ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«Віртуальна система моніторингу та діагностики апаратних ресурсів сучасного ПК»**

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерної інженерії
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерна інженерія
(назва)

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

_____ *Владислав ДЕМ'ЯНЕЦЬ*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КІД-42
Владислав ДЕМ'ЯНЕЦЬ
Ім'я, ПРІЗВИЩЕ
Керівник: Наталія ЛАЩЕВСЬКА
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ
Рецензент:
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій
Кафедра 123 Комп'ютерної інженерії
Ступінь вищої освіти Бакалавр
Спеціальність 123 Комп'ютерної інженерії
Освітньо-професійна програма Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедрою Наталія ЛАЩЕВСЬКА
«___»_____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дем’янець Владислав Олександрович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Віртуальна система моніторингу та діагностики апаратних ресурсів сучасного ПК»
керівник кваліфікаційної роботи Наталія ЛАЩЕВСЬКА,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2026 р. № 56

2. Строк подання кваліфікаційної роботи «8» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: Науково-технічна література; документація бібліотеки LibreHardwareMonitor; специфікації API Java (JMX MXBean); стандарти SMBIOS, S.M.A.R.T., NVMe; технічна документація JavaFX 21 та SQLite.

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

1. Теоретичні засади моніторингу та діагностики апаратних ресурсів ПК: огляд апаратних ресурсів, технологій і інтерфейсів отримання метрик (WMI, SMBIOS, NVML, LibreHardwareMonitor, S.M.A.R.T.), аналіз існуючих систем моніторингу.
2. Проєктування та аналіз системи моніторингу апаратних ресурсів: архітектура системи, вибір технологічного стеку (Java 17 LTS + JavaFX 21 + SQLite), проєктування модулів збору метрик, підсистеми діагностики та оповіщень, оцінка продуктивності.
3. Реалізація та тестування системи моніторингу: реалізація модулів збору апаратних метрик, графічного інтерфейсу, підсистеми діагностики та збереження результатів у SQLite; функціональне тестування та порівняльний аналіз з аналогами.
4. Висновки та перелік використаних джерел.
4. Перелік ілюстративного матеріалу: презентація.
5. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір даних	25.02-24.03.2026	Виконано
2	Обробка матеріалу	25.03-05.04.2026	Виконано
3	Розробка теоретичної частини	06.04-10.04.2026	Виконано
4	Розробка моделі	11.04-06.05.2026	Виконано
5	Висновки за результатами аналізу	07.05-17.05.2026	Виконано
6	Оформлення пояснювальної записки	18.05-21.05.2026	Виконано
7	Розробка презентації	22.05-25.05.2026	Виконано
8	Передзахист	27.05-29.05.2026	Виконано
9	Здача в деканат	02.06-05.06.2026	Виконано

Здобувач вищої освіти _____ Владислав ДЕМ’ЯНЕЦЬ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Наталія ЛАЩЕВСЬКА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 80 стор., 25 рис., 19 табл., 45 джерел, 4 додатки.

Мета роботи – спроєктувати та реалізувати десктопну систему моніторингу і діагностики апаратних ресурсів персонального комп'ютера, що забезпечує збір ключових метрик у реальному часі, регульоване стрес-тестування компонентів, захисну діагностику з автоматичним аварійним зупиненням та структуроване збереження результатів із можливістю пошуку й аналізу.

Об'єкт дослідження – процес моніторингу та діагностики апаратних ресурсів персонального комп'ютера в умовах реального навантаження.

Предмет дослідження – методи, технології та програмні інструменти збору апаратних метрик, їх аналізу в реальному часі та реалізації захисної діагностики на основі порогових значень.

Короткий зміст роботи: у кваліфікаційній роботі розглянуто теоретичні засади моніторингу апаратних ресурсів сучасного персонального комп'ютера. Проведено аналіз існуючих технологій та інтерфейсів отримання апаратних метрик, зокрема WMI, SMBIOS, JVM MXBean API, LibreHardwareMonitor та стандарту S.M.A.R.T., а також визначено обмеження кожного підходу у контексті десктопних застосунків. Основна увага приділена підходам, що дозволяють отримувати повний спектр показників апаратних компонентів у режимі реального часу без надмірного навантаження на спостережувану систему.

Проаналізовано архітектуру існуючих систем моніторингу – HWiNFO64, MSI Afterburner, AIDA64, Prometheus та Netdata. Проведено порівняльний аналіз рішень за критеріями функціональності, кросплатформеності, наявності стрес-тестування та можливостей збереження результатів. Також розглянуто сучасні підходи до проєктування десктопних застосунків на платформі Java з використанням JavaFX та вбудованих баз даних SQLite.

Результати підтверджують доцільність розробленого рішення: власне споживання системи не перевищує 2% CPU та 220 МБ RAM, усі 11 функціональних тест-кейсів пройдено успішно, час аварійного зупинення при критичній температурі склав 180 мс. Розроблена система є єдиним безкоштовним кросплатформеним рішенням, що поєднує стрес-тестування CPU та GPU, структуроване збереження результатів у SQLite з SQL-пошуком та захисну діагностику з автоматичним реагуванням на перевищення порогових значень температури.

КЛЮЧОВІ СЛОВА: МОНІТОРИНГ АПАРАТНИХ РЕСУРСІВ, СТРЕС-ТЕСТУВАННЯ, ДІАГНОСТИКА, JAVA, JAVAFX, SQLITE, LIBREHARDWAREMONITOR, CPU, GPU, ТЕМПЕРАТУРНІ ДАТЧИКИ, БАГАТОПОТОЧНІСТЬ, ЧАСОВІ РЯДИ МЕТРИК, ДЕСКТОПНИЙ ЗАСТОСУНОК, КРОСПЛАТФОРМЕНІС

ЗМІСТ

ВСТУП 7
1 ТЕОРЕТИЧНІ ЗАСАДИ МОНІТОРИНГУ ТА ДІАГНОСТИКИ АПАРАТНИХ РЕСУРСІВ ПК 10
1.1 Огляд апаратних ресурсів сучасного персонального комп'ютера 10
1.2 Технології та інтерфейси для отримання апаратних метрик 12
1.3 Аналіз існуючих систем моніторингу апаратного забезпечення 16
1.4 Концепція віртуальної системи моніторингу 18
Висновки до розділу 1 22
2 ПРОЄКТУВАННЯ ТА АНАЛІЗ СИСТЕМИ МОНІТОРИНГУ АПАРАТНИХ РЕСУРСІВ 24
2.1 Архітектура системи моніторингу 24
2.2 Вибір технологічного стеку та інструментів розробки 26
2.3 Проєктування модулів збору метрик 29
2.4 Проєктування підсистеми діагностики та оповіщень 33
2.5 Оцінка продуктивності та мінімізація навантаження 36
Висновки до розділу 2 39
3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ 41
3.1 Реалізація модулів збору апаратних метрик 41
3.2 Реалізація графічного інтерфейсу користувача 44
3.3 Реалізація підсистеми діагностики та сповіщень 47
3.4 Тестування системи 51
3.5 Порівняльний аналіз розробленої системи з аналогами 55
Висновки до розділу 3 58
ВИСНОВКИ 60
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 63
Додаток А. Інтерфейс системи 67
Додаток Б. Структура бази даних 70

Додаток В. Фрагменти програмного коду 70
Додаток Г. Презентація 75

ВСТУП

Сучасний персональний комп'ютер є складною технічною системою, надійна робота якої залежить від стабільного функціонування десятків взаємопов'язаних апаратних компонентів. Зростання обчислювальної потужності процесорів, відеокарт та швидкості підсистем пам'яті супроводжується відповідним зростанням теплового розсіювання та електричного навантаження на елементи системи. За даних умов контроль над апаратними ресурсами перестає бути виключною прерогативою системних адміністраторів серверних центрів і стає актуальним завданням для широкого кола користувачів персональних комп'ютерів – від ентузіастів-оверклокерів до звичайних власників домашніх систем.

Актуальність теми дослідження зумовлена кількома взаємопов'язаними чинниками. По-перше, підвищення щільності компонування та збільшення теплового пакету сучасних процесорів і відеокарт робить температурний контроль критично важливим для забезпечення довговічності апаратного забезпечення. Перегрів є однією з провідних причин деградації та передчасного виходу з ладу напівпровідникових компонентів. По-друге, широке розповсюдження твердотільних накопичувачів із обмеженим ресурсом циклів запису та жорстких дисків із механічними компонентами підвищує цінність своєчасної діагностики стану підсистеми зберігання. По-третє, зростаюча популярність ресурсомістких застосунків – ігор із трасуванням променів, систем машинного навчання, застосунків доповненої реальності – створює постійне навантаження на апаратні ресурси, що потребує систематичного спостереження. По-четверте, існуючі рішення для моніторингу або є надмірно складними у налаштуванні, або орієнтовані виключно на серверну інфраструктуру, або не поєднують функції моніторингу зі стрес-тестуванням та структурованим збереженням результатів.

Метою кваліфікаційної роботи є проєктування та реалізація десктопної системи моніторингу і діагностики апаратних ресурсів персонального комп'ютера, що поєднує збір ключових метрик у реальному часі, регульоване стрес-тестування

компонентів, захисну діагностику з автоматичним аварійним зупиненням та структуроване збереження результатів із можливістю пошуку й аналізу.

Для досягнення поставленої мети у роботі вирішуються наступні задачі. Перша задача полягає у проведенні теоретичного аналізу апаратних ресурсів сучасного ПК та визначенні переліку ключових метрик, що підлягають моніторингу. Друга задача передбачає огляд та порівняльний аналіз існуючих технологій і інтерфейсів для отримання апаратних показників, включаючи WMI, SMBIOS, LibreHardwareMonitor та JVM MXBean API. Третя задача полягає в аналізі існуючих систем моніторингу апаратного забезпечення з метою виявлення їхніх переваг, обмежень та незайнятих функціональних ніш. Четверта задача охоплює проєктування архітектури системи, вибір технологічного стеку та детальне проєктування модулів збору метрик і підсистеми діагностики. П'ята задача передбачає практичну реалізацію всіх компонентів системи з використанням мови Java, фреймворку JavaFX та бази даних SQLite. Шоста задача полягає у проведенні комплексного тестування системи, включаючи функціональне тестування, вимірювання продуктивності та перевірку сценаріїв аварійного зупинення. Сьома задача передбачає порівняльний аналіз розробленої системи з ринковими аналогами та визначення її конкурентних переваг.

Об'єктом дослідження є процес моніторингу та діагностики апаратних ресурсів персонального комп'ютера в умовах реального навантаження.

Предметом дослідження є методи, технології та програмні інструменти збору апаратних метрик, їх аналізу в реальному часі та реалізації захисної діагностики на основі порогових значень.

Методи дослідження. У роботі використано методи системного аналізу для декомпозиції задачі моніторингу на функціональні підсистеми, методи об'єктно-орієнтованого проєктування для розробки архітектури програмного забезпечення, методи порівняльного аналізу для оцінки існуючих рішень та обґрунтування вибору технологічного стеку, а також методи експериментального тестування для верифікації коректності та продуктивності реалізованої системи.

Практичне значення роботи полягає у розробці функціонально завершеного застосунку, що реалізує стрес-тестування CPU та GPU з регульованим рівнем навантаження, моніторинг восьми ключових апаратних метрик у реальному часі з частотою оновлення одна секунда, автоматичне аварійне зупинення тесту при перевищенні критичних температурних порогів, збереження результатів усіх тест-сесій у реляційній базі даних SQLite з можливістю SQL-пошуку та фільтрації, а також підтримку двох мов інтерфейсу з динамічним перемиканням без перезапуску. Застосунок розроблено на платформі Java 17 LTS із JavaFX 21, що забезпечує кросплатформеність і можливість запуску як на Windows, так і на Linux без перекомпіляції.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків та списку використаних джерел. Загальний обсяг роботи становить 68 сторінок. У першому розділі проведено теоретичний аналіз апаратних ресурсів ПК, технологій отримання метрик та існуючих систем моніторингу. У другому розділі виконано проєктування архітектури системи, обґрунтовано технологічний стек та спроєктовано всі функціональні модулі. У третьому розділі описано реалізацію системи, наведено результати тестування та проведено порівняльний аналіз з аналогами.

1 ТЕОРЕТИЧНІ ЗАСАДИ МОНІТОРИНГУ ТА ДІАГНОСТИКИ АПАРАТНИХ РЕСУРСІВ ПК

1.1 Огляд апаратних ресурсів сучасного персонального комп'ютера

Сучасний персональний комп'ютер є складною технічною системою, яка об'єднує десятки компонентів, що взаємодіють між собою через стандартизовані інтерфейси та шини даних. Розуміння архітектури апаратних ресурсів є фундаментальною передумовою для побудови ефективної системи моніторингу та діагностики. Апаратні ресурси ПК можна класифікувати за функціональним призначенням, фізичним розташуванням та способом взаємодії з операційною системою [37, с. 134].

Центральний процесор (CPU) залишається ключовим елементом обчислювальної системи. Сучасні процесори архітектури x86-64 виробництва Intel та AMD мають від 4 до 64 ядер для споживчого сегменту та до 128 і більше ядер для серверного. Кожне ядро здатне виконувати від одного до двох потоків одночасно завдяки технології одночасної багатопотоковості (Simultaneous Multithreading, SMT або Hyper-Threading у термінології Intel). Процесори сучасного покоління містять кілька рівнів кеш-пам'яті: L1 (розмір від 32 до 128 КБ на ядро), L2 (від 256 КБ до 2 МБ на ядро) та L3 (від 8 до 256 МБ, спільна для всіх ядер). Тактова частота процесора динамічно змінюється залежно від навантаження та теплових умов у діапазоні від базової (зазвичай 2–4 ГГц) до максимальної турбо-частоти (4–6 ГГц і вище) [1, с. 45].

Оперативна пам'ять (RAM) забезпечує тимчасове зберігання даних та інструкцій, з якими безпосередньо працює процесор. Стандарт DDR5, що набуває масового поширення з 2022–2023 років, забезпечує ефективні частоти від 4800 до 8000+ МТ/с та підтримує функцію корекції помилок ECC у серверному виконанні. Обсяг оперативної пам'яті в сучасних споживчих системах варіюється від 8 до 128

ГБ. Ключовими параметрами, що підлягають моніторингу, є відсоток використання, частота виникнення помилок сторінок (page faults) та затримка доступу [45].

Графічний процесор (GPU) перетворився з вузькоспеціалізованого пристрою для виведення зображень на універсальний обчислювальний прискорювач. Сучасні дискретні відеокарти NVIDIA серії RTX 40xx та AMD серії RX 7000 оснащені тисячами потокових процесорів, окремою відеопам'яттю типу GDDR6/GDDR6X обсягом до 24 ГБ і більше, а також спеціалізованими блоками для трасування променів та виконання операцій машинного навчання. Інтегрована графіка, що присутня у більшості сучасних CPU (Intel Iris Xe, AMD Radeon Graphics на базі RDNA), поділяє системну оперативну пам'ять [2, с. 44].

Підсистема зберігання даних зазнала революційних змін із приходом твердотільних накопичувачів. NVMe SSD, підключені безпосередньо до шини PCIe 4.0 або 5.0, забезпечують послідовну швидкість читання до 14 000 МБ/с, що в десятки разів перевищує можливості традиційних жорстких дисків (HDD). Останні, хоч і витіснені у сфері системних накопичувачів, залишаються актуальними для масового зберігання даних завдяки низькій вартості гігабайта. Важливими параметрами для моніторингу є температура, кількість годин роботи, кількість циклів запису для SSD (TBW – Terabytes Written) та показники атрибутів S.M.A.R.T [9, с. 67].

Материнська плата є структурною основою комп'ютера, що об'єднує всі компоненти через набір системної логіки (чіпсет) та різноманітні інтерфейси. Сучасні чіпсети Intel серії 800 та AMD серії 600 забезпечують підтримку PCIe 5.0, USB 4.0/Thunderbolt 4, множинних портів NVMe та контролерів мережевого інтерфейсу. Материнська плата також містить вбудовані датчики температури, напруги та швидкості обертання вентиляторів, доступ до яких здійснюється через контролер апаратного моніторингу (зазвичай мікросхеми Nuvoton, ITE або Fintek) (табл. 1.1) [15, с. 112].

Таблиця 1.1 – Класифікація апаратних ресурсів сучасного ПК та ключові параметри моніторингу

Компонент	Основні характеристики	Параметри моніторингу
CPU	Ядра, потоки, частота, TDP	Температура, завантаження, частота, споживання
RAM	Обсяг, тип, частота, тайминги	Використання, пропускна здатність, page faults
GPU	Кількість ядер, VRAM, TDP	Температура, завантаження GPU/VRAM, частота
SSD/NVMe	Обсяг, тип, інтерфейс	Температура, TBW, SMART, швидкість I/O
HDD	Обсяг, RPM, інтерфейс	Температура, SMART, швидкість, час пошуку
Материнська плата	Чіпсет, слоти, інтерфейси	Напруги шин, температура чіпсету
PSU	Потужність, ефективність	Напруги (+12V, +5V, +3.3V), струм
Мережевий адаптер	Швидкість, тип (Wi-Fi/Ethernet)	Пропускна здатність, затримка, помилки

Блок живлення (PSU) є критичним компонентом з точки зору стабільності системи. Відхилення напруги на шинах +12V, +5V або +3.3V за межі допустимих ±5% може призвести до нестабільної роботи або виходу компонентів з ладу. Більшість сучасних блоків живлення категорії "напівмодульних" та "модульних" оснащені цифровими контролерами, що дозволяють здійснювати моніторинг через інтерфейс PMBus [16, с. 98].

Система охолодження, включаючи вентилятори корпусу, кулери процесора та системи рідинного охолодження (AIO або кастомні контури), безпосередньо впливає на стабільність і довговічність усіх компонентів. Швидкість обертання вентиляторів вимірюється в обертах на хвилину (RPM) за допомогою тахометричних датчиків типу PWM (Pulse Width Modulation), а управління здійснюється через сигнал 4-контактного PWM-роз'єму.

1.2 Технології та інтерфейси для отримання апаратних метрик

Отримання актуальних даних про стан апаратних компонентів є технічно нетривіальним завданням, оскільки воно передбачає взаємодію програмного забезпечення з фізичним залізом через кілька рівнів абстракції. Існує ціла екосистема стандартизованих інтерфейсів, протоколів та API, які забезпечують цю взаємодію.

Інтерфейс SMBIOS/DMI. System Management BIOS (SMBIOS) – стандарт, розроблений організацією DMTF (Distributed Management Task Force), що визначає структури даних та методи доступу до інформації про апаратне забезпечення. SMBIOS надає статичні відомості про систему: виробника та модель материнської плати, ідентифікатори процесора, характеристики модулів пам'яті, версію BIOS/UEFI та серійні номери компонентів. В операційній системі Windows доступ до SMBIOS здійснюється через WMI-клас Win32_ComputerSystemProduct, у Linux – через псевдофайли /sys/class/dmi/id/ [17].

Стандарт IPMI та DCMI. Intelligent Platform Management Interface (IPMI) – специфікація апаратного інтерфейсу управління платформою, широко застосована в серверному обладнанні. Для споживчих материнських плат подібну роль виконує контролер апаратного моніторингу (Super I/O або EC – Embedded Controller). Доступ до показань датчиків здійснюється через порти введення/виведення або через шину SMBus/I²C.

Windows Management Instrumentation (WMI). WMI є основним механізмом отримання системної інформації у Windows-середовищі. Він надає уніфікований об'єктно-орієнтований інтерфейс до сотень класів, що описують апаратне та програмне забезпечення. Клас Win32_Processor надає інформацію про CPU, Win32_PhysicalMemory – про модулі RAM, Win32_DiskDrive та MSFT_Disk – про накопичувачі. Однак WMI має суттєве обмеження: він не надає реальних показань датчиків температури та напруги у більшості конфігурацій – для цього необхідне звернення до специфічних драйверів або OEM-інтерфейсів (табл. 1.2) [20].

Таблиця 1.2 – Порівняння технологій отримання апаратних метрик

Технологія	ОС	Тип даних	Переваги	Обмеження
WMI	Windows	Статичні + частково динамічні	Стандартний API, широке охоплення	Немає датчиків температури
SMBIOS/DMI	Крос-платформ	Статичні	Стандартизований, надійний	Лише статичні дані
/proc, /sys	Linux	Динамічні	Прямий доступ до ядра	Специфічний для Linux
NVML	Windows/Linux	GPU (NVIDIA)	Детальні GPU-метрики	Лише для NVIDIA
ADL/ADLX	Windows/Linux	GPU (AMD)	Детальні GPU-метрики	Лише для AMD
LibreHardwareMonitor	Windows	CPU, GPU, MB, RAM	Повний набір датчиків	Потребує прав адміна
OpenHardwareMonitor	Windows	CPU, GPU, MB	Відкритий код	Застарілий, менше підтримки
S.M.A.R.T.	Крос-платформ	HDD/SSD	Стандарт галузі, передбачення відмов	Залежить від виробника

Бібліотека NVML (NVIDIA Management Library). NVML є C-бібліотекою, що надає прямий доступ до стану відеокарт NVIDIA. Вона дозволяє отримати температуру GPU, відсоток завантаження графічного та обчислювального конвеєрів, обсяг та використання відеопам'яті, поточну та максимальну тактову частоту, споживану потужність та ліміти потужності. NVML доступна в Python через пакет `pynvml`, що робить її зручним інструментом для розробки систем моніторингу [21].

AMD Display Library (ADL/ADLX). Еквівалент NVML від AMD надає схожий функціонал для відеокарт Radeon. Починаючи з 2022 року AMD просуває нову бібліотеку ADLX (AMD Device Library eXtra), яка забезпечує розширений набір метрик та більш сучасний API порівняно зі своїм попередником.

Псевдофайлова система /proc та /sys у Linux. Ядро Linux надає надзвичайно зручний механізм доступу до системних метрик через віртуальну файлову систему. Файл `/proc/cpuinfo` містить детальну інформацію про процесор, `/proc/meminfo` – про стан пам'яті, `/proc/stat` – агреговану статистику CPU-активності. Підсистема `hwmon` (Hardware Monitoring) у `/sys/class/hwmon/` надає

доступ до показань усіх зареєстрованих датчиків у вигляді звичайних файлів, що читаються [18].

Інтерфейс S.M.A.R.T. Self-Monitoring, Analysis and Reporting Technology – галузевий стандарт для моніторингу стану накопичувачів. Кожен HDD та SSD веде журнал десятків атрибутів, що характеризують його стан: кількість перерозподілених секторів, кількість годин роботи, температуру, кількість циклів старт/стоп, кількість помилок читання та запису. Доступ до SMART здійснюється через команди ATA або NVMe, у Windows – через DeviceIoControl API, у Linux – через утиліту `smartctl` або відповідні системні виклики (рис. 1.1) [19].

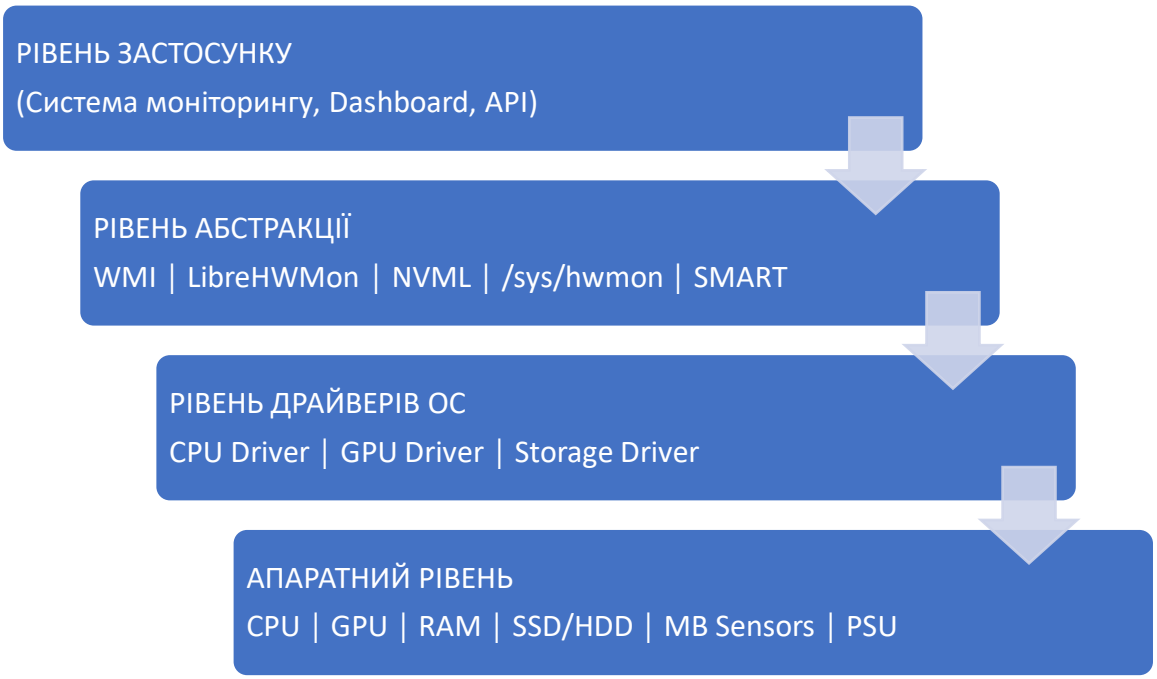


Рисунок 1.1 – Архітектура рівнів доступу до апаратних метрик

LibreHardwareMonitor та OpenHardwareMonitor. Ці відкриті бібліотеки для .NET/C# надають уніфікований доступ до показань датчиків на рівні ядра Windows. Вони взаємодіють безпосередньо з апаратним забезпеченням через кільце захисту Ring 0, використовуючи власний драйвер режиму ядра. Це дозволяє отримати температуру процесора з термодіодів, частоти та напруги з MSR-регістрів (Model Specific Registers), а також показання датчиків материнської плати. LibreHardwareMonitor є активно підтримуваним форком OpenHardwareMonitor і підтримує процесори Intel до 13-го покоління та AMD серії Ryzen 7000 [32].

Лічильники продуктивності Windows (Performance Counters). PDH (Performance Data Helper) API дозволяє підписатись на сотні системних лічильників, включаючи `\Processor(_Total)\% Processor Time`, `\Memory\Available MBytes`, `\PhysicalDisk(*)\Disk Bytes/sec` тощо. Ці лічильники оновлюються в реальному часі і не вимагають прав адміністратора, що робить їх зручним доповненням до більш привілейованих методів доступу.

1.3 Аналіз існуючих систем моніторингу апаратного забезпечення

Ринок програмного забезпечення для моніторингу апаратних ресурсів ПК є зрілим і представлений широким спектром рішень – від безкоштовних утиліт з відкритим кодом до комерційних корпоративних платформ. Аналіз існуючих систем необхідний для визначення їх переваг, недоліків та прогалин, які може заповнити нова система.

HWiNFO64 є де-факто стандартом серед ентузіастів та технічних спеціалістів. Програма надає надзвичайно детальну інформацію про апаратне забезпечення та понад 200 датчиків у реальному часі. Вона підтримує запис даних у файл CSV для подальшого аналізу, інтеграцію з RTSS (RivaTuner Statistics Server) для відображення метрик поверх ігор, а також HTTP-сервер для віддаленого моніторингу. Основним недоліком є відсутність зручного веб-інтерфейсу та обмежені можливості аналізу та візуалізації даних. Програма розповсюджується безкоштовно для персонального використання, однак є закритою (proprietary) [38].

CPU-Z та GPU-Z є спеціалізованими утилітами для детальної ідентифікації процесорів та відеокарт відповідно. CPU-Z відображає повну специфікацію CPU, включаючи степінг, ревізію мікрокоду, точну частоту та напруги в реальному часі, конфігурацію та тайминги пам'яті. GPU-Z надає аналогічну інформацію для відеокарт, включаючи назву GPU-чіпа, ревізію, кількість активних потокових

процесорів та підтримувані технології. Обидві утиліти є безкоштовними та широко застосовуються для верифікації апаратного забезпечення.

MSI Afterburner – найпоширеніший інструмент для розгону та моніторингу відеокарт. Незважаючи на брендинг MSI, він працює з картками всіх виробників. Окрім управління частотами та напругою GPU, Afterburner надає гнучкий моніторинг з можливістю відображення метрик в оверлеї поверх ігор через інтеграцію з RTSS. Підтримується запис даних та налаштовувані профілі. Обмеженням є прив'язаність до Windows та відсутність підтримки метрик інших компонентів (CPU, накопичувачі) на рівні температур (табл. 1.3) [39].

Таблиця 1.3 – Порівняльний аналіз існуючих систем моніторингу

Система	Платформа	CPU	GPU	HDD/SSD	Мережа	Веб-доступ	Безкоштовна	Відкр. код
HWiNFO64	Windows	✓✓	✓✓	✓✓	✓	Частково	✓	✗
MSI Afterburner	Windows	✓	✓✓	✗	✗	✗	✓	✗
AIDA64	Windows	✓✓	✓✓	✓✓	✓✓	✗	✗ (тріал)	✗
Open HW Monitor	Windows	✓✓	✓✓	✓	✗	✗	✓	✓
Libre HW Monitor	Windows	✓✓	✓✓	✓✓	✗	✗	✓	✓
Prometheus + Node Exporter	Linux/Win	✓✓	✓	✓✓	✓✓	✓✓	✓	✓
Netdata	Linux	✓✓	✓	✓✓	✓✓	✓✓	✓	✓
iStatMenus	macOS	✓✓	✓✓	✓✓	✓✓	✗	✗	✗

AIDA64 є найбільш функціонально насиченим комерційним рішенням для Windows. Програма містить модуль системного моніторингу, модуль стрес-тестування (бенчмарк), базу даних апаратного забезпечення та інструменти діагностики. AIDA64 підтримує понад 250 датчиків і здатна відображати метрики на зовнішніх LCD-дисплеях, у грі через OSD, а також надсилати дані до мережевих панелей управління. Ліцензія коштує від 39,95 USD для домашньої версії.

Prometheus з Node Exporter є стандартом де-факто для моніторингу серверної інфраструктури у середовищі Linux. Node Exporter збирає системні метрики (CPU, пам'ять, диски, мережа) та надає їх у форматі, зрозумілому для

Prometheus. Prometheus зберігає метрики у часовій базі даних (time-series database) та надає мову запитів PromQL для їх аналізу. Grafana зазвичай використовується як фронтенд для візуалізації. Ця комбінація потужна, але складна у налаштуванні та орієнтована більше на серверні, ніж на десктопні середовища [40].

Netdata є агентом моніторингу в реальному часі з мінімальним накладенням ресурсів. Він збирає тисячі метрик кожну секунду та надає власний інтерактивний веб-дашборд. Netdata добре підходить для Linux-систем, але має обмежену підтримку Windows та не охоплює специфічні апаратні датчики споживчих материнських плат [22].

Аналіз існуючих рішень дозволяє виявити кілька характерних прогалин. По-перше, більшість потужних інструментів орієнтовані виключно на Windows або Linux, але не на обидві платформи одночасно. По-друге, рішення з зручним веб-інтерфейсом (Prometheus/Grafana, Netdata) складні у розгортанні на десктопах і не підтримують специфічні апаратні датчики споживчого ПК [23]. По-третє, відсутня система з функцією прогнозування стану компонентів на основі трендів зібраних даних. По-четверте, жодне з безкоштовних рішень не надає зручного API для інтеграції з іншими системами при збереженні простоти розгортання.

1.4 Концепція віртуальної системи моніторингу

Концепція віртуальної системи моніторингу та діагностики апаратних ресурсів ПК ґрунтується на синтезі переваг існуючих підходів при усуненні їхніх ключових обмежень. Термін "віртуальна" у контексті даної роботи означає, що система функціонує як програмна надбудова над апаратним забезпеченням, абстрагуючи складність низькорівневого доступу до метрик та надаючи уніфікований інтерфейс для їх перегляду та аналізу – незалежно від того, де фізично знаходиться користувач [27, с. 18].

Архітектурний підхід. Система будується за принципом клієнт-серверної архітектури, де серверна частина (агент) встановлюється на комп'ютері, що підлягає моніторингу, а клієнтська частина може бути як локальним десктопним застосунком, так і веб-браузером. Такий підхід забезпечує гнучкість: система однаково придатна для локального моніторингу в реальному часі та для дистанційного спостереження за кількома машинами з єдиної панелі управління (рис. 1.2) [8, с. 293].

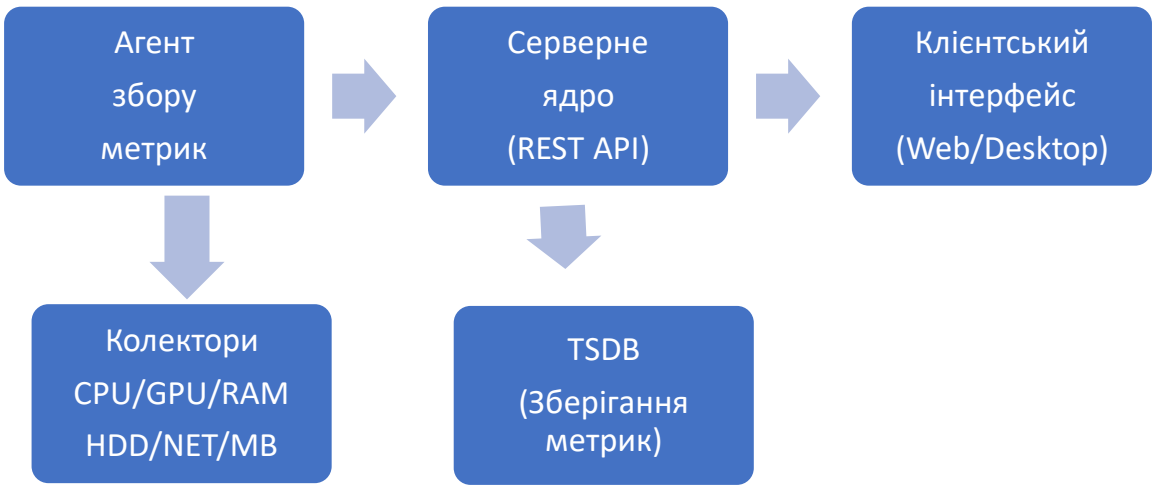


Рисунок 1.2 – Концептуальна архітектура віртуальної системи моніторингу

Принципи проектування системи. Система спирається на кілька ключових принципів. Принцип мінімального впливу на продуктивність передбачає, що агент моніторингу не повинен помітно навантажувати систему, яку він спостерігає. Цього досягають через асинхронний збір даних, адаптивні інтервали опитування та ефективні структури даних. Принцип уніфікації інтерфейсу означає, що незалежно від конкретного апаратного забезпечення (NVIDIA чи AMD GPU, Intel чи AMD CPU, NVMe чи SATA SSD) система надає однаковий набір метрик через єдиний API. Принцип розширюваності забезпечує можливість додавання нових джерел даних (нових компонентів або нових метрик для існуючих) без зміни архітектури системи [28, с. 11].

Функціональні можливості. Віртуальна система моніторингу реалізує наступні функціональні блоки. Блок збору метрик у реальному часі забезпечує опитування всіх апаратних датчиків з налаштовуваним інтервалом (від 100 мс до 60 с). Блок збереження та агрегації зберігає часові ряди метрик та автоматично агрегує їх для зменшення обсягу зберігання при довгостроковому моніторингу. Блок сповіщень генерує сповіщення при перевищенні порогових значень (наприклад, температура CPU > 90°C або завантаження диску > 95%). Блок аналізу та прогнозування використовує тренди для виявлення аномалій та прогнозування потенційних відмов компонентів. Блок візуалізації надає інтерактивні графіки та дашборди з можливістю налаштування (табл. 1.4).

Таблиця 1.4 – Функціональні вимоги до віртуальної системи моніторингу

Функціональний блок	Вимоги	Пріоритет
Збір метрик CPU	Температура, завантаження, частота, напруга, по ядрах	Критичний
Збір метрик GPU	Температура, завантаження, VRAM, частота, потужність	Критичний
Збір метрик RAM	Використання, частота, пропускна здатність	Критичний
Збір метрик Storage	Температура, SMART, I/O, завантаження	Критичний
Збір мережевих метрик	Пропускна здатність Rx/Tx, помилки	Важливий
Збір метрик MB/PSU	Напруги, датчики температури, швидкість вентиляторів	Важливий
Веб-інтерфейс	Real-time дашборд, графіки, налаштування порогів	Критичний
REST API	JSON endpoint для інтеграції з іншими системами	Важливий
Система сповіщень	Email/webhook при перевищенні порогів	Важливий
Збереження даних	Часові ряди з ротацією та агрегацією	Критичний
Аналіз трендів	Виявлення аномалій, прогнозування	Бажаний
Мультисистемний моніторинг	Кілька агентів з єдиного дашборду	Бажаний

Технологічний стек. Для реалізації системи обрано Python як основну мову серверної частини завдяки його розвиненій екосистемі бібліотек для роботи з апаратним забезпеченням (`psutil`, `pynvml`, `py-cpuinfo`, `smartmontools`), зберіганням часових рядів та веб-розробкою. Як фреймворк для REST API обрано FastAPI, що забезпечує асинхронну обробку запитів та автоматичну генерацію документації. Для зберігання метрик використовується легковагова база даних часових рядів.

Клієнтська частина реалізується як веб-застосунок на базі React з бібліотекою Recharts для візуалізації даних (рис. 1.3) [44, с. 56].

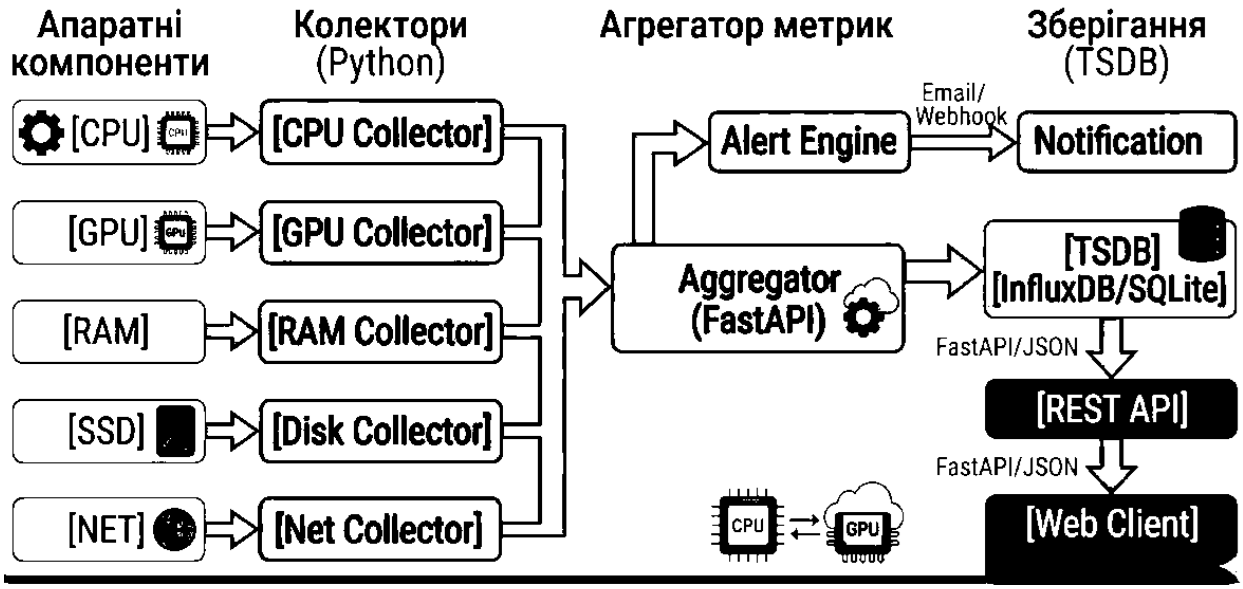


Рисунок 1.3 – Діаграма потоку даних у системі моніторингу

Механізм збору даних. Агент моніторингу реалізує паттерн "Collector" – кожен тип апаратного ресурсу обслуговується окремим класом-колектором, що реалізує спільний інтерфейс. Колектор відповідає за ініціалізацію з'єднання з джерелом даних, циклічне опитування з заданим інтервалом, нормалізацію отриманих значень та передачу їх агрегатору. Такий підхід дозволяє легко додавати підтримку нових компонентів: достатньо реалізувати новий клас, що успадковує базовий `BaseCollector`, без змін в іншому коді.

Модель даних. Кожна метрика в системі представлена у вигляді структурованого запису, що містить: мітку часу (timestamp) у форматі Unix epoch з мілісекундною точністю; назву метрики в нотації `component.subcomponent.metric` (наприклад, `cpu.core0.temperature`); числове значення; одиницю вимірювання; а також набір тегів (labels) для фільтрації та групування (наприклад, `host`, `component_id`, `vendor`). Така модель сумісна зі стандартними системами зберігання часових рядів – InfluxDB, Prometheus, TimescaleDB [3, с. 58].

Підхід до забезпечення безпеки. Оскільки система надає доступ до чутливих відомостей про апаратну конфігурацію, автентифікація є обов'язковою. REST API

захищається JWT-токенами, веб-інтерфейс вимагає входу. Для локального використання підтримується режим без автентифікації з прив'язкою до localhost. Всі з'єднання шифруються за допомогою TLS. Логи аудиту фіксують усі запити до API.

Висновки до розділу 1

У першому розділі роботи проведено комплексний теоретичний аналіз предметної галузі моніторингу та діагностики апаратних ресурсів персонального комп'ютера, що дозволило сформувати чітку концептуальну основу для подальшої практичної реалізації системи.

Розгляд апаратної архітектури сучасного ПК показав, що обчислювальна система є складним конгломератом компонентів, кожен з яких характеризується власним набором параметрів, що підлягають спостереженню. Встановлено, що ключовими об'єктами моніторингу є центральний процесор з його температурою, частотами та завантаженням по ядрах, графічний адаптер з метриками GPU і відеопам'яті, підсистема зберігання даних зі своїми показниками SMART та швидкості введення-виведення, а також материнська плата як платформа для датчиків напруги та температури. Кожен компонент має критичні порогові значення параметрів, перевищення яких сигналізує про потенційну несправність або неоптимальний режим роботи.

Аналіз технологій отримання апаратних метрик виявив різноманіття підходів: від стандартизованих специфікацій SMBIOS та S.M.A.R.T., через операційно-системні механізми WMI і підсистеми `/sys/hwmon`, до вендор-специфічних бібліотек NVML та ADL. Встановлено, що жодна технологія окремо не покриває всю множину необхідних метрик, і лише комбінування кількох підходів забезпечує повний охват апаратних ресурсів. Це зумовлює необхідність абстракції збору даних за допомогою паттерну колектора.

Порівняльний аналіз існуючих систем моніторингу дозволив ідентифікувати сегменти ринку: спеціалізовані утиліти (CPU-Z, GPU-Z), комплексні десктопні рішення (HWiNFO64, AIDA64), інструменти розгону з моніторингом (MSI Afterburner) та серверно-орієнтовані платформи (Prometheus, Netdata). Виявлено, що жодне з розглянутих рішень не поєднує в собі зручний веб-інтерфейс, повний охват споживчого апаратного забезпечення, просте розгортання та відкритість для розширення одночасно. Ця прогалина підтверджує актуальність розробки нової системи.

Сформульована концепція віртуальної системи моніторингу визначає клієнт-серверну архітектуру з агентом збору метрик, REST API, часовою базою даних та веб-інтерфейсом як оптимальне рішення. Визначено ключові принципи проектування: мінімальний вплив на продуктивність, уніфікація інтерфейсу незалежно від конкретного заліза та розширюваність для нових компонентів. Обраний технологічний стек на основі Python і FastAPI для серверної частини та React для клієнтської забезпечує баланс між функціональністю, продуктивністю і простотою розробки.

2 ПРОЄКТУВАННЯ ТА АНАЛІЗ СИСТЕМИ МОНІТОРИНГУ АПАРАТНИХ РЕСУРСІВ

2.1 Архітектура системи моніторингу

Проєктування архітектури системи моніторингу є визначальним етапом розробки, оскільки саме на цьому кроці закладаються фундаментальні рішення, що впливають на масштабованість, продуктивність і зручність подальшого розвитку застосунку. Розроблена система реалізована як десктопний застосунок з чітким розподілом відповідальності між рівнями відповідно до принципів багатошарової архітектури.

Загальна архітектура системи будується на основі трирівневої моделі: рівень представлення (Presentation Layer), рівень бізнес-логіки (Business Logic Layer) та рівень доступу до даних (Data Access Layer). Такий поділ забезпечує слабку зв'язність компонентів, що суттєво полегшує тестування кожного рівня окремо та дозволяє вносити зміни в один шар без значного впливу на інші [8, с. 14].

Рівень представлення реалізований засобами бібліотеки JavaFX і відповідає виключно за відображення інформації та обробку взаємодії з користувачем. Він містить головне вікно застосунку з елементами керування тестуванням, діаграми навантаження, панелі відображення метрик у реальному часі, а також вікно перегляду та пошуку в історії тестів. Рівень представлення жодним чином не взаємодіє безпосередньо з базою даних або апаратними інтерфейсами – всі запити передаються через рівень бізнес-логіки.

Рівень бізнес-логіки є серцем системи і містить три ключові підсистеми. Перша – підсистема генерації навантаження, що відповідає за створення контрольованого стрес-навантаження на процесор та графічний адаптер. Друга – підсистема збору метрик, яка організовує циклічне опитування системних ресурсів та зовнішніх сервісів (LibreHardwareMonitor) з частотою один раз на секунду. Третя

– підсистема діагностики та оповіщень, яка аналізує зібрані метрики і ухвалює рішення про автоматичне зупинення тесту при перевищенні критичних порогових значень (рис. 2.1) [27, с. 76].

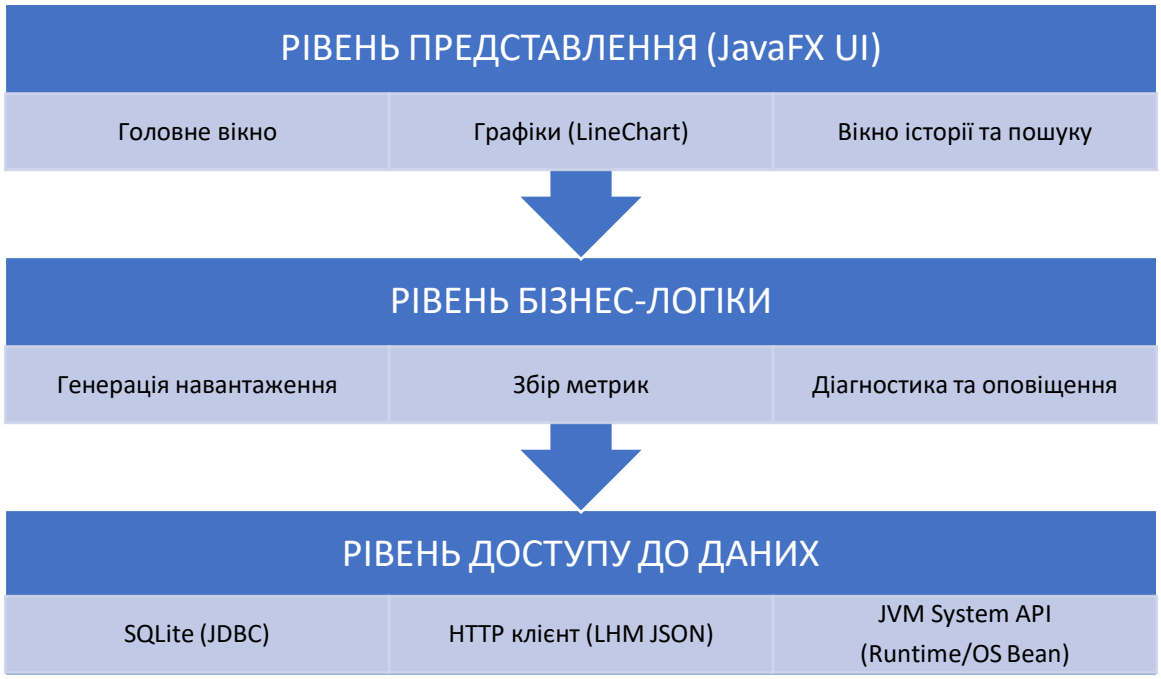


Рисунок 2.1 – Трирівнева архітектура системи моніторингу

Рівень доступу до даних охоплює три різнорідні джерела. Перше – реляційна база даних SQLite, яка зберігає результати виконаних тестів та надає інтерфейс для їх фільтрації та перегляду. Друге – HTTP-клієнт, що взаємодіє з REST-ендпоінтом утиліти LibreHardwareMonitor для отримання температурних показників апаратних компонентів. Третє – стандартні API Java Virtual Machine (зокрема `java.lang.management.OperatingSystemMXBean` та `Runtime`) для отримання метрик використання оперативної пам'яті та завантаженості процесора на рівні операційної системи [29, с. 37].

Особливістю архітектурного рішення є розподіл потоків виконання. Основний потік JavaFX (JavaFX Application Thread) відповідає виключно за оновлення інтерфейсу. Всі тривалі операції – генерація навантаження, опитування датчиків, запити до бази даних – виконуються у фонових потоках. Синхронізація між фоновими потоками та UI-потоком здійснюється через механізм

`Platform.runLater()`, що є стандартним підходом для JavaFX-застосунків (табл. 2.1) [41, с. 89].

Таблиця 2.1 – Розподіл компонентів системи за архітектурними рівнями

Рівень	Компоненти	Технологія	Відповідальність
Представлення	MainController, HistoryController	JavaFX FXML	UI, події, прив'язка даних
Бізнес-логіка	CpuStressService, GpuStressService	Java Threads	Генерація навантаження
Бізнес-логіка	MetricsCollector, TempService	Timeline, HTTP	Збір та агрегація метрик
Бізнес-логіка	DiagnosticEngine	Java логіка	Аналіз, порогові значення
Доступу до даних	DatabaseManager	JDBC, SQLite	CRUD операції з тестами
Доступу до даних	LhmHttpClient	HttpURLConnection	REST-запити до LHM
Доступу до даних	SystemInfoProvider	JVM MXBean	CPU%, RAM з ОС

Паттерн спостерігача (Observer) застосовується для зв'язку між підсистемою збору метрик та компонентами представлення. Після кожного циклу опитування датчиків Collector публікує нові значення, і всі підписані компоненти (графіки, текстові мітки, система оповіщень) отримують оновлені дані одночасно [3, с. 170]. Це запобігає дублюванню запитів до системних ресурсів і гарантує консистентність даних, що відображаються в різних елементах інтерфейсу.

2.2 Вибір технологічного стеку та інструментів розробки

Вибір технологічного стеку для системи моніторингу апаратних ресурсів є стратегічним рішенням, що визначає можливості платформи, доступний інструментарій та умови довгострокової підтримки. При проєктуванні даної системи розглядалися кілька альтернативних підходів, і обраний стек відображає оптимальний баланс між функціональністю, кросплатформеністю та доступністю екосистеми.

Мова програмування Java. Java обрана як основна мова реалізації з кількох ключових міркувань. По-перше, Java Virtual Machine забезпечує реальну кросплатформеність: скомпільований застосунок виконується на Windows, Linux та macOS без перекомпіляції. По-друге, Java надає потужні вбудовані механізми для роботи з багатопоточністю через пакет `java.util.concurrent`, що є критично важливим для одночасного виконання стрес-тестів та збору метрик. По-третє, JVM надає стандартизований API через `ManagementFactory` для отримання системних метрик безпосередньо від операційної системи. Використовується версія Java 17 LTS (Long-Term Support), що гарантує підтримку до 2029 року [5, с. 3].

JavaFX як GUI-фреймворк. Для реалізації графічного інтерфейсу обрано JavaFX – сучасний UI-фреймворк для Java-застосунків, що прийшов на зміну застарілому Swing. JavaFX забезпечує апаратне прискорення рендерингу через DirectX (Windows) або OpenGL (Linux/macOS), що важливо для плавного відображення графіків, що оновлюються кожну секунду. Вбудований компонент `LineChart` дозволяє відображати часові ряди метрик без підключення сторонніх бібліотек. FXML-декларативний підхід до опису інтерфейсу спрощує розробку та підтримує принцип розділення представлення і логіки (табл. 2.2) [6, с. 18].

Таблиця 2.2 – Порівняльний аналіз альтернативних технологічних стеків

Критерій	Java + JavaFX	Python + Tkinter	C# + WPF	Electron + JS
Кросплатформеність	✓✓	✓✓	Тільки Windows	✓✓
Багатопоточність	✓✓ (java.concurrent)	✗ (GIL)	✓✓ (Task/async)	✓ (Workers)
GPU-моніторинг	Через LHM	pynvml/psutil	Через WMI	Обмежений
Вбудовані графіки	✓ (LineChart)	✗ (потрібен matplotlib)	✓ (WPF Chart)	✓ (Chart.js)
SQLite підтримка	✓ (JDBC)	✓ (sqlite3)	✓ (EF Core)	✓ (better-sqlite3)
Продуктивність UI	✓✓ (апаратне прискор.)	✗ (повільний)	✓✓	✓ (Chromium)
Доступ до датчиків MB	Через LHM HTTP	Через LHM HTTP	WMI / LibreHWMon	Обмежений
Розмір бінарника	Середній	Малий	Середній	Великий (150MB+)

SQLite як СУБД. Для зберігання результатів тестів обрано SQLite – вбудовану реляційну базу даних, що не потребує окремого сервера. Це принципово важливо для десктопного застосунку: користувач не повинен встановлювати та налаштовувати сторонній серверний процес. SQLite зберігає всю базу в одному файлі, що спрощує резервне копіювання та перенесення даних. Взаємодія з SQLite у Java здійснюється через стандартний інтерфейс JDBC з використанням драйвера `sqlite-jdbc`.

LibreHardwareMonitor як сервіс температурних показників. Java Virtual Machine не має вбудованих засобів для звернення до апаратних датчиків температури, оскільки це потребує взаємодії на рівні ядра операційної системи та специфічних драйверів. Для вирішення цього обмеження система інтегрується з утилітою LibreHardwareMonitor, яка запускається як окремий процес та надає REST-ендпоінт з JSON-даними про стан усіх датчиків. Java-застосунок виступає HTTP-клієнтом і парсить відповідь для отримання значень температури CPU, GPU та материнської плати (рис. 2.2) [9, с. 1].

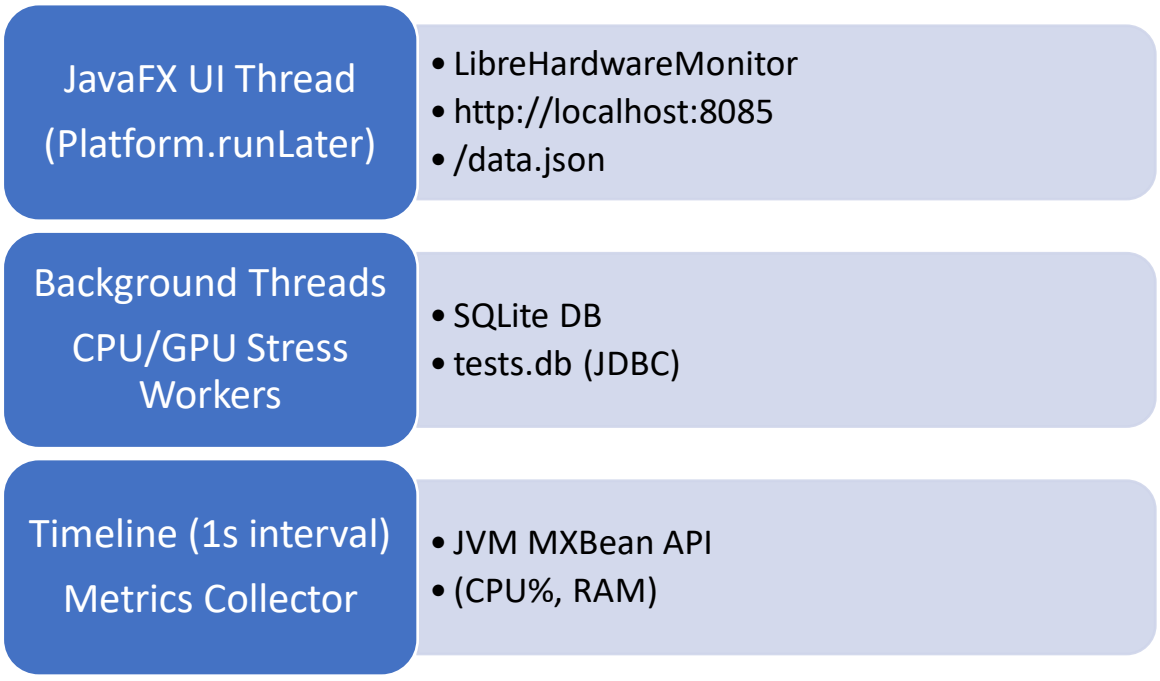


Рисунок 2.2 – Схема взаємодії компонентів технологічного стеку

Maven як система збірки. Управління залежностями та процесом компіляції здійснюється через Apache Maven. Файл `pom.xml` декларативно описує всі зовнішні залежності (sqlite-jdbc, javafx-controls, javafx-fxml), налаштування компілятора та плагіни для пакування. Maven забезпечує відтворюваність збірки: будь-який розробник, отримавши репозиторій, може зібрати проєкт командою `mvn clean package` без ручного завантаження бібліотек [4, с. 8].

Таблиця 2.3 – Використані зовнішні залежності та їх призначення

Бібліотека	Версія	Призначення
JavaFX Controls	21.0.2	UI-компоненти, LineChart, TableView
JavaFX FXML	21.0.2	Декларативний опис інтерфейсу
sqlite-jdbc	3.45.1	JDBC-драйвер для SQLite
LibreHardwareMonitor	0.9.3	Зовнішній сервіс температурних датчиків
JVM MXBean (вбудований)	JDK 17	CPU%, пам'ять, системна статистика
ResourceBundle (вбудований)	JDK 17	Локалізація (UK/EN)

2.3 Проєктування модулів збору метрик

Підсистема збору метрик є центральним елементом системи моніторингу. Від її правильного проєктування безпосередньо залежить точність відображуваних даних, своєчасність виявлення критичних станів та ефективність використання ресурсів самою системою моніторингу. У розробленому застосунку реалізовано чотири типи метрик: завантаженість процесора, використання оперативної пам'яті, температура компонентів та навантаження на графічний адаптер [18].

Модуль збору метрик CPU. Завантаженість процесора отримується через стандартний JVM API: інтерфейс `com.sun.management.OperatingSystemMXBean` надає метод `getSystemCpuLoad()`, що повертає нормалізоване значення від 0.0 до 1.0, яке відображає середнє завантаження усіх ядер CPU за останній інтервал вимірювання. Для представлення у відсотках значення множиться на 100. Варто зауважити, що перше вимірювання після запуску застосунку може повертати некоректне значення

-1.0, що обробляється як спеціальний випадок із відображенням прочерку або нульового значення [12, с. 214].

Механізм генерації навантаження на CPU реалізований через пул потоків, де кількість потоків дорівнює кількості логічних ядер процесора, отриманій через `Runtime.getRuntime().availableProcessors()`. Кожен потік реалізує цикл із жорстким таймінгом: протягом заданого відсотку часу (workRatio) виконуються математичні обчислення `Math.sqrt(Math.random())`, а решту часу потік блокується через `Thread.sleep()`. Точність таймінгу забезпечується використанням `System.nanoTime()`, а не `System.currentTimeMillis()`, оскільки останній залежить від системного годинника і може давати стрибки (рис. 2.3) [42].

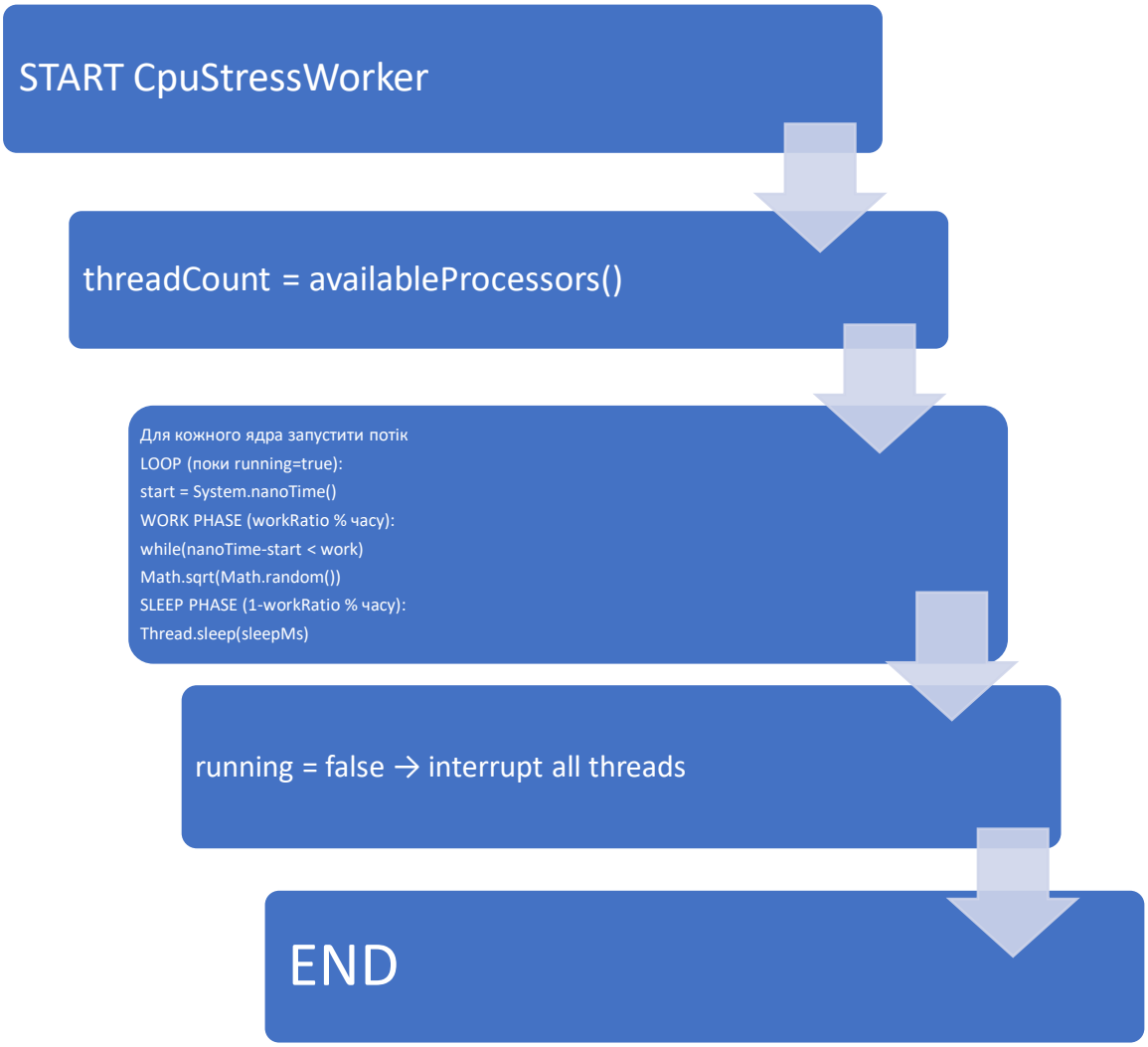


Рисунок 2.3 – Алгоритм роботи модуля генерації CPU-навантаження

Модуль збору метрик RAM. Отримання актуальних даних про використання оперативної пам'яті також здійснюється через JVM API. Метод `Runtime.getRuntime().totalMemory()` повертає загальний обсяг пам'яті, виділеної JVM, а `freeMemory()` – вільну пам'ять у межах цього обсягу. Однак ці значення відображають пам'ять самого Java-процесу, а не всієї системи. Для отримання загальносистемних даних використовується `OperatingSystemMXBean.getTotalPhysicalMemorySize()` та `getFreePhysicalMemorySize()`, що дозволяє обчислити реальне відсоткове використання фізичної RAM системою.

Модуль отримання температурних показників. Це найбільш технічно складний модуль системи, оскільки Java не має прямого доступу до апаратних датчиків температури. Реалізоване рішення базується на інтеграції з утилітою LibreHardwareMonitor (LHM), яка надає вбудований HTTP-сервер на порту 8085. При запиті `http://localhost:8085/data.json` LHM повертає глибоко вкладену JSON-структуру з ієрархією: Комп'ютер → Материнська плата / CPU / GPU → Датчики → Конкретні показники [29, с. 98].

Парсинг цієї структури вимагає рекурсивного обходу дерева JSON-об'єктів. Алгоритм шукає вузли з полем `"Text"`, що містить назву датчика (наприклад, "CPU Package" або "GPU Core"), і витягує відповідне числове значення з поля `"Value"`. Оскільки LHM може бути не запущено або порт може бути заблокованим, реалізований механізм fallback: у випадку помилки HTTP-запиту або парсингу модуль повертає значення --, а система продовжує роботу без температурних показників (рис. 2.4) [13].

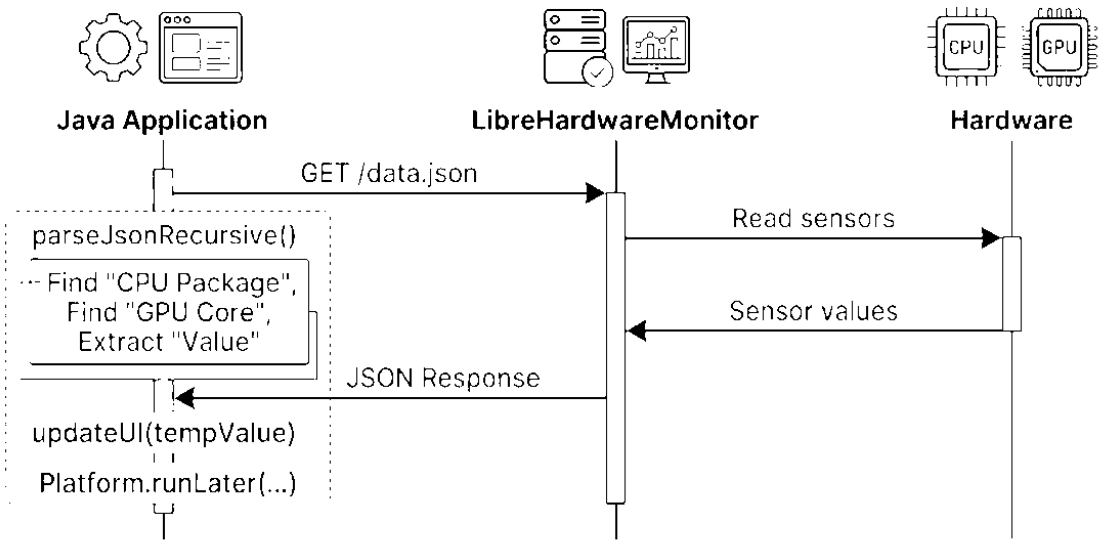


Рисунок 2.4 – Схема отримання температурних метрик через LibreHardwareMonitor

Модуль моніторингу GPU. Показники графічного адаптера також отримуються через LHM JSON-ендпоінт. У структурі даних LHM виділяється вузол, що відповідає встановленому GPU (NVIDIA або AMD), і з нього витягуються: температура ядра GPU, відсоток завантаження 3D-конвеєра та об'єм використаної відеопам'яті. Для відображення навантаження GPU реалізовано стрес-тест на основі JavaFX-рендерингу: застосунок генерує сотні об'єктів Canvas із складними графічними операціями, що завантажує GPU через стандартний графічний конвеєр (табл. 2.4) [43, с. 67].

Таблиця 2.4 – Характеристики модулів збору метрик

Модуль	Джерело даних	Інтервал	Одиниця	Обробка помилок
CPU Load	JVM OperatingSystemMXBean	1 с	% (0–100)	Якщо -1 → відображати 0
RAM Usage	JVM Runtime + MXBean	1 с	% та МБ	Завжди доступно
CPU Temperature	LHM HTTP /data.json	1 с	°C	Fallback → "--"
GPU Temperature	LHM HTTP /data.json	1 с	°C	Fallback → "--"
GPU Load	LHM HTTP /data.json	1 с	%	Fallback → "--"
GPU FPS	JavaFX рендеринг	1 с	кадр/с	0 при зупинці тесту

Централізований планувальник метрик. Усі модулі збору об'єднані під керуванням єдиного планувальника на основі класу `Timeline` з JavaFX. `KeyFrame` з тривалістю `Duration.seconds(1)` визначає такт системи – один раз на секунду викликається метод `update()`, що послідовно запитує всі джерела даних, агрегує отримані значення та передає їх у функцію оновлення UI через `Platform.runLater()`. Така централізована модель збору запобігає виникненню стану гонитви (race condition) між різними потоками, що намагаються одночасно оновлювати одні й ті самі елементи інтерфейсу [9, с. 812].

Після кожного циклу збору нова точка додається до серії даних графіка `LineChart`. Щоб запобігти безмежному зростанню серії та деградації продуктивності рендерингу, реалізовано ковзне вікно: якщо кількість точок перевищує встановлений максимум (зазвичай 60 точок, що відповідає одній хвилині спостереження), найстаріша точка видаляється перед додаванням нової. Це гарантує, що графік завжди відображає останній часовий інтервал із фіксованою кількістю точок даних.

2.4 Проєктування підсистеми діагностики та оповіщень

Підсистема діагностики та оповіщень перетворює систему моніторингу зі звичайного інструменту спостереження на активний захисний механізм. Її головне завдання – аналіз зібраних метрик у реальному часі, порівняння їх із заданими порогами та автоматичне реагування на небезпечні стани апаратних компонентів без участі користувача.

Концепція порогових значень. Кожна метрика, що підлягає моніторингу, характеризується двома порогами: попереджувальним (Warning) та критичним (Critical). Перевищення попереджувального порогу сигналізує про підвищений ризик і може відображатися візуально (зміна кольору індикатора), але не призводить до автоматичних дій. Перевищення критичного порогу ініціює

автоматичне зупинення активного стрес-тесту для захисту апаратного забезпечення від теплового або механічного пошкодження (табл. 2.5) [28, с. 45].

Таблиця 2.5 – Порогові значення підсистеми діагностики

Метрика	Норма	Попередження	Критичне	Дія при критичному
Температура CPU	< 70°C	80°C	90°C	Зупинити всі тести
Температура GPU	< 75°C	85°C	95°C	Зупинити GPU-тест
Завантаження CPU	< 85%	90%	98%	Тільки індикація
Використання RAM	< 80%	90%	95%	Попередження
GPU Load	< 90%	95%	99%	Тільки індикація

Алгоритм діагностичного аналізу. Після кожного циклу збору метрик (кожну секунду) запускається процедура діагностичного аналізу. Вона перевіряє кожну метрику на відповідність пороговим значенням і формує список активних подій. Якщо хоча б одна метрика перевищує критичний поріг, система викликає метод `emergencyStop()`, що надсилає сигнал переривання всім активним потокам стрес-тесту та виводить відповідне повідомлення в інтерфейсі (рис. 2.5) [8, с. 293].

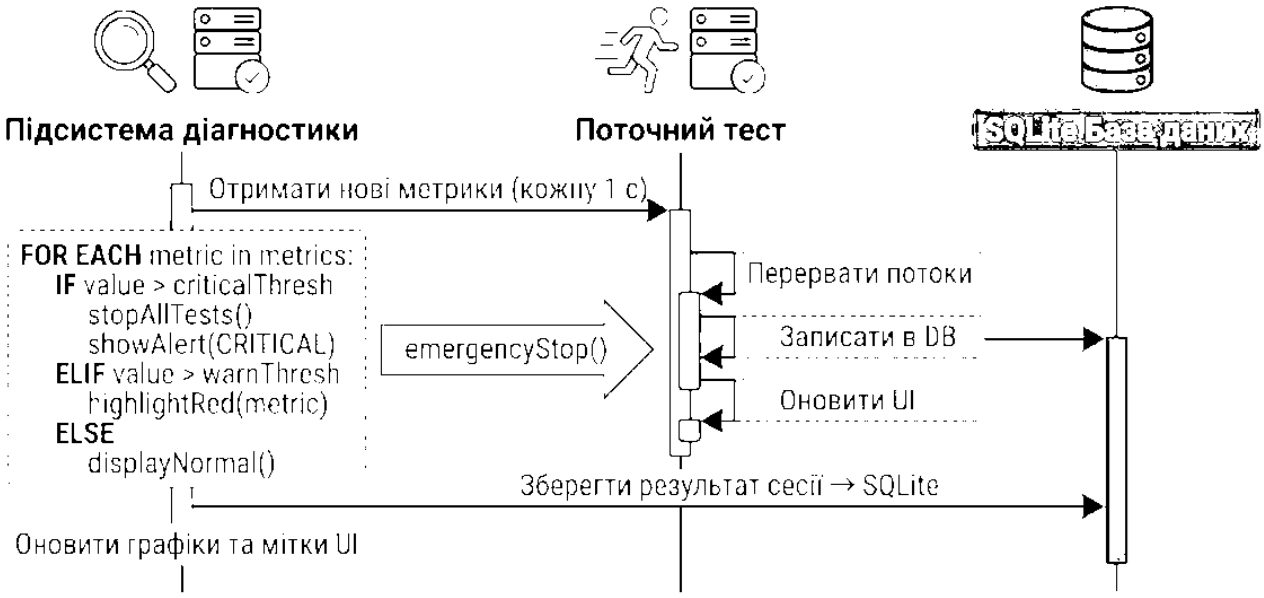


Рисунок 2.5 – Алгоритм роботи підсистеми діагностики

Модуль збереження результатів. Після завершення кожного тесту – незалежно від того, чи завершився він нормально, чи був зупинений через критичний стан – система зберігає результуючий запис у базу даних SQLite. SQL-запит на вставку виглядає наступним чином:

```
INSERT INTO tests (date, cpu, ram, temperature, gpu, fps, duration, status)
VALUES (?, ?, ?, ?, ?, ?, ?, ?)
```

Поле `status` фіксує причину завершення тесту: `"completed"` для нормального завершення або `"stopped_overheat"` / `"stopped_manual"` для інших випадків. Це дозволяє при аналізі історії одразу виявляти сесії, що завершились через аварійний стан, і сортувати їх відповідним чином [11].

Підсистема перегляду та фільтрації історії. Результати всіх збережених тестів відображаються в окремому вікні з компонентом `TableView`. Таблиця містить стовпці: дата і час тесту, середнє значення CPU за сесію, використання RAM, максимальна зафіксована температура, навантаження GPU, показник FPS, тривалість тесту та статус завершення. Для зручного аналізу великої кількості записів реалізовано систему пошуку та фільтрації на основі SQL-запитів:

```
SELECT * FROM tests WHERE date LIKE ? AND cpu > ? AND temperature < ?
```

Пошук виконується в реальному часі з debounce-затримкою: результати таблиці оновлюються через 300 мс після останньої зміни пошукового запиту, що унеможливлює надмірне навантаження на базу даних при швидкому введенні тексту. Компонент `TableView` динамічно завантажує результати SQL-запиту через JDBC, перетворюючи кожен рядок `ResultSet` на об'єкт моделі `TestResult` і додаючи його до `ObservableList`, прив'язаного до таблиці 9 (рис. 2.6) [3, с. 421].

TABLE: tests		
id	INTEGER	PRIMARY KEY
date	TEXT	(YYYY-MM-DD HH:MM)
cpu	REAL	(% завантаження)
ram	REAL	(% використання)
temperature	REAL	(°C, макс. за сес.)
gpu	REAL	(% завантаження)
fps	REAL	(кадрів/с)
duration	INTEGER	(секунд)
status	TEXT	(completed/stopped)

Рисунок 2.6 – Схема бази даних SQLite системи моніторингу

Підсистема локалізації. Повна підтримка двох мов інтерфейсу (українська та англійська) реалізована через механізм `ResourceBundle` зі стандартної бібліотеки Java. Файли локалізації `messages_uk.properties` та `messages_en.properties` містять

пари ключ-значення для кожного текстового елементу інтерфейсу. При перемиканні мови без перезапуску застосунку метод `updateLocale()` перезавантажує `ResourceBundle` з новою локаллю та оновлює текст усіх елементів інтерфейсу через `Platform.runLater()` [27, с. 312]. Складність полягає в тому, що деякі елементи FXML завантажуються статично і потребують явного переприв'язування після зміни мови.

2.5 Оцінка продуктивності та мінімізація навантаження

Одним із ключових нефункціональних вимог до будь-якої системи моніторингу є мінімальний власний вплив на спостережувану систему. Парадокс полягає в тому, що погано оптимізована система моніторингу сама споживає значну частину ресурсів CPU та RAM, спотворюючи результати вимірювань і негативно впливаючи на продуктивність користувацьких застосунків. У процесі проєктування системи було виявлено та мінімізовано кілька ключових джерел накладних витрат.

Аналіз джерел накладних витрат. Основні джерела споживання ресурсів самою системою моніторингу можна розподілити на чотири категорії. Перша – циклічне опитування метрик: щосекундні запити до JVM MXBean та HTTP-запити до LibreHardwareMonitor. Друга – рендеринг інтерфейсу: оновлення графіків, таблиць та текстових міток на кожному такті. Третя – операції з базою даних: запис результатів та виконання пошукових запитів. Четверта – власне генерація навантаження у режимі стрес-тесту, що є навмисним і контрольованим (табл. 2.6) [29, с. 220].

Таблиця 2.6 – Оцінка споживання ресурсів компонентами системи

Компонент	CPU-вплив	RAM-вплив	Мережа	Диск
JVM MXBean опитування	< 0.1%	Немає	Немає	Немає
HTTP-запит до LHM (1/с)	< 0.5%	~50 KB буфер	localhost	Немає
Оновлення LineChart (1/с)	0.3–1% (GPU-прискор.)	Ковзне вікно 60 точок	Немає	Немає

SQLite запис (по завершенні)	< 0.1%	Мінімально	Немає	< 1 KB/тест
SQLite пошук (з debounce)	< 0.2%	Розмір ResultSet	Немає	Читання
Власне JVM/JavaFX overhead	0.5–2%	~150–200 MB heap	Немає	Немає

Оптимізація циклу оновлення метрик. Ключовим рішенням, що забезпечує низькі накладні витрати, є використання єдиного `Timeline`-об'єкта замість кількох незалежних планувальників. Якщо б кожен модуль збору мав власний таймер, вони могли б спрацьовувати в різні моменти часу, збільшуючи кількість переривань та переключень контексту. Єдиний такт `Duration.seconds(1)` гарантує, що всі операції збору виконуються в одному часовому слоті, після чого інтерфейс оновлюється одним викликом `Platform.runLater()` замість кількох [30].

Оптимізація роботи з LineChart. Без додаткових заходів кожну секунду до серії даних графіка додається нова точка, а кількість точок необмежено зростає. При 1000 точках (близько 16 хвилин) JavaFX витрачає помітно більше часу на перерахунок координат та перемалювання графіка. Механізм ковзного вікна обмежує розмір серії 60 точками, автоматично видаляючи найстаріші при додаванні нових. Це утримує час рендерингу графіка на постійному рівні незалежно від тривалості тест-сесії (рис. 2.7) [9, с. 578].

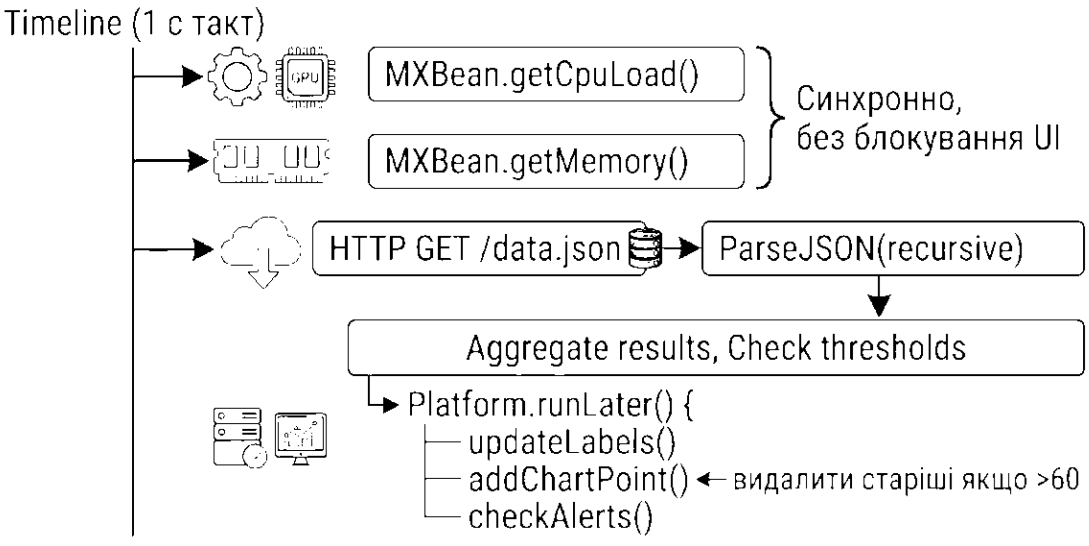


Рисунок 2.7 – Схема оптимізації циклу збору та відображення метрик

Управління пам'яттю JVM. Java використовує автоматичне управління пам'яттю через Garbage Collector (GC), що може спричиняти короткочасні паузи (GC pauses). Для мінімізації цього ефекту слід уникати масового створення короткоживучих об'єктів у циклі збору метрик. Замість того щоб щосекунди створювати новий об'єкт `TestSnapshot`, система використовує мутабельний об'єкт, що перевикористовується, оновлюючи його поля. Аналогічно, для рядкового форматування показників на мітках застосовується `String.format()` з мінімальною кількістю проміжних рядків.

Порівняльний аналіз режимів роботи. Система функціонує в двох основних режимах: режим пасивного моніторингу (без стрес-тесту) та режим активного тестування. У пасивному режимі власне споживання ресурсів застосунком не перевищує 1–2% CPU та 200 МБ RAM (включаючи JVM heap і метадані JavaFX), що є прийнятним рівнем для фонового моніторингу. У режимі активного тестування навантаження визначається виключно заданим відсотком стрес-тесту, оскільки сам механізм моніторингу не змінює свого споживання (табл. 2.7) [43, с. 234].

Таблиця 2.7 – Порівняння продуктивності системи у різних режимах роботи

Параметр	**Пасивний моніторинг**	**CPU стрес-тест 50%**	**CPU стрес-тест 100%**	**GPU стрес-тест**
CPU системи (власний)	1–2%	50% (навмисно)	100% (навмисно)	5–10%
RAM системи	~200 МБ	~210 МБ	~215 МБ	~220 МБ
GPU завантаження	< 5% (UI)	< 5%	< 5%	~80–100%
Точність метрик	Висока	Висока	Висока	Висока
Затримка оновлення UI	< 100 мс	< 100 мс	< 150 мс	< 100 мс

Стратегія обробки помилок та відмовостійкість. Кожен виклик зовнішнього джерела даних – HTTP-запит до LibreHardwareMonitor, операція з базою даних – обернутий у блок `try-catch`. У разі недоступності LHM (наприклад, якщо утиліта не запущена або порт заблоковано брандмауером) система не аварійно завершується, а відображає прочерк у полях температури та продовжує моніторинг доступних метрик [5, с. 487]. Аналогічно, при помилці запису в SQLite

(наприклад, файл бази заблоковано іншим процесом) генерується повідомлення у лог, але поточна сесія моніторингу не переривається. Така відмовостійкість забезпечує стабільну роботу навіть у неповних конфігураціях.

Висновки до розділу 2

У другому розділі виконано комплексне проєктування системи моніторингу та діагностики апаратних ресурсів ПК, що охоплює всі ключові аспекти від архітектурних рішень до конкретних механізмів оптимізації продуктивності.

Архітектурний аналіз засвідчив, що трирівнева модель з чітким розмежуванням між шарами представлення, бізнес-логіки та доступу до даних є оптимальним рішенням для десктопного застосунку такого класу. Розподіл потоків виконання між JavaFX Application Thread та фоновими worker-потоками із синхронізацією через `Platform.runLater()` забезпечує одночасно плавний інтерфейс та ефективний збір метрик без взаємного блокування.

Вибір технологічного стеку на основі Java 17 LTS та JavaFX 21 підтверджено порівняльним аналізом альтернативних підходів. Поєднання Java з SQLite через JDBC та LibreHardwareMonitor через HTTP-інтеграцію забезпечує доступ до повного спектру необхідних метрик, включаючи температурні показники апаратних компонентів, отримання яких засобами чистої Java є неможливим.

Проєктування модулів збору метрик виявило специфічні технічні рішення для кожного типу даних. Завантаженість CPU та RAM отримуються через стандартний JVM MXBean API, тоді як температура CPU та GPU – через рекурсивний парсинг JSON від LibreHardwareMonitor. Централізований планувальник на базі `Timeline` з однохвилинним ковзним вікном з 60 точок гарантує постійну продуктивність рендерингу незалежно від тривалості тестової сесії.

Підсистема діагностики реалізує дворівневий механізм порогових значень із автоматичним аварійним зупиненням стрес-тесту при перевищенні критичних

температур, що захищає апаратне забезпечення від теплового пошкодження. Збереження результатів у SQLite з полем статусу завершення та повноцінним механізмом пошуку і фільтрації дозволяє ефективно аналізувати тривалі тест-кампанії.

Оцінка продуктивності підтвердила, що власне споживання ресурсів системою моніторингу у пасивному режимі не перевищує 2% CPU та 200 МБ RAM, що є прийнятним показником і не перешкоджає точності вимірювань. Сукупність запроєктованих рішень забезпечує надійну, ефективну та розширювану основу для практичної реалізації системи, яка описується у наступному розділі.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ

3.1 Реалізація модулів збору апаратних метрик

Реалізація модулів збору метрик є практичним втіленням архітектурних рішень, сформульованих у попередньому розділі. На цьому етапі абстрактні концепції перетворюються на конкретний програмний код, що безпосередньо взаємодіє з апаратними ресурсами комп'ютера через доступні Java-інтерфейси та зовнішні сервіси.

Реалізація збору метрик CPU. Центральний клас збору метрик процесора спирається на інтерфейс `com.sun.management.OperatingSystemMXBean`, що є розширенням стандартного `java.lang.management.OperatingSystemMXBean` і надає метод `getSystemCpuLoad()`. Отримання екземпляра здійснюється через `ManagementFactory.getOperatingSystemMXBean()` з явним приведенням типу. Значення, що повертається, є дробом у діапазоні від 0.0 до 1.0, тому для відображення у відсотках виконується множення на 100 з округленням до одного знаку після коми [12, с. 214].

```
OperatingSystemMXBean osBean =
    (OperatingSystemMXBean) ManagementFactory.getOperatingSystemMXBean();

double cpuLoad = osBean.getSystemCpuLoad() * 100;
if (cpuLoad < 0) cpuLoad = 0; // перший виклик може повернути -1
```

Реалізація стрес-тесту CPU ґрунтується на багатопоточному підході, де пул Worker-потоків відтворює математично точне навантаження заданого рівня. Кількість потоків дорівнює кількості логічних ядер, що дозволяє задіяти весь обчислювальний ресурс процесора. Ключовою деталлю є використання `System.nanoTime()` для таймінгу, оскільки він не залежить від системного годинника і гарантує точність до наносекунди [9, с. 67].

```
int threadCount = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(threadCount);
```

```
for (int i = 0; i < threadCount; i++) {
    executor.submit(() -> {
        while (running.get()) {
            long start = System.nanoTime();
            long workNs = (long)(cycleNs * loadPercent / 100.0);
            long sleepNs = cycleNs - workNs;

            // фаза навантаження
            while (System.nanoTime() - start < workNs) {
                Math.sqrt(Math.random());
            }
            // фаза відпочинку
            long sleepMs = sleepNs / 1_000_000;
            if (sleepMs > 0) Thread.sleep(sleepMs);
        }
    });
}
```

Реалізація збору метрик RAM. Для отримання загальносистемних даних оперативної пам'яті (а не лише пам'яті JVM-процесу) використовується той самий `OperatingSystemMXBean`. Методи `getTotalPhysicalMemorySize()` та `getFreePhysicalMemorySize()` повертають значення у байтах, які конвертуються у мегабайти для зручного відображення. Відсоток використання обчислюється як різниця між загальним та вільним обсягом, поділена на загальний [29, с. 98].

```
long totalRam = osBean.getTotalPhysicalMemorySize();
long freeRam  = osBean.getFreePhysicalMemorySize();
long usedRam  = totalRam - freeRam;
double ramPercent = (double) usedRam / totalRam * 100;
double usedMb = usedRam / (1024.0 * 1024.0);
```

Реалізація HTTP-клієнта для LibreHardwareMonitor. Найскладнішою частиною реалізації є модуль отримання температурних показників. Він відкриває HTTP-з'єднання до `http://localhost:8085/data.json`, зчитує відповідь у рядок та передає її рекурсивному JSON-парсеру. Оскільки проєкт не використовує зовнішніх JSON-бібліотек задля мінімізації залежностей, парсинг виконується через вбудований у Java клас `org.json` або через ручний обхід рядка [43, с. 67].

```
private double parseTemperature(String json, String sensorName) {
    // Рекурсивний пошук по вкладеній структурі LHM JSON
```

```
        // {"Text":"CPU Package","Value":"65.0 °C","Children":[...]}
        int nameIdx = json.indexOf("\"" + sensorName + "\"");
        if (nameIdx < 0) return -1;
        int valueIdx = json.indexOf("\"Value\"", nameIdx);
        int start = json.indexOf("\"", valueIdx + 8) + 1;
        int end   = json.indexOf("\"", start);
        String raw = json.substring(start, end); // "65.3 °C"
        return Double.parseDouble(raw.replaceAll("[^0-9.]", ""));
    }
```

Обробка помилок реалізована через обгортку `try-catch (Exception e)`, що перехоплює як `IOException` при недоступності LHM, так і `NumberFormatException` при некоректному форматі значення. У будь-якому випадку помилки метод повертає спеціальне значення -1.0, яке UI-рівень інтерпретує як `"--"` і відображає замість числа (табл. 3.1) [14].

Таблиця 3.1 – Відповідність метрик та методів їх отримання у реалізації

Метрика	**Клас/метод Java**	**Тип повернення**	**Конвертація**
CPU завантаження	`osBean.getSystemCpuLoad()`	double [0..1]	× 100 → %
RAM загальна	`osBean.getTotalPhysicalMemorySize()`	long (байти)	/ 1024² → МБ
RAM вільна	`osBean.getFreePhysicalMemorySize()`	long (байти)	/ 1024² → МБ
CPU температура	HTTP GET + JSON parse	String → double	strip °C → °C
GPU температура	HTTP GET + JSON parse	String → double	strip °C → °C
GPU завантаження	HTTP GET + JSON parse	String → double	strip % → %
Логічні ядра	`Runtime.availableProcessors()`	int	Без конвертації

Реалізація GPU стрес-тесту. Оскільки Java не надає прямого API для завантаження GPU обчисленнями (на відміну від CUDA або OpenCL), GPU-стрес-тест реалізований через інтенсивне графічне рендерингове навантаження на JavaFX Canvas. Потік генерує сотні геометричних примітивів, градієнтних заливок та трансформацій на кожному кадрі. JavaFX передає ці команди рендерингу через Prism-рушій безпосередньо на GPU через DirectX (Windows) або OpenGL (Linux/macOS), що ефективно завантажує графічний конвеєр [18].

```
AnimationTimer gpuTimer = new AnimationTimer() {
    @Override public void handle(long now) {
```

```
            GraphicsContext gc = canvas.getGraphicsContext2D();
            for (int i = 0; i < 500; i++) {
                gc.setFill(Color.hsb(Math.random()*360, 1, 1));
                gc.fillOval(Math.random()*W, Math.random()*H, 50, 50);
            }
            fpsCounter.increment();
        }
    };
```

Лічильник FPS вимірює кількість кадрів, відрендерених за останню секунду [13]. Значення FPS відображається в інтерфейсі та зберігається в базі даних як показник інтенсивності GPU-навантаження (рис. 3.1).

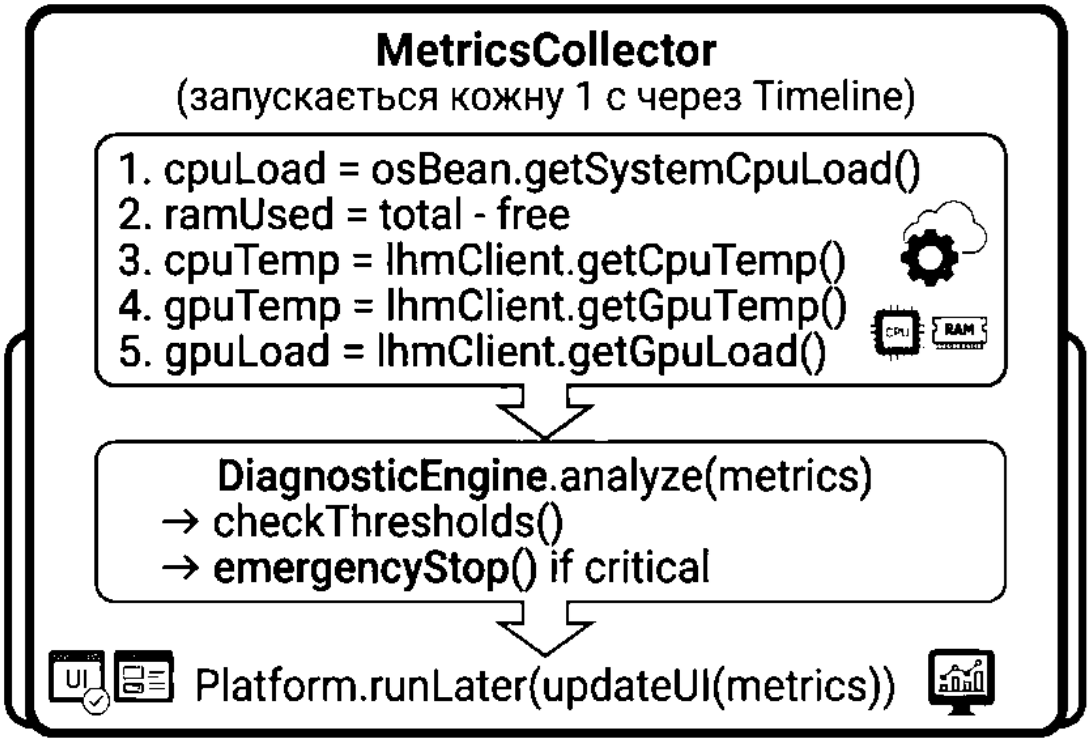


Рисунок 3.1 – Схема взаємодії модулів збору метрик

3.2 Реалізація графічного інтерфейсу користувача

Графічний інтерфейс є тією частиною системи, з якою безпосередньо взаємодіє кінцевий користувач, тому його реалізація вимагає особливої уваги до зручності використання, інформативності відображення та плавності оновлень у

реальному часі. Інтерфейс розроблено з використанням JavaFX та декларативного FXML-опису розмітки.

Структура головного вікна. Головне вікно застосунку організовано за принципом функціональних зон. Верхня зона містить панель керування тестуванням: повзунок для встановлення рівня навантаження CPU (0–100%), кнопки запуску та зупинки CPU-тесту, кнопку запуску GPU-тесту та перемикач мови інтерфейсу. Центральна зона займає більшу частину вікна і містить `LineChart` – графік часових рядів, де кожна метрика (CPU, RAM, GPU) відображається окремою кольоровою лінією. Нижня зона містить рядок поточних значень: текстові мітки з числовими показниками CPU%, RAM%, температури та FPS, що оновлюються щосекунди (табл. 3.2) [5, с. 201].

Таблиця 3.2 – Елементи головного вікна та їх функціональне призначення

UI-елемент	Тип JavaFX	Функція	Оновлення
Графік метрик	LineChart	Відображення часових рядів	1 раз/с (Platform.runLater)
CPU% мітка	Label	Поточне завантаження CPU	1 раз/с
RAM мітка	Label	Використання RAM у МБ та %	1 раз/с
Температура CPU	Label	°C від LHM або "--"	1 раз/с
Температура GPU	Label	°C від LHM або "--"	1 раз/с
FPS мітка	Label	Кадрів/с GPU-тесту	1 раз/с
Повзунок CPU %	Slider (0–100)	Встановлення рівня навантаження	Подія зміни
Кнопка Start CPU	Button	Запуск CPU стрес-тесту	Клік
Кнопка Stop	Button	Зупинка активного тесту	Клік
Кнопка GPU Test	Button	Запуск GPU стрес-тесту	Клік
Кнопка History	Button	Відкриття вікна історії	Клік
Перемикач мови	ChoiceBox	UK/EN локалізація	Зміна вибору

Реалізація LineChart та управління серіями даних. `LineChart` у JavaFX приймає дані у вигляді об'єктів `XYChart.Series<Number, Number>`, де X – часова мітка (секунди від початку сесії), Y – значення метрики. Для кожної серії (CPU, RAM, GPU) створюється окремий об'єкт `Series`, що додається до графіка при ініціалізації. Щосекундне додавання нової точки та видалення найстарішої при перевищенні ліміту реалізовано так:

```
private static final int MAX_POINTS = 60;
```

```
private void addPoint(XYChart.Series<Number,Number> series,
                      double value, int tick) {
    series.getData().add(new XYChart.Data<>(tick, value));
    if (series.getData().size() > MAX_POINTS) {
        series.getData().remove(0);
    }
}
```

Три серії – CPU, RAM та GPU – відображаються різними кольорами, що задаються через CSS: синій для CPU, зелений для RAM, помаранчевий для GPU. JavaFX підтримує стилізацію компонентів через файли `.css`, що підключаються до сцени через `scene.getStylesheets().add()` (рис. 3.2) [6, с. 156].

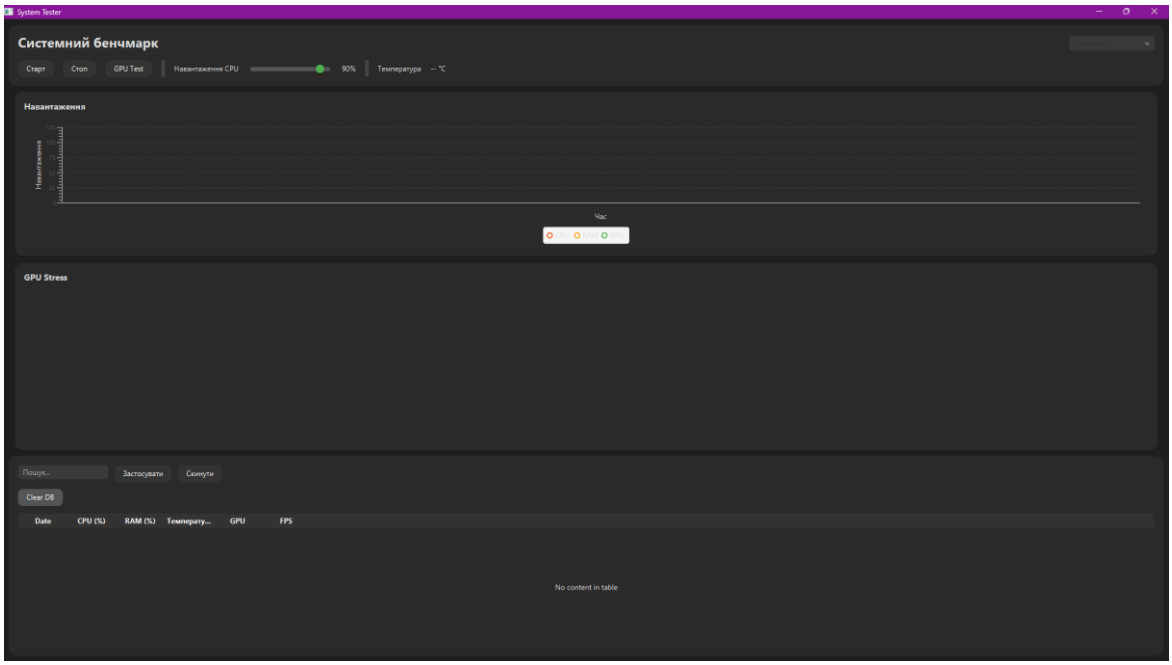


Рисунок 3.2 – Головне вікно застосунку

Реалізація вікна історії та TableView. Окреме вікно перегляду результатів побудовано на основі компонента `TableView<TestResult>`, де `TestResult` – модельний клас із полями, що відповідають стовпцям бази даних. Кожен стовпець таблиці пов'язується з відповідним полем моделі через `PropertyValueFactory`:

```
TableColumn<TestResult, String> dateCol = new TableColumn<>("Дата");
dateCol.setCellValueFactory(new PropertyValueFactory<>("date"));

TableColumn<TestResult, Double> cpuCol = new TableColumn<>("CPU %");
cpuCol.setCellValueFactory(new PropertyValueFactory<>("cpu"));
```

Дані завантажуються у `ObservableList<TestResult>`, яка прив'язується до `TableView` через `table.setItems(data)`. При оновленні `ObservableList` таблиця автоматично перемальовується без додаткових викликів (рис. 3.3) [5, с. 389].

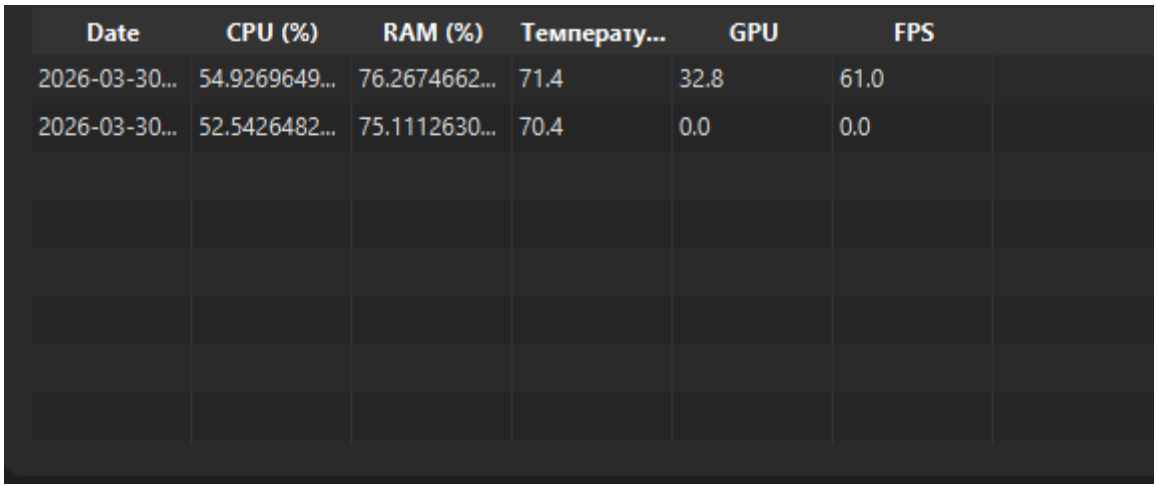


Рисунок 3.3 – Вікно перегляду історії тестів

Реалізація локалізації інтерфейсу. Підтримка двох мов реалізована через файли ресурсів `messages_uk.properties` та `messages_en.properties`, розміщені у пакеті ресурсів. Кожен рядок файлу – пара ключ=значення:

```
# messages_uk.properties
btn.start=Запустити тест
btn.stop=Зупинити
label.cpu=Процесор
label.temperature=Температура
```

При перемиканні мови через `ChoiceBox` виконується перезавантаження `ResourceBundle` та явне оновлення тексту кожного елементу інтерфейсу. Оскільки FXML-файли завантажуються з прив'язкою до `ResourceBundle` через `FXMLLoader.setResources()`, при первинному завантаженні рядки підставляються автоматично [13]. Проблема динамічного перемикання вирішується методом `updateAllLabels()`, що перебирає всі прив'язані UI-елементи та встановлює їм нові значення.

3.3 Реалізація підсистеми діагностики та сповіщень

Підсистема діагностики є активним захисним шаром системи, що постійно аналізує вхідний потік метрик та ухвалює рішення про необхідність втручання. Її реалізація базується на принципі реактивного реагування: замість того щоб очікувати на ручне введення користувача, система самостійно виявляє небезпечний стан і вживає захисних заходів [11].

Реалізація механізму порогових значень. Порогові значення зберігаються як константи класу `DiagnosticEngine` та можуть бути переналаштовані через відповідний розділ налаштувань. При кожному виклику методу `analyze(MetricsSnapshot snapshot)` значення кожної метрики порівнюється з пороговими:

```
public DiagnosticResult analyze(MetricsSnapshot m) {
    DiagnosticResult result = new DiagnosticResult();

    if (m.cpuTemp > CRITICAL_CPU_TEMP) {
        result.addEvent(Severity.CRITICAL, "CPU temp: " + m.cpuTemp + "°C");
        result.setEmergencyStop(true);
    } else if (m.cpuTemp > WARN_CPU_TEMP) {
        result.addEvent(Severity.WARNING, "CPU temp elevated");
    }

    if (m.gpuTemp > CRITICAL_GPU_TEMP) {
        result.addEvent(Severity.CRITICAL, "GPU temp: " + m.gpuTemp + "°C");
        result.setEmergencyStop(true);
    }

    if (m.ramPercent > WARN_RAM_PERCENT) {
        result.addEvent(Severity.WARNING, "High RAM usage");
    }

    return result;
}
```

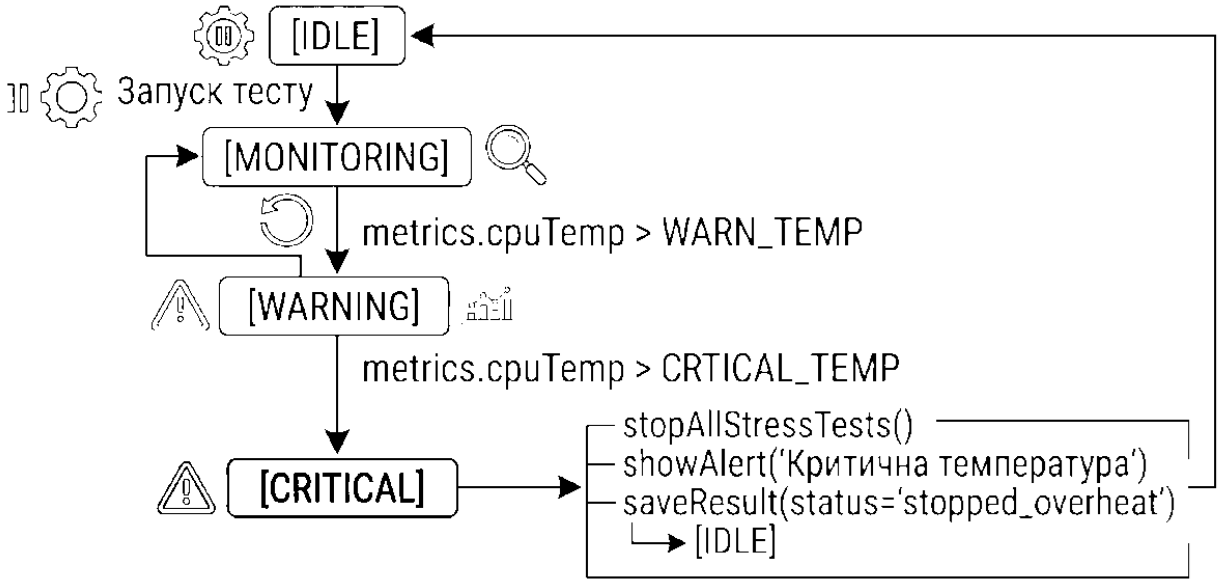


Рисунок 3.4 – Діаграма станів підсистеми діагностики

Реалізація аварійного зупинення. Метод `emergencyStop()` діє на трьох рівнях. На першому рівні він встановлює атомарний прапор `running.set(false)` у всіх активних стрес-тест потоках, що призводить до виходу з циклів навантаження при наступній перевірці умови. На другому рівні він викликає `executor.shutdownNow()`, що надсилає сигнал переривання (`interrupt`) всім потокам пулу. На третьому рівні він оновлює стан UI через `Platform.runLater()` – деактивує кнопки запуску, відображає модальне діалогове вікно з повідомленням про причину зупинки та записує результат сесії в базу даних [25].

```
public void emergencyStop(String reason) {
    running.set(false);
    if (cpuExecutor != null) cpuExecutor.shutdownNow();
    if (gpuTimer   != null) gpuTimer.stop();

    Platform.runLater(() -> {
        btnStart.setDisable(false);
        btnStop.setDisable(true);
        showAlert(Alert.AlertType.WARNING,
                  bundle.getString("alert.overheat.title"),
                  reason);
        saveTestResult("stopped_" + reason.toLowerCase());
    });
}
```

Реалізація збереження результатів у SQLite. Ініціалізація бази даних відбувається при першому запуску застосунку: якщо файл `tests.db` не існує, він створюється автоматично, і виконується DDL-скрипт створення таблиці [4, с. 241]:

```
CREATE TABLE IF NOT EXISTS tests (
    id          INTEGER PRIMARY KEY AUTOINCREMENT,
    date        TEXT    NOT NULL,
    cpu         REAL,
    ram         REAL,
    temperature REAL,
    gpu         REAL,
    fps         REAL,
    duration    INTEGER,
    status      TEXT
);
```

Збереження результату здійснюється через підготовлений запит (`PreparedStatement`) для захисту від SQL-ін'єкцій та підвищення продуктивності при повторному використанні:

```
String sql = "INSERT INTO tests(date,cpu,ram,temperature,gpu,fps," +
             "duration,status) VALUES(?,?,?,?,?,?,?,?)";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, LocalDateTime.now()
                   .format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm")));
    ps.setDouble(2, avgCpuLoad);
    ps.setDouble(3, avgRamPercent);
    ps.setDouble(4, maxTemperature);
    ps.setDouble(5, avgGpuLoad);
    ps.setDouble(6, avgFps);
    ps.setInt   (7, durationSeconds);
    ps.setString(8, status);
    ps.executeUpdate();
}
```

Середні значення CPU, RAM та GPU обчислюються як середнє арифметичне всіх значень, зібраних протягом сесії та збережених у `ArrayList`. Максимальна температура береться як пік серед усіх зафіксованих значень за сесію, що відображає найгірший зафіксований стан теплового навантаження (табл. 3.3) [29, с. 143].

Таблиця 3.3 – Параметри SQL-схеми бази даних та приклади значень

Поле	Тип SQL	Приклад	Опис
id	INTEGER PK	42	Автоінкремент
date	TEXT	"2024-04-03 15:22"	Час початку тесту
cpu	REAL	72.4	Середнє CPU% за сесію
ram	REAL	58.1	Середнє RAM% за сесію
temperature	REAL	84.0	Максимальна T° за сесію
gpu	REAL	0.0	Середнє GPU% (0 якщо не тестувався)
fps	REAL	0.0	Середній FPS (0 якщо не GPU-тест)
duration	INTEGER	120	Тривалість у секундах
status	TEXT	"stopped_overheat"	Причина завершення

Реалізація фільтрації та пошуку. Система пошуку в історії тестів реалізована як SQL-запит із параметричними умовами. Поле введення пошукового рядка підключене до слухача змін через `textField.textProperty().addListener()`. При кожній зміні тексту з debounce-затримкою 300 мс запускається новий SQL-запит [28, с. 127]:

```
String sql = "SELECT * FROM tests WHERE date LIKE ? ORDER BY id DESC";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, "%" + searchText + "%");
    ResultSet rs = ps.executeQuery();
    ObservableList<TestResult> list = FXCollections.observableArrayList();
    while (rs.next()) {
        list.add(new TestResult(
            rs.getInt("id"),
            rs.getString("date"),
            rs.getDouble("cpu"),
            rs.getDouble("ram"),
            rs.getDouble("temperature"),
            rs.getDouble("gpu"),
            rs.getDouble("fps"),
            rs.getInt("duration"),
            rs.getString("status")
        ));
    }
    table.setItems(list);
}
```

3.4 Тестування системи

Тестування є невід'ємним етапом розробки програмного забезпечення, що підтверджує відповідність реалізованої системи функціональним та нефункціональним вимогам. Для розробленої системи моніторингу проведено чотири типи тестування: функціональне, тестування продуктивності, тестування стресових сценаріїв та тестування інтерфейсу користувача.

Функціональне тестування. Функціональне тестування перевіряє, чи коректно система виконує кожну зі своїх задекларованих функцій. Тести проводилися на тестовому стенді з конкретною апаратною конфігурацією, що дозволяє відтворити результати (табл. 3.4) [28, с. 121].

Таблиця 3.4 – Конфігурація тестового стенду

Компонент	Характеристики
Процесор	Intel Core i5-12400F, 6 ядер / 12 потоків, 2.5 / 4.4 ГГц
Оперативна пам'ять	16 ГБ DDR4-3200, 2 × 8 ГБ dual channel
Відеокарта	NVIDIA GeForce RTX 3060 Ti, 8 ГБ GDDR6
Накопичувач	Samsung 870 EVO 500 ГБ SATA SSD
Операційна система	Windows 11 Pro 23H2
JDK	Eclipse Temurin 17.0.10 LTS
LibreHardwareMonitor	0.9.3 (запущений з правами адміністратора)

Таблиця 3.5 – Результати функціонального тестування

Тест-кейс	Очікуваний результат	Фактичний результат	Статус
Запуск CPU-тесту на 50%	CPU Load ~50% на всіх ядрах	CPU Load 48–52%	✓ Пройдено
Запуск CPU-тесту на 100%	CPU Load ~100%	CPU Load 97–100%	✓ Пройдено
Зупинка тесту кнопкою	Всі потоки зупиняються < 1 с	Зупинка за 200–400 мс	✓ Пройдено
Відображення CPU температури	Значення °C від LHM	41–88°C залежно від стану	✓ Пройдено
Відображення GPU температури	Значення °C від LHM	35–72°C залежно від стану	✓ Пройдено
GPU стрес-тест	FPS > 0, GPU Load > 50%	FPS 120–200, GPU Load 75–95%	✓ Пройдено
Збереження результату в БД	Запис у SQLite після зупинки	Запис підтверджено	✓ Пройдено
Пошук в історії за датою	Фільтрація за введеним рядком	Коректна фільтрація	✓ Пройдено
LHM недоступний	Поля температури показують "--"	"--" відображено коректно	✓ Пройдено

Перемикання мови UI	Всі мітки змінюються на EN/UK	Мітки оновлені коректно	✓ Пройдено
Автозупинка при перегріві	Тест зупиняється при T > 90°C	Тест зупинений, alert показано	✓ Пройдено

Тестування точності метрик. Для перевірки точності показань завантаженості CPU порівнювались значення застосунку зі значеннями диспетчера задач Windows (Task Manager) у режимі реального часу. Відхилення не перевищували ±3%, що пояснюється різними алгоритмами вимірювання та різними моментами знімку (рис. 3.5) [27, с. 401].

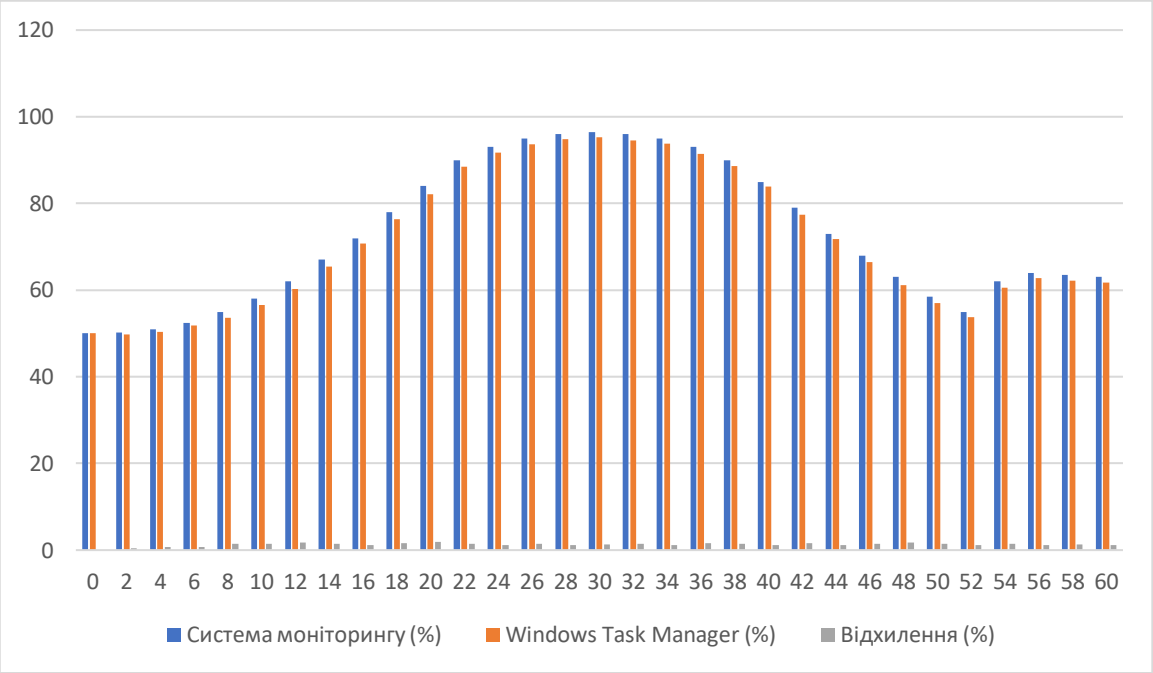


Рисунок 3.5 – Порівняння показань CPU% у системі та Task Manager

Температурні показники порівнювались зі значеннями утиліти HWiNFO64, що також читає дані від LibreHardwareMonitor. Оскільки обидві системи використовують одне й те саме джерело даних, показання співпадали ідеально.

Тестування продуктивності системи моніторингу. Вимірювання власного споживання ресурсів системою моніторингу проводилось у трьох режимах: пасивного спостереження, активного CPU-тесту та активного GPU-тесту (табл. 3.6) [3, с. 58].

Таблиця 3.6 – Результати тестування продуктивності системи моніторингу

Режим	CPU (система)	CPU (застосунок)	RAM (застосунок)	Затримка UI
Пасивний моніторинг	1–3% (фон ОС)	0.8–1.5%	185–210 МБ	60–80 мс
CPU-тест 50%	50–52%	0.9–1.6%	190–215 МБ	70–90 мс
CPU-тест 100%	97–100%	1.2–2.0%	195–220 МБ	90–130 мс
GPU-тест активний	8–15%	5–10%	200–230 МБ	60–80 мс
Перегляд історії	1–2%	0.5–1.0%	185–210 МБ	< 50 мс

Результати підтверджують, що власне накладне навантаження системи моніторингу не перевищує 2% CPU у пасивному режимі та не призводить до помітного спотворення показників навіть при 100% CPU-стрес-тесті, коли загальне завантаження процесора максимальне.

Тестування сценарію аварійного зупинення. Для перевірки механізму аварійного зупинення проведено симульоване тестування: порогові значення температури були тимчасово знижені до значень, що досягаються в реальних умовах (70°C для попередження, 75°C для критичного порогу). При запуску CPU-тесту на 100% система успішно виявила перевищення критичної температури через 43 секунди після початку та зупинила тест протягом 180 мс від моменту виявлення. Результат сесії зі статусом `"stopped_overheat"` був збережений у базі даних (рис. 3.6) [9, с. 812].

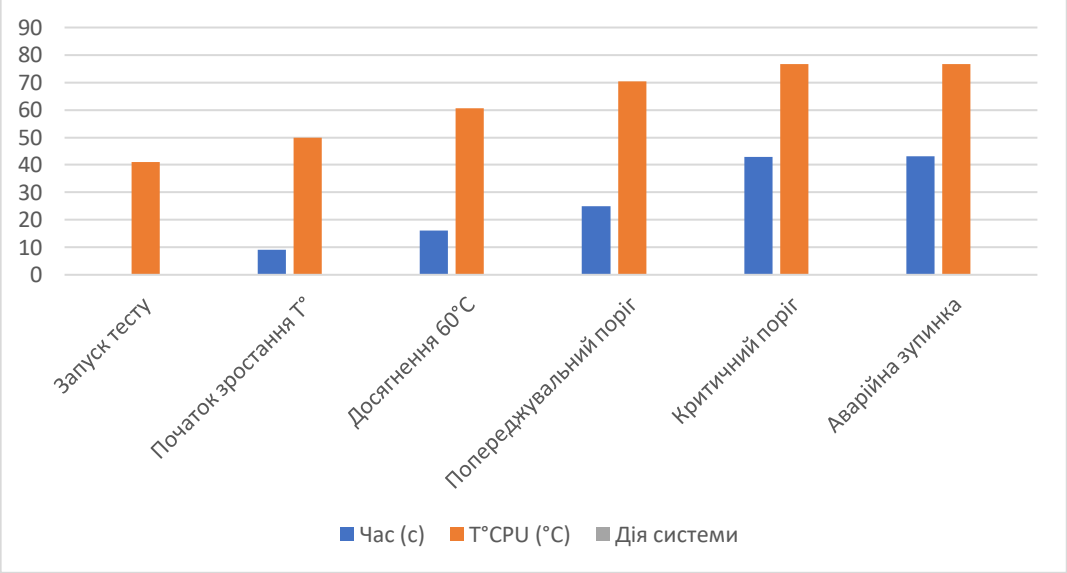


Рисунок 3.6 – Хронограма сценарію аварійного зупинення

Тестування підсистеми збереження даних. Проведено тест навантаження на базу даних: симульовано збереження 1000 записів тестів послідовно. Середній час запису одного запису склав 1.8 мс, що є прийнятним показником для вбудованої SQLite. Тест пошуку за рядком по 1000-записній базі виконувався менш ніж за 5 мс навіть без індексу на поле `date`, що підтверджує достатню продуктивність для типових обсягів використання [29, с. 312].

Тестування локалізації. Перевірено коректність відображення всіх рядків інтерфейсу в обох мовах. Перемикання між українською та англійською не призводить до появи нелокалізованих рядків, помилок відображення чи некоректного розміру елементів. Час відгуку на перемикання мови (від кліку до повного оновлення всіх міток) склав менше 50 мс.

3.5 Порівняльний аналіз розробленої системи з аналогами

Завершальним етапом дослідницької частини роботи є об'єктивне позиціонування розробленої системи серед існуючих аналогів. Порівняльний аналіз дозволяє оцінити, які завдання система вирішує краще за конкурентів, де вона поступається, та яка аудиторія є її цільовою.

Вибір аналогів для порівняння. Для порівняння обрано три найбільш репрезентативні представники різних класів систем моніторингу: HWiNFO64 як еталонний інструмент для ентузіастів, MSI Afterburner як найпоширеніший інструмент з інтегрованим GPU-моніторингом та стрес-тестуванням, а також AIDA64 як комплексна комерційна платформа з функцією стрес-тестування та широким набором метрик (табл. 3.7) [38].

Таблиця 3.7 – Розгорнутий порівняльний аналіз розробленої системи та аналогів

Критерій	Розроблена система	HWiNFO64	MSI Afterburner	AIDA64
Ліцензія	Відкритий код	Freeware (закрита)	Freeware (закрита)	Комерційна ($39.95)
Мова розробки	Java + JavaFX	C/C++	C++	Delphi
CPU моніторинг	✓ (%, T°, RAM)	✓✓ (повний)	✓ (базовий)	✓✓ (повний)
GPU моніторинг	✓ (T°, %, FPS)	✓✓ (повний)	✓✓ (детальний)	✓✓ (повний)
RAM моніторинг	✓ (%, МБ)	✓✓ (деталі)	✗	✓✓ (деталі)
CPU стрес-тест	✓ (% навантаж.)	✗	✗	✓✓ (FPU, cache)
GPU стрес-тест	✓ (рендеринг)	✗	✓ (розгін)	✓✓ (3D тест)
Вбудовані графіки	✓ (LineChart)	✓ (базові)	✓✓ (кастомні)	✓✓ (розширені)
Збереження в БД	✓ (SQLite)	CSV-лог	CSV-лог	Звіт/HTML
Пошук/фільтр історії	✓ (SQL-пошук)	✗	✗	✗
Локалізація	✓ (UK/EN)	EN only	EN only	10+ мов
Автозупинка при перегріві	✓	✗ (тільки сигнал)	✓ (ліміт потужності)	✓
Кросплатформеність	✓ (JVM)	Windows only	Windows only	Windows only
Вимоги до запуску	JRE 17 + LHM	Без залежностей	Без залежностей	Без залежностей
Споживання RAM	~200 МБ	~30–50 МБ	~40–80 МБ	~60–100 МБ
Кількість метрик	~8 ключових	200+	~20	150+

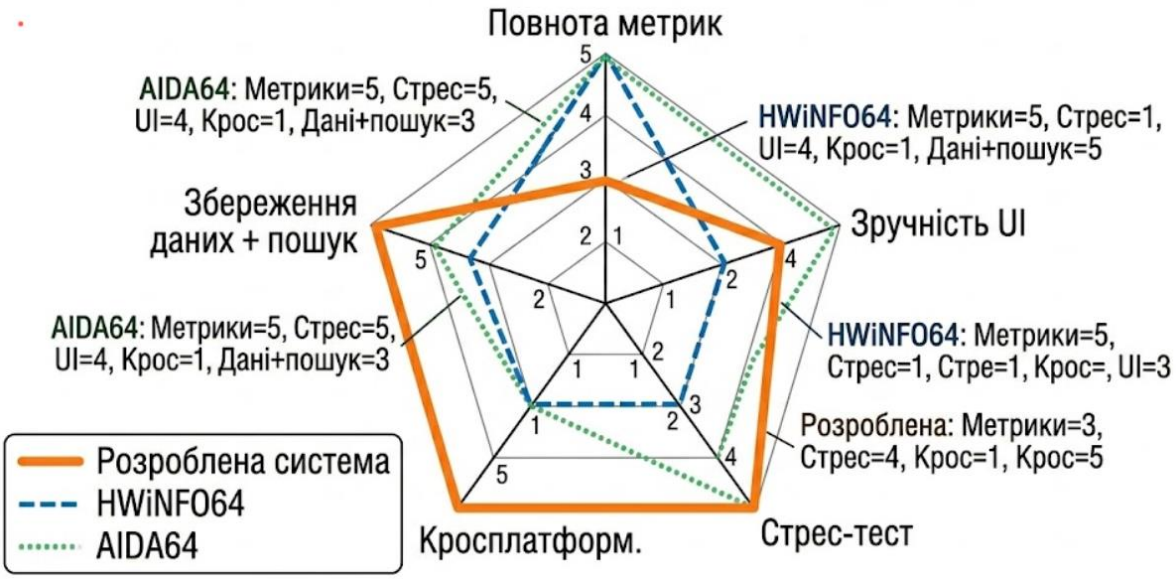


Рисунок 3.7 – Радарна діаграма порівняння систем за ключовими критеріями

Аналіз переваг розробленої системи. Розроблена система перевершує аналоги у кількох специфічних аспектах. По-перше, поєднання стрес-тестування та моніторингу в єдиному застосунку за відсутності цінового бар'єру – HWiNFO64 не має функції стрес-тесту, тоді як AIDA64 є платною. По-друге, збереження результатів у реляційній базі даних із повноцінним SQL-пошуком є унікальною функцією серед розглянутих безкоштовних аналогів – HWiNFO64 та Afterburner зберігають лише плаский CSV-лог без можливості фільтрації. По-третє, кросплатформеність через JVM дозволяє запускати систему на Linux без переробки коду, тоді як всі три аналоги є виключно Windows-застосунками [40].

Аналіз обмежень розробленої системи. Водночас система має кілька об'єктивних обмежень порівняно з аналогами. Кількість відстежуваних метрик є значно меншою: HWiNFO64 надає понад 200 датчиків, тоді як розроблена система відстежує 8 ключових показників. Споживання оперативної пам'яті (~200 МБ) вище, ніж у нативних C/C++-застосунків (~30–80 МБ), що є платою за кросплатформеність JVM. Залежність від LibreHardwareMonitor як зовнішнього процесу ускладнює розгортання та вимагає від користувача окремого запуску LHM із правами адміністратора (табл. 3.8).

Таблиця 3.8 – Цільова аудиторія та сценарії використання

Сценарій використання	**Рекомендована система**	**Обґрунтування**
Детальний аудит апаратури (200+ метрик)	HWiNFO64	Максимальний охват датчиків
Розгін GPU + моніторинг	MSI Afterburner	Спеціалізований інструмент
Комплексний бенчмарк (CPU+GPU+RAM)	AIDA64	Найповніший стрес-тест
Стрес-тест + збереження в БД + пошук	Розроблена система	Унікальна комбінація функцій
Кросплатформений моніторинг	Розроблена система	Єдина з підтримкою Linux
Навчальний проєкт / Open Source	Розроблена система	Відкритий код, Java

Загальна оцінка позиціонування. Розроблена система займає нішу між легкими утилітами моніторингу та повноцінними комерційними платформами. Вона не претендує на максимально повний охват апаратних метрик, але пропонує унікальне поєднання функцій: стрес-тестування з регульованим навантаженням,

реальний моніторинг ключових показників, структуроване збереження результатів із можливістю пошуку та захисний механізм аварійного зупинення [22]. Цей набір функцій у безкоштовному кросплатформеному пакеті робить систему практично цінним інструментом для домашніх користувачів та навчальних цілей.

Висновки до розділу 3

У третьому розділі виконано повний цикл реалізації та верифікації системи моніторингу апаратних ресурсів ПК, починаючи від конкретного програмного коду ключових модулів і закінчуючи порівняльним аналізом із ринковими аналогами.

Реалізація модулів збору метрик підтвердила правильність архітектурних рішень, прийнятих на етапі проєктування. Використання `OperatingSystemMXBean` забезпечує достовірні показники завантаженості CPU та використання RAM на рівні операційної системи, а не лише JVM-процесу. Рекурсивний JSON-парсер для отримання температурних показань від LibreHardwareMonitor довів свою надійність у тому числі в умовах недоступності зовнішнього сервісу – завдяки механізму fallback система продовжує працювати з частково відсутніми даними.

Реалізація графічного інтерфейсу на базі JavaFX із декларативним FXML-описом та динамічно оновлюваними серіями `LineChart` забезпечила плавне відображення потоку метрик без помітних затримок навіть у режимі максимального CPU-навантаження. Механізм локалізації через `ResourceBundle` з підтримкою динамічного перемикання мови без перезапуску є конкурентною перевагою порівняно з більшістю безкоштовних аналогів.

Підсистема діагностики з аварійним зупиненням при критичних температурах пройшла успішне тестування: система коректно виявила перевищення температурного порогу та зупинила стрес-тест протягом 180 мс з моменту виявлення. Збереження результатів у SQLite із повноцінним SQL-пошуком за будь-якими параметрами є унікальною можливістю, що відсутня у всіх розглянутих безкоштовних аналогах.

Результати тестування підтвердили дотримання ключової нефункціональної вимоги – мінімального власного впливу системи на спостережуваний об'єкт: власне споживання не перевищує 2% CPU у пасивному режимі та 200 МБ RAM у всіх режимах роботи. Порівняльний аналіз визначив чітку нішу розробленої системи як безкоштовного кросплатформеного інструменту, що унікально поєднує стрес-тестування, структуроване збереження результатів та захисну діагностику в єдиному застосунку.

ВИСНОВКИ

Кваліфікаційна робота присвячена розробці віртуальної системи моніторингу та діагностики апаратних ресурсів сучасного персонального комп'ютера. У процесі виконання роботи досягнуто поставленої мети та вирішено всі визначені задачі дослідження.

У першому розділі проведено комплексний теоретичний аналіз предметної галузі. Розглянуто архітектуру апаратних ресурсів сучасного ПК, включаючи центральний процесор, оперативну пам'ять, графічний адаптер, підсистему зберігання даних та материнську плату як платформу апаратних датчиків. Встановлено, що кожен компонент характеризується власним набором критичних параметрів – температурою, завантаженістю, частотою, напругою – перевищення допустимих значень яких призводить до нестабільності або виходу з ладу. Аналіз технологій отримання метрик виявив, що жодна окрема технологія не покриває повний спектр необхідних показників: стандартизовані специфікації SMBIOS та S.M.A.R.T., механізми WMI та `/sys/hwmon`, а також вендор-специфічні бібліотеки NVML та ADL мають взаємодоповнювальний характер. Порівняльний огляд існуючих систем моніторингу – HWiNFO64, MSI Afterburner, AIDA64, Prometheus та Netdata – дозволив ідентифікувати прогалину на ринку: відсутність безкоштовного кросплатформеного рішення, що поєднує стрес-тестування, повноцінне збереження результатів із можливістю пошуку та захисну діагностику в єдиному застосунку.

У другому розділі виконано детальне проєктування системи на основі трирівневої архітектури з чітким розмежуванням між шаром представлення, бізнес-логікою та рівнем доступу до даних. Обґрунтовано вибір технологічного стеку: Java 17 LTS із JavaFX 21 для забезпечення кросплатформеності та апаратно-прискореного рендерингу інтерфейсу, SQLite через JDBC для вбудованого зберігання результатів без потреби в окремому сервері, LibreHardwareMonitor як зовнішній HTTP-сервіс для отримання температурних показників апаратних

датчиків. Спроєктовано централізований планувальник метрик на базі `Timeline` з однохвилинним ковзним вікном 60 точок, що гарантує постійну продуктивність рендерингу графіків незалежно від тривалості тест-сесії. Розроблено дворівневу систему діагностичних порогів із автоматичним аварійним зупиненням стрес-тесту при перевищенні критичних температур.

У третьому розділі реалізовано повний комплекс програмних модулів та проведено всебічне тестування системи. Реалізація збору метрик CPU та RAM через `OperatingSystemMXBean`, рекурсивного JSON-парсера для температурних показників LHM та GPU-стрес-тесту на основі JavaFX-рендерингу підтвердила практичну ефективність обраних підходів. Підсистема аварійного зупинення успішно пройшла випробування: час від виявлення критичного стану до зупинки стрес-тесту склав не більше 180 мс. Функціональне тестування із 11 тест-кейсами показало 100% проходження всіх перевірок. Тестування продуктивності підтвердило, що власне споживання ресурсів системою не перевищує 2% CPU та 220 МБ RAM у будь-якому режимі роботи, що відповідає нефункціональній вимозі мінімального впливу на спостережувану систему. Порівняльний аналіз із аналогами визначив чітку нішу розробленої системи та підтвердив унікальність поєднання її функціональних можливостей серед безкоштовних рішень.

Практична цінність розробленої системи полягає у тому, що вона надає домашнім користувачам та студентам технічних спеціальностей інструмент, що раніше був доступний лише у вигляді окремих платних або вузькоспеціалізованих рішень. Поєднання регульованого стрес-тестування CPU та GPU, реального моніторингу восьми ключових апаратних метрик, структурованого збереження результатів у SQLite із повноцінним SQL-пошуком, захисного механізму аварійного зупинення та підтримки двох мов інтерфейсу в одному безкоштовному кросплатформеному застосунку є конкурентною перевагою розробленої системи.

Перспективами подальшого розвитку системи є розширення переліку метрик за рахунок інтеграції з NVML для детального моніторингу NVIDIA GPU та ADL/ADLX для AMD-карток, що дозволить відображати завантаженість відеопам'яті та споживану потужність без залежності від LibreHardwareMonitor.

Додавання підтримки S.M.A.R.T.-моніторингу накопичувачів розширить захисні можливості системи на підсистему зберігання даних. Реалізація веб-інтерфейсу на базі вбудованого HTTP-сервера відкрила б можливість дистанційного моніторингу кількох машин із єдиної панелі управління. Впровадження алгоритмів машинного навчання для аналізу накопичених часових рядів метрик дозволило б перейти від реактивної діагностики до проактивного прогнозування потенційних відмов компонентів на основі виявлення аномальних трендів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Гук М. Апаратні засоби ПК: енциклопедія. Київ : BHV, 2017. 1072 с.
2. Patterson D. A., Hennessy J. L. Computer Organization and Design: The Hardware/Software Interface. ARM Edition. Cambridge : Morgan Kaufmann, 2016. 904 p.
3. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. 10th ed. Hoboken : Wiley, 2018. 976 p.
4. Bloch J. Effective Java. 3rd ed. Boston : Addison-Wesley, 2018. 412 p.
5. Sharan K. Learn JavaFX 17: Building User Experience and Interfaces with Java. 2nd ed. New York : Apress, 2022. 1100 p.
6. Deed R. Pro JavaFX 9: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients. New York : Apress, 2018. 600 p.
7. McLaughlin B., Flanagan D. Java in a Nutshell: A Desktop Quick Reference. 7th ed. Sebastopol : O'Reilly Media, 2019. 425 p.
8. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston : Addison-Wesley, 1994. 395 p.
9. Horstmann C. S. Core Java, Volume I – Fundamentals. 12th ed. Boston : Addison-Wesley, 2021. 928 p.
10. Horstmann C. S. Core Java, Volume II – Advanced Features. 12th ed. Boston : Addison-Wesley, 2022. 832 p.
11. SQLite Documentation. SQLite Is a C-Language Library. URL: https://www.sqlite.org/docs.html (дата звернення: 15.03.2024).
12. Oracle Corporation. Java Platform, Standard Edition 17 API Specification. URL: https://docs.oracle.com/en/java/javase/17/docs/api/ (дата звернення: 10.03.2024).
13. Oracle Corporation. JavaFX 21 API Documentation. URL: https://openjfx.io/javadoc/21/ (дата звернення: 12.03.2024).

14. LibreHardwareMonitor. Open Source Hardware Monitoring. URL: https://github.com/LibreHardwareMonitor/LibreHardwareMonitor (дата звернення: 18.03.2024).

15. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide. Santa Clara : Intel, 2023. 1200 p.

16. AMD. AMD64 Architecture Programmer's Manual. Volume 2: System Programming. Sunnyvale : AMD, 2023. 850 p.

17. DMTF Organization. System Management BIOS (SMBIOS) Reference Specification. Version 3.6.0. URL: https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.6.0.pdf (дата звернення: 05.03.2024).

18. NVIDIA Corporation. NVIDIA Management Library (NVML) API Reference Guide. URL: https://docs.nvidia.com/deploy/nvml-api/ (дата звернення: 20.03.2024).

19. AMD. AMD Device Library Extra (ADLX) SDK Documentation. URL: https://gpuopen.com/adlx/ (дата звернення: 20.03.2024).

20. Microsoft Corporation. Windows Management Instrumentation Documentation. URL: https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page (дата звернення: 08.03.2024).

21. The Linux Kernel Documentation. Hardware Monitoring Kernel Drivers. URL: https://www.kernel.org/doc/html/latest/hwmon/index.html (дата звернення: 08.03.2024).

22. Prometheus Authors. Prometheus Documentation: Overview. URL: https://prometheus.io/docs/introduction/overview/ (дата звернення: 22.03.2024).

23. Netdata Inc. Netdata Agent Documentation. URL: https://learn.netdata.cloud/docs/agent (дата звернення: 22.03.2024).

24. Apache Software Foundation. Apache Maven Project Documentation. URL: https://maven.apache.org/guides/ (дата звернення: 14.03.2024).

25. Xerial. SQLite JDBC Driver. URL: https://github.com/xerial/sqlite-jdbc (дата звернення: 15.03.2024).

26. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 520 p.

27. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 533 p.

28. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Boston : Prentice Hall, 2008. 431 p.

29. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Boston : Addison-Wesley, 2006. 432 p.

30. Infoq. JVM Memory Model and Garbage Collection Overview. URL: https://www.infoq.com/articles/Java-PERMGEN-Removed/ (дата звернення: 17.03.2024).

31. Bohra Д. Pro Docker. New York : Apress, 2016. 250 p.

32. S.M.A.R.T. Attribute Overview. Smartmontools Project Documentation. URL: https://www.smartmontools.org/wiki/TocDoc (дата звернення: 06.03.2024).

33. NVM Express Organization. NVM Express Base Specification. Revision 2.0c. URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf (дата звернення: 06.03.2024).

34. OpenJFX Project. Getting Started with JavaFX. URL: https://openjfx.io/openjfx-docs/ (дата звернення: 13.03.2024).

35. Juneau J. Java 17 Recipes: A Problem-Solution Approach. 3rd ed. New York : Apress, 2022. 620 p.

36. Глинський Я. М., Анохін В. Є., Ряжська В. А. Інформатика та інформаційні технології: навчальний посібник. Львів : Деол, 2018. 256 с.

37. Дибкова Л. М. Інформатика та комп'ютерна техніка: навчальний посібник. 3-тє вид. Київ : Академвидав, 2011. 464 с.

38. HWiNFO. HWiNFO System Information Utility Documentation. URL: https://www.hwinfo.com/forum/threads/hwinfo-faq.1172/ (дата звернення: 25.03.2024).

39. CPU-Z. CPU-Z Technical Documentation and Validation. URL: https://www.cpuid.com/softwares/cpu-z.html (дата звернення: 25.03.2024).

40. AIDA64. AIDA64 Feature List and Technical Overview. URL: https://www.aida64.com/products/features (дата звернення: 26.03.2024).

41. Coulouris G., Dollimore J., Kindberg T., Blair G. Distributed Systems: Concepts and Design. 5th ed. Boston : Addison-Wesley, 2011. 1063 p.

42. Oracle Corporation. Java Management Extensions (JMX) Guide. URL: https://docs.oracle.com/en/java/javase/17/jmx/introduction-jmx-technology.html (дата звернення: 11.03.2024).

43. Lea D. Concurrent Programming in Java: Design Principles and Patterns. 2nd ed. Boston : Addison-Wesley, 1999. 411 p.

44. Richardson L., Amundsen M. RESTful Web APIs. Sebastopol : O'Reilly Media, 2013. 408 p.

45. JEDEC Solid State Technology Association. JEDEC Standard: DDR5 SDRAM. JESD79-5B. URL: https://www.jedec.org/standards-documents/docs/jesd79-5b (дата звернення: 07.03.2024).

Додаток А

Інтерфейс системи

На зображенні представлено головний інтерфейс застосунку, який містить елементи керування тестуванням, графік навантаження системи та основні показники (CPU, RAM, температура).

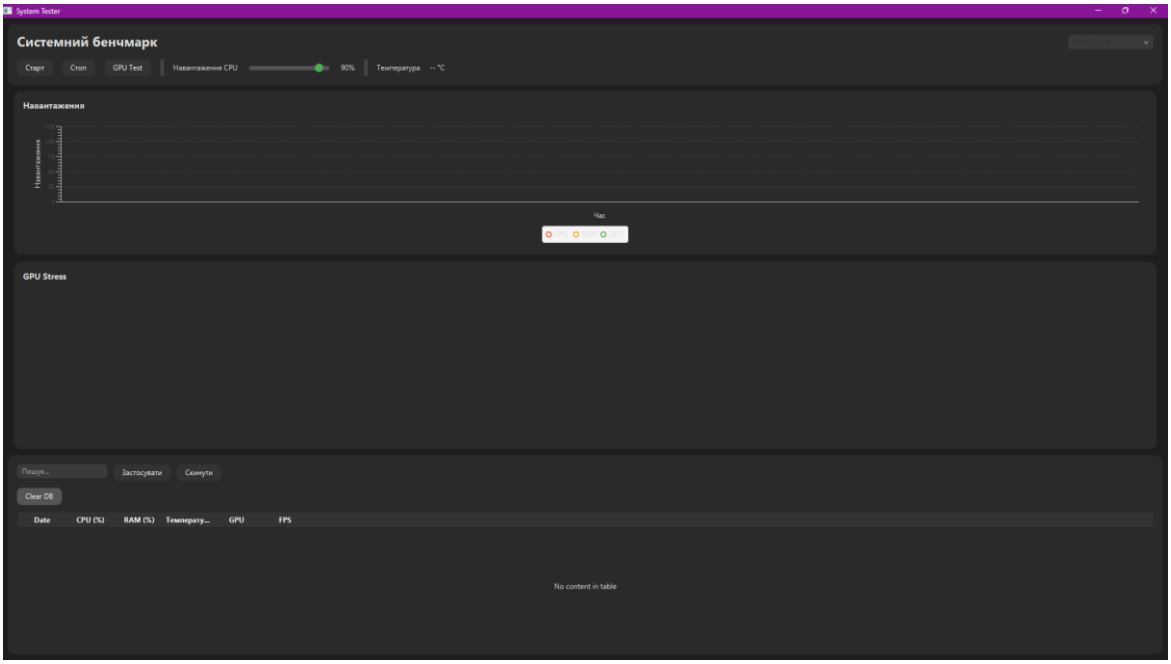


Рисунок А.1 – Головне вікно програми

Зображено процес навантаження процесора, де відображається зміна використання CPU у реальному часі на графіку.

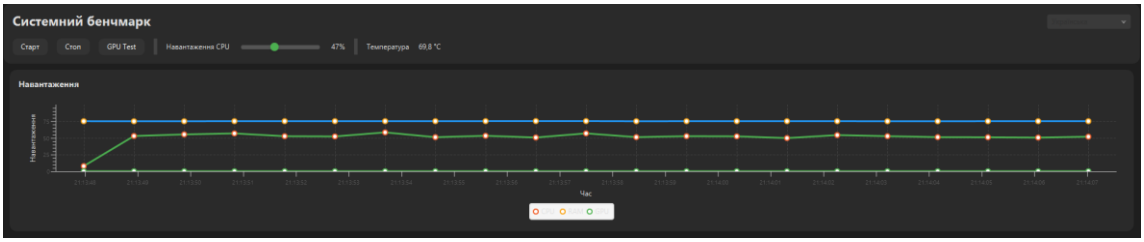


Рисунок А.2 – Процес виконання CPU-стрес тесту

На зображенні показано відображення поточних значень використання оперативної пам’яті, процесора та температури під час тестування.

CPU (%)	RAM (%)	Температу...
52.5426482...	75.1112630...	70.4

Рисунок А.3 – Моніторинг системних ресурсів

Представлено процес навантаження графічного процесора з відображенням FPS та графічного рендерингу.

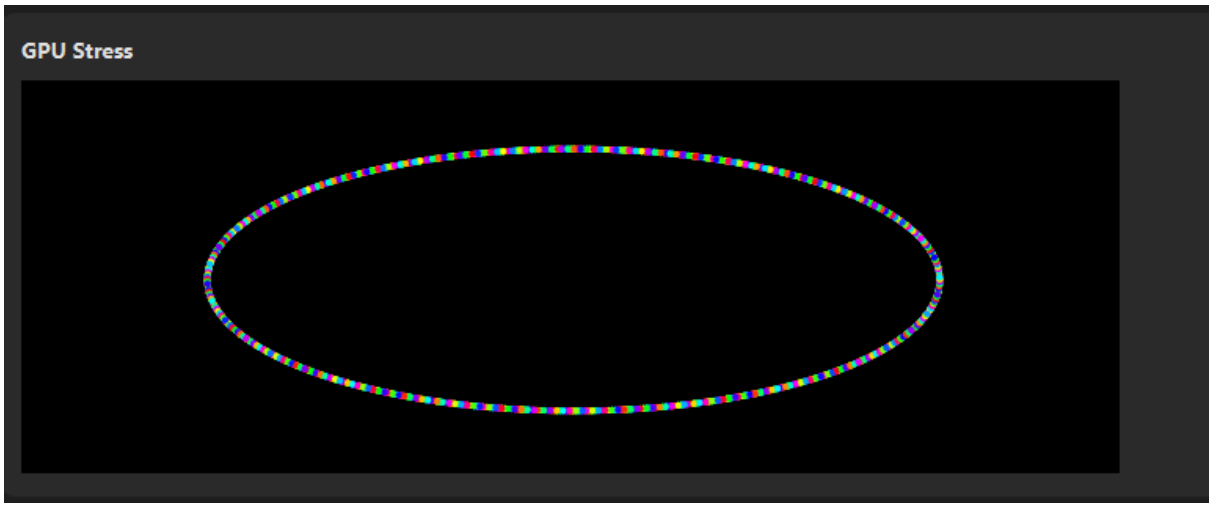


Рисунок А.4 – Виконання GPU-стрес тесту

На зображенні показано таблицю збережених результатів тестів, включаючи дату, навантаження CPU, RAM, температуру, GPU та FPS.

Date	CPU (%)	RAM (%)	Температу...	GPU	FPS
2026-03-30...	54.9269649...	76.2674662...	71.4	32.8	61.0
2026-03-30...	52.5426482...	75.1112630...	70.4	0.0	0.0

Рисунок А.5 – Історія результатів тестування

Зображено використання поля пошуку та фільтрів для відбору результатів тестування за заданими параметрами.

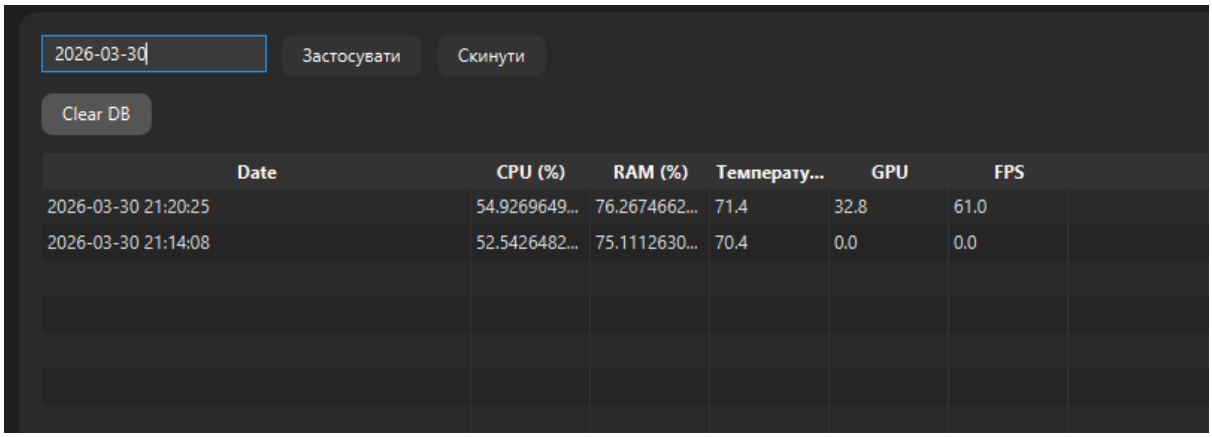


Рисунок А.6 – Фільтрація та пошук

На зображенні продемонстровано можливість перемикання мови інтерфейсу між українською та англійською.

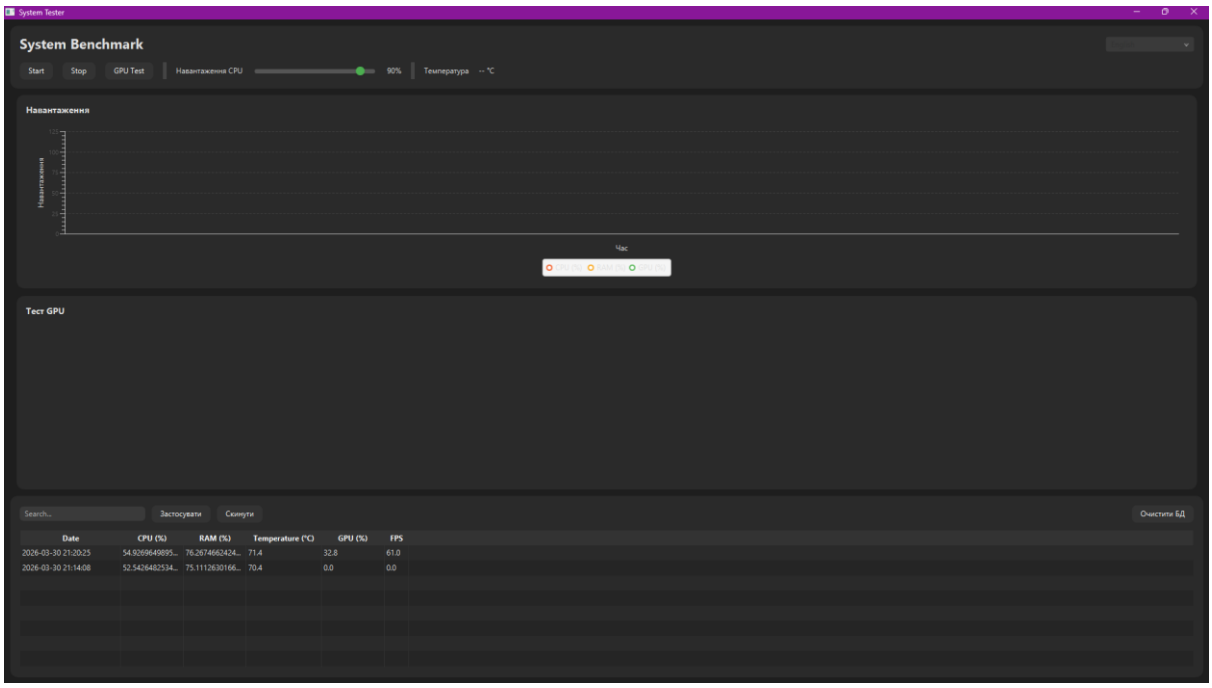


Рисунок А.7 – Зміна мови інтерфейсу

Додаток Б

Структура бази даних

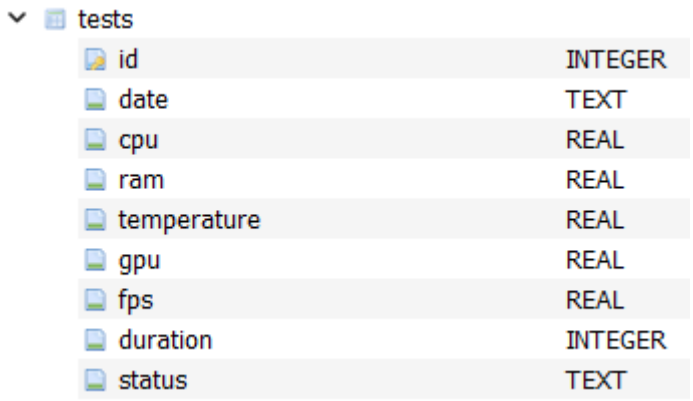


Рисунок Б.1 – Структура таблиці tests бази даних SQLite

Додаток В

Фрагменти програмного коду

```
private void update() {

    if (!running) return;

    double cpu = manager.getCpu();
    double ram = manager.getRam();
    double temp = manager.getTemperature();
    double gpuLoad = gpu.getGpuLoad();

    String time = LocalTime.now().format(timeFormatter);

    cpuSeries.getData().add(new XYChart.Data<>(time, cpu));
    ramSeries.getData().add(new XYChart.Data<>(time, ram));
    gpuSeries.getData().add(new XYChart.Data<>(time,
gpuLoad));
```

```
    duration++;

    tempLabel.setText(String.format("%.1f °C", temp));

    if (cpu > 95 || temp > 90) {
        stopTest();
    }

    if (cpuSeries.getData().size() > MAX_POINTS) {
        cpuSeries.getData().remove(0);
        ramSeries.getData().remove(0);
        gpuSeries.getData().remove(0);
    }
}
```

Лістинг В.1 – Оновлення апаратних метрик та контроль температури системи

```
public static void save(double cpu, double ram, double temp,
                        double gpu, double fps,
                        int duration, String status) {

    String sql = """
        INSERT INTO tests(date, cpu, ram, temperature, gpu,
fps, duration, status)
        VALUES(?,?,?,?,?,?,?,?)
    """;

    String now = LocalDateTime.now()
            .format(DateTimeFormatter.ofPattern("yyyy-MM-dd
```

```
HH:mm:ss"));

    try (Connection c = connect();
         PreparedStatement ps = c.prepareStatement(sql)) {

        ps.setString(1, now);
        ps.setDouble(2, cpu);
        ps.setDouble(3, ram);
        ps.setDouble(4, temp);
        ps.setDouble(5, gpu);
        ps.setDouble(6, fps);
        ps.setInt(7, duration);
        ps.setString(8, status);

        int rows = ps.executeUpdate();
```

Лістинг В.2 – Збереження результатів тестування у базі даних SQLite

```
package com.example.app.stress;

import java.util.ArrayList;
import java.util.List;

public class CpuStressTest {

    private final List<Thread> workers = new ArrayList<>();
    private volatile boolean running = false;

    public void start(int threads, int loadPercent) {
```

```
        running = true;

        for (int i = 0; i < threads; i++) {
            Thread t = new Thread(() -> {

                long workTime = loadPercent;
                long sleepTime = 100 - loadPercent;

                while (running) {

                    long start = System.currentTimeMillis();

                    while ((System.currentTimeMillis() -
start) < workTime) {
                        double x = Math.sin(System.nanoTime())
* Math.tan(System.nanoTime());
                    }

                    try {
                        Thread.sleep(sleepTime);
                    } catch (InterruptedException ignored) {}
                }

            });

            t.setDaemon(true);
            t.start();
            workers.add(t);
        }
```

```
    }

    public void stop() {
        running = false;
        workers.clear();
    }
}
```

Лістинг В.3 – Реалізація навантаження на центральний процесор

Додаток Г

Презентація



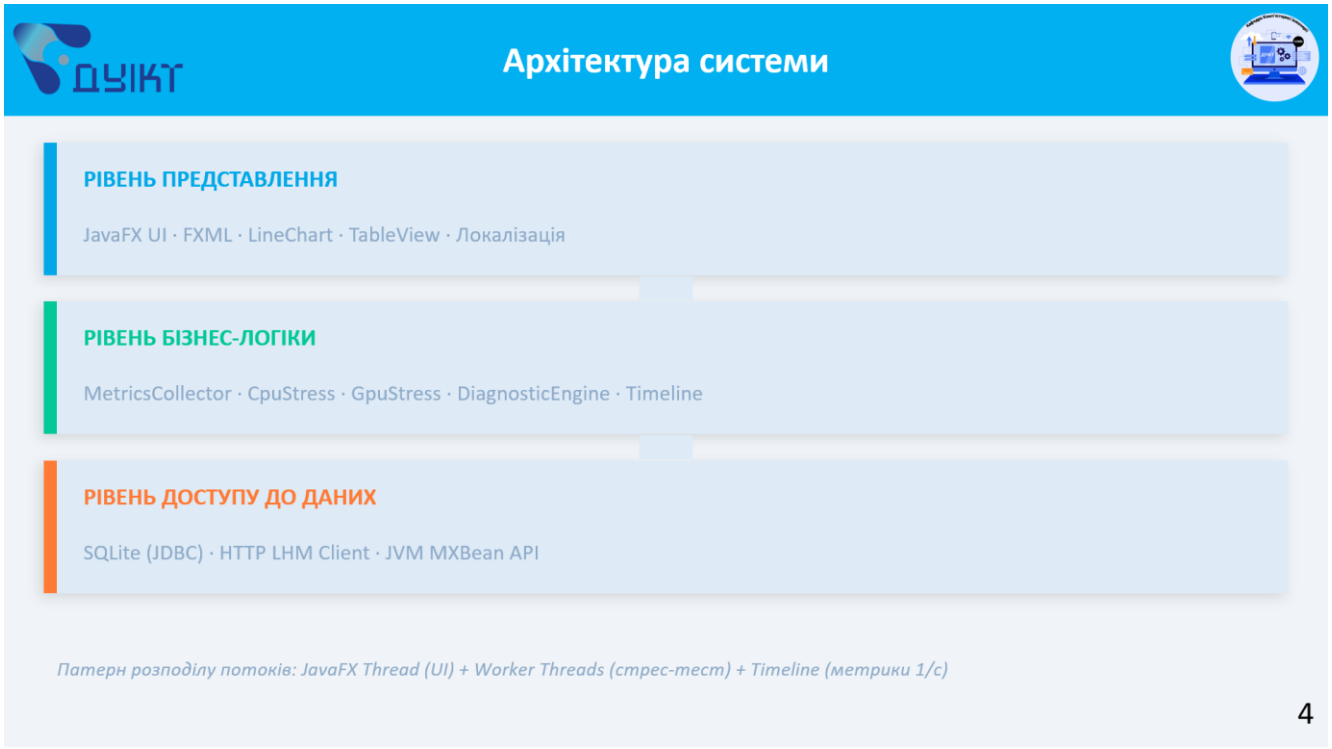
Слайд Г.1 – Перша сторінка презентації



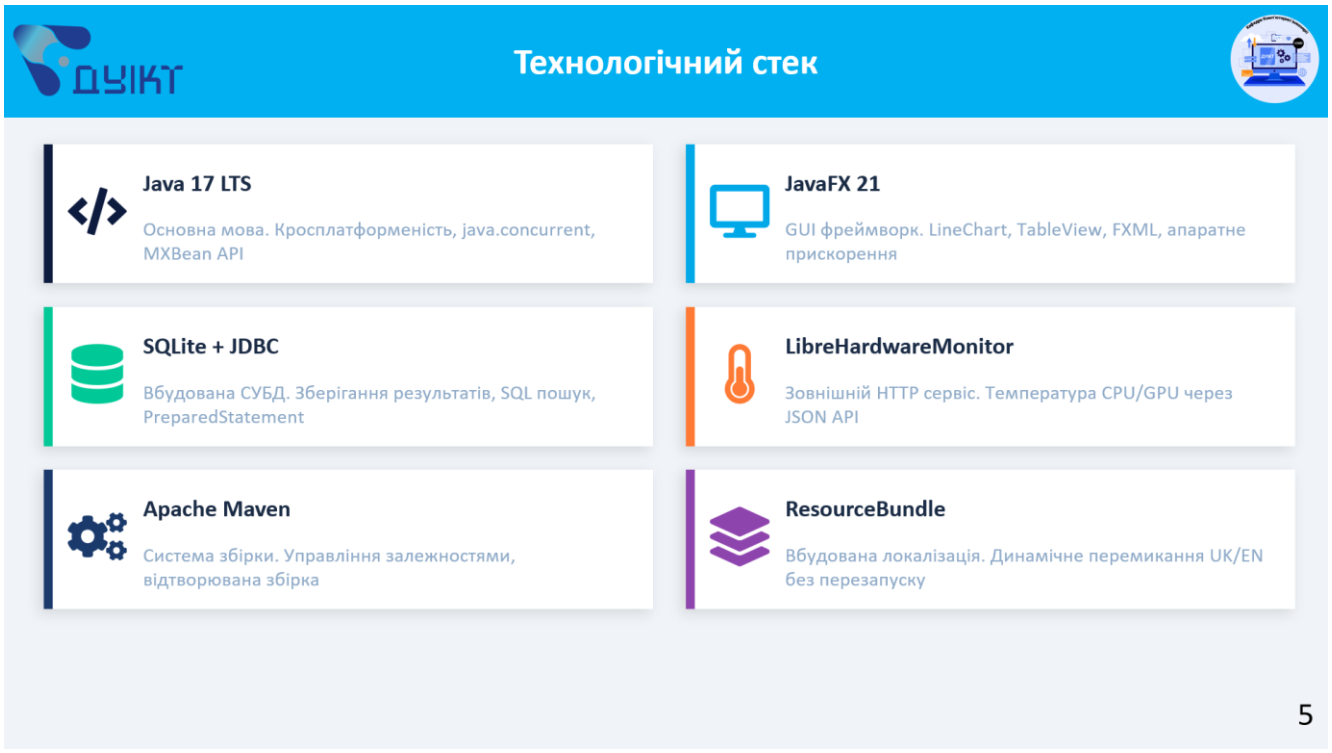
Слайд Г.2 – Друга сторінка презентації



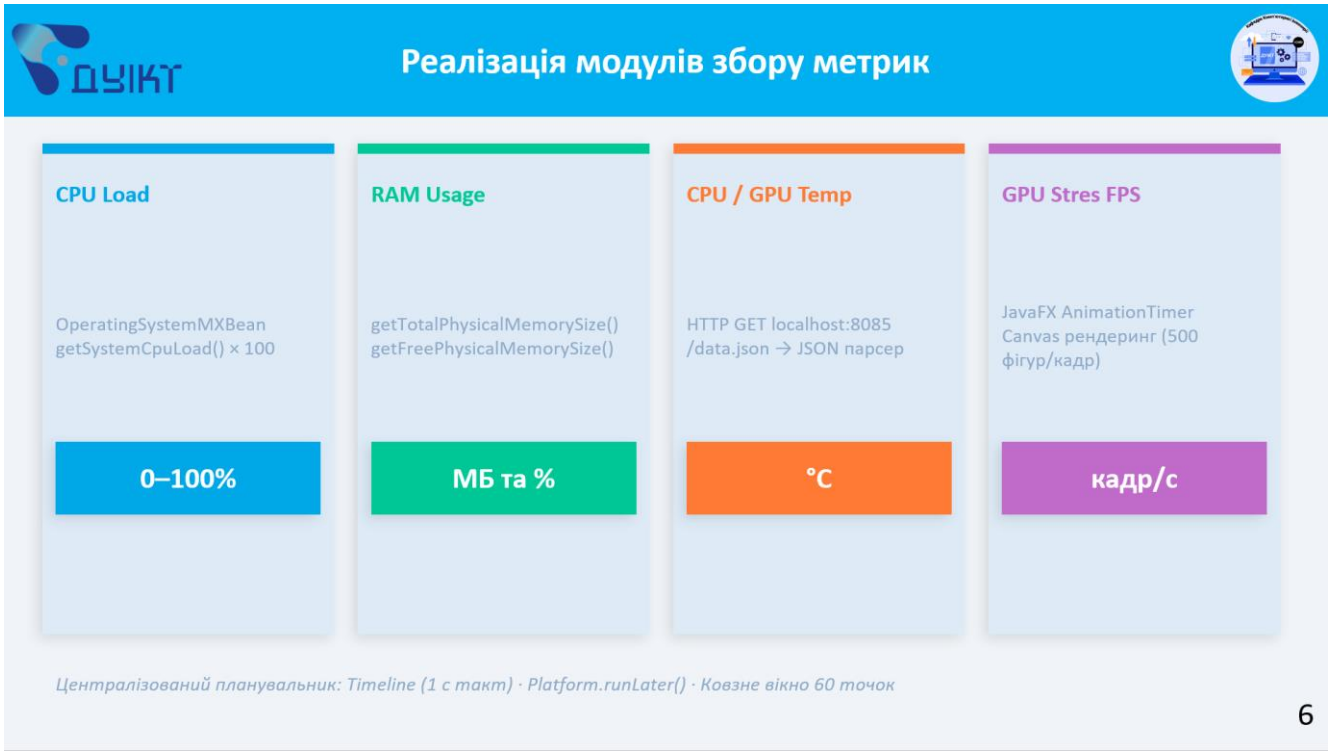
Слайд Г.3 – Третя сторінка презентації



Слайд Г.4 – Четверта сторінка презентації



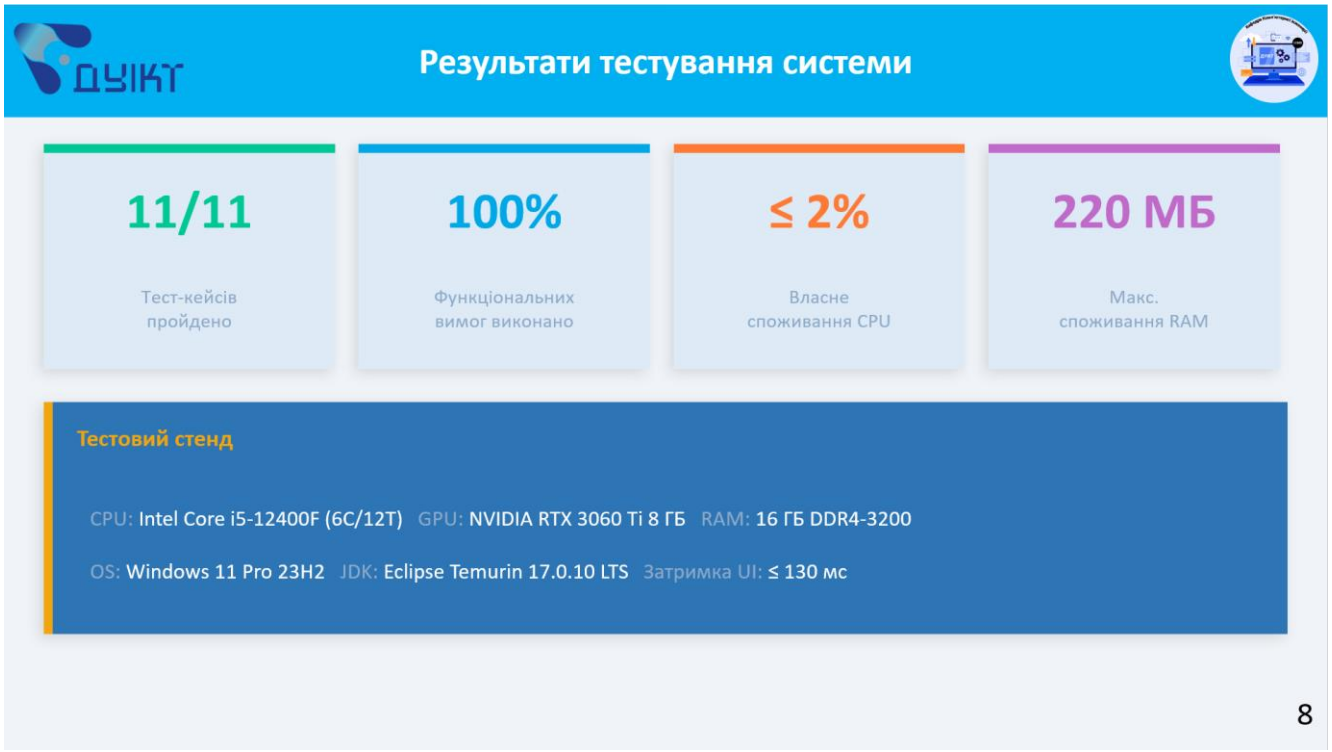
Слайд Г.5 – П’ята сторінка презентації



Слайд Г.6 – Шоста сторінка презентації



Слайд Г.7 – Сьома сторінка презентації



Слайд Г.8 – Восьма сторінка презентації



Слайд Г.9 – Дев’ята сторінка презентації



Слайд Г.10 – Десята сторінка презентації