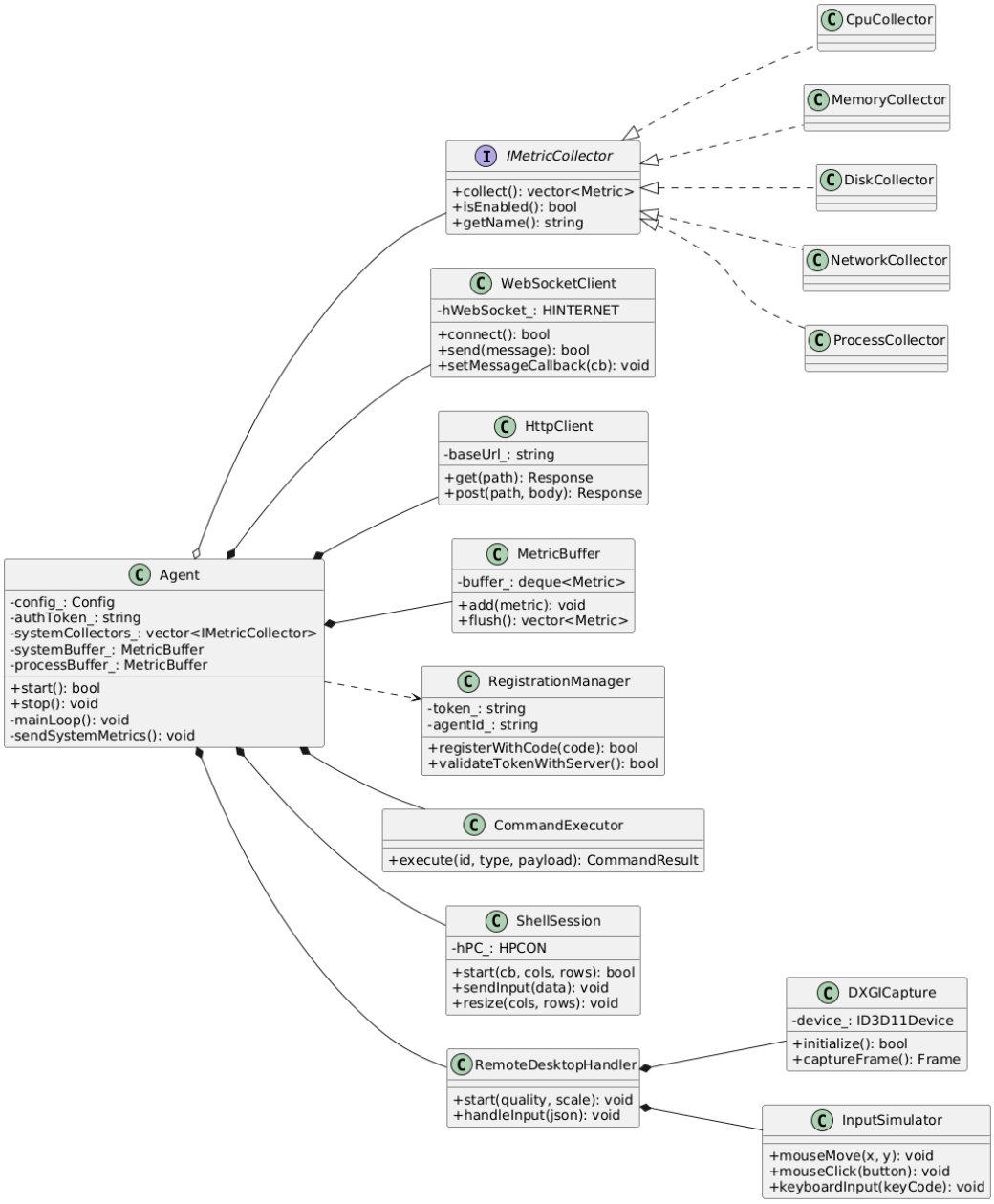


Рис. 4.1 Структура агентського компонента

Центральним класом є Agent, що агрегує всі підсистеми збору метрик, транспортного шару та виконання команд. Колектори метрик реалізують спільний інтерфейс IMetricCollector, що дозволяє додавати нові типи метрик без модифікації основного циклу. Транспортна підсистема представлена двома класами - WebSocketClient та HttpClient. Виконання команд дистанційного управління делегується класам CommandExecutor, ShellSession та RemoteDesktopHandler. Далі розглянуто реалізацію кожного з ключових компонентів.

4.1.1 Збір системних метрик

Збір показників завантаженості процесора, оперативної пам'яті, дискової підсистеми та мережевих інтерфейсів реалізовано через окремі колектори, що дотримуються спільного інтерфейсу IMetricCollector із трьома обов'язковими методами: collect(), isEnabled() та getName(). Такий поліморфний підхід дозволяє додавати нові типи метрик без модифікації основного циклу збору в класі Agent.

Як приклад розглянемо реалізацію колектора процесорних метрик CpuCollector, що використовує бібліотеку PDH для отримання значення стандартного лічильника Windows. Ініціалізація колектора виконується в конструкторі через послідовність викликів PdhOpenQuery та PdhAddEnglishCounter. Використання саме PdhAddEnglishCounter (а не локалізованого PdhAddCounter) забезпечує стабільність роботи у Windows-системах з різними мовами інтерфейсу, оскільки англійські імена лічильників є інваріантними. Принциповою особливістю PDH є необхідність двох послідовних викликів PdhCollectQueryData для отримання валідного значення відсотка завантаженості - перший виклик зберігає базовий вимір, другий обчислює різницю. Тому в конструкторі виконується “холостий” виклик відразу після додавання лічильника.

При кожному наступному виклику методу collect() колектор виконує PdhCollectQueryData для оновлення значень та PdhGetFormattedCounterValue для отримання поточного значення у вигляді числа з плаваючою комою. Отримане значення обгортається в структуру Metric з ім'ям cpu.usage.percent, типом MetricType::CPU та поточною часовою міткою. Аналогічна структура використана для інших колекторів.

4.1.2 Реєстрація агента

Процес реєстрації агента у системі відбувається у класі RegistrationManager. При першому запуску агент через консольний інтерфейс отримує від оператора 6-символьний код підключення, попередньо згенерований на стороні сервера через веб-інтерфейс. Метод registerWithCode спочатку обчислює унікальний апаратний ідентифікатор поточного сервера через утиліту HardwareId::getOrCreateAgentId(), після чого формує JSON-тіло запиту з кодом і апаратним ідентифікатором та надсилає POST-запит на ендпоінт /api/registration/activate.

У випадку успіху сервер повертає JSON-відповідь з персональним токеном автентифікації агента та згенерованим на стороні сервера ідентифікатором. Отриманий токен зберігається у системному реєстрі Windows за шляхом HKEY_LOCAL_MACHINE\SOFTWARE\ApmMonitoring\AuthToken через виклики RegCreateKeyExA та RegSetValueExA. Запис потребує адміністраторських привілеїв при першому запуску, що гарантує неможливість встановлення агента непривілейованим користувачем.

При наступних запусках агент намагається завантажити збережений токен з реєстру через loadTokenFromRegistry і, якщо він присутній, валідує його у сервера через виклик ендпоінта /api/agents/{agentId}/mode. У разі повернення статусу 401 (недійсний токен) або 404 (агент видалено з сервера) реєстрація вважається недійсною і агент переходить у режим очікування нового коду. У разі

недоступності сервера токен вважається валідним - це дозволяє агенту продовжувати роботу при тимчасових проблемах із зв'язком.

4.1.3 Передача метрик через WebSocket

Транспортна підсистема агента побудована за принципом гібридної надійності - основним каналом передачі є WebSocket-з'єднання з сервером, а у разі його недоступності виконується автоматичний перехід на HTTP. Клас WebSocketClient інкапсулює роботу з протоколом RFC 6455 через бібліотеку WinHTTP. Підключення виконується послідовністю викликів WinHttpOpen, WinHttpConnect та WinHttpOpenRequest, після чого через WinHttpSendRequest виконується початковий HTTP-запит з заголовком Upgrade: websocket. Якщо сервер відповідає кодом 101 (Switching Protocols), виклик WinHttpWebSocketCompleteUpgrade перетворює HTTP-з'єднання на WebSocket-сокет. Після успішного встановлення з'єднання у фоновому потоці запускається цикл прийому повідомлень receiveLoop, що блокує потік на виклику WinHttpWebSocketReceive і викликає зареєстрований callback при надходженні кожного повідомлення.

Метод Agent::sendSystemMetrics ілюструє стратегію вибору каналу передачі. Спочатку метод вибирає метрики з буфера systemBuffer_, формує JSON-пакет та намагається надіслати його через WebSocket. Якщо WebSocket-канал активний, відправка відбувається миттєво; якщо ж він недоступний, метод переходить на HTTP-fallback через виклик httpClient_->post.

У разі невдалої відправки через обидва канали метрики повертаються у буфер за умови, що його розмір не перевищує 400 елементів. Ця перевірка запобігає неконтрольованому накопиченню даних та споживанню пам'яті при тривалій недоступності сервера. Робочий цикл агента в окремому потоці mainLoop виконує спробу перепідключення WebSocket з експоненційно зростаючим інтервалом - від 10 до 120 секунд - що зменшує навантаження на

мережу при тривалих збоях, але забезпечує швидке відновлення зв'язку при коротких обривах.

4.1.4 Виконання команд дистанційного управління

Виконання команд агентом здійснюється класом CommandExecutor, що отримує тип команди та параметри у вигляді JSON-об'єкта і делегує виконання спеціалізованим обробникам. Найскладнішими у реалізації є інтерактивний термінал та захоплення екрану.

Інтерактивний термінал на основі ConPTY - реалізація терміналу у класі ShellSession базується на Pseudo Console API, доданому у Windows 10 версії 1809. На відміну від класичного підходу зі створенням консольного вікна, ConPTY надає віртуальну консоль, до якої можна писати дані як у потік та читати вихідні байти у вигляді ANSI-послідовностей, що ідеально для передачі через WebSocket до браузерного термінал-емулятора xterm.js. Ініціалізація сесії передбачає створення двох пар анонімних каналів, створення псевдоконсолі вказаного розміру через CreatePseudoConsole, налаштування атрибутів процесу через InitializeProcThreadAttributeList з прив'язкою псевдоконсолі через атрибут PROC_THREAD_ATTRIBUTE_PSEUDOCONSOLE та запуск процесу cmd.exe з EXTENDED_STARTUPINFO_PRESENT.

Після успішного запуску процесу cmd.exe у фоновому потоці виконується цикл читання вихідних даних з псевдоконсолі через ReadFile, який передає кожну порцію байтів зворотному виклику OnOutput для подальшої передачі через WebSocket. Введення з боку користувача надходить до агента через WebSocket-повідомлення типу shell.input і записується у вхідний канал псевдоконсолі через WriteFile, що передає дані безпосередньо до cmd.exe.

Захоплення екрану через DXGI Desktop Duplication - для функції Remote Desktop використано API DXGI Desktop Duplication, що забезпечує апаратно-прискорене захоплення вмісту екрану з мінімальними накладними витратами. Альтернативний варіант через GDI працює на старіших версіях ОС, але є значно

повільнішим і блокує робочий стіл при кожному захопленні. Ініціалізація DXGICapture полягає у створенні D3D11-пристрою через D3D11CreateDevice, отриманні DXGI-адаптера та виходу через IDXGIOutput::QueryInterface(IDXGIOutput1) і виклику DuplicateOutput для початку захоплення.

При невдачі апаратної ініціалізації здійснюється відкат на програмний рендерер WARP через D3D_DRIVER_TYPE_WARP. Захоплені кадри стискаються в JPEG, кодуються у Base64 та передаються через WebSocket як повідомлення типу desktop.frame . Введення миші та клавіатури з боку клієнта обробляється класом InputSimulator через виклики SendInput для імітації нативних подій вводу.

4.2 Реалізація серверної частини

Серверна частина платформи побудована з чітким розділенням відповідальності між класами шарів автентифікації, маршрутизації та реального часу. Структуру ключових серверних класів наведено на рисунку 4.2.

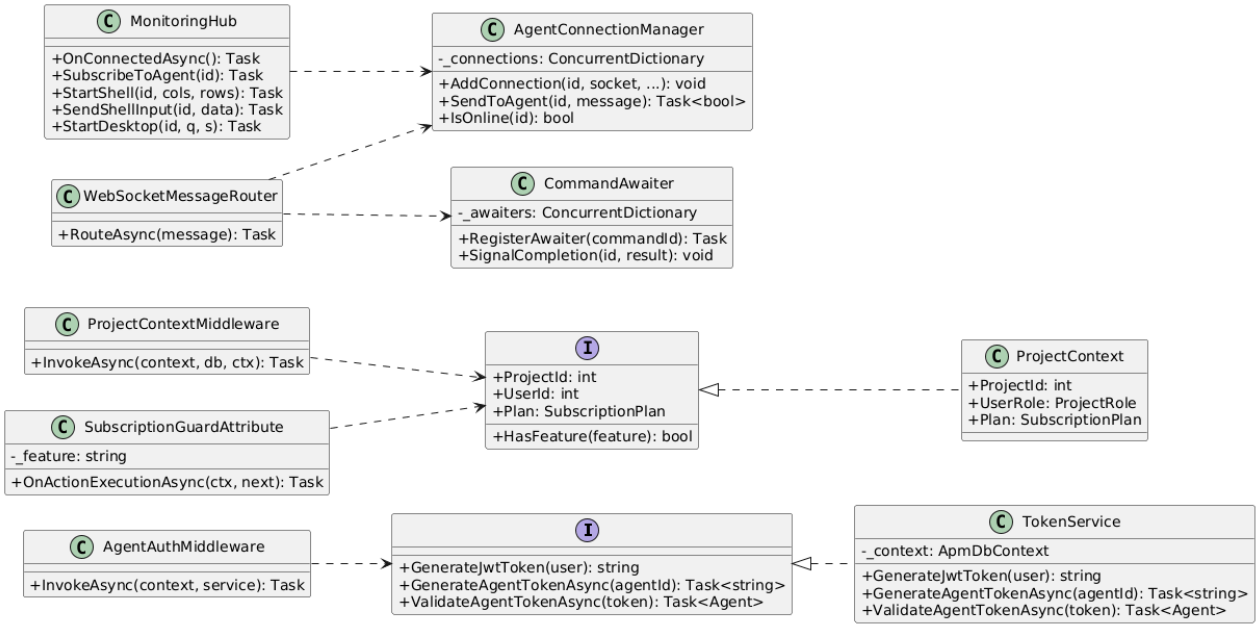


Рис. 4.2 Структура серверних класів

На рисунку 4.2 показано три основні групи компонентів. Першу групу формують класи автентифікації - інтерфейс *ITokenService* з реалізацією *TokenService*, що обслуговує як JWT-токени користувачів, так і непрозорі токени агентів. Другу групу складають middleware-компоненти *AgentAuthMiddleware* та *ProjectContextMiddleware*, що виконують перевірки та збагачення контексту запиту, а також атрибут-фільтр *SubscriptionGuardAttribute*. Третю групу - інфраструктура реального часу: *MonitoringHub*, *AgentConnectionManager* та *WebSocketMessageRouter*. Далі детально розглянуто реалізацію цих компонентів.

4.2.1 Реалізація автентифікації

Серверна частина системи реалізує дві незалежні схеми автентифікації - для користувачів та для агентів - об'єднані у класі TokenService. Для користувачів використовується стандарт JSON Web Token згідно з RFC 7519, що дозволяє реалізувати stateless-автентифікацію без зберігання сесій на стороні сервера. Метод GenerateJwtToken формує підписаний токен з трьома claims, необхідними для подальшої авторизації: ідентифікатором користувача, ім'ям входу та електронною адресою.

Підпис обчислюється алгоритмом HMAC-SHA256 з використанням секретного ключа, що зберігається у конфігураційному файлі та може бути замінений через змінні середовища при розгортанні. Термін дії токена становить 24 години, після чого користувач має повторно увійти в систему.

Для агентів реалізована окрема схема на основі непрозорих токенів - випадкових 64-символьних послідовностей у шістнадцятковій нотації, що генеруються через криптографічно безпечну функцію RandomNumberGenerator. На відміну від JWT, агентський токен не містить вбудованої інформації - це звичайний випадковий ідентифікатор, який валідується через звернення до бази даних. У БД зберігається лише SHA-256 хеш токена, що захищає від компрометації навіть у разі витоку бази.

Розподіл агентських токенів між запитами обробляє middleware-компонент AgentAuthMiddleware, що активується для запитів на ендпоінти з префіксом /api/metrics, /api/processes та /api/agents/sysinfo. Він витягує токен із заголовка Authorization, передає його у ValidateAgentTokenAsync та зберігає в HttpContext.Items ідентифікатор агента, його власника та проекту для подальшого використання в контролерах і ProjectContextMiddleware.

4.2.2 Реалізація обробника WebSocket-з'єднань для агентів

Обробник WebSocket-з'єднань агентів реалізовано як окремий маршрут /ws/agent через метод-розширення MapAgentWebSocket. Він не використовує SignalR, оскільки агент є C++-програмою, а не браузером - використання “сирого” WebSocket дозволяє агенту обходитися вбудованою в Windows бібліотекою WinHTTP без додаткових клієнтських залежностей. Обробник реалізує чотирикроковий протокол з'єднання: прийом авторизаційного повідомлення, валідація токена, реєстрація з'єднання у AgentConnectionManager та надсилання поточної політики безпеки агенту.

Після успішної авторизації обробник встановлює статус is_online в таблиці agents, надсилає поточну політику безпеки через повідомлення типу policy.sync та сповіщає всіх зацікавлених учасників про підключення агента через широкомовну розсилку SignalR-події AgentStatusChanged. Далі управління передається до циклу ReceiveLoop, який обробляє вхідні повідомлення від агента - пакети метрик, результати команд, кадри віддаленого робочого столу - та маршрутизує їх до відповідних обробників через клас WebSocketMessageRouter.

4.2.3 Реалізація SignalR-хаба

Для двосторонньої взаємодії з веб-клієнтами реалізовано SignalR-хаб MonitoringHub, що монтується за маршрутом /hubs/monitoring і захищений атрибутом [Authorize]. Хаб виконує дві ключові функції: керування групами для

широкомовної розсилки повідомлень та надання клієнтам RPC-методів для дистанційного управління агентами.

При встановленні нового з'єднання з боку клієнта метод OnConnectedAsync автоматично додає клієнта до особистої групи user_{userId} для отримання повідомлень про власні агенти, а також до груп усіх проектів, у яких користувач є учасником. Це дозволяє ефективно маршрутизувати події без перевірки прав доступу при кожному надходженні даних - клієнт отримує лише ті повідомлення, які стосуються його контекстів.

Окрім управління групами, хаб надає клієнтам набір RPC-методів для віддаленого управління агентами: StartShell, SendShellInput, StartDesktop, SendDesktopInput, BrowseRemoteFiles та інші. Кожен з цих методів виконує перевірку рольових прав через допоміжний метод GetUserRoleLevel, після чого формує JSON-повідомлення відповідного типу і передає його агенту через AgentConnectionManager:

Метод GetMinRoleLevelForPermission зчитує налаштування проекту з таблиці project_settings і повертає мінімальну роль, дозволену для конкретної дії - це реалізує гнучке налаштування прав, описане в розділі 3.

4.2.4 Маршрутизація метрик до підписаних клієнтів

Метрики, що надходять від агентів через WebSocket, мають бути в реальному часі доставлені тим веб-клієнтам, які наразі переглядають дашборд відповідного агента. Маршрутизація реалізована через WebSocketMessageRouter, який після збереження метрик у БД викликає метод SignalR-хаба для широкомовної розсилки повідомлення в групу agent_{agentId}.

Перевірка прав на доступ до агента виконується методом UserHasAccessToAgent, який повертає істинне значення в одному з двох випадків: користувач є власником агента або користувач є учасником проекту, у якому розміщений агент. Така централізація логіки доступу гарантує, що жоден

клієнт не отримає метрики недоступного йому агента, незалежно від того, як він спробує зловжити SignalR-методами.

4.2.5 Реалізація контролерів REST API

Контролери серверної частини реалізовані за стандартним патерном ASP.NET Core: вони отримують залежності через ін'єкцію в конструктор, виконують перевірку прав доступу за допомогою атрибутів та сервісів, делегують бізнес-логіку прикладним сервісам та повертають серіалізовані JSON-відповіді. Як приклад розглянемо ендпоінт переведення агента між проектами в AgentsController.

Ендпоінт демонструє типову послідовність дій контролерів системи: завантаження сутності з бази, перевірка прав власності, перевірка членства в цільовому проекті, оновлення стану в БД, оновлення активного WebSocket-з'єднання через AgentConnectionManager та сповіщення підписаних клієнтів через SignalR-розсилку. Аналогічно побудовані інші ендпоінти - наприклад, активація агента в RegistrationController, додавання учасника в ProjectsController, отримання історії метрик в MetricsController. Уся бізнес-логіка, що виходить за межі простої перевірки і збереження, виноситься у відповідні сервіси прикладного шару.

4.2.6 Управління схемою бази даних

Зміни структури бази даних виконуються через систему міграцій Entity Framework Core. Кожне внесення зміни до Entity-класів супроводжується створенням нової міграції командою dotnet ef migrations add <Name>, після чого міграція може бути застосована до існуючої бази через dotnet ef database update. Цей підхід забезпечує відтворюваність схеми у різних середовищах та можливість відкатити зміни в разі помилок. Особливістю розгортання у поточній системі є необхідність додаткового кроку синхронізації послідовностей

PostgreSQL для гіпертаблиць TimescaleDB - оскільки гіпертаблиці мають складений первинний ключ (id, timestamp), EF Core не може автоматично налаштувати послідовність генерації значень для поля id. Цей крок виконується у фоновому режимі при запуску застосунку через виконання SQL-скрипту, що створює послідовності system_metrics_id_seq і process_metrics_id_seq за необхідністю та прив'язує їх як значення за замовчуванням до відповідних стовпців.

4.3 Реалізація веб-інтерфейсу

Веб-інтерфейс платформи реалізовано як односторінковий застосунок на React 19, у якому центральне місце займає сервіс реального часу для взаємодії з SignalR-хабом серверної частини. Структуру ключових клієнтських класів та їх взаємодію з React-компонентами наведено на рисунку 4.3.

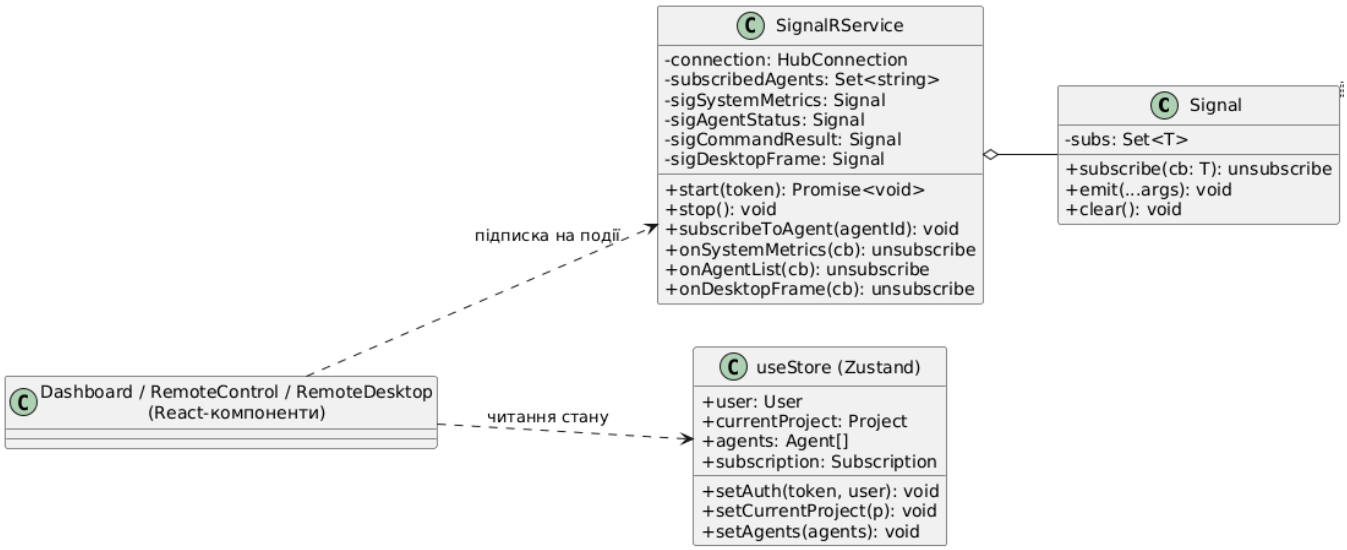


Рис. 4.3 Структура клієнтського SignalR-сервісу

На рисунку 4.3 відображено чотири основні елементи клієнтської архітектури. Центральним є клас SignalRService - singleton-обгортка над бібліотекою @microsoft/signalr, що зберігає єдине з'єднання з хабом, реєструє

підписки та диспетчеризує вхідні події до зацікавлених компонентів. Для розподілу подій між множиною підписників використано допоміжний шаблонний клас Signal<T>, кожен екземпляр якого керує власним набором обробників конкретної події. Глобальний стан застосунку зосереджено у Zustand-сторі useStore, що містить дані автентифікованого користувача, поточний обраний проект, перелік агентів, активну підписку та інші розподілені дані. React-компоненти сторінок звертаються до SignalRService для підписки на події реального часу та до useStore для читання й оновлення глобального стану. Така архітектура забезпечує одностороннє підключення до серверної частини незалежно від кількості відкритих сторінок та коректне очищення підписок при розмонтуванні компонентів.

4.3.1 Реалізація централізованого SignalR-сервісу

Для забезпечення двосторонньої комунікації веб-клієнта з серверною частиною у режимі реального часу реалізовано окремий сервіс signalRService, що інкапсулює встановлення з'єднання, автоматичне перепідключення, керування підписками та диспетчеризацію вхідних повідомлень. Цей сервіс ініціалізується одноразово при першому використанні і використовується всіма сторінками застосунку, що потребують реального часу - дашборд, Remote Shell, Remote Desktop, файловий менеджер, сторінка проектів. Така централізація дозволяє підтримувати єдине WebSocket-з'єднання на весь час життя SPA незалежно від кількості сторінок, які підписуються на події.

Особливість реалізації - використання внутрішнього класу Signal для управління множиною підписників на одну подію. Кожна сторінка може зареєструвати власний обробник для конкретного типу повідомлень і отримати функцію скасування підписки для виклику при розмонтуванні. Такий патерн дозволяє множинам компонентів незалежно реагувати на одні й ті самі події без конфліктів.

При встановленні з'єднання сервіс передає JWT-токен через query-параметр access_token, оскільки браузерний WebSocket API не дозволяє додавати власні HTTP-заголовки. Бібліотека @microsoft/signalr автоматично виконує перепідключення при тимчасових обривах та перемикається на резервні транспорти (Server-Sent Events, Long Polling), якщо WebSocket недоступний.

4.3.2 Реалізація дашборду з даними у реальному часі

Сторінка Dashboard.tsx відповідає за відображення системних метрик обраного агента у вигляді карток-показників та інтерактивних графіків. Її ключова особливість полягає у поєднанні початкового завантаження історичних даних через REST API з безперервним оновленням через SignalR-підписку.

При зміні обраного агента компонент виконує чотири дії: завантажує початкові дані дашборду через analyticsApi.getDashboard, отримує перелік топ-процесів, скасовує попередню SignalR-підписку та підписується на оновлення метрик нового агента через виклик signalRService.subscribeToAgent. Підписка серверної частини на нові метрики реалізована як виклик RPC-методу хаба, що додає клієнта до групи agent_{agentId}. У відповідь на кожне нове повідомлення MetricsUpdate сторінка перезавантажує дані дашборду, щоб коректно відобразити оновлені часові ряди.

Додатково реалізовано резервний механізм періодичного оновлення з інтервалом 30 секунд - він гарантує актуальність даних навіть у випадку, якщо SignalR-з'єднання тимчасово розірване і повідомлення MetricsUpdate не доставляються. Графіки часових рядів відображаються через бібліотеку Recharts у компоненті ChartCard, що адаптується під різні типи метрик. Загальний вигляд розробленого дашборду наведено на рисунку 4.4.

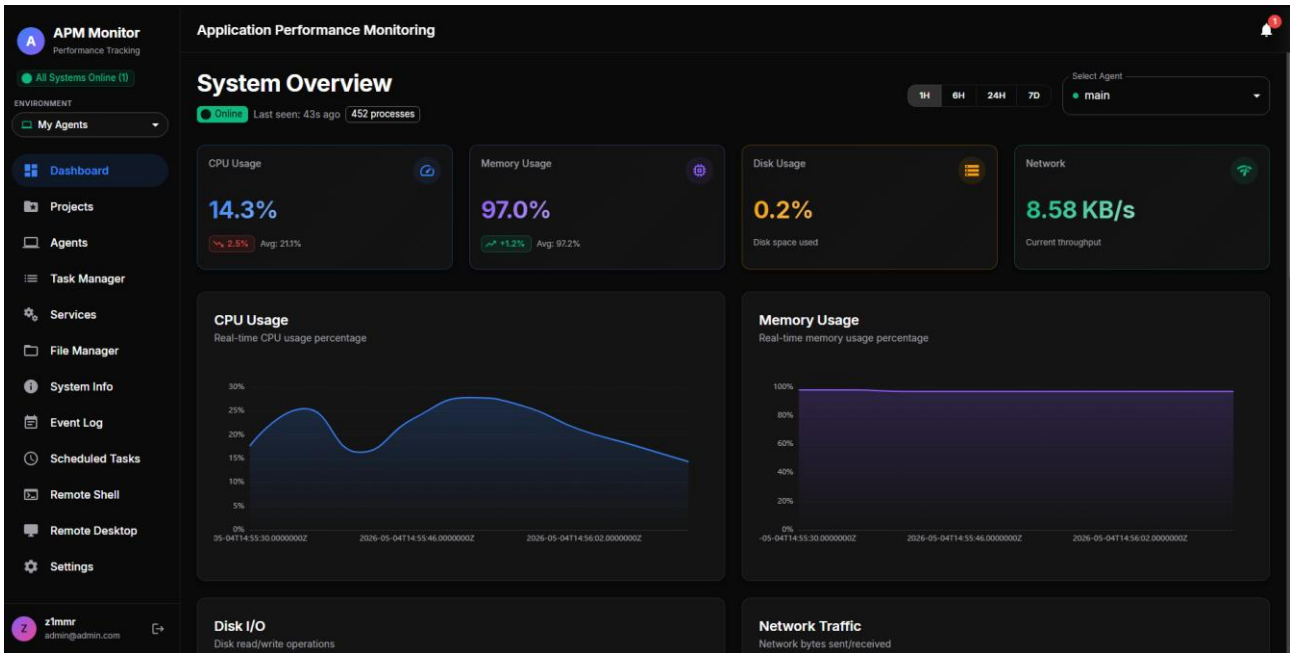


Рис. 4.4 Головний дашборд з метриками агента у реальному часі

4.3.3 Реалізація Remote Shell на основі xterm.js

Реалізація інтерактивного дистанційного терміналу побудована на бібліотеці xterm.js - повноцінному термінал-емуляторі, що працює у браузері та підтримує ANSI escape-послідовності, кольорові схеми, копіювання-вставку та зміну розміру. У сторінці RemoteControl.tsx компонент Terminal створюється одноразово при монтуванні та зберігається у useRef, щоб уникнути перестворення при перерендеренні React-компонента:

При натисканні користувачем кнопки запуску терміналу клієнт викликає метод StartShell SignalR-хаба. Сервер передає це повідомлення агенту, який створює нову ConPTY-сесію та починає надсилати вихідні байти cmd.exe назад через WebSocket. Ці байти доходять до сервера, передаються до клієнта через SignalR-подію ShellOutput, і клієнт записує їх безпосередньо у термінал викликом term.write(data). Завдяки тому, що xterm.js розуміє той самий формат ANSI-послідовностей, який генерує cmd.exe через ConPTY, користувач отримує повноцінний термінал з кольорами, курсором, історією прокрутки та правильною обробкою керівних клавіш.

При зміні розміру вікна браузера компонент перераховує оптимальну кількість колонок і рядків через FitAddon.fit() і викликає метод ResizeShell хаба для синхронізації розміру ConPTY на стороні агента - це гарантує, що cmd.exe відображається без обрізання тексту або зайвих переносів рядків. Інтерфейс модуля Remote Shell з відкритою сесією до Windows-сервера наведено на рисунку 4.5.

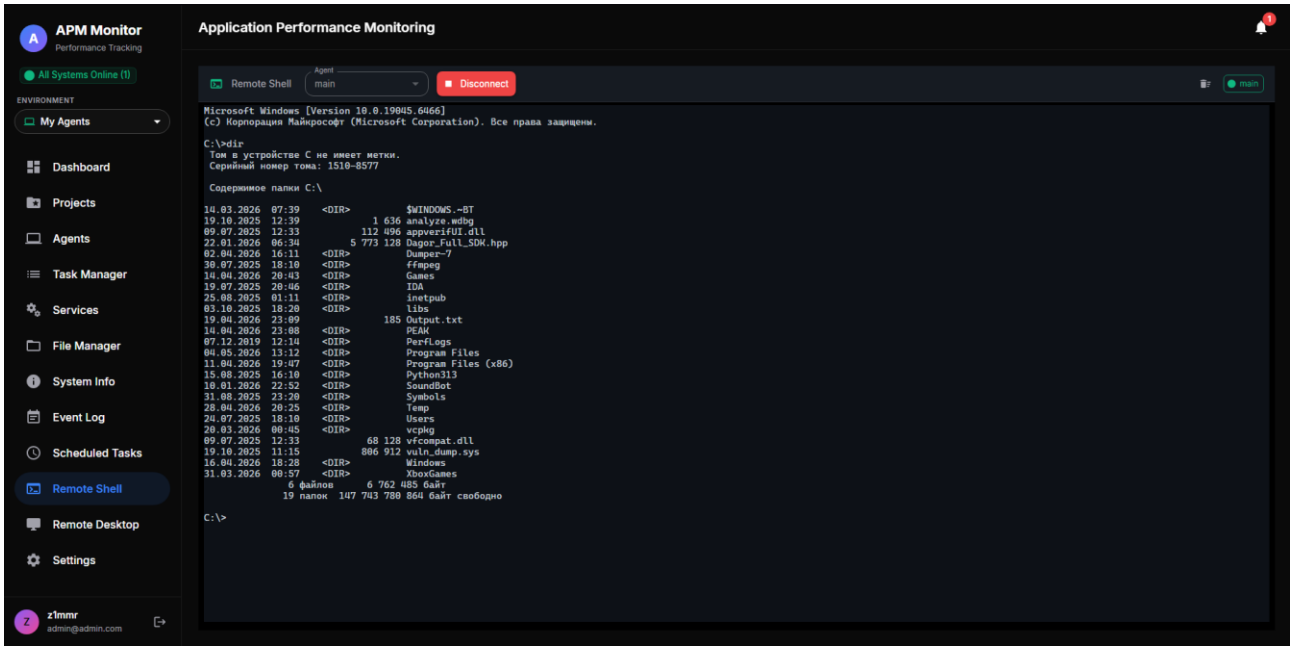


Рис. 4.5 Інтерфейс Remote Shell з відкритою сесією до Windows-сервера

4.3.4 Реалізація Remote Desktop через HTML5 Canvas

Сторінка RemoteDesktop.tsx реалізує функцію віддаленого робочого столу шляхом відображення JPEG-кадрів, що надходять від агента через SignalR, на HTML5 <canvas>-елементі. Для оптимізації пропускної здатності агент надсилає два типи кадрів: повний кадр full (повний знімок екрана при першому підключенні або після значних змін) та tiles - набір прямокутних плиток зі змінними областями екрана для оновлень, коли більшість екрану залишилася без змін. Клієнтський обробник розрізняє ці типи і відповідно малює або весь зображення, або лише змінені плитки.

Введення з боку користувача - рухи миші, кліки, натискання клавіш - обробляється через нативні DOM-події mousemove, mousedown, keydown на елементі <canvas> і передається на агент через SignalR-метод SendDesktopInput у вигляді JSON-об'єктів з координатами та параметрами. Якісне забезпечення інтерактивності реалізовано адаптивним стисненням JPEG: агент динамічно змінює рівень якості залежно від мережевої затримки та кількості змін на екрані. Користувач може вручну налаштувати якість і масштаб через панель Settings - параметри передаються методом UpdateDesktopSettings і застосовуються на стороні агента “на льоту” без необхідності перепідключення.

Окремо реалізована синхронізація буфера обміну між локальним комп'ютером користувача та віддаленим сервером: при натисканні відповідних кнопок викликаються методи GetRemoteClipboard / SetRemoteClipboard SignalR-хаба, що дозволяє швидко обмінюватися текстовими фрагментами між середовищами. Інтерфейс модуля Remote Desktop з активною сесією до Windows-сервера наведено на рисунку 4.6.

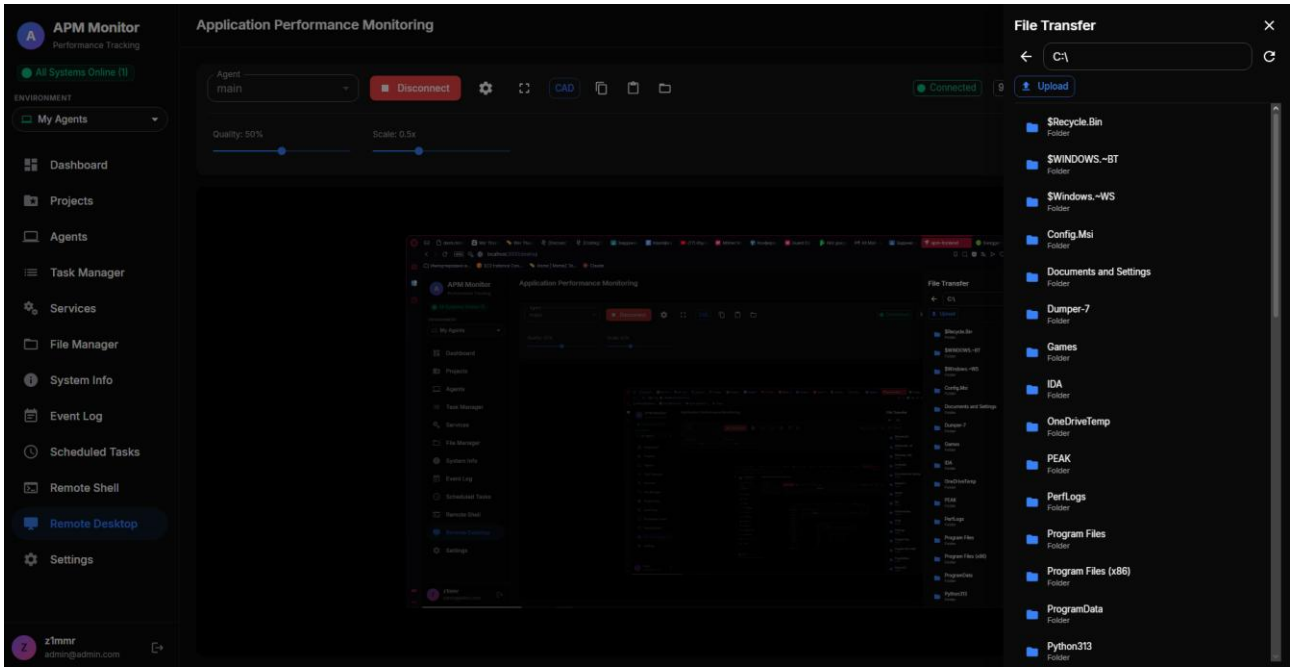


Рис. 4.6 Інтерфейс Remote Desktop з активною сесією до Windows-сервера

4.3.5 Реалізація інших функціональних модулів

Сторінки TaskManager.tsx, Services.tsx, FileManager.tsx, EventLog.tsx та ScheduledTasks.tsx мають типову структуру взаємодії з агентом через REST API: при відкритті сторінки клієнт надсилає POST-запит на ендпоінт /api/agents/{id}/remote/command з відповідним типом команди, сервер передає команду агенту через WebSocket, агент виконує її та повертає результат, який клієнт отримує через polling або SignalR-подію CommandCompleted. Загальну логіку відправки команди винесено у спільний хелпер executeCommand у сервісі api.ts.

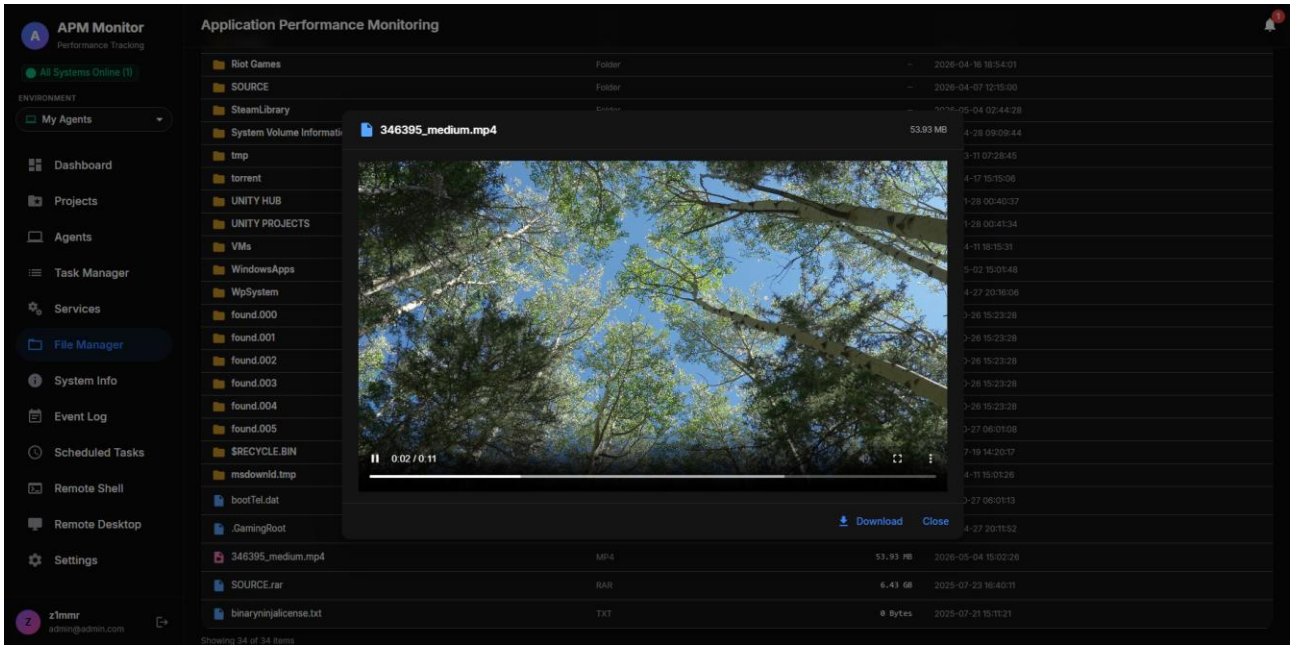


Рис. 4.7 Інтерфейс файлового менеджера

4.3.6 Управління глобальним станом

Глобальний стан застосунку зберігається у Zustand-сторі, що містить дані автентифікованого користувача, список доступних проектів, поточний обраний проект, перелік агентів, активну підписку, налаштування проекту, отримані запрошення та нотифікації. Критичні поля стану автоматично зберігаються у localStorage.

5. ТЕСТУВАННЯ

5.1 Методологія тестування

Тестування системи виконувалося переважно у форматі ручної перевірки сценаріїв з використанням наявних інструментів розробника, що зумовлено обсягом роботи та одноосібним характером розробки. Такий підхід забезпечує покриття критичних сценаріїв роботи системи без значних витрат часу на створення та супроводження автоматизованих тестових наборів - пріоритет надавався реальній перевірці поведінки системи в умовах, наближених до експлуатаційних.

Для функціонального тестування REST API серверної частини використано клієнт Postman - широко відомий інструмент розробників, що дозволяє формувати HTTP-запити з довільними заголовками, тілом та параметрами, а також зручно зберігати колекції запитів для повторного використання. У Postman створено колекцію запитів, що покриває основні групи ендпоінтів: автентифікацію, управління проектами і агентами, операції з метриками та командами дистанційного управління. Для кожного запиту перевірявся не лише код відповіді, а й структура повернутого JSON-об'єкта та виконання очікуваних змін у базі даних. Для тестування поведінки веб-інтерфейсу та коректності SignalR-комунікації застосовано вбудовані інструменти розробника браузера DevTools - вкладки Network для аналізу HTTP-трафіку та WebSocket-з'єднань, Console для виявлення помилок виконання JavaScript, Performance для оцінки часу рендерингу компонентів реального часу.

Тестування коректної роботи C++-агента виконувалося шляхом запуску його на Windows-машині та перевірки відповідності зібраних метрик показникам штатних інструментів операційної системи - Task Manager і Performance Monitor.

Для перевірки коректної роботи команд дистанційного управління використовувалися реальні сценарії: завершення процесів, запуск і зупинка служб, навігація файловою системою, інтерактивна робота через Remote Shell.

Тестування продуктивності та обчислення кількісних показників - затримки REST API, пропускної здатності WebSocket, споживання ресурсів агентом - здійснювалося шляхом ручного вимірювання за допомогою тих же DevTools, а для контролю споживання ресурсів агентом використовувався Task Manager Windows. Для імітації навантаження на сервер декількома агентами одночасно запускалося кілька екземплярів агента.

Тестування безпеки спрямоване на верифікацію коректної роботи трьох рівнів захисту: автентифікації за JWT, рольової авторизації в проектах та функціональної авторизації за рівнем підписки. Для кожного з рівнів формулювалися негативні сценарії - спроби виконання операцій без необхідних прав, з некоректними або підробленими токенами, від імені користувачів з обмеженою роллю - і перевірялося, що сервер коректно повертає відповідні помилки HTTP-статусу 401 та 403.

5.2 Функціональне тестування

Функціональне тестування проводилося за методом ручної перевірки сценаріїв з відстеженням очікуваного та фактичного результату для кожного тест-кейсу. Перелік протестованих сценаріїв охоплює всі ключові функції системи, від реєстрації користувачів і агентів до взаємодії учасників проекту з різними ролями.

У таблиці 5.1 наведено зведену інформацію про виконані тест-кейси.

Таблиця 5.1

Результати функціонального тестування

ID	Тест-кейс	Очікуваний результат	Статус
TC-01	Реєстрація нового користувача з валідними даними	Створення облікового запису, видача JWT-токена	Виконано
TC-02	Спроба реєстрації з існуючою електронною адресою	Помилка 400 з повідомленням про дублікат	Виконано
TC-03	Вхід з валідними обліковими даними	Видача JWT-токена з терміном дії 24 год	Виконано
TC-04	Вхід з невірним паролем	Помилка 401	Виконано
TC-05	Створення нового проекту	Створення запису у БД	Виконано
TC-06	Запрошення нового учасника до проекту	Створення запрошення, надсилання SignalR-події	Виконано
TC-07	Прийняття запрошення новим учасником	Створення запису у project_members з призначеною роллю	Виконано

Продовження таблиці 5.1

Результати функціонального тестування

ТС-08	Зміна ролі учасника власником проекту	Оновлення поля role, сповіщення SignalR	Виконано
ТС-09	Спроба зміни ролі учасника користувачем без прав	Помилка 403	Виконано
ТС-10	Створення коду підключення для нового агента	Створення 6-символьного коду з терміном 10 хв	Виконано
ТС-11	Активація агента з валідним кодом	Реєстрація агента, видача токена автентифікації	Виконано
ТС-12	Активація з простроченим кодом	Помилка 400 “Code expired”	Виконано
ТС-13	Збір системних метрик агентом і їх відображення на дашборді	Затримка відображення менше 5 секунд	Виконано
ТС-14	Переведення агента з My Agents у проект	Зміна project_id, зникнення з My Agents, поява в проекті	Виконано

Продовження таблиці 5.1

Результати функціонального тестування

TC-15	Відкриття Remote Shell учасником з дозволом	Запуск ConPTY-сесії, інтерактивна робота	Виконано
TC-16	Спроба Remote Shell учасником без дозволу	Запит відхилено хабом, без передачі агенту	Виконано
TC-17	Завершення процесу через Task Manager	Виклик команди kill_process, видалення з переліку	Виконано
TC-18	Запуск, зупинка, перезапуск служби Windows	Зміна стану служби в реальному часі	Виконано
TC-19	Навігація файловою системою сервера	Відображення вмісту каталогу через file_list	Виконано
TC-20	Завантаження файлу з сервера	Передача через streaming-ендпоінт з підтримкою range	Виконано
TC-21	Підключення Remote Desktop користувачем з дозволом	Запуск DXGI-захоплення, відображення на canvas	Виконано

Продовження таблиці 5.1

Результати функціонального тестування

ТС-22	Видалення агента власником через дашборд	Скидання реєстрації агента, очікування нового коду	Виконано
ТС-23	Вихід учасника з проекту через кнопку Leave	Видалення з project_members, втрата доступу до агентів	Виконано
ТС-24	Зміна підписки користувачем	Розблокування додаткових функцій	Виконано

За результатами проведеного тестування всі 24 сценарії успішно пройшли перевірку. Особливу увагу приділено сценаріям зміни контексту - переведення агента між My Agents та проектом, зміна підписки та налаштувань ролей - оскільки саме вони перевіряють коректну роботу системи синхронізації станів між клієнтами через SignalR.

5.3 Тестування продуктивності

Тестування продуктивності спрямоване на перевірку відповідності розробленої системи нефункціональним вимогам, сформульованим у підрозділі 2.2. Вимірювання проводилися у локальному середовищі розробки на робочій станції зі стандартною конфігурацією; тестовий агент, сервер та клієнт розгорнуто на одному комп'ютері з Windows.

Для вимірювання часу відповіді REST API формувалися серії запитів через інструменти розробника браузера, після чого аналізувалися значення показника Time для кожного запиту. Для отримання усереднених показників фіксувалися значення для типових запитів - отримання списку агентів, переліку

проектів, даних дашборду, переліку учасників проекту, налаштувань підписки. Для тестування пропускної здатності та коректної роботи WebSocket-каналу запускалися агенти з подальшим спостереженням за оновленням метрик на дашборді. Споживання ресурсів агентом вимірювалося через вкладку Details у Task Manager Windows для процесу ApmAgent.exe. Розмір виконуваного файлу контролювався через стандартні засоби файлового менеджера Windows. Час завантаження веб-інтерфейсу фіксувався через DevTools при перезавантаженні сторінки без використання кешу.

Узагальнені результати тестування з порівнянням до встановлених у підрозділі 2.2 цільових значень наведено в таблиці 5.2.

Таблиця 5.2

Результати тестування продуктивності

Показник	Цільове значення	Виміряне значення	Статус
Час відповіді REST API (мінімум)	≤ 500 мс	5 мс	Виконано
Час відповіді REST API (середнє)	≤ 500 мс	30 мс	Виконано
Час відповіді REST API (максимум)	≤ 500 мс	78 мс	Виконано
Періодичність оновлення метрик	≤ 5 с	5 с	Виконано
Завантаження CPU агентом	≤ 2 %	1,2 %	Виконано
Споживання оперативної пам'яті агентом	≤ 50 МБ	27,5 МБ	Виконано
Розмір виконуваного файлу агента	≤ 2 МБ	990 КБ	Виконано
Час повного завантаження веб-інтерфейсу	≤ 3 с	2,15 с	Виконано
DOMContentLoaded	≤ 3 с	1,92 с	Виконано

Результати тестування продуктивності повністю задовольняють визначеним нефункціональним вимогам, причому за більшістю показників фактичні значення суттєво перевершують цільові. Час відповіді REST API знаходиться в діапазоні 5–78 мілісекунд із середнім значенням близько 30–50 мс, що приблизно у десять разів швидше за встановлений ліміт у 500 мс. Така низька затримка забезпечується поєднанням декількох факторів: ефективною компіляцією .NET 9 з оптимізаціями JIT, мінімальним конвеєром middleware-компонентів, оптимізованими індексами PostgreSQL та використанням гіпертаблиць TimescaleDB для часових рядів, що дозволяє швидко виконувати запити за діапазонами часу. Дашборд оновлює відображення метрик з періодичністю 5 секунд, що відповідає налаштованому інтервалу збору на стороні агента; реальна затримка доставки даних від моменту збору до відображення є суттєво меншою за рахунок push-моделі через WebSocket-з'єднання.

Особливо позитивні показники продемонстрував агентський компонент. Споживання процесорного часу на рівні 1,2 % та обсяг оперативної пам'яті у 27,5 МБ є мінімальними значеннями для системи моніторингу і не спотворюють відображувані метрики сервера, залишаючись значно нижчими за визначені ліміти у 2 % та 50 МБ відповідно. Розмір виконуваного файлу агента становить лише 990 КБ - це безпосередній результат архітектурного рішення на користь нативного C++ зі статичним лінкуванням та використання вбудованих у Windows бібліотек (WinHTTP, PDH, IPHLPAPI) замість сторонніх залежностей. Для порівняння, аналогічні рішення на базі .NET або Go зазвичай мають розмір у десятки мегабайт через включення runtime-середовища та додаткових залежностей.

5.4 Тестування безпеки

Тестування безпеки виконувалося шляхом моделювання негативних сценаріїв - спроб обходу механізмів автентифікації та авторизації - з перевіркою

того, що сервер у кожному випадку повертає відповідну помилку та не виконує запитувану операцію.

Перевірка автентифікації за JWT – здійснено серію спроб надіслати запити до захищених ендпоінтів без заголовка Authorization, з порожнім токеном, з підробленим JWT та з простроченим токеном. У всіх випадках сервер повернув статус 401 Unauthorized. Це підтверджує коректну роботу middleware-компонента JwtAuthentication та усталеного валідатора TokenValidationParameters.

Перевірка рольової авторизації - на тестовому проекті створено акаунти з різними ролями і виконано спроби викликати методи SignalR-хаба та REST-ендпоінти, що потребують підвищених прав. Спроби запуску Remote Shell з роллю Viewer завершилися мовчазним відхиленням на стороні хаба. Спроби зміни ролі учасника користувачем без прав Admin повернули статус 403 Forbidden. Спроби видалення агента не власником повернули статус 403 з повідомленням “Only the agent owner can delete it”. Окремо перевірено, що клієнтський інтерфейс коректно приховує недоступні дії - Viewer не бачить кнопки Kill у Task Manager, кнопки Start/Stop у Services тощо. Це створює два рівні захисту: на стороні клієнта та на стороні сервера.

Перевірка функціональної авторизації за підпискою - створено тестових користувачів з різними рівнями підписки і виконано спроби виклику ендпоінтів, захищених атрибутом [SubscriptionGuard]. Спроби виклику команди kill_process користувачем без доступа повернули статус 403 . Спроби виконання Remote Desktop користувачем без доступа повернули статус 403. Окремо перевірено сценарій, коли учасник проекту отримує доступ до Pro-функцій завдяки підписці творця проекту - запит виконався успішно, що підтверджує коректну роботу контекстної логіки в ProjectContextMiddleware.

Перевірка автентифікації агентів - виконано спроби надсилання метрик на ендпоінт /api/metrics без агентського токена, з валідним JWT-токеном користувача та з відкликаним токеном. У всіх випадках сервер повернув статус 401 з повідомленням “Agent token required” або “Invalid agent token”. Це

підтверджує коректну роботу AgentAuthMiddleware та механізму перевірки хешу токена в TokenService.

Перевірка захисту транспортного рівня - при розгортанні системи у виробничому середовищі необхідно забезпечити TLS 1.2+ для всіх з'єднань. У середовищі розробки використовувалися самопідписані сертифікати, що дозволяє локально перевірити коректність обробки шифрованих з'єднань. Спроби підключення через незашифрований протокол HTTP до серверу, налаштованого виключно на HTTPS, відхилялися reverse-proxy на рівні нижче за сервер застосунку.

Узагальнені результати тестування безпеки підтверджують коректну роботу всіх трьох рівнів захисту. Сервер послідовно дотримується принципу “безпека за замовчуванням” - при будь-яких сумнівах щодо прав запит відхиляється, а інформація про точну причину відмови не розкривається атакувальнику, що відповідає рекомендаціям OWASP [11] щодо мінімізації витоку даних через повідомлення про помилки.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи бакалавра проведено комплексне дослідження сучасного стану галузі систем моніторингу продуктивності серверів та розроблено власну SaaS-платформу класу APM, орієнтовану на моніторинг Windows-серверів з вбудованими інструментами дистанційного управління. Поставлені на початку роботи мета і задачі виконано в повному обсязі.

Аналіз предметної галузі та порівняльний огляд чотирьох найпоширеніших APM-рішень за ключовими критеріями виявив відсутність на ринку рішення, яке б одночасно поєднувало SaaS-розгортання, рольову модель доступу в проектах та інструменти дистанційного виконання команд. Виявлена прогалина обґрунтувала доцільність розробки нової платформи, орієнтованої саме на цей сегмент.

Для проектованої системи сформульовано функціональні вимоги та кількісні нефункціональні вимоги до продуктивності, безпеки і зручності використання. На основі вимог обґрунтовано вибір технологічного стеку: C++17 з вбудованими в Windows бібліотеками PDH і WinHTTP для агентського компонента, фреймворк ASP.NET Core .NET 9 для серверної частини, PostgreSQL з розширенням TimescaleDB для зберігання даних, React 19 з TypeScript і Material UI v7 для веб-інтерфейсу. Окремим архітектурним рішенням є гібридний підхід до двосторонньої комунікації – “сирий” WebSocket для агента та SignalR для веб-клієнта, що зберігає легковагість агентського компонента і одночасно дає переваги високорівневої абстракції на стороні браузера.

Спроектована трирівнева архітектура має чіткий розподіл відповідальності між компонентами. Агентський компонент має модульну структуру з підсистемами збору метрик, реєстрації, транспорту та виконання команд. На стороні сервера побудовано конвеєр обробки запитів з власними middleware-компонентами автентифікації агентів та резолвінгу контексту проекту.

Реляційна модель бази даних включає чотирнадцять таблиць, серед яких дві гіпертаблиці TimescaleDB обслуговують часові ряди метрик з автоматичним видаленням даних після тридцяти днів зберігання.

Агентський компонент реалізовано мовою C++ із застосуванням незалежних колекторів метрик. Реєстрація на сервері відбувається за одноразовим кодом підключення та апаратним ідентифікатором, а передача даних здійснюється у режимі push через WebSocket з автоматичним відкатом на HTTP при недоступності основного каналу. Інструменти дистанційного управління реалізовано на нативних Windows API: інтерактивний термінал - через Pseudo Console API, віддалений робочий стіл - через апаратно-прискорений механізм DXGI Desktop Duplication з відкатом на програмний рендерер WARP за відсутності графічного драйвера.

Серверну частину побудовано як багатопроектне рішення з REST-контролерами, SignalR-хабом з груповою маршрутизацією повідомлень та окремим обробником WebSocket-з'єднань для агентів. Безпеку реалізовано на двох рівнях: автентифікація користувачів за стандартом JWT, окрема схема для агентів на основі непрозорих токенів з SHA-256 хешуванням у базі даних.

Веб-інтерфейс реалізовано як односторінковий застосунок на React 19 з централізованим SignalR-сервісом. Дашборд оновлює графіки метрик у реальному часі, інтерактивний термінал побудовано на основі xterm.js з прямим транзитом ANSI-послідовностей від ConPTY, віддалений робочий стіл відображається на HTML5 Canvas з адаптивним стисненням JPEG-кадрів. Глобальний стан застосунку зосереджено в Zustand-сторі з персистентністю критичних полів у локальному сховищі браузера.

Тестування платформи проведено в трьох напрямах. Функціональне тестування покрило двадцять чотири ключових сценаріїв роботи системи, всі тест-кейси пройшли перевірку успішно. Виміряні показники продуктивності суттєво перевершили цільові: середній час відповіді REST API становить 30–50 мілісекунд при ліміті 500 мс, кадри Remote Desktop надходять зі швидкістю 40–50 кадрів на секунду, веб-інтерфейс повністю завантажується за 2,15 секунди.

Розмір виконуваного файлу агента - 990 кілобайт при споживанні менше двох відсотків процесорного часу та близько тридцяти мегабайт оперативної пам'яті. Тестування безпеки негативними сценаріями підтвердило коректну роботу всіх трьох рівнів захисту: спроби виконати операції без необхідних прав послідовно відхиляються сервером з відповідними кодами HTTP 401 та 403.

Розроблена платформа має практичну цінність для адміністраторів Windows-серверів та команд DevOps, оскільки об'єднує в одному інтерфейсі функціональність, для якої традиційно потрібно кілька окремих програмних засобів. Підтримка командної роботи через проекти роблять рішення придатним як для індивідуальних адміністраторів, так і для корпоративних команд з розподіленою відповідальністю. Серед напрямів подальшого розвитку слід відзначити підтримку Linux і macOS на стороні агента, інтеграцію з зовнішніми системами сповіщень - Telegram, Slack, електронною поштою, та додавання алгоритмів автоматичного виявлення аномалій у метриках.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Попсуй М.С., Золотухіна О.А. Аналіз засобів моніторингу продуктивності додатків ОС Windows. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.234 -237.
2. Попсуй М.С., Золотухіна О.А. Забезпечення безпеки інформації в АРМ-системі. Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. – К.: ДУІКТ, 2026, С.230-232.

ПЕРЕЛІК ПОСИЛАНЬ

1. Aceto G., Botta A., De Donato W., Pescapè A. Cloud monitoring: A survey. *Computer Networks*. 2013. Vol. 57, No. 9. P. 2093–2115. DOI: 10.1016/j.comnet.2013.04.001.
2. Datadog Inc. Datadog Documentation [Електронний ресурс]. URL: https://docs.datadoghq.com (дата звернення: 02.04.2026).
3. Fette I., Melnikov A. The WebSocket Protocol : RFC 6455. IETF, 2011. URL: https://datatracker.ietf.org/doc/html/rfc6455 (дата звернення: 08.04.2026).
4. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : PhD thesis. University of California, Irvine, 2000. 180 p. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 07.04.2026).
5. Grafana Labs. Grafana Documentation [Електронний ресурс]. URL: https://grafana.com/docs/grafana/latest (дата звернення: 03.04.2026).
6. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : RFC 7519. IETF, 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 07.04.2026).
7. Meta Platforms Inc. React Documentation [Електронний ресурс]. URL: https://react.dev (дата звернення: 08.04.2026).
8. Microsoft Corporation. ASP.NET Core documentation [Електронний ресурс]. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core (дата звернення: 07.04.2026).
9. Microsoft Corporation. Introduction to ASP.NET Core SignalR [Електронний ресурс]. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction (дата звернення: 08.04.2026).

10. Microsoft Corporation. Windows API Index [Електронний ресурс]. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list (дата звернення: 13.04.2026).
11. OWASP Foundation. OWASP Top Ten - Web Application Security Risks [Електронний ресурс]. URL: https://owasp.org/Top10 (дата звернення: 14.04.2026).
12. Paessler AG. PRTG Network Monitor - Documentation [Електронний ресурс]. URL: https://www.paessler.com/manuals/prtg (дата звернення: 02.04.2026).
13. PostgreSQL Global Development Group. PostgreSQL 17 Documentation [Електронний ресурс]. URL: https://www.postgresql.org/docs/17 (дата звернення: 07.04.2026).
14. Prometheus Authors. Prometheus Documentation [Електронний ресурс]. URL: https://prometheus.io/docs/introduction/overview (дата звернення: 03.04.2026).
15. Provos N., Mazières D. A future-adaptable password scheme. *Proceedings of USENIX Annual Technical Conference, FREENIX Track*. Monterey, CA, 1999. P. 81–91.
16. Sandhu R. S., Coyne E. J., Feinstein H. L., Youman C. E. Role-Based Access Control Models. *Computer*. 1996. Vol. 29, No. 2. P. 38–47. DOI: 10.1109/2.485845.
17. Turnbull J. The Art of Monitoring. Lulu.com, 2016. 390 p.
18. Zabbix LLC. Zabbix Documentation 7.0 [Електронний ресурс]. URL: https://www.zabbix.com/documentation/7.0/en (дата звернення: 01.04.2026).
19. Microsoft Corporation. Event Tracing for Windows (ETW) [Електронний ресурс]. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/etw/event-tracing-portal (дата звернення: 13.04.2026).
20. Richardson C. Microservices Patterns: With Examples in Java. Shelter Island : Manning Publications, 2018. 520 p. ISBN 978-1-61729-454-9.

21. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 612 p. ISBN 978-1-4920-3402-5.
22. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley Professional, 2021. 464 p. ISBN 978-0-13-688609-9.
23. Majors C., Fong-Jones L., Miranda G. Observability Engineering: Achieving Production Excellence. Sebastopol : O'Reilly Media, 2022. 401 p. ISBN 978-1-4920-7644-5.
24. Khononov V. Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. Sebastopol : O'Reilly Media, 2021. 340 p. ISBN 978-1-0981-0013-1.
25. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol : O'Reilly Media, 2017. 614 p. ISBN 978-1-4493-7332-0.
26. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p. ISBN 978-0-13-449416-6.
27. Russinovich M., Solomon D. A., Ionescu A. Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More. 7th ed. Redmond : Microsoft Press, 2017. 800 p. ISBN 978-0-7356-8418-8.
28. Rescorla E. The Transport Layer Security (TLS) Protocol Version 1.3 : RFC 8446. IETF, 2018. URL: https://datatracker.ietf.org/doc/html/rfc8446 (дата звернення: 14.04.2026).
29. Pahl C., Jamshidi P. Microservices: A Systematic Mapping Study. Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER). Rome, 2016. P. 137–146. DOI: 10.5220/0005785501370146.
30. Pourmajidi W., Miranskyy A. Logging in the Age of Web Services. IEEE Software. 2018. Vol. 35, No. 4. P. 7–11. DOI: 10.1109/MS.2018.2902301.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка APM-системи для моніторингу продуктивності Windows-серверів

Виконав студент 4 курсу
групи ПД-44
Попсуй Микола Сергійович
Керівник роботи
доцент кафедри, канд. техн. наук, доцент Золотухіна Оксана Анатоліївна
Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи спрощення процесу моніторингу продуктивності серверів та застосунків у розподілених IT-інфраструктурах на базі операційної системи Windows шляхом розробки модульної SaaS-платформи.

Об'єкт дослідження процес моніторингу продуктивності серверів та застосунків у розподілених IT-інфраструктурах на базі операційної системи Windows.

Предмет дослідження архітектура та методи реалізації SaaS-платформи для моніторингу продуктивності додатків і серверів у Windows-орієнтованих середовищах.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз предметної галузі та порівняльний аналіз існуючих АРМ-систем.
2. Сформулювати функціональні та нефункціональні вимоги до системи й обґрунтувати вибір технологічного стеку.
3. Спроектувати архітектуру платформи та модель бази даних.
4. Реалізувати агентський компонент.
5. Реалізувати серверну частину.
6. Реалізувати веб-інтерфейс з відображенням метрик у реальному часі та інструментами дистанційного управління.
7. Провести функціональне тестування, тестування продуктивності та безпеки розробленої системи.

3

АНАЛІЗ АНАЛОГІВ

Критерій	Datadog	PRTG	Zabbix	Prometheus + Grafana	APM System
Модель розгортання	Хмарне	On-premise	On-premise	On-premise	Хмарне
Модель збору даних	Push	Pull	Pull / Push	Pull (scraping exporters)	Push
Цільова ОС агента	Windows, Linux, macOS, Docker, K8s	Windows	Windows, Linux, Unix	Windows	Windows
Залежності агента	Go-runtime	Сторонні бібліотеки	Сторонні бібліотеки	Окремі exporters	Без зовнішніх залежностей
Затримка відображення метрик	До 2 хвилин	Залежить від інтервалу опитування	Залежить від режиму	Залежить від scrape_interval	До 5 секунд
Інструменти дистанційного управління серверами	Відсутні	Відсутні	Відсутні	Відсутні	Присутні
Командна робота	Підтримується	Відсутня	Відсутня	Тільки в Grafana Enterprise	Підтримується
Рольова модель доступу	Присутні	Відсутня	Присутні	Тільки у Grafana Enterprise	4-рівнева ієрархія з кастомізацією
Складність первинного розгортання	Низька	Середня	Висока	Висока	Низька

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1. Збір системних метрик у реальному часі.
2. Реєстрація агентів за одноразовим кодом підключення.
3. JWT-автентифікація та авторизація користувачів.
4. Управління проектами з рольовою моделлю.
5. Призначення агентів до проектів із real-time синхронізацією через SignalR.
6. Дистанційне управління серверами через захищені сесії.

Нефункціональні вимоги:

1. Агент моніторингу не повинен мати зовнішніх залежностей
2. Завантаження процесора підконтрольного сервера агентом не повинно перевищувати 2%
3. Споживання оперативної пам'яті агентом не повинно перевищувати 50 МБ
4. Система повинна підтримувати горизонтальне масштабування серверної частини.
5. Час первинного завантаження інтерфейсу не повинен перевищувати 3 секунди.
6. Затримка доставки метрик від агента до інтерфейсу не повинна перевищувати 5 секунд.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Агент:
- C++ (Windows API, WinHTTP)
- spdlog
- WebSocket client

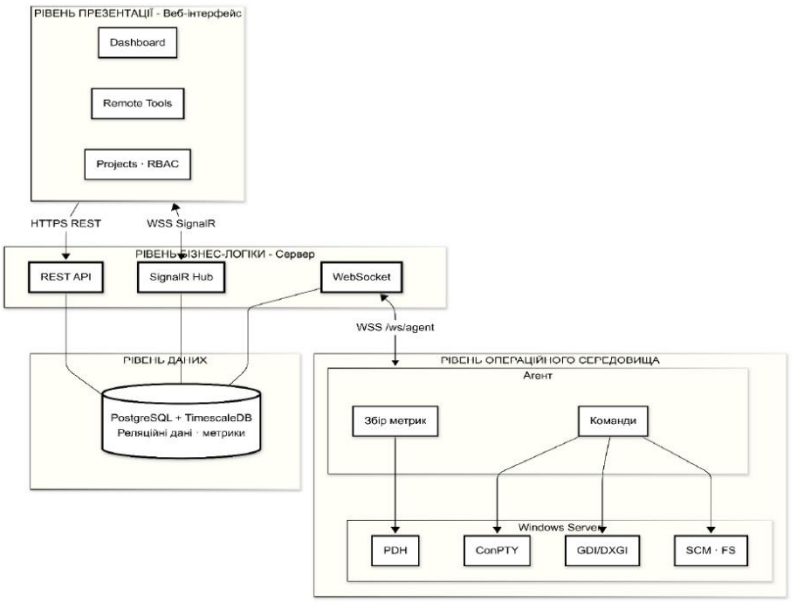
Бекенд:
- C# / .NET 9
- ASP.NET Core Web API
- Entity Framework Core
- PostgreSQL
- SignalR
- JWT / bcrypt



Фронтенд:
- React 19 + TypeScript
- Material UI v7
- Vite
- Zustand
- SignalR client

6

АРХІТЕКТУРА СИСТЕМИ

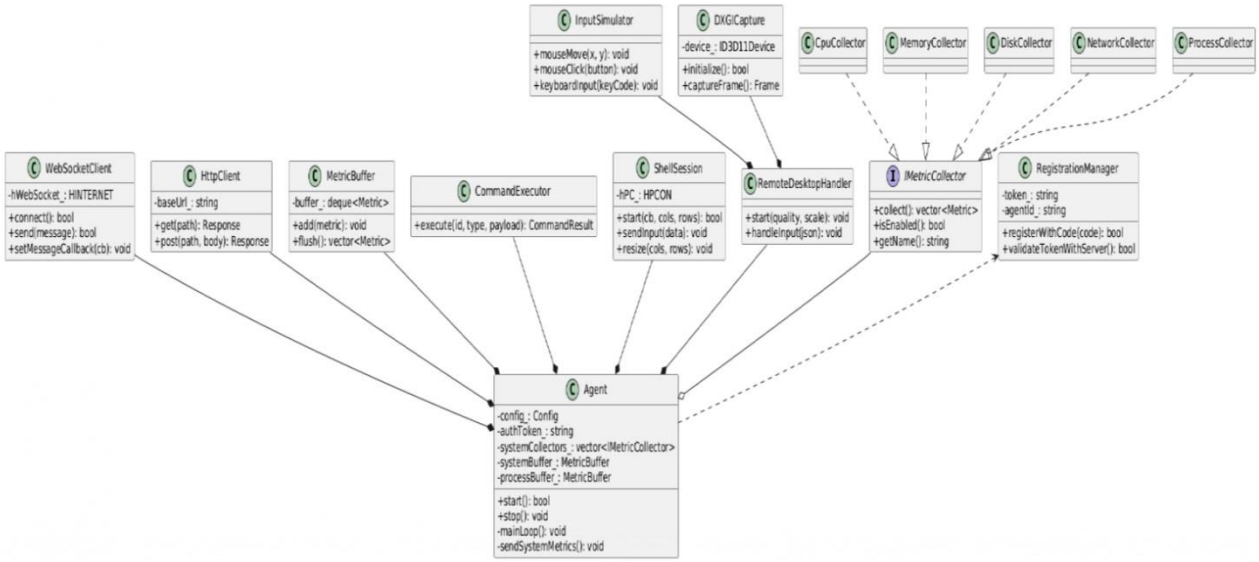


7

РЕЛЯЦІЙНА МОДЕЛЬ БАЗИ ДАНИХ

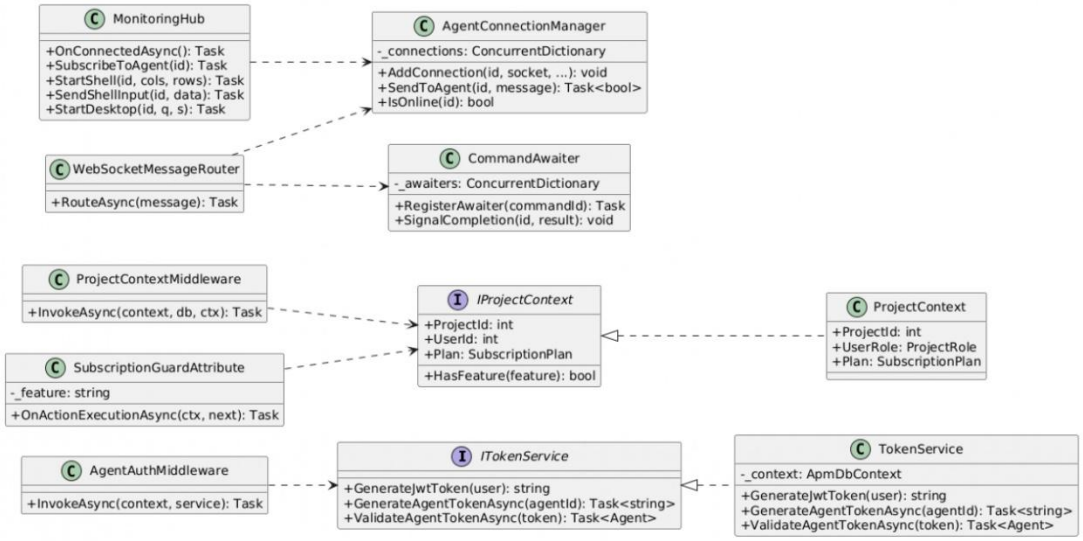
8

СТРУКТУРА АГЕНТСЬКОГО КОМПОНЕНТА



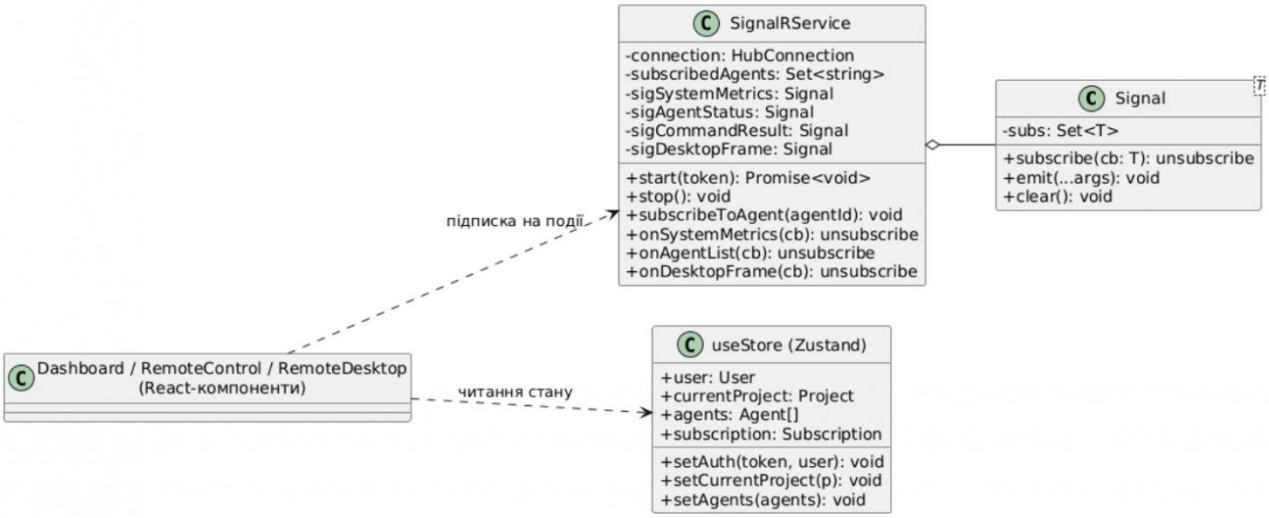
9

СТРУКТУРА СЕРВЕРНИХ КЛАСІВ



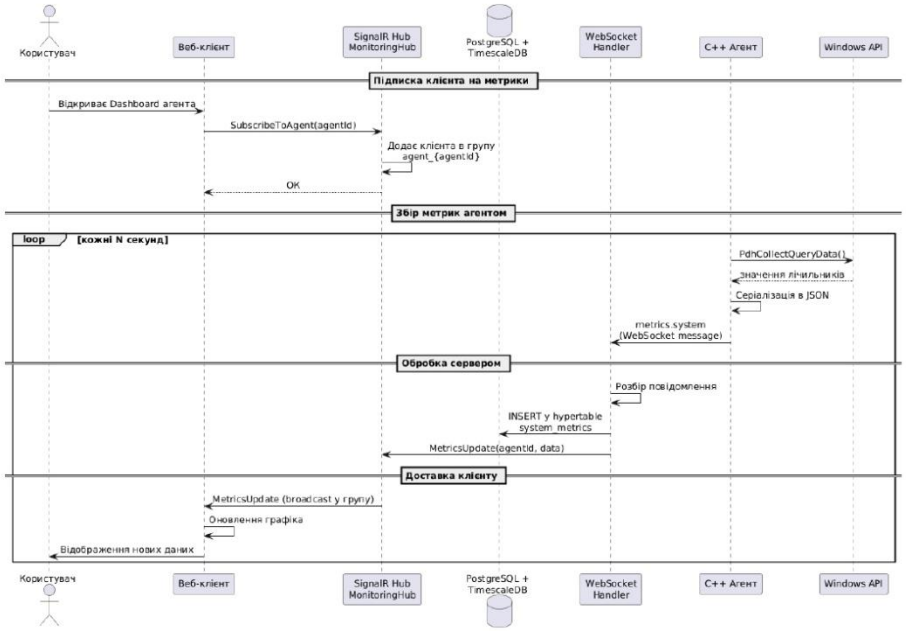
10

СТРУКТУРА КЛІЄНТСЬКОГО SIGNALR-СЕРВІСУ



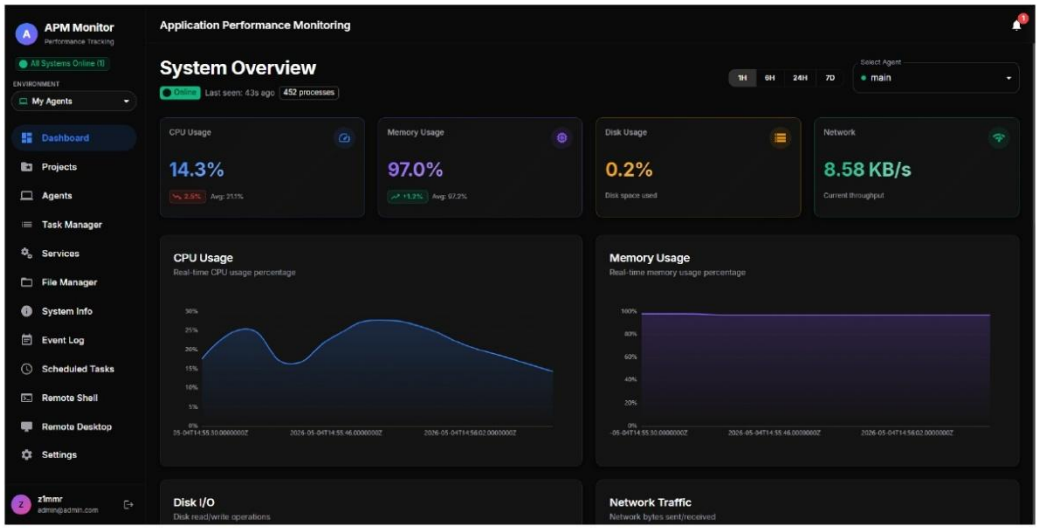
11

ПЕРЕДАЧА МЕТРИК У РЕАЛЬНОМУ ЧАСІ



12

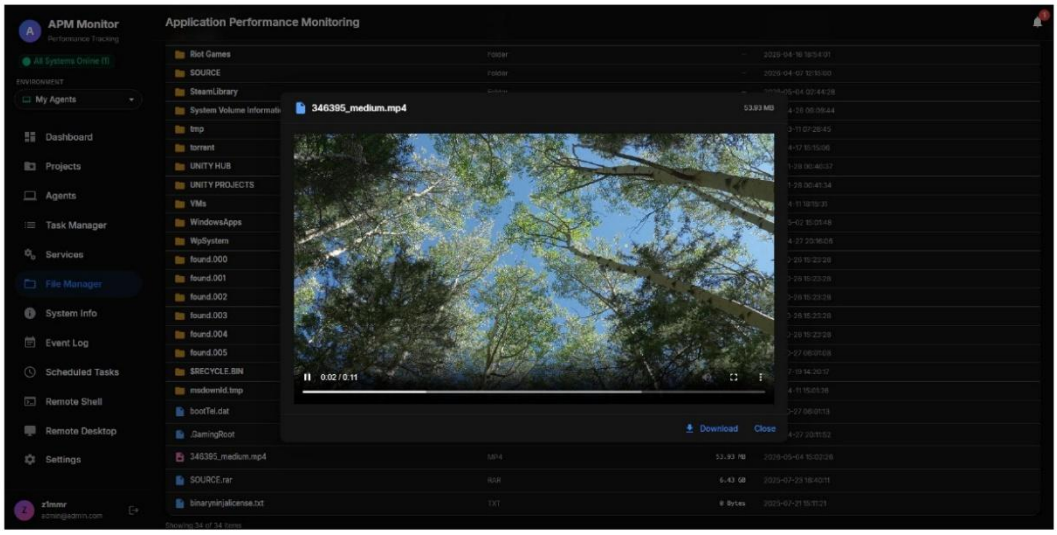
ЕКРАННІ ФОРМИ



Dashboard

13

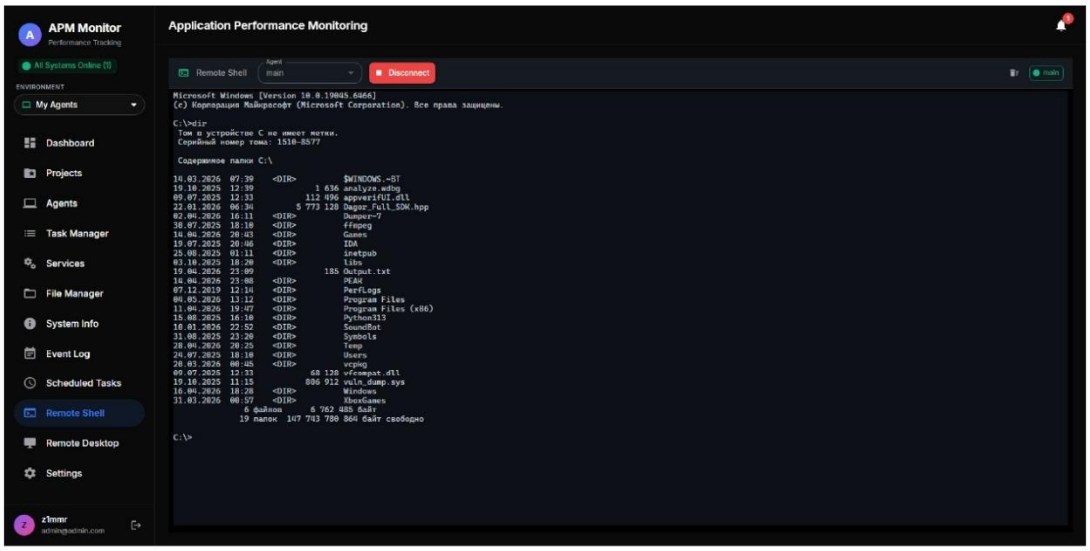
ЕКРАННІ ФОРМИ



File Manager

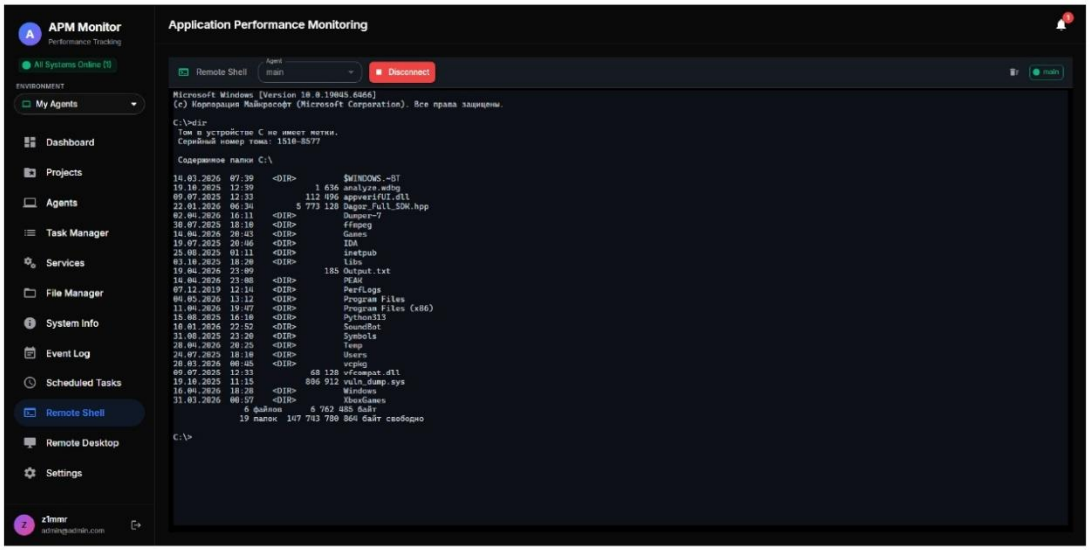
14

ЕКРАННІ ФОРМИ



Remote Shell

ЕКРАННІ ФОРМИ



Remote Shell

15

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Попсуй М.С., Золотухіна О.А. Аналіз засобів моніторингу продуктивності додатків ОС Windows. Матеріали ІІ Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. – К.: ДУІКТ, 2025, С.234 -237.

2. Попсуй М.С., Золотухіна О.А. Забезпечення безпеки інформації в АРМ-системі. Матеріали Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, ДУІКТ. Збірник тез. *(подано до друку).*

17

ВИСНОВКИ

1. Проведено аналіз предметної галузі та порівняльний огляд чотирьох провідних АРМ-систем - Datadog, PRTG, Zabbix, Prometheus + Grafana - за ключовими критеріями. Виявлено відсутність на ринку рішення, яке б одночасно поєднувало SaaS-розгортання, рольову модель доступу та інструменти дистанційного управління - це обґрунтувало доцільність розробки нової платформи.
2. Сформульовано функціональні та кількісні нефункціональні вимоги до системи; обґрунтовано технологічний стек: C++17 з PDH і WinHTTP для агента, ASP.NET Core .NET 9 з SignalR для серверної частини, PostgreSQL з TimescaleDB для зберігання даних, React 19 з TypeScript і Material UI v7 для веб-інтерфейсу.
3. Спроектовано трирівневу архітектуру системи з реляційною моделлю бази даних, що включає гіпертаблиці TimescaleDB для часових рядів метрик з 30-денною ретенцією.
4. Реалізовано нативний C++-агент розміром 990 КБ без зовнішніх залежностей (≤ 2 % CPU, ~ 30 МБ RAM). Реєстрація відбувається за одноразовим кодом підключення та апаратним ідентифікатором. Інструменти дистанційного управління - на нативних Windows API: ConPTY для терміналу, DXGI Desktop Duplication для віддаленого робочого столу.
5. Реалізовано серверну частину на ASP.NET Core .NET 9 з REST API, SignalR-хабом та окремим обробником WebSocket-з'єднань для агентів. Безпеку реалізовано на двох рівнях: автентифікація користувачів за стандартом JWT та окрема схема для агентів на основі непрозорих токенів з SHA-256 хешуванням.
6. Реалізовано веб-інтерфейс на React 19 з централізованим SignalR-сервісом: дашборд з метриками реального часу, повноцінний термінал на xterm.js з прямим транзитом ANSI-послідовностей від ConPTY, віддалений робочий стіл на HTML5 Canvas з адаптивним стисненням JPEG.
7. Проведено комплексне тестування системи у трьох напрямах: функціональне, продуктивності та безпеки. Усі 24 функціональні тест-кейси, що охоплюють реєстрацію, моніторинг метрик, дистанційне управління та командну роботу, пройдено успішно. Виміряні показники продуктивності: середній час відповіді REST API становить 30–50 мс, повне завантаження веб-інтерфейсу - 2,15 секунди, розмір виконуваного файлу агента - 990 КБ при споживанні менше 2 % процесорного часу та близько 30 МБ оперативної пам'яті. Тестування безпеки негативними сценаріями підтвердило коректну роботу всіх рівнів.

18

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Лістинги агентського компонента

Інтерфейс колектора метрик
IMetricCollector.h:

```
class IMetricCollector {
public:
    virtual ~IMetricCollector() = default;
    virtual std::vector<Metric> collect() = 0;
    virtual bool isEnabled() const = 0;
    virtual std::string getName() const = 0;
};
```

Реалізація колектора процесорних метрик
CpuCollector.cpp:

```
CpuCollector::CpuCollector(const std::string& agentId)
        : agentId_(agentId)
        , enabled_(true)
        , initialized_(false)
        , cpuQuery_(nullptr)
        , cpuTotal_(nullptr)
    {
        initialized_ = initialize();
    }

    CpuCollector::~CpuCollector() {
        cleanup();
    }

    bool CpuCollector::initialize() {
        PDH_STATUS status = PdhOpenQuery(NULL, NULL,
&cpuQuery_);
        if (status != ERROR_SUCCESS) {
            spdlog::error("Failed to open PDH query: {}", status);
            return false;
        }

        status = PdhAddEnglishCounter(cpuQuery_,
L"\\Processor(_Total)\\% Processor Time", NULL,
&cpuTotal_);
        if (status != ERROR_SUCCESS) {
            spdlog::error("Failed to add CPU counter: {}",
status);
            PdhCloseQuery(cpuQuery_);
            return false;
        }

        PdhCollectQueryData(cpuQuery_);

        spdlog::info("CPU Collector initialized successfully");
        return true;
    }

    void CpuCollector::cleanup() {
        if (cpuQuery_) {
            PdhCloseQuery(cpuQuery_);
            cpuQuery_ = nullptr;
        }
    }

    double CpuCollector::getCpuUsage() {
        if (!initialized_) {
            return -1.0;
        }

        PDH_STATUS status =
PdhCollectQueryData(cpuQuery_);
        if (status != ERROR_SUCCESS) {
            spdlog::error("Failed to collect query data: {}",
status);
            return -1.0;
        }

        PDH_FMT_COUNTERVALUE counterVal;
        status = PdhGetFormattedCounterValue(cpuTotal_,
PDH_FMT_DOUBLE, NULL, &counterVal);
        if (status != ERROR_SUCCESS) {
            spdlog::error("Failed to get formatted counter value:
{}", status);
            return -1.0;
        }

        return counterVal.doubleValue;
    }

    std::vector<Metric> CpuCollector::collect() {
        std::vector<Metric> metrics;

        if (!enabled_ || !initialized_) {
            return metrics;
        }

        double cpuUsage = getCpuUsage();
        if (cpuUsage >= 0) {
            Metric metric;
            metric.agentId = agentId_;
            metric.metricName = "cpu.usage.percent";
            metric.type = MetricType::CPU;
            metric.value = cpuUsage;
            metric.timestamp = std::chrono::system_clock::now();
            metric.metadata = nlohmann::json::object();

            metrics.push_back(metric);
            spdlog::debug("CPU Usage: {:.2f}%", cpuUsage);
        }

        return metrics;
    }

}
```

Основний цикл агента
Agent.cpp, фрагмент:

```
void Agent::mainLoop() {
    spdlog::info("Agent main loop started");

    reportSystemInfo();

    connectWebSocket();
    lastWsReconnectAttempt_ =
std::chrono::system_clock::now();

    while (running_) {
        auto now = std::chrono::system_clock::now();

        if (!useWebSocket_) {
```

```
        auto wsElapsed =
std::chrono::duration_cast<std::chrono::seconds>(
            now - lastWsReconnectAttempt_).count();
        if (wsElapsed >= wsReconnectInterval_) {
            connectWebSocket();
            lastWsReconnectAttempt_ = now;
            if (!useWebSocket_) {
                wsReconnectInterval_ =
(std::min)(wsReconnectInterval_ * 2, 120);
                spdlog::debug("WebSocket reconnect backoff:
{}s", wsReconnectInterval_);
            }
        }
    }
    else if (wsClient_ && !wsClient_->isConnected()) {
        spdlog::warn("WebSocket disconnected, falling back
to HTTP");
        useWebSocket_ = false;
        wsReconnectInterval_ = 10;
        lastWsReconnectAttempt_ = now;
    }

    int collectionInterval = policy_.getCollectionInterval();

    auto elapsedSystem =
std::chrono::duration_cast<std::chrono::seconds>(
        now - lastSystemCollection_).count();

    if (elapsedSystem >= collectionInterval) {
        collectSystemMetrics();
        sendSystemMetrics();
        lastSystemCollection_ = now;
    }

    if (processCollector_) {
        auto elapsedProcess =
std::chrono::duration_cast<std::chrono::seconds>(
            now - lastProcessCollection_).count();

        if (elapsedProcess >= collectionInterval) {
            collectProcessMetrics();
            sendProcessSnapshot();
            lastProcessCollection_ = now;
        }
    }

    std::this_thread::sleep_for(std::chrono::seconds(1));
  }

  spdlog::info("Agent main loop stopped");
}

void Agent::sendSystemMetrics() {
  if (systemBuffer_->isEmpty()) return;

  auto metrics = systemBuffer_->flush();

  nlohmann::json jsonArray = nlohmann::json::array();
  for (const auto& metric : metrics) {
    jsonArray.push_back(metric.toJson());
  }

  nlohmann::json payload;
  payload["agentId"] = config_.getAgentId();
  payload["timestamp"] =
std::chrono::system_clock::to_time_t(
    std::chrono::system_clock::now());
  payload["metrics"] = jsonArray;

  if (useWebSocket_ && wsClient_ && wsClient_-
>isConnected()) {
    payload["type"] = "metrics.system";
    if (wsClient_->send(payload.dump())) {
      spdlog::debug("WS: System metrics sent ({}
metrics)", metrics.size());
      return;
    }
    spdlog::warn("WS: Send failed, falling back to HTTP");
    useWebSocket_ = false;
  }

  // HTTP fallback
  try {
    auto response = httpClient_->post("/api/metrics",
payload);
    if (response.success) {
      spdlog::debug("HTTP: System metrics sent (status:
{})", response.statusCode);
    }
    else {
      spdlog::error("HTTP: Failed to send system metrics
(status: {})", response.statusCode);
      if (systemBuffer_->size() < 400) {
        systemBuffer_->add(metrics);
      }
    }
  }
  catch (const std::exception& e) {
    spdlog::error("HTTP: Exception sending metrics: {}",
e.what());
    if (systemBuffer_->size() < 400) {
      systemBuffer_->add(metrics);
    }
  }
}

void Agent::connectWebSocket() {
  if (useWebSocket_ || !wsClient_) return;

  wsClient_->stopReceiveLoop();

  spdlog::info("Attempting WebSocket connection...");

  try {
    if (wsClient_->connect()) {
      wsClient_->startReceiveLoop();
      useWebSocket_ = true;
      wsReconnectInterval_ = 10;
      spdlog::info("WebSocket transport active");
    }
    else {
      spdlog::debug("WebSocket connection failed, using
HTTP");
    }
  }
  catch (const std::exception& e) {
    spdlog::debug("WebSocket connection error: {}",
e.what());
  }
}
```

Інтерактивний термінал на основі ConPTY
ShellSession.cpp:

```
bool ShellSession::start(OutputCallback onOutput, short cols,
short rows) {
  if (active_.load()) {
    spdlog::warn("ShellSession: Already active");
    return false;
```

```
    }

    onOutput_ = onOutput;

    HANDLE ptyInputRead = nullptr, ptyInputWrite = nullptr;
    HANDLE ptyOutputRead = nullptr, ptyOutputWrite =
nullptr;

    if (!CreatePipe(&ptyInputRead, &ptyInputWrite, nullptr,
0)) {
        spdlog::error("ShellSession: Failed to create input pipe
({})", GetLastError());
        return false;
    }

    if (!CreatePipe(&ptyOutputRead, &ptyOutputWrite,
nullptr, 0)) {
        spdlog::error("ShellSession: Failed to create output pipe
({})", GetLastError());
        CloseHandle(ptyInputRead);
        CloseHandle(ptyInputWrite);
        return false;
    }

    COORD size = { cols, rows };
    HRESULT hr = CreatePseudoConsole(size, ptyInputRead,
ptyOutputWrite, 0, &hPC_);
    if (FAILED(hr)) {
        spdlog::error("ShellSession: CreatePseudoConsole
failed (hr=0x{:08X})", static_cast<unsigned>(hr));
        CloseHandle(ptyInputRead);
        CloseHandle(ptyInputWrite);
        CloseHandle(ptyOutputRead);
        CloseHandle(ptyOutputWrite);
        return false;
    }

    CloseHandle(ptyInputRead);
    CloseHandle(ptyOutputWrite);

    hPipeOut_ = ptyInputWrite;
    hPipeIn_ = ptyOutputRead;

    SIZE_T attrListSize = 0;
    InitializeProcThreadAttributeList(nullptr, 1, 0,
&attrListSize);

    auto attrList =
reinterpret_cast<PPROC_THREAD_ATTRIBUTE_LIST>(
HeapAlloc(GetProcessHeap(), 0, attrListSize));
    if (!attrList) {
        spdlog::error("ShellSession: HeapAlloc failed");
        stop();
        return false;
    }

    if (!InitializeProcThreadAttributeList(attrList, 1, 0,
&attrListSize)) {
        spdlog::error("ShellSession:
InitializeProcThreadAttributeList failed ({})",
GetLastError());
        HeapFree(GetProcessHeap(), 0, attrList);
        stop();
        return false;
    }

    if (!UpdateProcThreadAttribute(attrList, 0,
PROC_THREAD_ATTRIBUTE_PSEUDOCONSOLE,
        hPC_, sizeof(HPCON), nullptr, nullptr)) {
        spdlog::error("ShellSession: UpdateProcThreadAttribute
failed ({})", GetLastError());
        DeleteProcThreadAttributeList(attrList);
        HeapFree(GetProcessHeap(), 0, attrList);
        stop();
        return false;
    }

    STARTUPINFOEXW si = {};
    si.StartupInfo.cb = sizeof(si);
    si.lpAttributeList = attrList;

    PROCESS_INFORMATION pi = {};
    wchar_t cmdLine[] = L"cmd.exe";
    const wchar_t* startDir = L"C:\\";

    if (!CreateProcessW(nullptr, cmdLine, nullptr, nullptr,
FALSE,
        EXTENDED_STARTUPINFO_PRESENT, nullptr,
startDir,
        &si.StartupInfo, &pi)) {
        spdlog::error("ShellSession: CreateProcessW failed
({})", GetLastError());
        DeleteProcThreadAttributeList(attrList);
        HeapFree(GetProcessHeap(), 0, attrList);
        stop();
        return false;
    }

    hProcess_ = pi.hProcess;
    CloseHandle(pi.hThread);
    DeleteProcThreadAttributeList(attrList);
    HeapFree(GetProcessHeap(), 0, attrList);

    active_.store(true);
    readThread_ = std::thread(&ShellSession::readLoop, this);

    spdlog::info("ShellSession: Started with ConPTY (pid={},
{}x{})", pi.dwProcessId, cols, rows);
    return true;
}
```

Лістинги серверної частини

Сервіс генерації токенів
TokenService.cs:

```
public class TokenService : ITokenService
{
    private readonly ApmDbContext _context;
    private readonly IConfiguration _configuration;

    public TokenService(ApmDbContext context,
IConfiguration configuration)
    {
        _context = context;
        _configuration = configuration;
    }

    public async Task<string> GenerateAgentTokenAsync(int
agentId)
    {
        var rawToken =
Convert.ToHexString(RandomNumberGenerator.GetBytes(3
2)).ToLowerInvariant();

        var tokenHash = HashToken(rawToken);

        var agentToken = new AgentToken
```

```
        {
            AgentId = agentId,
            TokenHash = tokenHash,
            CreatedAt = DateTime.UtcNow
        };

        _context.AgentTokens.Add(agentToken);
        await _context.SaveChangesAsync();

        return rawToken;
    }

    public async Task<Agent?>
ValidateAgentTokenAsync(string token)
    {
        var tokenHash = HashToken(token);

        var agentToken = await _context.AgentTokens
            .Include(t => t.Agent)
            .FirstOrDefaultAsync(t => t.TokenHash ==
tokenHash && !t.IsRevoked);

        if (agentToken == null)
            return null;

        agentToken.LastUsedAt = DateTime.UtcNow;
        await _context.SaveChangesAsync();

        return agentToken.Agent;
    }

    public string GenerateJwtToken(User user)
    {
        var jwtSettings =
_configuration.GetSection("JwtSettings");
        var secretKey = jwtSettings["SecretKey"] ?? throw new
InvalidOperationException("JWT SecretKey not
configured");
        var issuer = jwtSettings["Issuer"] ?? "ApmBackend";
        var audience = jwtSettings["Audience"] ??
"ApmFrontend";
        var expiryHours = int.Parse(jwtSettings["ExpiryHours"]
?? "24");

        var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey
));
        var credentials = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);

        var claims = new[]
        {
            new Claim(ClaimTypes.NameIdentifier,
user.Id.ToString()),
            new Claim(ClaimTypes.Name, user.Username),
            new Claim(ClaimTypes.Email, user.Email),
            new Claim("userId", user.Id.ToString())
        };

        var token = new JwtSecurityToken(
            issuer: issuer,
            audience: audience,
            claims: claims,
            expires: DateTime.UtcNow.AddHours(expiryHours),
            signingCredentials: credentials
        );

        return new
JwtSecurityTokenHandler().WriteToken(token);
    }

    public async Task RevokeAgentTokensAsync(int agentId)
    {
        var tokens = await _context.AgentTokens
            .Where(t => t.AgentId == agentId && !t.IsRevoked)
            .ToListAsync();

        foreach (var token in tokens)
        {
            token.IsRevoked = true;
            token.RevokedAt = DateTime.UtcNow;
        }

        await _context.SaveChangesAsync();
    }

    private static string HashToken(string token)
    {
        var bytes =
SHA256.HashData(Encoding.UTF8.GetBytes(token));
        return Convert.ToHexString(bytes).ToLowerInvariant();
    }
}
```

SignalR-хаб з RPC-методами
MonitoringHub.cs:

```
[Authorize]
public class MonitoringHub : Hub<IMonitoringHubClient>
{
    private readonly ILogger<MonitoringHub> _logger;
    private readonly ApmDbContext _dbContext;

    public MonitoringHub(ILogger<MonitoringHub> logger,
ApmDbContext dbContext)
    {
        _logger = logger;
        _dbContext = dbContext;
    }

    private int? GetUserId()
    {
        var claim = Context.User?.FindFirst("userId")?.Value;
        return claim != null ? int.Parse(claim) : null;
    }

    public override async Task OnConnectedAsync()
    {
        var userId = GetUserId();
        if (userId != null)
        {
            await
Groups.AddToGroupAsync(Context.ConnectionId,
$"user_{userId}");
            var projectIds = await _dbContext.ProjectMembers
                .Where(m => m.UserId == userId.Value)
                .Select(m => m.ProjectId)
                .ToListAsync();

            foreach (var projectId in projectIds)
            {
                await
Groups.AddToGroupAsync(Context.ConnectionId,
$"project_{projectId}");
            }
```

```
            _logger.LogInformation("SignalR: User {UserId}
connected (conn: {ConnId}, projects: {Count})",
                userId, Context.ConnectionId, projectIds.Count);
        }

        await base.OnConnectedAsync();
    }

    public override async Task
OnDisconnectedAsync(Exception? exception)
    {
        var userId = GetUserId();
        _logger.LogInformation("SignalR: User {UserId}
disconnected (conn: {ConnId})", userId,
Context.ConnectionId);
        await base.OnDisconnectedAsync(exception);
    }

    private async Task<bool> UserHasAccessToAgent(int
userId, string agentId)
    {
        var agent = await
_dbContext.Agents.FirstOrDefaultAsync(a => a.AgentId ==
agentId);
        if (agent == null) return false;

        if (agent.OwnerId == userId) return true;

        if (agent.ProjectId.HasValue)
            return await _dbContext.ProjectMembers
                .AnyAsync(m => m.ProjectId ==
agent.ProjectId.Value && m.UserId == userId);

        return false;
    }

    private async Task<int> GetUserRoleLevel(int userId,
string agentId)
    {
        var agent = await
_dbContext.Agents.FirstOrDefaultAsync(a => a.AgentId ==
agentId);
        if (agent == null) return -1;

        if (agent.OwnerId == userId) return 99;

        if (!agent.ProjectId.HasValue) return -1;

        var member = await _dbContext.ProjectMembers
            .FirstOrDefaultAsync(m => m.ProjectId ==
agent.ProjectId.Value && m.UserId == userId);
        if (member == null) return -1;

        return member.Role switch
        {
            "Owner" => 3,
            "Admin" => 2,
            "Operator" => 1,
            "Viewer" => 0,
            _ => 0
        };
    }

    private async Task<int>
GetMinRoleLevelForPermission(string agentId, string
permission, int defaultLevel = 1)
    {
        var agent = await
_dbContext.Agents.FirstOrDefaultAsync(a => a.AgentId ==
agentId);
        if (agent?.ProjectId == null) return defaultLevel;

        var settings = await _dbContext.ProjectSettings
            .FirstOrDefaultAsync(s => s.ProjectId ==
agent.ProjectId.Value);
        if (settings == null) return defaultLevel;

        var roleName = permission switch
        {
            "Shell" => settings.MinRoleForShell,
            "RemoteDesktopInput" =>
settings.MinRoleForRemoteDesktopInput,
            _ => "Operator"
        };

        return roleName switch
        {
            "Owner" => 3,
            "Admin" => 2,
            "Operator" => 1,
            "Viewer" => 0,
            _ => defaultLevel
        };
    }

    public async Task JoinProjectGroup(int projectId)
    {
        var userId = GetUserId();
        if (userId == null) return;

        var isMember = await _dbContext.ProjectMembers
            .AnyAsync(m => m.ProjectId == projectId &&
m.UserId == userId.Value);
        if (!isMember)
        {
            _logger.LogWarning("SignalR: JoinProjectGroup
denied — user {UserId} is not a member of project
{ProjectId}",
                userId, projectId);
            return;
        }

        await
Groups.AddToGroupAsync(Context.ConnectionId,
$"project_{projectId}");
        _logger.LogInformation("SignalR: User {UserId} joined
project_{ProjectId} group", userId, projectId);
    }

    public async Task LeaveProjectGroup(int projectId)
    {
        await
Groups.RemoveFromGroupAsync(Context.ConnectionId,
$"project_{projectId}");
        _logger.LogDebug("SignalR: Connection {ConnId} left
project_{ProjectId} group", Context.ConnectionId,
projectId);
    }

    public async Task SubscribeToAgent(string agentId)
    {
        var userId = GetUserId();
        if (userId == null)
        {
            _logger.LogWarning("SignalR: SubscribeToAgent
denied — no userId in token");
```

```
            return;
        }

        if (!await UserHasAccessToAgent(userId.Value,
agentId))
        {
            _logger.LogWarning("SignalR: User {UserId} tried
to subscribe to agent {AgentId} without access",
                userId, agentId);
            return;
        }

        await
Groups.AddToGroupAsync(Context.ConnectionId,
$"agent_{agentId}");
        _logger.LogDebug("SignalR: User {UserId} subscribed
to agent {AgentId}", userId, agentId);
    }

    public async Task UnsubscribeFromAgent(string agentId)
    {
        await
Groups.RemoveFromGroupAsync(Context.ConnectionId,
$"agent_{agentId}");
        _logger.LogDebug("SignalR: Connection {ConnId}
unsubscribed from agent {AgentId}", Context.ConnectionId,
agentId);
    }

    public async Task StartDesktop(string agentId, int quality
= 50, float scale = 0.5f)
    {
        var userId = GetUserId();
        if (userId == null) return;

        var roleLevel = await GetUserRoleLevel(userId.Value,
agentId);
        if (roleLevel < 1)
        {
            _logger.LogWarning("SignalR: StartDesktop denied
— user {UserId} has insufficient role for agent {AgentId}",
userId, agentId);
            return;
        }

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null ||
!connectionManager.IsOnline(agentId))
        {
            _logger.LogWarning("SignalR: StartDesktop denied
— agent {AgentId} is offline", agentId);
            return;
        }

        var message =
System.Text.Json.JsonSerializer.Serialize(new
        {
            type = "desktop.start",
            quality,
            scale
        });
        await connectionManager.SendToAgent(agentId,
message);
        _logger.LogInformation("SignalR: Desktop start sent to
agent {AgentId} (q={Quality}, scale={Scale})",
            agentId, quality, scale);
    }

    public async Task StopDesktop(string agentId)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value,
agentId)) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null) return;

        var message =
System.Text.Json.JsonSerializer.Serialize(new { type =
"desktop.stop" });
        await connectionManager.SendToAgent(agentId,
message);
        _logger.LogInformation("SignalR: Desktop stop sent to
agent {AgentId}", agentId);
    }

    public async Task UpdateDesktopSettings(string agentId,
int? quality = null, float? scale = null)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value,
agentId)) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null ||
!connectionManager.IsOnline(agentId)) return;

        var settings = new Dictionary<string, object> { ["type"]
= "desktop.settings" };
        if (quality.HasValue) settings["quality"] =
quality.Value;
        if (scale.HasValue) settings["scale"] = scale.Value;

        var message =
System.Text.Json.JsonSerializer.Serialize(settings);
        await connectionManager.SendToAgent(agentId,
message);
        _logger.LogDebug("SignalR: Desktop settings sent to
agent {AgentId}", agentId);
    }

    public async Task StartShell(string agentId, int cols = 120,
int rows = 30)
    {
        var userId = GetUserId();
        if (userId == null) return;

        var roleLevel = await GetUserRoleLevel(userId.Value,
agentId);
        var minLevel = await
GetMinRoleLevelForPermission(agentId, "Shell");
        if (roleLevel < minLevel)
        {
```

```
            _logger.LogWarning("SignalR: StartShell denied — user {UserId} role {Role} < min {Min} for agent {AgentId}",
                userId, roleLevel, minLevel, agentId);
            return;
        }

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "shell.start", cols, rows });
        await connectionManager.SendToAgent(agentId, message);
        _logger.LogInformation("SignalR: Shell start sent to agent {AgentId} ({Cols}x{Rows})", agentId, cols, rows);
    }

    public async Task ResizeShell(string agentId, int cols, int rows)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value, agentId)) return;

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "shell.resize", cols, rows });
        await connectionManager.SendToAgent(agentId, message);
    }

    public async Task SendShellInput(string agentId, string data)
    {
        var userId = GetUserId();
        if (userId == null) return;

        var roleLevel = await GetUserRoleLevel(userId.Value, agentId);
        var minLevel = await GetMinRoleLevelForPermission(agentId, "Shell");
        if (roleLevel < minLevel) return;

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "shell.input", data });
        await connectionManager.SendToAgent(agentId, message);
    }

    public async Task StopShell(string agentId)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value, agentId)) return;

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "shell.end" });
        await connectionManager.SendToAgent(agentId, message);
        _logger.LogInformation("SignalR: Shell stop sent to agent {AgentId}", agentId);
    }

    public async Task GetRemoteClipboard(string agentId)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value, agentId)) return;

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "desktop.clipboard.get" });
        await connectionManager.SendToAgent(agentId, message);
    }

    public async Task SetRemoteClipboard(string agentId, string text)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value, agentId)) return;

        var connectionManager = Context.GetHttpContext()?.RequestServices.GetRequiredService<ApmBackend.API.WebSocket.AgentConnectionManager>();
        if (connectionManager == null || !connectionManager.IsOnline(agentId)) return;

        var message = System.Text.Json.JsonSerializer.Serialize(new { type = "desktop.clipboard.set", text });
```

```
        await connectionManager.SendToAgent(agentId,
message);
    }

    public async Task BrowseRemoteFiles(string agentId,
string path)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value,
agentId)) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null ||
!connectionManager.IsOnline(agentId)) return;

        var message =
System.Text.Json.JsonSerializer.Serialize(new { type =
"desktop.file.browse", path });
        await connectionManager.SendToAgent(agentId,
message);
    }

    public async Task DownloadRemoteFile(string agentId,
string path)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value,
agentId)) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null ||
!connectionManager.IsOnline(agentId)) return;

        var message =
System.Text.Json.JsonSerializer.Serialize(new { type =
"desktop.file.download", path });
        await connectionManager.SendToAgent(agentId,
message);
    }

    public async Task UploadRemoteFile(string agentId, string
path, string data)
    {
        var userId = GetUserId();
        if (userId == null) return;

        if (!await UserHasAccessToAgent(userId.Value,
agentId)) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null ||
!connectionManager.IsOnline(agentId)) return;

        var message =
System.Text.Json.JsonSerializer.Serialize(new { type =
"desktop.file.upload", path, data });
        await connectionManager.SendToAgent(agentId,
message);
    }

    public async Task SendDesktopInput(string agentId, object
inputData)
    {
        var userId = GetUserId();
        if (userId == null) return;

        var roleLevel = await GetUserRoleLevel(userId.Value,
agentId);
        var minLevel = await
GetMinRoleLevelForPermission(agentId,
"RemoteDesktopInput");
        if (roleLevel < minLevel) return;

        var connectionManager =
Context.GetHttpContext()?.RequestServices.GetRequiredSer
vice<ApmBackend.API.WebSocket.AgentConnectionManag
er>();
        if (connectionManager == null) return;

        var inputJson =
System.Text.Json.JsonSerializer.Deserialize<System.Text.Js
on.JsonElement>(

System.Text.Json.JsonSerializer.Serialize(inputData));

        var message =
System.Text.Json.JsonSerializer.Serialize(new
        {
            type = "desktop.input",
            inputType =
inputJson.GetProperty("inputType").GetString(),
            x = inputJson.TryGetProperty("x", out var xVal) ?
xVal.GetInt32() : 0,
            y = inputJson.TryGetProperty("y", out var yVal) ?
yVal.GetInt32() : 0,
            button = inputJson.TryGetProperty("button", out var
btnVal) ? btnVal.GetInt32() : 0,
            deltaY = inputJson.TryGetProperty("deltaY", out var
dyVal) ? dyVal.GetInt32() : 0,
            keyCode = inputJson.TryGetProperty("keyCode", out
var kcVal) ? kcVal.GetInt32() : 0
        });
        await connectionManager.SendToAgent(agentId,
message);
    }
}
```