НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету Інформаційних технологій	**Завідувач кафедри** Комп'ютерних систем, мереж та кібербезпеки
_____Ігор БОЛБОТ (підпис)	_____Дмитро КАСАТКІН (підпис)
«___»_____2026 р.	«___»_____2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: « Розроблення комп'ютерної системи моніторингу продуктивності обчислювальної системи на основі засобів Python і бібліотеки Psutil »

Спеціальність F7 «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Гарант освітньої програми
канд. фіз.-мат. наук, доцент _____ (підпис) **Євгеній НІКІТЕНКО**

Керівник бакалаврської кваліфікаційної роботи
доктор технічних наук, професор _____ (підпис) **Вадим ШКАРУПИЛО**

Виконав _____ (підпис) **Владислав МЕЛЬНИЧЕНКО**

КИЇВ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
канд. пед. наук, доцент_____Дмитро КАСАТКІН
«_____»_____20____ р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Мельниченку Владиславу Юрійовичу
(прізвище, ім'я, по батькові)

Спеціальність «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Тема кваліфікаційної бакалаврської роботи
« Розроблення комп'ютерної системи моніторингу продуктивності обчислювальної системи на основі засобів Python і бібліотеки Psutil »
затверджена наказом ректора НУБіП України від «10»___12_____2026 р. №___3072"С"___
Термін подання завершеної роботи на кафедру «__»_____2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи

Перелік питань, що підлягають дослідженню:
Аналіз методів та існуючих програмних засобів моніторингу продуктивності обчислювальних систем. Проєктування архітектури системи моніторингу та обґрунтування вибору технологічного стека (Python, PyQt5, psutil). Програмна реалізація модулів збору апаратних метрик, багатопотокових алгоритмів та графічного інтерфейсу користувача. Верифікація точності збору системних метрик та тестування стабільності розробленого програмного продукту.
Перелік графічного матеріалу (за необхідності).

Дата видачі завдання "__10__"_____02_____2026 р.

Керівник бакалаврської кваліфікаційної роботи
доктор технічних наук, професор _____ (підпис) **Вадим ШКАРУПИЛО**

Завдання взяв до виконання _____ (підпис) **Владислав МЕЛЬНИЧЕНКО**

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області та огляд існуючих систем моніторингу	14.02.2026 р. – 28.02.2026 р.	Виконано
2	Проєктування архітектури комп'ютерної системи	01.03.2026 р. – 20.03.2026 р.	Виконано
3	Програмна реалізація комп'ютерної системи моніторингу	21.03.2026 р. – 15.04.2026 р.	Виконано
4	Тестування розробленого програмного продукту	16.04.2026 р. – 28.04.2026 р.	Виконано
5	Оформлення пояснювальної записки та підготовка до захисту	29.04.2026 р. – 05.05.2026 р.	Виконано

Керівник бакалаврської кваліфікаційної роботи
доктор технічних наук, професор _____ (підпис) **Вадим ШКАРУПИЛО**

Завдання взяв до виконання _____ (підпис) **Владислав МЕЛЬНИЧЕНКО**

РЕФЕРАТ

Пояснювальна записка: 61 сторінка, 20 рисунків, 5 таблиць, 5 лістингів, 25 джерел.

МОНІТОРИНГ ПРОДУКТИВНОСТІ, ОБЧИСЛЮВАЛЬНА СИСТЕМА, PYTHON, PYQT5, PSUTIL, БАГАТОПОТОКОВІСТЬ, СИСТЕМНІ МЕТРИКИ, ГРАФІЧНИЙ ІНТЕРФЕЙС

Об'єкт дослідження — процес моніторингу продуктивності апаратних ресурсів обчислювальної системи під управлінням ОС Microsoft Windows.

Предмет дослідження — методи та засоби збору, обробки і візуалізації системних метрик на основі мови Python, бібліотек PyQt5, pyqtgraph та psutil.

Мета роботи — розроблення десктопного застосунку для моніторингу продуктивності обчислювальної системи, що забезпечує збір і відображення апаратних показників у реальному часі без застосування систем керування базами даних.

У першому розділі проаналізовано існуючі рішення для моніторингу систем та обґрунтовано вибір технологічного стека (Python, PyQt5, pyqtgraph, psutil, WMI).

У другому розділі спроєктовано трирівневу архітектуру застосунку, обґрунтовано використання in-memory структур collections.deque замість СУБД та розроблено UML-діаграми (прецедентів, компонентів, діяльності).

У третьому розділі реалізовано модулі збору метрик (psutil, WMI, Win32 API), асинхронні QThread-воркери та багатопотокові алгоритми бенчмаркінгу. Створено графічний інтерфейс із live-графіками, динамічною локалізацією (UA/EN) і модулем генерації HTML-звітів.

У четвертому розділі проведено верифікацію коректності метрик відносно Windows Task Manager, навантажувальне тестування та оцінку стабільності багатопотокової архітектури протягом 1 години безперервної роботи.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 7
ВСТУП 9
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ 11
1.1. Поняття моніторингу продуктивності обчислювальних систем та його роль у системному адмініструванні 11
1.2. Порівняльний аналіз існуючих систем моніторингу: Windows Task Manager, HWiNFO64, AIDA64, MSI Afterburner 12
1.3. Аналіз засобів розробки: мова Python, бібліотека psutil, GUI-фреймворк PyQt5 та бібліотека pyqtgraph 13
1.4. Постановка задачі та формування функціональних і нефункціональних вимог до розроблюваної системи 14
РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ 17
2.1. Архітектура програмного забезпечення: модульна структура, шаблони проектування Observer і Worker та розподіл відповідальності між рівнями ... 17
2.2. Проектування підсистеми збору метрик: psutil як основне джерело даних, WMI/Win32 API та інтеграція з PowerShell для отримання показників недоступних через psutil 19
2.3. Проектування багатопотокової архітектури: QThread-воркери, фонові threading.Thread з TTL-кешуванням WMI-даних та ProcessPoolExecutor для обходу GIL у CPU-бенчмарку 21
2.4. Проектування графічного інтерфейсу на базі PyQt5: структура головного вікна, система вкладок QStackedWidget та механізм міжпотокової взаємодії через сигнали і слоти 22
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ 25
3.1. Реалізація модулів збору апаратних метрик: CPU (частота через WMI), GPU (рушії та VRAM через PowerShell), RAM, диск, мережа, температура (LibreHardwareMonitor) та акумулятор 25
3.2. Реалізація фонових QThread-воркерів: MonitorWorker (головний поллер 1 с), CpuBenchWorker (ST/MT стрес-тест із ProcessPoolExecutor), DiskBenchmarkWorker та SpeedtestWorker 29
3.3. Розробка графічного інтерфейсу: sidebar-навігація, кругові gauge-віджети, live-графіки pyqtgraph на deque-буферах, система інтернаціоналізації (i18n UA/EN) та генерація HTML-звіту 31

3.4. Тестування системи: верифікація коректності метрик відносно показників Windows Task Manager, аналіз накладних витрат багатопоточності та оцінка часу відгуку інтерфейсу 34
3.5. Розгортання системи та керівництво користувача 36
РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОЇ КОМП'ЮТЕРНОЇ СИСТЕМИ 41
4.1. Методика та середовище тестування 41
4.2. Верифікація коректності збору системних метрик в реальному часі 42
4.3. Навантажувальне тестування (стрес-тест CPU) та бенчмаркінг диска 48
4.4. Тестування стабільності GUI та багатопотокової архітектури 53
ВИСНОВКИ 57
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 59

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ОС — операційна система

AC — змінний струм (Alternating Current)

API — прикладний програмний інтерфейс (Application Programming Interface)

BIOS — базова система введення-виведення (Basic Input/Output System)

CPU — центральний процесор (Central Processing Unit)

CSS — каскадні таблиці стилів (Cascading Style Sheets)

DC — постійний струм (Direct Current)

DIMM — двобічний модуль оперативної пам'яті (Dual Inline Memory Module)

DLL — динамічно підключувана бібліотека (Dynamic Link Library)

DWM — диспетчер вікон робочого столу (Desktop Window Manager)

GB — гігабайт (Gigabyte)

GHz — гігагерц (Gigahertz)

GPU — графічний процесор (Graphics Processing Unit)

GUI — графічний інтерфейс користувача (Graphical User Interface)

HTML — мова розмітки гіпертексту (HyperText Markup Language)

I/O — введення/виведення (Input/Output)

JSON — текстовий формат обміну даними (JavaScript Object Notation)

KB — кілобайт (Kilobyte)

MB — мегабайт (Megabyte)

MHz — мегагерц (Megahertz)

MOps — мільйон операцій (Million Operations)

mWh — міліват-година (Milliwatt-hour)

PB — петабайт (Petabyte)

PID — ідентифікатор процесу (Process Identifier)

QSS — таблиці стилів Qt (Qt Style Sheets)

RAM — оперативна пам'ять (Random Access Memory)

RSS — розмір резидентного набору (Resident Set Size)

SMBIOS — базова система введення-виведення керування системою (System

Management BIOS)

SPD — послідовне визначення наявності (Serial Presence Detect)

SWAP — файл підкачки (Swap Memory)

TB — терабайт (Terabyte)

TCP — протокол керування передачею (Transmission Control Protocol)

TTL — час життя (Time To Live)

UDP — протокол датаграм користувача (User Datagram Protocol)

UEFI — уніфікований розширюваний інтерфейс мікропрограми (Unified Extensible Firmware Interface)

UI — інтерфейс користувача (User Interface)

UTC — всесвітній координований час (Coordinated Universal Time)

VRAM — відеопам'ять (Video Random Access Memory)

WMI — інструментарій керування Windows (Windows Management Instrumentation)

ВСТУП

Ефективний контроль апаратних ресурсів є умовою надійної роботи будь-якої обчислювальної системи. Стандартні засоби операційних систем — зокрема Диспетчер задач Windows — надають базовий перелік показників, проте позбавлені розширених можливостей діагностики, гнучкого налаштування та програмованого збору даних [5]. Комерційні рішення для моніторингу обчислювальних систем нерідко є закритими, прив'язаними до конкретного постачальника або надлишково складними для задач одинарної робочої станції [2]. Потреба у відкритому, розширюваному та портативному застосунку з підтримкою двомовного інтерфейсу і формування звітів зумовлює актуальність теми дипломної роботи.

Мета роботи — розроблення комп'ютерної системи моніторингу продуктивності обчислювальної системи з графічним інтерфейсом реального часу на базі Python і бібліотеки Psutil.

Завдання дослідження:

1. Проаналізувати задачу моніторингу обчислювальних систем, провести порівняльний огляд існуючих рішень та обґрунтувати вибір технологічного стека.
2. Спроєктувати модульну архітектуру застосунку із дотриманням принципу поділу відповідальностей та визначити підхід до зберігання метрик реального часу.
3. Реалізувати модулі збору системних метрик із задіянням бібліотеки Psutil, протоколу WMI та Win32 API, а також асинхронні воркери для бенчмаркінгу процесора і диска.
4. Розробити графічний інтерфейс із живими графіками, кастомними QPainter-віджетами, двомовною підсистемою інтернаціоналізації та генерацією HTML-звіту.
5. Провести верифікацію збору метрик, навантажувальне тестування алгоритмів бенчмаркінгу та оцінку стабільності застосунку при тривалій

роботі.

Об'єкт дослідження — процес моніторингу продуктивності апаратних ресурсів обчислювальної системи під управлінням Microsoft Windows.

Предмет дослідження — методи і засоби збору, обробки та візуалізації системних метрик на основі мови програмування Python, бібліотек PyQt5, pyqtgraph і Psutil.

Розроблений програмний продукт є повністю функціонуючим десктопним застосунком для операційної системи Windows, що не потребує зовнішньої СУБД або серверної інфраструктури. Застосовані архітектурні рішення — трирівневий поділ на колектори, воркери та шар представлення, механізм Signals & Slots для потокобезпечного оновлення інтерфейсу, обхід GIL Python через ProcessPoolExecutor — становлять практичну цінність як зразок організації багатопотокового десктопного програмного забезпечення на PyQt5. Двомовна підсистема інтернаціоналізації та механізм генерації самодостатніх HTML-звітів розширюють практичну застосовність продукту без залучення додаткових залежностей [8, 23].

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1. Поняття моніторингу продуктивності обчислювальних систем та його роль у системному адмініструванні

Моніторинг обчислювальної системи являє собою безперервний процес збору, аналізу та візуалізації показників, що характеризують стан апаратних і програмних ресурсів комп'ютера. До таких показників належать завантаженість центрального процесора, обсяг використовуваної оперативної пам'яті, інтенсивність дискових операцій, мережевий трафік, температурний режим компонентів, рівень заряду акумулятора у портативних пристроях, перелік запущених процесів та споживані ними ресурси [2]. У розроблюваному програмному продукті моніторинг виконує дві ключові функції — інформаційну (надання користувачеві об'єктивних даних про поточний стан машини) та діагностичну (виявлення аномалій, що передують відмові).

Класифікувати засоби моніторингу доцільно за критерієм глибини охоплення метрик і середовища застосування. До першої групи належать вбудовані інструменти операційних систем — Диспетчер задач Windows, утиліта top у Linux. Їхньою перевагою є нульова вартість і відсутність потреби в інсталяції; недоліком — обмежений набір показників, відсутність історичних даних та неможливість розширення [5]. Друга група — професійні застосунки на кшталт AIDA64, HWMonitor, Process Explorer. Вони пропонують глибокий аналіз, проте часто є комерційними продуктами із закритим кодом і обмеженою інтеграцією зі сторонніми робочими процесами.

Окремий клас становлять серверні системи моніторингу — Zabbix, Prometheus, Nagios. Їх застосовують для інфраструктури центрів обробки даних, де важливі агрегація показників із тисяч вузлів та довготривале зберігання у часових базах даних. Для робочої станції такі рішення надлишкові [2, 5].

Прогалину між поверхневими вбудованими інструментами та переускладненими корпоративними платформами заповнюють настільні застосунки моніторингу — саме до цього сегмента належить розроблюваний програмний продукт. Його перевагами є модульна архітектура, відкритість кодової бази, можливість двомовної локалізації та формат-незалежний експорт звітів.

1.2. Порівняльний аналіз існуючих систем моніторингу: Windows Task Manager, HWiNFO64, AIDA64, MSI Afterburner

Серед мов загального призначення для реалізації системного застосунку моніторингу було розглянуто C++, C#, Java та Python. Вибір зупинено на Python з декількох міркувань. Інтерпретаторна природа мови забезпечує швидкий цикл розробки: зміни в коді виконуються без проміжного етапу компіляції [11]. Динамічна типізація знижує синтаксичні бар'єри під час роботи зі словниками метрик, які мають змінну структуру — різні моделі ЦП повертають різну кількість ядер, машини мають неоднакову кількість дисків, наявність акумулятора залежить від форм-фактора.

Сучасний Python — мова з потужними засобами об'єктно-орієнтованого програмування. Підтримуються інкапсуляція, поліморфізм, успадкування, а також механізм декораторів і протоколи дескрипторів, що дозволяють реалізовувати патерни проектування лаконічно [1, 16]. Класи проектуються відповідно до принципу єдиної відповідальності: кожен колектор займається лише одним типом метрики, кожен робочий потік відповідає за конкретну задачу. Стандартна бібліотека містить готові примітиви для роботи з процесами, потоками, мережею, файловою системою, що знімає потребу в зовнішніх залежностях для базових операцій [20]. Низка типових патернів проектування з канонічного каталогу [10] застосовується в проєкті природно — спостерігач реалізується через сигнали Qt, одинак — через модулі-сінглетони, фабрика — у місцях створення вкладок інтерфейсу.

Не менш суттєвою є екосистема пакетів PyPI. Для розроблюваного

застосунку залучено десятки сторонніх бібліотек — psutil, PyQt5, pyqtgraph, speedtest-cli — кожна з яких має сталі версії та документацію [25]. Стиль написання коду в проєкті відповідає принципам Clean Code: модулі мають єдину відповідальність, імена функцій описові, побічні ефекти локалізовані [7]. Окремою перевагою стало те, що Python дозволяє писати швидко без втрати читабельності — характеристика, важлива для академічного проєкту, який має бути зрозумілим також рецензентам, а не лише авторові [8, 23].

Серед недоліків мови — глобальне блокування інтерпретатора (GIL), яке обмежує справжній паралелізм у межах одного процесу. Проблему обходять переходом до багатопроцесного виконання через модуль multiprocessing та інтерфейс concurrent.futures.ProcessPoolExecutor, що буде продемонстровано на прикладі CPU-бенчмарка в розділі 3 [18, 19]. Швидкодія Python поступається компільованим мовам, однак для застосунку, де основну частину часу займає очікування системних викликів — зчитування лічильників WMI, опитування psutil, мережеві запити speedtest, — накладні витрати інтерпретатора є несуттєвими.

1.3. Аналіз засобів розробки: мова Python, бібліотека psutil, GUI-фреймворк PyQt5 та бібліотека pyqtgraph

Вибір графічної бібліотеки прямо впливає на швидкість відмальовування, можливості стилізації та архітектуру обробки подій у застосунку. Розглянуто три кандидати: Tkinter, PyQt5 та PySide6.

Tkinter входить до стандартної поставки Python і не потребує окремого встановлення. Однак він базується на застарілому графічному рушії Tk, який погано пристосований до високої частоти оновлень. Графіки реального часу, де щосекунди перемальовуються десятки точок на декількох осях, у Tkinter відмальовуються з помітними затримками. Стилізація обмежена темами ttk; повноцінний контроль над виглядом елементів відсутній [9]. Розробник позбавлений механізмів апаратного прискорення — увесь рендеринг

виконується центральним процесором. Створення кастомних віджетів, наприклад кругового індикатора або спідометра швидкості, у Tkinter потребує суттєвих зусиль і дає посередній візуальний результат.

PyQt5 — обгортка над фреймворком Qt 5, написаним мовою C++ [24]. Архітектурно Qt побудовано навколо концепції сигналів і слотів — типобезпечного механізму прив'язки джерела події до її обробника [22]. У контексті розроблюваного застосунку це має визначальне значення: робочий потік MonitorWorker раз на секунду формує словник метрик і випромінює сигнал data_updated, на який підписані всі вкладки інтерфейсу. Жодних колбеків, жодного спільного стану — слоти викликаються автоматично через чергу подій Qt, що перенаправляє виконання між потоками. Такий підхід узгоджується з принципами слабкої зв'язаності компонентів [3].

Qt підтримує апаратне прискорення через OpenGL та растровий рендерер; стилізація виконується через мову QSS, синтаксично подібну до CSS [9]. Це дозволило реалізувати glassmorphism-оформлення з прозорими панелями та згладженими кутами без зовнішніх залежностей. Для побудови графіків задіяно бібліотеку pyqtgraph, оптимізовану під високу частоту оновлень: повне перемальовування 60-точкового буфера займає менш ніж 5 мс на процесорі середнього класу [17]. Альтернативна matplotlib для цього сценарію непридатна — вона спроєктована для статичних публікаційних графіків, а не для живого моніторингу.

PySide6 функціонально близький до PyQt5, відрізняється ліцензією (LGPL замість GPL чи комерційної) та прив'язкою до Qt 6. Перехід на нього потребував би оновлення коду на нові API, тоді як Qt 5 залишається стабільним і підтримуваним. У роботі обрано PyQt5 також з міркувань наявності більшої кількості навчальних матеріалів українською та ширшої бази готових прикладів [4, 24].

1.4. Постановка задачі та формування функціональних і нефункціональних вимог до розроблюваної системи

Базовим інструментом збору метрик у проєкті виступає бібліотека psutil — кросплатформна обгортка над низькорівневими системними викликами [25]. Її API надає функції для отримання завантаженості ЦП (cpu_percent), даних про пам'ять (virtual_memory, swap_memory), дискові розділи (disk_partitions, disk_usage, disk_io_counters), мережеві з'єднання (net_io_counters, net_connections), перелік процесів (process_iter), стан акумулятора (sensors_battery). Реалізація приховує платформні відмінності за єдиним інтерфейсом, що пришвидшує написання коду і робить логіку колекторів простою для супроводу.

Попри широке покриття, psutil має на Windows низку обмежень. Не повертаються поточна тактова частота ядер ЦП (cpu_freq повертає лише номінальне значення без врахування турборежимів), завантаженість графічного процесора, температура компонентів. У версіях для Windows відсутні дані про активний час дискового контролера. Перелічені метрики необхідні для повноцінного огляду стану системи, тому довелося звертатися до нативних механізмів операційної системи.

Першим таким механізмом є Windows Management Instrumentation (WMI) — реалізація стандарту WBEM, що надає об'єктну модель доступу до апаратних і програмних компонентів машини [15]. WMI містить класи лічильників продуктивності, серед яких — Win32_PerfFormattedData_Counters_ProcessorInformation з полем PercentProcessorPerformance, яке дозволяє обчислити поточну тактову частоту як добуток номінальної частоти на відсоток продуктивності [13]. Для GPU застосовано клас Win32_PerfFormattedData_GPUPerformanceCounters_GPUEngine, що повертає утилізацію кожного графічного двигуна окремо — 3D, copy, video decode [14].

Доступ до WMI з Python організовано через виклики PowerShell за допомогою модуля subprocess. Командлет Get-WmiObject приймає назву класу і повертає об'єкти, які серіалізуються у JSON для подальшого розбору засобами

стандартного модуля json [12]. Такий підхід має одну ваду — кожен запит до WMI коштує сотні мілісекунд, що неприйнятно для оновлення раз на секунду. Розв'язанням стало кешування результатів: для кожного WMI-полера створено окремий фоновий потік (threading.Thread) із TTL-кешем тривалістю 2.0–4.0 секунди залежно від характеру метрики [21]. Поки активний потік чекає на завершення нового запиту, інтерфейс отримує попереднє значення з кешу — це дозволяє зберігати плавність анімацій без перерв.

Окремі дані, недоступні через psutil та WMI, отримуються прямими викликами Win32 API через бібліотеку ctypes — інформація про підключені монітори, точна версія операційної системи, конфігурація мережевих адаптерів. Реєстр Windows зчитується модулем winreg для отримання назв процесора та материнської плати у тому форматі, який відображає sysinfo-вкладка застосунку. Багатошарова стратегія збору метрик дозволила досягти повноти охоплення, недосяжної для жодної окремо взятої бібліотеки [2, 5, 6].

РОЗДІЛ 2. ПРОЕКТУВАННЯ СИСТЕМИ МОНІТОРИНГУ

2.1. Архітектура програмного забезпечення: модульна структура, шаблони проектування Observer і Worker та розподіл відповідальності між рівнями

Кодова база системи моніторингу організована за принципом поділу відповідальностей між трьома основними шарами. Перший шар — `app/collector/` — містить модулі, кожен з яких відповідає за збір метрик однієї категорії: cpu.py, memory.py, disk.py, network.py, gpu.py, temperature.py, processes.py, system_info.py, battery.py. Функції колекторів є чистими у тому сенсі, що не змінюють глобального стану, а лише повертають словники з метриками; такий підхід робить їх легко тестованими ізольовано від інтерфейсу [7, 16].

Другий шар — `app/workers/` — реалізує асинхронні робочі процеси на базі класу QThread. MonitorWorker запускається при старті програмного продукту і раз на секунду опитує всі колектори, формує агрегований словник метрик та випромінює сигнал data_updated [22]. CpuBenchWorker, DiskBenchmarkWorker і SpeedtestWorker — спеціалізовані потоки, що виконуються лише на запит користувача й завершуються після видачі результату.

Третій шар — `app/ui/` — містить підмодулі tabs/, widgets/ та модулі main_window.py, title_bar.py, style.qss, icon.py. У шарі представлення жодного безпосереднього звертання до psutil чи WMI не виконується; усі дані надходять виключно через сигнал з MonitorWorker. Така ізоляція відповідає принципу інверсії залежностей та сприяє слабкій зв'язаності між шарами [3, 10].

Допоміжні модулі — `app/i18n.py` (двомовна локалізація UA/EN), `app/html_report.py` (генератор HTML-звіту) та `app/utils.py` (функції форматування bytes_to_human, mhz_to_str) — винесено в окремий рівень утиліт без приналежності до конкретного шару [1].

Узагальнено функціональні можливості, доступні користувачеві, представлено

діаграмою прецедентів. На ній відображено єдиного актора — користувача — та повний перелік базових сценаріїв, що ініціюються в інтерфейсі. До групи моніторингу належать прецеденти «Перегляд оглядової панелі» та «Перегляд метрик підсистем», що охоплюють навігацію між усіма інформаційними вкладками. До групи активних операцій віднесено «Запуск бенчмарків», «Експорт HTML-звіту», «Зміна мови інтерфейсу» та «Керування процесами» (зокрема їх примусове завершення). Суцільні лінії зв'язку на діаграмі позначають класичні асоціації між актором і прецедентами; відношень включення (include) чи розширення (extend) у поточній архітектурі не передбачено (рис. 2.1).

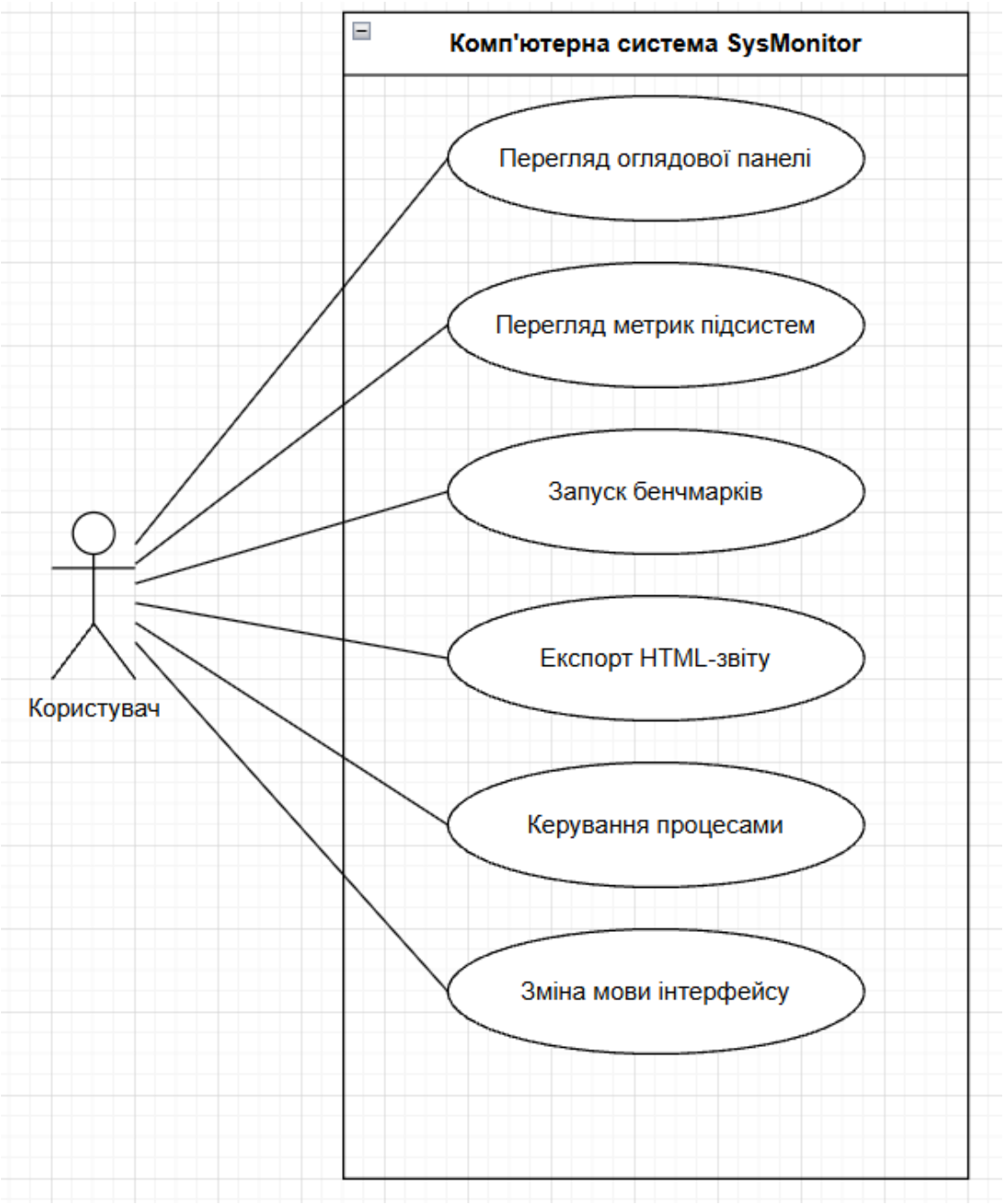

Рисунок 2.1 — Діаграма прецедентів комп'ютерної системи SysMonitor

2.2. Проектування підсистеми збору метрик: psutil як основне джерело даних, WMI/Win32 API та інтеграція з PowerShell для отримання показників недоступних через psutil

Принципове проєктне рішення полягає у відмові від використання будь-якої системи керування базами даних. Жодних SQLite, PostgreSQL, файлових

журналів метрик чи логів. Усі дані живуть виключно в оперативній пам'яті процесу і зникають після його завершення. Рішення прийнято на підставі трьох міркувань.

Перше — мінімізація навантаження на дискову підсистему. Програмний продукт сам по собі є інструментом моніторингу дискових операцій; будь-який запис у файлову базу даних спотворював би показники, які цей продукт має об'єктивно вимірювати [5]. У сценарії, де метрики оновлюються раз на секунду, постійний запис у СУБД додав би сотні дрібних транзакцій на секунду, що було б помітно у графіках disk_io_counters [2].

Друге — для задач візуалізації реального часу персистентне зберігання непотрібне. Користувача цікавить графік за останні 60 секунд, а не за останній місяць. Для такого сценарію оптимальною є структура collections.deque з параметром maxlen=60: при додаванні нового елемента, коли довжина черги досягає максимуму, найстаріший елемент видаляється автоматично [20]. Складність операції — O(1), що принципово важливо для оновлень з частотою 1 Гц. Деки створено для кожного відображуваного графіка — завантаженість ЦП у відсотках, перцентилі ядер, використання пам'яті, мережевий вхідний та вихідний трафік, дисковий read/write, температура.

Третє — in-memory підхід спрощує архітектуру і скорочує кількість зовнішніх залежностей. Відсутність драйвера СУБД, файлу бази, схеми міграції, ORM-шару, конфігурації підключення робить програмний продукт самодостатнім і портативним: достатньо одного виконуваного файлу для запуску [23].

Серед обмежень підходу — неможливість аналізу історичних даних після закриття програми. Користувач, який хоче дослідити патерн зміни завантаженості за весь робочий день, цього зробити не зможе. Компенсує прогалину функція експорту поточного зрізу системи у формат HTML — користувач може зберегти знімок стану перед закриттям. Для довгострокового спостереження доцільно використовувати спеціалізовані серверні рішення на кшталт Prometheus, які виходять за межі задачі дипломного проєкту.

2.3. Проектування багатопотокової архітектури: QThread-воркери, фонові threading.Thread з TTL-кешуванням WMI-даних та ProcessPoolExecutor для обходу GIL у CPU-бенчмарку

Графічний інтерфейс побудовано за двопанельною компоновкою: ліва панель — фіксований сайдбар із кнопками навігації, правий блок — динамічний контейнер на базі класу QStackedWidget, що містить дев'ять вкладок. QStackedWidget зберігає в собі стек віджетів-сторінок і відображає лише одну з них одночасно; перемикання здійснюється методом setCurrentIndex без перестворення віджетів [24]. Це принципово важливо для збереження стану — графіки кожної вкладки продовжують оновлюватися навіть тоді, коли вкладка не видима, тому що сигнал data_updated розсилається в усі підключені слоти незалежно від видимості сторінки.

Кожна вкладка є екземпляром окремого класу, успадкованого від QWidget. Клас має метод update_ui(data: dict), який викликається сигналом MonitorWorker. У межах методу віджети-споживачі (мітки, прогрес-бари, кругові індикатори, графіки pyqtgraph) отримують свіжі значення. Структура data — плаский словник з вкладеними словниками: `data["cpu"]["percent"]`, `data["memory"]["available"]`, `data["disk"][drive]["used"]`. Така схема є компромісом між читабельністю та продуктивністю — серіалізація чи десеріалізація не виконується, усі звертання — за прямою адресою у пам'яті [4].

Для нестандартних візуальних елементів — кругових манометрів та спідометра швидкості — реалізовано власні підкласи QWidget із перевизначеним методом paintEvent. Малювання виконується через QPainter з активним згладжуванням QPainter.Antialiasing, що дає згладжені дуги без сходинок навіть на дисплеях з невисоким DPI. QSS-стилі застосовуються до стандартних віджетів — кнопок сайдбара, заголовків панелей, рамок графіків [9].

На наведеній діаграмі компонентів відображено логічну архітектуру системи та взаємодію її основних шарів. Структурно систему поділено на шар представлення (MainWindow та QStackedWidget), шар робочих процесів

(MonitorWorker та спеціалізовані бенчмарки), шар колекторів (модулі збору метрик) та рівень зовнішніх ресурсів операційної системи. Суцільними лініями на схемі позначено прямі програмні виклики та залежності між модулями (від запитувача до постачальника даних). Натомість асинхронну передачу результатів від фонових потоків до інтерфейсу користувача реалізовано через механізм сигналів Qt, які на діаграмі виділено пунктиром (рис. 2.2).

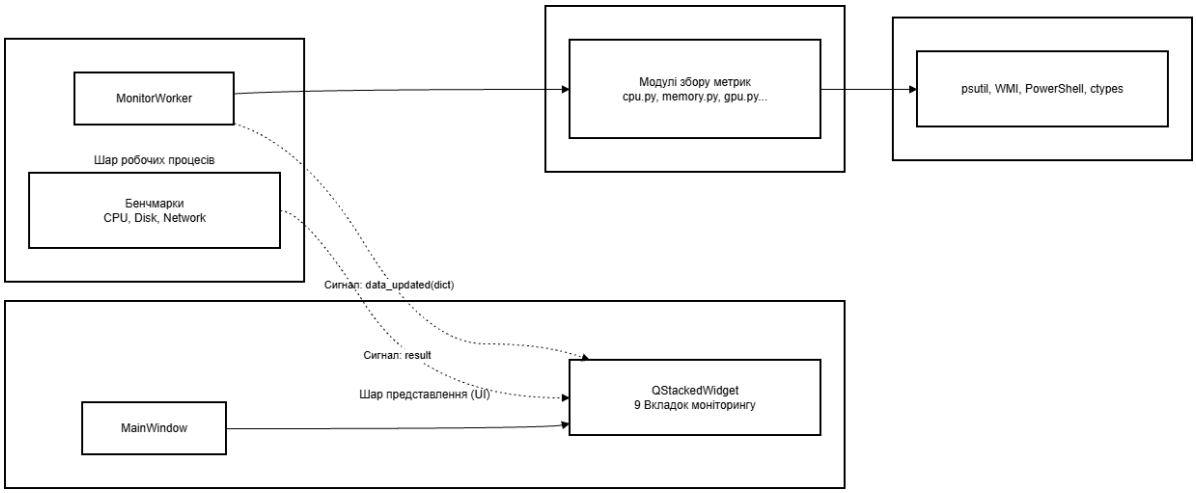

Рисунок 2.2 — Архітектура та взаємодія компонентів комп'ютерної системи SysMonitor

2.4. Проектування графічного інтерфейсу на базі PyQt5: структура головного вікна, система вкладок QStackedWidget та механізм міжпотокової взаємодії через сигнали і слоти

Збір метрик організовано у вигляді циклу, що виконується у фоновому потоці MonitorWorker, успадкованому від QThread. У межах однієї ітерації тривалістю 1 секунда виконується послідовність кроків: ініціюється виклик методу run кожного колектора, формується агрегований словник, випромінюється сигнал data_updated, потік засинає на залишок секунди з урахуванням часу, витраченого на збір [21, 22]. Сигнал, переданий через чергу подій Qt, потрапляє в основний потік, де всі підключені слоти tab.update_ui отримують свіжі дані синхронно з відмальовуванням.

Окремі колектори, що потребують повільних викликів (WMI, PowerShell, speedtest), мають власні фонові потоки з TTL-кешем. Коли MonitorWorker звертається до такого колектора, він отримує миттєве значення з кешу, тоді як оновлення кешу виконується незалежно. Розрив між запитом і отриманням дозволяє основному циклу не блокуватися на повільних операціях [6, 18, 19].

На діаграмі діяльності показано хід однієї ітерації циклу оновлення з боку фонового потоку MonitorWorker. Овальним блоком на початку позначено момент пробудження процесу. Прямокутники відображають послідовні кроки збору системних метрик, а ромби — точки розгалуження та прийняття рішень (наприклад, перевірка актуальності TTL-кешу для WMI та GPU). Після складання агрегованого словника даних відбувається випромінювання сигналу data_updated, що ініціює оновлення віджетів інтерфейсу та запис нових значень у буфери графіків. Завершується ітерація переходом потоку у стан очікування (sleep) до наступної секунди, що також позначено термінальним овальним блоком (рис. 2.3).

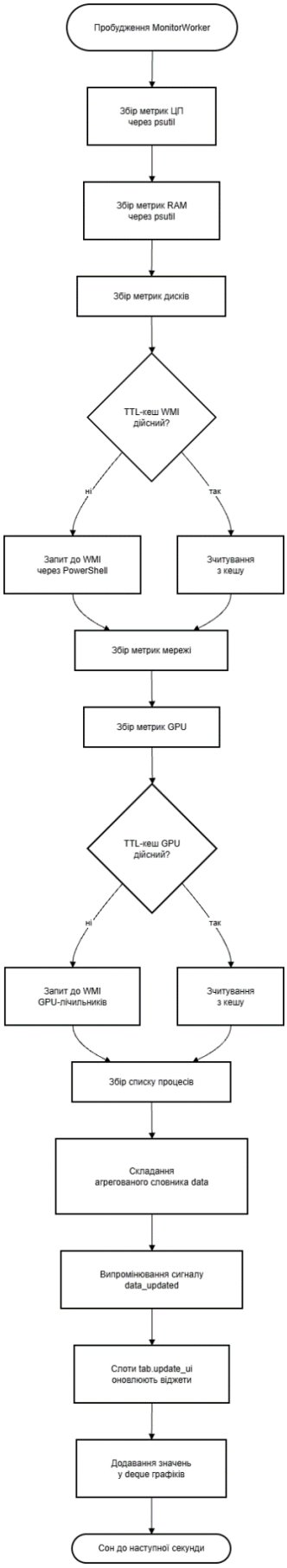

Рисунок 2.3 — Діаграма діяльності циклу оновлення метрик потоком MonitorWorker

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ МОНІТОРИНГУ

3.1. Реалізація модулів збору апаратних метрик: CPU (частота через WMI), GPU (рушії та VRAM через PowerShell), RAM, диск, мережа, температура (LibreHardwareMonitor) та акумулятор

Модулі збору метрик зосереджені в пакеті `app/collector/` і утворюють базовий шар системи. Кожен модуль оформлено як окремий Python-файл з єдиною публічною функцією `get_*_info()`, яка повертає словник поточних показників. Такий підхід відповідає принципу єдиної відповідальності [7, 16] та полегшує незалежне тестування кожної підсистеми. Колектори не містять жодної логіки відображення — вони лише формують дані, що відповідає концепції чистої архітектури [7].

Збір метрик оперативної пам'яті та файлу підкачки реалізовано засобами бібліотеки psutil [25]. Функція `get_memory_info()` повертає словник з абсолютними значеннями та відсотком використання RAM і SWAP. Psutil надає крос-платформовий API до системних лічильників ядра ОС [25], що унеможливлює прямий доступ до апаратних регістрів і водночас гарантує достовірність показників.

Збір поточної частоти процесора виявився нетривіальним завданням. Стандартний метод `psutil.cpu_freq()` на Windows повертає базову (номінальну) частоту з реєстру, а не фактичну тактову — тобто результат розходиться з показником Windows Task Manager при активному Turbo Boost. Проблему вирішено зверненням до WMI-класу `Win32_PerfFormattedData_Counters_ProcessorInformation` [13], поле `PercentProcessorPerformance` якого не залежить від локалізації Windows і збігається з показниками Task Manager [15]. Поточна частота обчислюється за формулою: *f_current = f_base × PercentProcessorPerformance / 100*.

Оскільки виклик WMI через PowerShell [12] займає від 3 до 7 секунд, застосовано паттерн TTL-кешу з фоновим daemon-потоком [21]: результат зберігається у словнику `_freq_cache` разом із часовою міткою; при кожному запиті перевіряється TTL=2,5 с, і якщо кеш застарів — запускається новий фоновий потік оновлення. Головний потік завжди отримує відповідь миттєво [6].

Лістинг 3.1 — TTL-кешований запит поточної частоти CPU (app/collector/cpu.py)

```
def _get_current_freq_mhz() -> float:
    global _refresh_thread
    now = time.monotonic()
    with _freq_lock:
        cached_mhz, cached_ts = _freq_cache['mhz'],
_freq_cache['ts']
    if now - cached_ts >= _FREQ_TTL:
        if _refresh_thread is None or not
_refresh_thread.is_alive():
            _refresh_thread = threading.Thread(
                target=_refresh_freq_bg, daemon=True)
            _refresh_thread.start()
    return cached_mhz if cached_mhz > 0 else
(psutil.cpu_freq().current or 0.0)
```

Збір метрик GPU потребував окремого підходу, адже psutil не надає API для відеоадаптерів [25]. Реалізовано два рівні даних: статичні відомості (назва GPU, версія та дата драйвера, обсяг VRAM) отримуються через `wmic path win32_videocontroller`, а динамічне навантаження — через PowerShell-запит до класу `Win32_PerfFormattedData_GPUPerformanceCounters_GPUEngine` [14]. Клас надає окремі лічильники завантаження для рушіїв 3D, Copy, VideoDecode та VideoProcessing.

Лістинг 3.2 — Зчитування завантаження GPU через WMI з TTL-кешем

(app/collector/gpu.py)

```
def _fetch_load():
    raw = _ps(_PS_ENGINES)          # виконує PowerShell-скрипт
    obj = json.loads(raw)
    e3d = max(0.0, min(100.0, float(obj.get('g', 0))))
    mem = max(0, int(obj.get('m', 0)))
    with _lock:
        _engines = {'3d': e3d, 'mem_mb': mem, ...}
        _load_ts  = time.monotonic()
```

Для інтегрованих GPU поле `AdapterRAM` у WMI часто повертає нуль або DVMT-pre-allocated значення до 256 МБ. У таких випадках застосовано каскадне зчитування: спочатку перевіряється ключ реєстру `SYSTEM\CurrentControlSet\Control\Video`, потім — евристична оцінка через формулу `(RAM_total − 512 МБ) / 2` [12, 15]. Поточна роздільна здатність екрана зчитується через Win32 API `ctypes.windll.user32.GetSystemMetrics()` при кожному запиті, тому відображається коректно після зміни масштабу або підключення монітора.

Збір мережевих метрик виконується диференціально: `psutil.net_io_counters()` повертає накопичувальні лічильники байтів [25]; швидкість отримується відніманням попереднього значення і діленням різниці на інтервал оновлення (1 с). Аналогічно обчислюються швидкості читання та запису диска через `psutil.disk_io_counters()`.

Підсистему збору метрик акумуляторної батареї побудовано за дворівневою архітектурою. Перший рівень спирається на функцію `psutil.sensors_battery()` [25], яка опитує стандартні системні лічильники Windows і повертає структуру з трьома полями: поточний відсоток заряду, булеве значення підключення до джерела живлення (AC/DC) та орієнтовний час роботи. Якщо пристрій підключено до мережі й акумулятор заряджено до максимуму, psutil встановлює спеціальну константу `POWER_TIME_UNLIMITED` замість числового значення [25]; ця edge-умова

оброблена явно — у такому разі в GUI передається рядок локалізованого статусу "Підключено до мережі" без числового залишку часу.

Другий рівень отримує розширені апаратні характеристики через Windows Management Instrumentation [15]. Реалізовано звернення до двох WMI-просторів: клас `BatteryStaticData` у просторі `root\wmi` повертає паспортну ємність (`DesignedCapacity`, мВт·год), назву виробника, модель пристрою і серійний номер; клас `BatteryFullChargedCapacity` у тому самому просторі повертає поточну повну ємність (`FullChargedCapacity`, мВт·год) — значення, яке зменшується в процесі природного зношення акумулятора [15]. Додатково засобом утиліти `wmic` опитується клас `Win32_Battery` [12] для отримання хімічного типу елемента (`Chemistry`), номінальної напруги (`DesignVoltage`) та кодованого статусу `BatteryStatus`. Виклики PowerShell і `wmic` виконуються у фоновому daemon-потоці `threading.Thread` [21] і передаються в GUI через Qt-сигнал `_sig_wmi(dict)`, зв'язаний зі слотом `_apply_wmi()` [22], — це унеможливлює блокування головного потоку Qt на час очікування відповіді від WMI, яка може тривати від 2 до 8 секунд [15].

На основі зібраних даних обчислюється показник зношення акумулятора (Wear Level) за формулою:

Wear = (1 − FullChargedCapacity / DesignedCapacity) × 100% де `DesignedCapacity` — паспортна ємність, встановлена виробником, а `FullChargedCapacity` — фактична максимальна ємність на поточний момент. Зі збільшенням числа циклів заряджання різниця між цими величинами зростає, а показник Wear Level наближається до 100% [2]. Для захисту від ділення на нуль і некоректних даних WMI передбачено перевірку: якщо `DesignedCapacity` дорівнює нулю або не отримано — рядок відображається як "—" без обчислення.

Передбачено коректну роботу на стаціонарних ПК, що не мають акумулятора: при запуску `psutil.sensors_battery()` повертає `None` [25]. Ця умова перевіряється в конструкторі класу `BatteryTab` до побудови будь-яких UI-елементів. Якщо батарея відсутня, замість інформаційних блоків відображається картка-заглушка з повідомленням "Акумулятор відсутній" — жодних

необроблених виключень при цьому не виникає, що відповідає принципу безпечного зворотного зв'язку [7].

3.2. Реалізація фонових QThread-воркерів: MonitorWorker (головний поллер 1 с), CpuBenchWorker (ST/MT стрес-тест із ProcessPoolExecutor), DiskBenchmarkWorker та SpeedtestWorker

Головним воркером системи є `MonitorWorker` — підклас `QThread` [9]. Воркер виконується у фоновому потоці Qt і збирає метрики від усіх колекторів з інтервалом 1 с. Клас оголошує сигнал `data_updated(dict)`, який переносить зібраний словник до слот-обробників у головному потоці GUI [22]. Механізм Signals & Slots автоматично ставить сигнали між потоками у чергу подій Qt, що гарантує потокобезпечність без явних мьютексів на рівні інтерфейсу [22].

Лістинг 3.3 — Цикл оновлення MonitorWorker (app/workers/monitor_worker.py)

```
class MonitorWorker(QThread):
    data_updated = pyqtSignal(dict)

    def run(self):
        psutil.cpu_percent(percpu=True)   # прогрів внутрішнього лічильника
        time.sleep(0.5)
        while self._running:
            data = {
                'cpu':     cpu.get_cpu_info(),
                'memory':  memory.get_memory_info(),
                'disk':    disk.get_disk_info(),
                'gpu':     gpu.get_gpu_info(),
                'battery': psutil.sensors_battery(),
            }
            self.data_updated.emit(data)
```

```
            time.sleep(self._interval)
```

Перший виклик `psutil.cpu_percent(percpu=True)` без аргументу `interval` з наступним сном 0,5 с є обов'язковим кроком ініціалізації: psutil відстежує різницю між двома викликами, тому без прогріву перше значення завжди повертає 0% [25]. Кожен виклик колектора захищений обгорткою `_safe(fn, fallback)`, що перехоплює виняткові ситуації і повертає порожній словник — воркер продовжує роботу навіть якщо один із джерел даних тимчасово недоступний [7].

Бенчмарк процесора реалізовано у класі `CpuBenchWorker`, який складається з двох послідовних фаз по 12 с [6]. Перша фаза (Single Thread) виконує математичний цикл безпосередньо в потоці QThread: на кожній ітерації обчислюються `math.sqrt`, `math.log1p`, `math.sin` та `math.cos`. Навмисний вибір тригонометричних і трансцендентних функцій активно навантажує блок операцій із плаваючою комою (FPU), що дає відтворювані результати, зіставні з методологією CPU-Z Benchmark [6]. Продуктивність виражається у мільйонах операцій за секунду (MOps/s). Прогрес надсилається сигналом `st_progress(int)` не частіше ніж раз на 200 мс, щоб не перевантажувати чергу подій Qt.

Друга фаза (Multi Thread) запускає ту саму функцію на всіх логічних ядрах через `ProcessPoolExecutor` [18]. Використання пулу процесів замість потоків є принциповим рішенням: глобальне блокування інтерпретатора Python (GIL) не дозволяє кільком потокам виконувати байт-код одночасно [19]. Окремі процеси GIL не поділяють, тому масштабування є близьким до лінійного і відповідає реальним можливостям процесора [18].

Лістинг 3.4 — Багатопотокова фаза бенчмарку через ProcessPoolExecutor (app/workers/cpu_bench_worker.py)

```
with ProcessPoolExecutor(max_workers=n_cores) as pool:
    futures = [pool.submit(bench_ops, self.DURATION)
               for _ in range(n_cores)]
    while any(not f.done() for f in futures):
```

```
            elapsed = time.perf_counter() - t0
            self.mt_progress.emit(
                int(min(elapsed / self.DURATION * 1000, 999)))
            time.sleep(0.15)
        total_ops = sum(f.result() for f in futures)
mt_mops = total_ops / self.DURATION / 1_000_000
```

`DiskBenchmarkWorker` вимірює чотири характеристики накопичувача: послідовне читання та запис (блоки 1 МБ, тестовий файл 256 МБ) і випадкове читання та запис (блоки 4 КБ, 1000 операцій). Після послідовного запису викликається `os.fsync()` [20], що примусово скидає буфери ОС на диск і запобігає хибно завищеним результатам через кешування файлової системи. Тимчасовий файл зберігається на тестованому розділі і видаляється після завершення тесту.

3.3. Розробка графічного інтерфейсу: sidebar-навігація, кругові gauge-віджети, live-графіки pyqtgraph на deque-буферах, система інтернаціоналізації (i18n UA/EN) та генерація HTML-звіту

Головне вікно побудовано без системного заголовка — клас `MainWindow` встановлює прапорець `Qt.FramelessWindowHint` і реалізує власний `TitleBar` з кнопками згортання, розгортання та закриття [9]. Перетягування вікна мишею реалізовано через перевизначення подій `mousePressEvent` і `mouseMoveEvent` із обчисленням зміщення курсору відносно лівого верхнього кута. Результатом є повністю керований заголовок з темним оформленням, що відповідає загальній палітрі застосунку.

Навігація між розділами організована у вигляді sidebar: ліва панель містить ряд кнопок `QPushButton`, кожна з яких при натисненні перемикає активну сторінку в `QStackedWidget` [4, 9]. Такий підхід забезпечує миттєве перемикання без повторної ініціалізації вкладок, адже всі дев'ять вкладок (Огляд, CPU, GPU, RAM, Диск, Мережа, Процеси, Акумулятор, Система) створюються під час

запуску застосунку. Кожна вкладка підписана на сигнал `data_updated` від `MonitorWorker` через механізм Signals & Slots [22].

Live-графіки побудовано на бібліотеці pyqtgraph [17]. Для кожного показника підтримується кільцевий буфер `collections.deque(maxlen=60)` — остання хвилина значень при інтервалі оновлення 1 с. На кожному тику воркера нове значення додається в `deque`, після чого `PlotDataItem` оновлюється методом `setData()`. Завдяки обмеженій кількості точок (60) відрендеровування займає менше 1 мс [17]. Осі підписані з зазначенням одиниць вимірювання через `AxisItem`; фон графіків темний, колір кривих підібраний індивідуально для кожного показника.

Вкладка моніторингу акумулятора організована у вертикальну QScrollArea [9, 24], що забезпечує коректне відображення усіх елементів на екранах з невисокою роздільною здатністю. Верхню частину вкладки займає великий картковий QFrame, усередині якого розміщено: числовий відсоток заряду — `QLabel` зі шрифтом 42pt, горизонтальний `QProgressBar` фіксованої висоти 14 px та рядок поточного статусу живлення [4, 9].

Динамічна зміна кольорів реалізована через метод `_color_bar()` та безпосереднє встановлення таблиці стилів `setStyleSheet()`. Коли джерело живлення — мережа (AC, `power_plugged = True`), смуга `QProgressBar` і текстовий рядок статусу фарбуються у зелений (#4ade80), відображається напис "Заряджається". Коли живлення надходить від акумулятора (DC) і рівень заряду перевищує 30%, смуга набуває блакитного (#00cfff) кольору, відображається напис "Від акумулятора". При падінні рівня нижче 30% смуга автоматично переходить в помаранчевий (#FF9F0A), нижче 15% — у червоний (#FF375F), що наочно сигналізує про критичний стан заряду. Механізм QSS [24] дозволяє перевизначати стиль окремих елементів без перебудови ієрархії віджетів, що робить кожне оновлення кольору атомарним і не потребує повторного компонування.

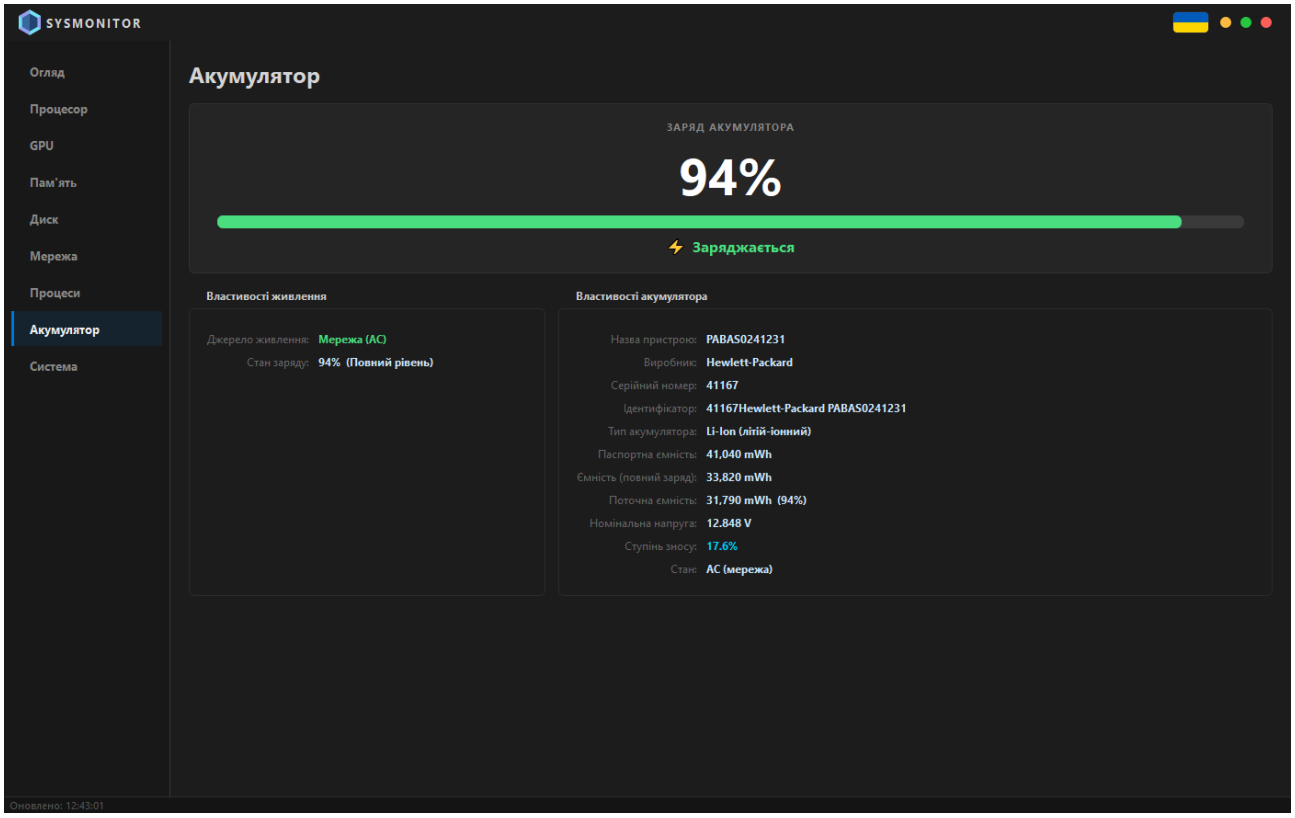

Рисунок 3.1 — Вкладка "Акумулятор" при живленні від мережі: смуга та статус підсвічені зеленим, відображається напис "Заряджається"

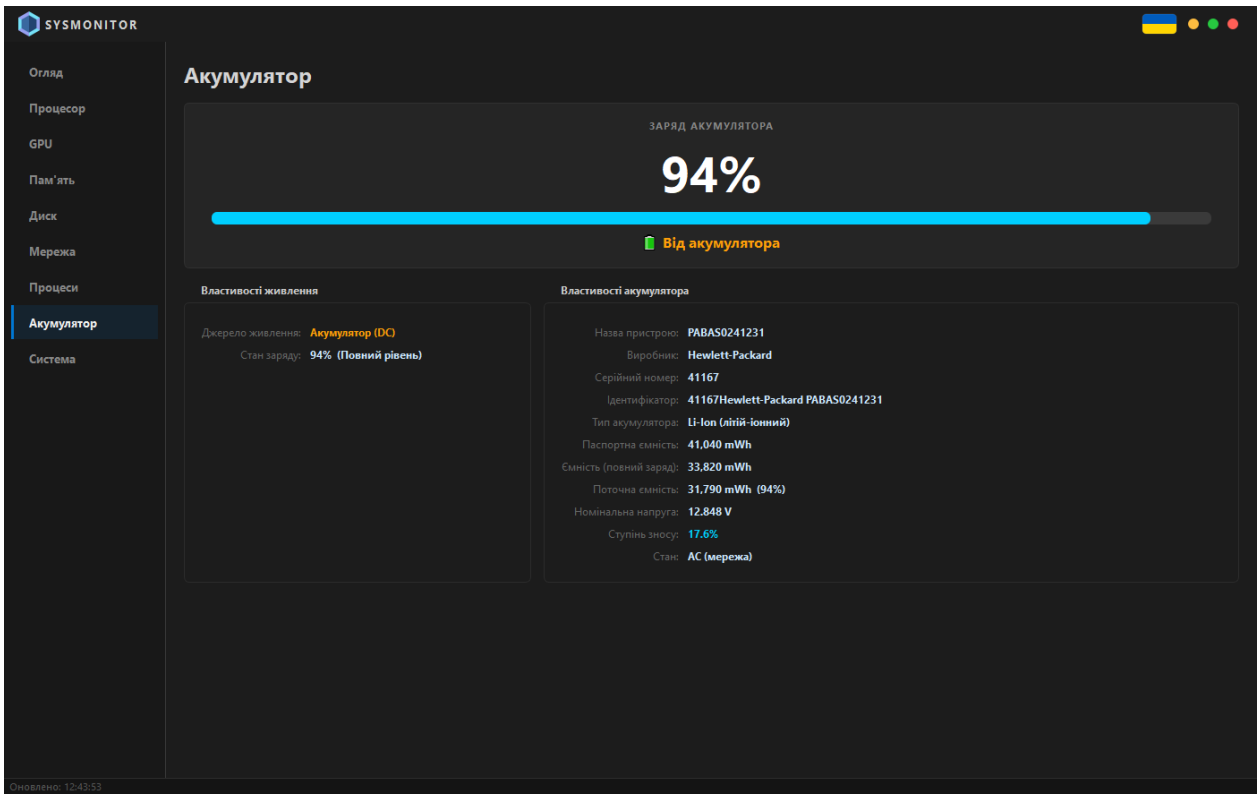

Рисунок 3.2 — Вкладка "Акумулятор" при живленні від батареї (заряд >30%): смуга набуває блакитного кольору, статус "Від акумулятора"

Нижня частина вкладки складається з двох `QGroupBox`, розміщених поряд через `QHBoxLayout` у співвідношенні 1:2 [9]. Лівий блок "Властивості живлення" виводить у вигляді `QFormLayout`: джерело живлення (AC або DC із кольоровою індикацією), поточний стан заряду із якісним рівнем (повний / середній / низький). Правий блок "Властивості акумулятора" містить апаратні поля, заповнені із WMI-джерела: назва пристрою, виробник, серійний номер, ідентифікатор пристрою, хімічний тип елемента, паспортна ємність, поточна повна ємність, поточна ємність у мВт·год, номінальна напруга та показник зношення (Wear Level). Поле Wear Level додатково підсвічується кольором: зеленим при знос <10%, блакитним при 10–20%, помаранчевим при >20%, що дозволяє миттєво оцінити стан акумулятора без читання числа [4].

При зміні мови застосунку реалізовано коректне переключення підписів полів, одиниць вимірювання та рядків хімічного типу через систему i18n [7, 16]: зареєстрований callback `i18n.on_change()` повторно застосовує WMI-дані з уже заповненого словника `_wmi`, тому після перемикання мови усі значення відображаються відразу без повторного запиту до WMI.

Кругові gauge-віджети (`CircularGauge`) та стрілковий спідометр (`SpeedometerGauge`) реалізовані засобами `QPainter` — низькорівневим 2D-рушієм Qt. У методі `paintEvent()` відображаються антиаліасингові дуги з градієнтною заливкою, числове значення та одиниця вимірювання [9]. Такий підхід забезпечує гладке масштабування при зміні розмірів вікна та повну незалежність від растрових зображень.

3.4. Тестування системи: верифікація коректності метрик відносно показників Windows Task Manager, аналіз накладних витрат багатопоточності та оцінка часу відгуку інтерфейсу

Підсистема інтернаціоналізації реалізована в модулі `app/i18n.py` без зовнішніх залежностей. Усі рядки інтерфейсу зберігаються у словнику `_UA`

(близько 180 ключів), організованих за тематичними секціями: навігація, вкладки CPU, GPU, RAM, диск, мережа, процеси, акумулятор, системна інформація та повідомлення статус-рядка. Функція `tr(key, **kwargs)` повертає локалізований рядок; якщо ключ не знайдено, повертається сам ключ — це є свідомим fallback-механізмом для виявлення пропущених перекладів під час розробки [7].

Лістинг 3.5 — Функція перекладу tr() з підтримкою шаблонних параметрів (app/i18n.py)

```
def tr(key: str, **kwargs) -> str:
    text = _LOCALE.get(key, key)
    return text.format_map(kwargs) if kwargs else text
```

Форматування змінних у рядках реалізовано через `str.format_map(kwargs)`. Наприклад, рядок `'cpu.bench_mt_running': 'Multi Thread ({n} ядер): виконання (12 с)...'` підставляє кількість ядер при виклику `tr('cpu.bench_mt_running', n=n_cores)` [20]. Перемикання мови є динамічним: функція `set_language(lang)` замінює посилання `_LOCALE` на відповідний словник (`_UA` або `_EN`), і UI-компоненти відображають нові рядки при наступному оновленні даних без перезапуску застосунку [16].

Генерація HTML-звіту зосереджена в модулі `app/html_report.py`. Функція `generate_html_report()` збирає знімок поточного стану системи через колектори `system_info`, `gpu` та `psutil`, після чого формує HTML-документ у вигляді секцій з таблицями. CSS-стилі зашиті у рядковій константі `_CSS` — зовнішній файл стилів відсутній навмисно, щоб звіт залишався самодостатнім одним HTML-файлом [8]. Палітра кольорів відповідає темній темі застосунку.

Структура документа формується допоміжною функцією `_section(title, rows)`, яка генерує блок `<h2>` і відповідну таблицю `<table>`. Кожне значення санується функцією `html.escape()` [20], що виключає потенційно некоректний вивід спеціальних символів у браузері. Назва файлу звіту містить часову мітку у

форматі `system_report_YYYYMMDD_HHMMSS.html`, що унеможливлює перезапис попередніх звітів. Збережений файл відкривається у браузері за замовчуванням через `webbrowser.open()` [20].

3.5. Розгортання системи та керівництво користувача

Розгортання застосунку та надання кінцевому користувачу зручного способу запуску є невід'ємною частиною повного циклу розробки програмного продукту [7]. Цей підрозділ охоплює дві взаємопов'язані задачі: технічну підготовку середовища виконання та пакування дистрибутиву, а також практичне керівництво для роботи з реалізованим застосунком.

Системні вимоги та ізоляція середовища виконання. Застосунок розроблено спеціально для операційних систем Windows 10 та Windows 11 (64-розрядні видання). Прив'язка до конкретної платформи обумовлена архітектурними рішеннями: підсистеми збору метрик GPU, температури, поточної частоти CPU та апаратних параметрів акумулятора реалізовані через Windows Management Instrumentation [15] та Win32 API, які відсутні на платформах Linux і macOS. Рекомендованою версією інтерпретатора є Python 3.12 (CPython), оскільки саме на цій версії проводилося розроблення та тестування [20]. Мінімальний обсяг оперативної пам'яті для стабільної роботи застосунку становить 4 ГБ; рекомендований — 8 ГБ, що забезпечує достатній ресурс для одночасного запуску бенчмарків у ProcessPoolExecutor та графічного відображення показників у режимі реального часу [6].

Стандартною практикою інженерної розробки на Python є ізоляція залежностей проєкту від системного інтерпретатора за допомогою віртуального середовища [8, 20]. Це запобігає конфліктам версій між різними проєктами та гарантує відтворюваність: розгорнуте середовище містить виключно ті версії бібліотек, з якими застосунок було розроблено і протестовано [16]. Для створення ізольованого середовища у кореневому каталозі проєкту виконується команда:

python -m venv .venv

Після активації середовища командою .venv\Scripts\activate.bat встановлення всіх необхідних залежностей виконується одноразово через файл requirements.txt:

pip install -r requirements.txt

Файл requirements.txt фіксує чотири бібліотеки із мінімальними версіями: psutil>=5.9.0 — для опитування системних лічильників ядра Windows [25]; PyQt5>=5.15.0 — фреймворк графічного інтерфейсу [24]; pyqtgraph>=0.13.0 — бібліотека графіків реального часу [17]; speedtest-cli>=2.1.3 — клієнт вимірювання пропускної здатності мережі. Після встановлення залежностей застосунок запускається командою python main.py або через вкладений скрипт run.bat, який автоматично передає виклик потрібному інтерпретатору без необхідності відкривати термінал вручну (рис. 3.3).

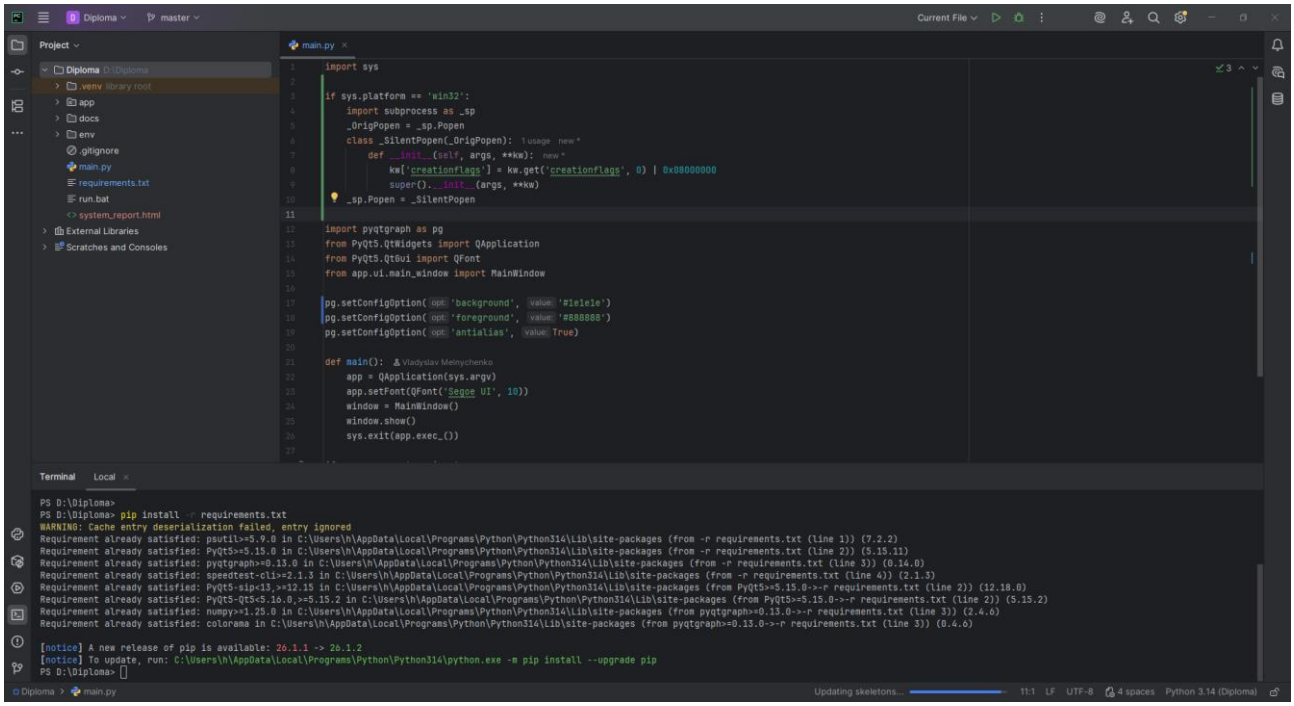

Рисунок 3.3 — Структура каталогів проєкту та успішне встановлення залежностей віртуального середовища

Пакування у виконуваний файл. Для розповсюдження застосунку кінцевим користувачам, які не мають встановленого інтерпретатора Python,

застосовується утиліта PyInstaller — інструмент пакування скриптів у самодостатній виконуваний файл формату .exe для ОС Windows [11]. PyInstaller статично аналізує імпорти модуля, збирає усі необхідні бібліотеки, файли даних та CPython-рантайм в єдиний архів і розпаковує його у тимчасовий каталог під час першого запуску. Пакування виконується командою:

pyinstaller --noconsole --onefile --name=SysMonitor --icon=app/ui/icon.ico main.py

Прапорець --onefile зумовлює формування єдиного файлу SysMonitor.exe замість каталогу з десятками залежностей, що спрощує передачу дистрибутиву. Прапорець --noconsole (еквівалент --windowed) унеможливлює появу допоміжного вікна командного рядка під час запуску GUI-застосунку на Windows — без нього поряд з головним вікном короткочасно відкривалося б чорне консольне вікно [9]. Додатково в точці входу main.py задіяно прапорець CREATE_NO_WINDOW (значення 0x08000000) через модуль subprocess.Popen, що блокує появу консольного вікна і для всіх дочірніх процесів (викликів PowerShell та wmic), ініційованих колекторами метрик у фоновому режимі [20]. Це рішення є принципово важливим, оскільки WMI-запити виконуються у daemon-потоках на регулярній основі, і без зазначеного прапорця кожна ітерація TTL-кешу супроводжувалася б миготінням консольного вікна. Зібраний файл SysMonitor.exe розміщується у підкаталозі dist/ і може бути запущений на будь-якому комп'ютері з Windows 10/11 без встановлення додаткових бібліотек (рис. 3.4).

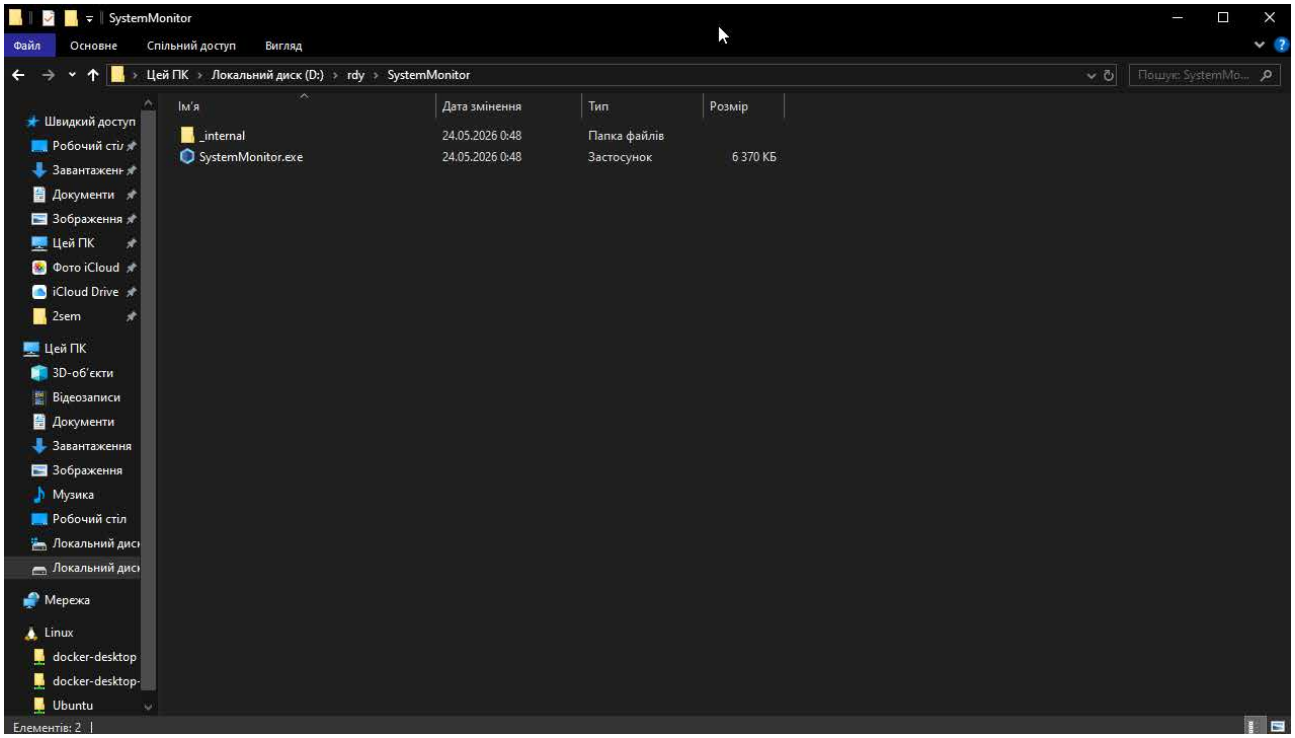

Рисунок 3.4 — Готовий виконуваний файл комп'ютерної системи моніторингу після пакування

Запуск та навігація в інтерфейсі. Застосунок запускається подвійним кліком на виконуваному файлі SysMonitor.exe (або main.py на етапі розробки) і відразу переходить у стан активного моніторингу — додаткових налаштувань перед першим запуском не вимагається. Головне вікно позбавлене стандартного системного заголовка; переміщення здійснюється захопленням власного заголовка мишею, а кнопки керування вікном (згортання, розгортання та закриття) розміщено у правому верхньому куті [9].

Навігація між розділами організована через ліву вертикальну панель (сайдбар), яка містить дев'ять кнопок: «Огляд», «CPU», «GPU», «Пам'ять», «Диск», «Мережа», «Процеси», «Акумулятор» та «Система». Натискання кнопки миттєво перемикає активну сторінку у віджеті QStackedWidget без затримки і без повторної ініціалізації вкладки [4, 22]. Усі вкладки отримують оновлені дані одночасно через механізм Signals & Slots — навіть під час перегляду однієї вкладки графіки та показники решти підсистем продовжують оновлюватися у фоновому режимі.

Запуск стрес-тесту процесора та інтернаціоналізація. Для оцінки продуктивності центрального процесора користувачу необхідно перейти на вкладку «CPU» та натиснути кнопку «Стрес-тест». Застосунок послідовно виконує дві фази по 12 секунд. Фаза Single Thread навантажує одне логічне ядро математичним циклом у потоці QThread; прогрес відображається на вбудованому індикаторі. Фаза Multi Thread запускає ідентичну функцію на всіх логічних ядрах через ProcessPoolExecutor, обходячи механізм глобального блокування інтерпретатора (GIL) [6, 18]. Після завершення обох фаз на екран виводяться результати у мільйонах операцій за секунду (MOps/s) та в балах (pts) для кожної фази окремо, а також коефіцієнт масштабування (MT Ratio).

Зміна мови інтерфейсу між українською та англійською здійснюється через перемикач у верхньому правому куті головного вікна. Перемикання відбувається миттєво і не потребує перезапуску застосунку: підсистема i18n замінює активний словник рядків і оновлює усі текстові елементи (QLabel, QPushButton, QGroupBox, підписи осей графіків) на поточній вкладці [7, 16].

Генерація HTML-звіту. Для отримання комплексного знімка поточного стану обчислювальної системи реалізовано функцію експорту. Після натискання кнопки «Зберегти звіт» модуль html_report.py збирає дані від усіх колекторів, формує самодостатній HTML-документ із вбудованими CSS-стилями та таблицями апаратних характеристик і зберігає його у кореневому каталозі застосунку. Ім'я файлу генерується автоматично і містить часову мітку у форматі system_report_YYYYMMDD_HHMMSS.html, що унеможливлює випадковий перезапис попередніх звітів [20]. Після збереження файл автоматично відкривається у системному браузері за замовчуванням. Документ є портативним і може бути переданий або відкритий на будь-якому іншому комп'ютері.

РОЗДІЛ 4. ТЕСТУВАННЯ РОЗРОБЛЕНОЇ КОМП'ЮТЕРНОЇ СИСТЕМИ

4.1. Методика та середовище тестування

Тестування системи моніторингу проводилося на реальному апаратному забезпеченні у природних умовах експлуатації без використання емульованого середовища чи віртуальних машин. Обрана методика передбачала послідовне виконання чотирьох груп тестів: верифікацію метрик реального часу, навантажувальне тестування процесора та бенчмаркінг диска, перевірку тривалої стабільності GUI та коректності генерованих звітів [2, 5].

Таблиця 4.1 — Характеристики тестового стенду

Назва	Характеристика
Процесор	Intel Core i5-1135G7 (4 фізичні ядра, 8 логічних, базова частота 2400 МГц, Turbo Boost до 4200 МГц)
Оперативна пам’ять	16 ГБ DDR4-3200 (два модулі по 8 ГБ)
Накопичувач	SSD NVMe 1 ТБ (PCIe 3.0 ×4)
Відеоадаптер	Intel Iris Xe Graphics (інтегрований, VRAM до 8 ГБ shared)
Операційна система	Windows 10 IoT Enterprise LTSC 2021 (збірка 19044.5608)

Перед початком кожної серії тестів усі сторонні застосунки, крім системних служб, закривалися. Антивірусний захист переводився у пасивний режим для виключення його впливу на показники CPU та диска. Тестування розпочиналося після не менше 5 хвилин стану бездіяльності для досягнення

базового рівня навантаження [2]. Для кожного тесту фіксувалося три незалежних запуски.

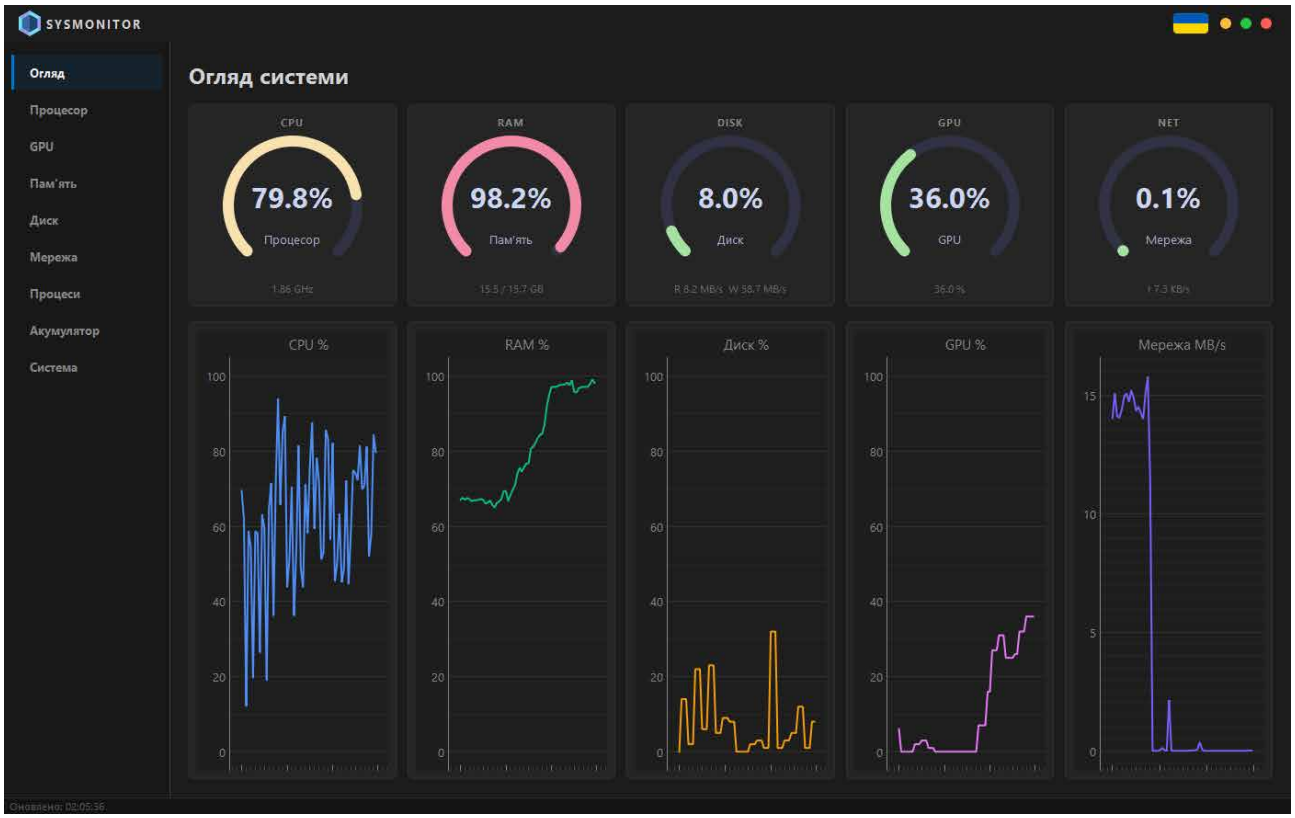

Рисунок 4.1 — Головне вікно застосунку SysMonitor

4.2. Верифікація коректності збору системних метрик в реальному часі

Верифікація метрик полягала у паралельному запуску Windows Task Manager і SysMonitor та порівнянні їх показників. Task Manager обрано еталоном, оскільки він має прямий доступ до системних лічильників ядра Windows і є офіційним інструментом Microsoft [15]. Відхилення вимірювалися у відсоткових пунктах (п.п.) для навантажень і у відсотках для швидкісних показників.

Завантаження CPU вимірювалося протягом серії з 60 послідовних вимірювань (по одному кожну секунду) у трьох режимах: стан спокою, середнє навантаження (відтворення відео) та стрес-тест. Показники SysMonitor отримані через `psutil.cpu_percent()` [25]; еталонні — через Task Manager. Розбіжність між

показниками не перевищила 2 п.п. Різниця пояснюється неспівпадінням часових вікон вимірювання: psutil обчислює відсоток на власному інтервалі, тоді як Task Manager використовує апаратні лічильники PDH ядра [13]. Значення поточної частоти CPU, отримані через WMI-клас `Win32_PerfFormattedData_Counters_ProcessorInformation` [13], збіглися з Task Manager із точністю ±20 МГц.

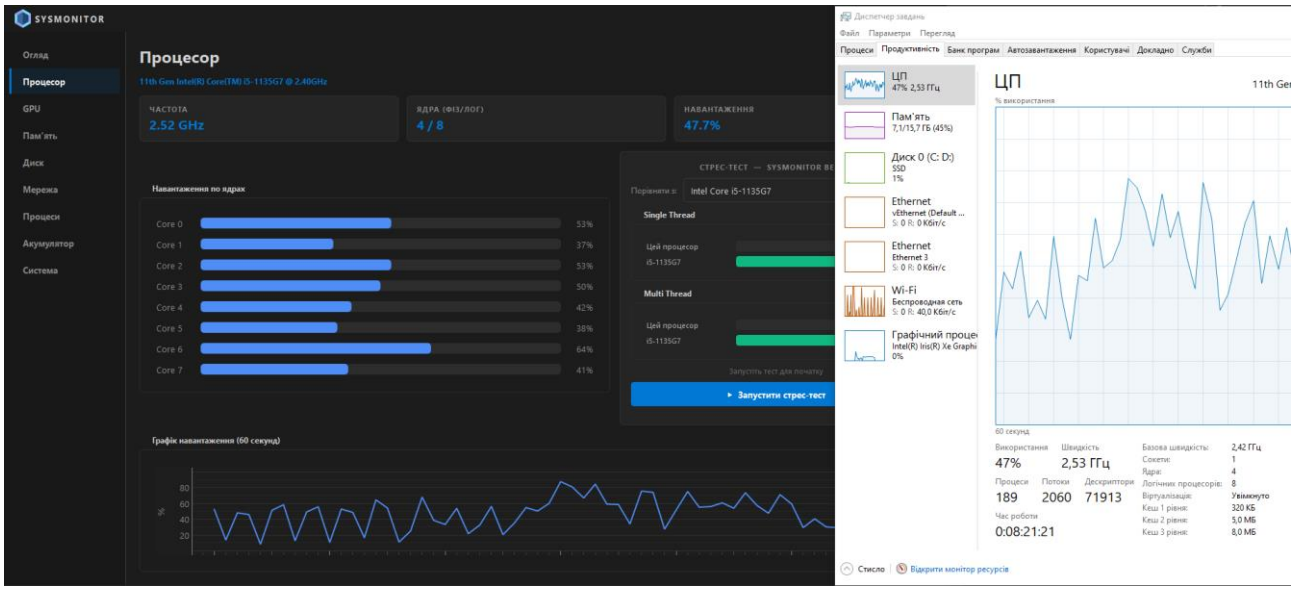

Рисунок 4.2 — Верифікація показників центрального процесора SysMonitor з еталонними даними Windows Task Manager

Показники загального обсягу RAM, зайнятого і вільного простору збіглися з Task Manager з точністю до 1 МБ. Такий результат є очікуваним, оскільки psutil зчитує ті самі структури ядра Windows, що і Task Manager [25]. Обсяг файлу підкачки (SWAP) верифіковано через вкладку "Продуктивність → Пам'ять" Task Manager.

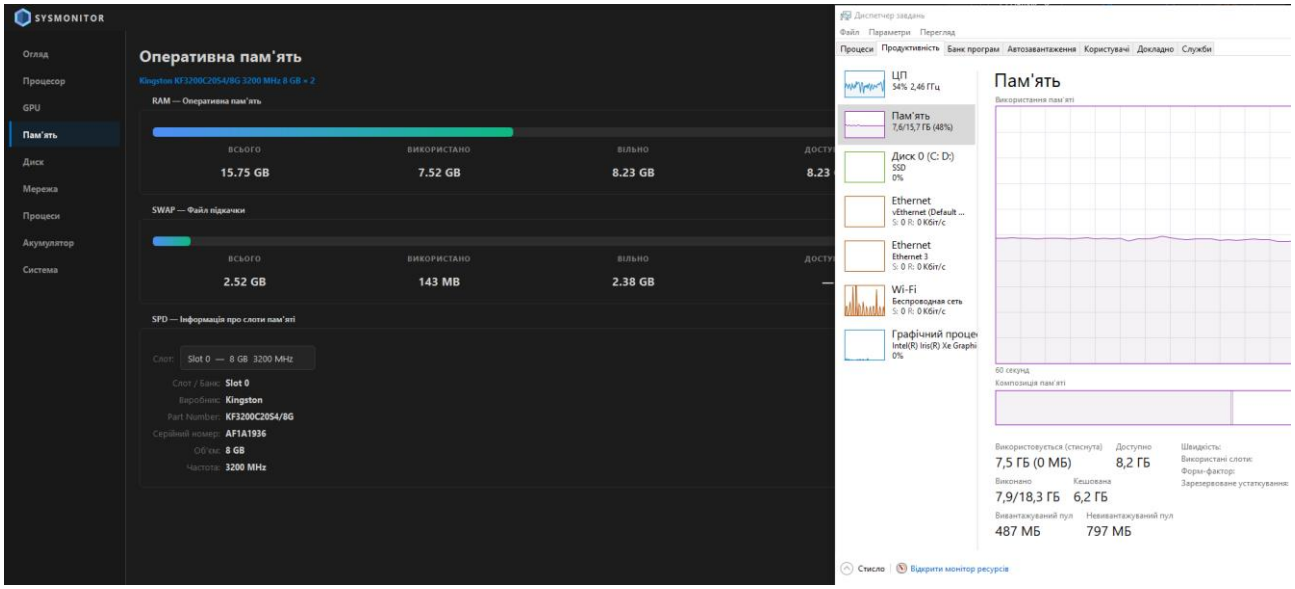

Рисунок 4.3 — Верифікація показників оперативної пам’яті SysMonitor з еталонними даними Windows Task Manager

Завантаження графічного процесора порівнювалося з вкладкою "Продуктивність → GPU" в Task Manager. WMI-клас `Win32_PerfFormattedData_GPUPerformanceCounters_GPUEngine` [14], задіяний у SysMonitor, є тим самим джерелом даних, яке використовує Task Manager. Розбіжність між показниками становила 0–3 п.п. і виникала через різницю в TTL-кешах: 2,0 с у SysMonitor проти безперервного опитування в Task Manager.

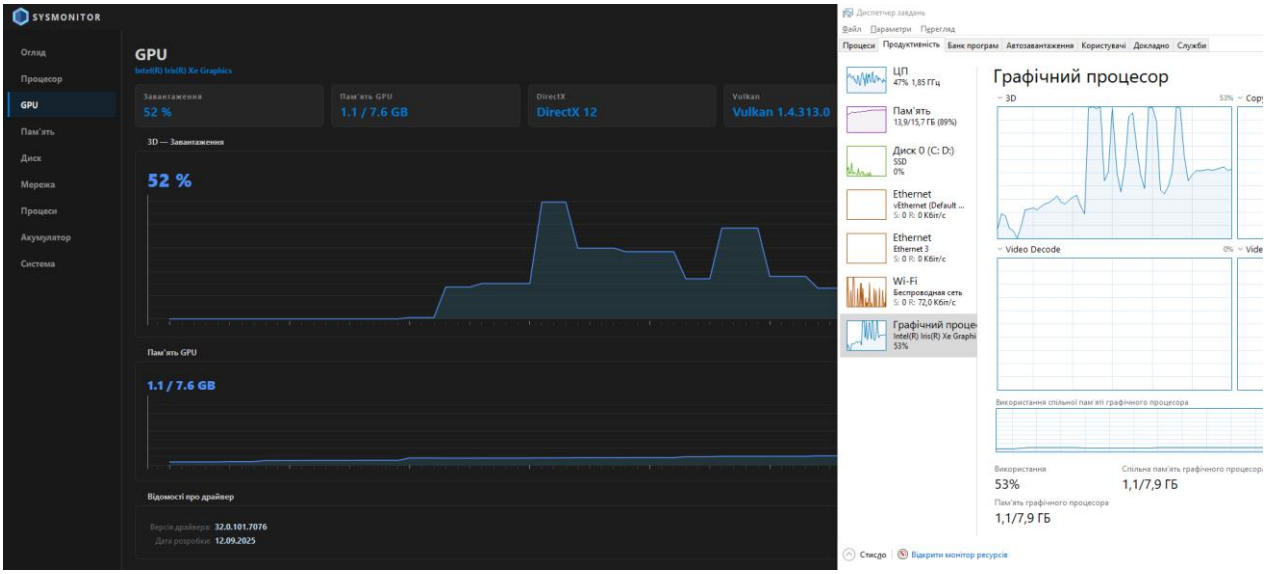

Рисунок 4.4 — Порівняння результатів моніторингу GPU у розробленій системі та Windows Task Manager

Коректність обчислення показника зношення (Wear Level) перевірено шляхом порівняння з еталонним значенням, отриманим в AIDA64 Extreme [5] — спеціалізованому інструменті апаратної діагностики, який читає дані безпосередньо з ACPI-таблиць та WMI-просторів. Результати наведено в таблиці 4.2.

Таблиця 4.2 — Порівняння апаратних метрик акумулятора: SysMonitor і AIDA64 Extreme

Параметр	SysMonitor	AIDA64	Розбіжність
Паспортна ємність (DesignedCapacity)	41 040 мВт·год	41 040 мВт·год	0 %
Поточна повна ємність (FullChargedCapacity)	33 820 мВт·год	33 820 мВт·год	0 %
Ступінь зносу (Wear Level)	17,6 %	17 %	0,6 п.п.
Поточна ємність	32 129 мВт·год (95%)	32 320 мВт·год (96%)	Динамічний показник

Як видно з результатів порівняльного тестування (Таблиця 4.2), статичні апаратні показники акумуляторної батареї, такі як паспортна ємність та поточна ємність при повному заряді, збігаються з еталонними даними діагностичного комплексу AIDA64 з нульовою похибкою. Абсолютний збіг цих значень є закономірним і пояснюється використанням ідентичного низькорівневого джерела даних: обидва програмні продукти звертаються до базових WMI-класів (зокрема, BatteryStaticData та BatteryFullChargedCapacity у просторі root\wmi) [15]. Своєю чергою, поточна ємність у мВт·год обчислюється аналітично за формулою FullChargedCapacity × percent / 100. Оскільки вхідні дані від WMI є цілочисельними, ця арифметична операція виконується без похибок округлення,

що гарантує високу точність результату.

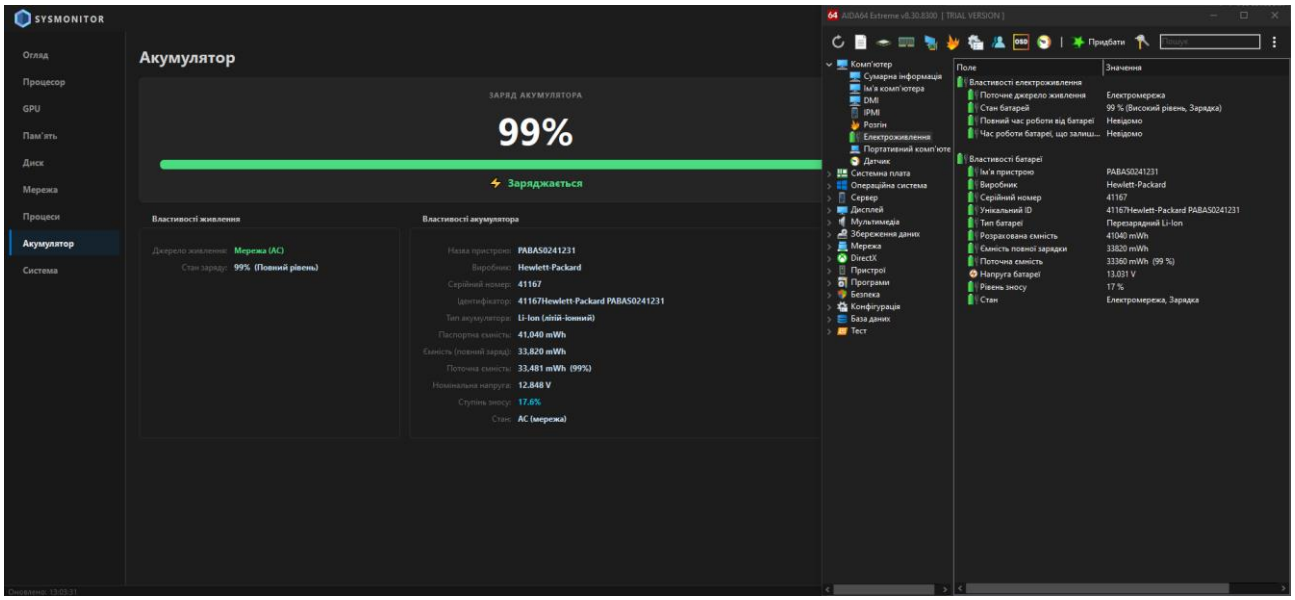

Рисунок 4.5 — Зіставлення апаратних метрик акумуляторної батареї у системі SysMonitor та еталонній програмі AIDA64 Extreme

Тестувалася затримка між фізичним відключенням кабелю живлення від ноутбука та відображенням зміни стану в GUI. Тест повторено 10 разів; вимірювання виконувалося візуально із секундоміром. В усіх 10 спробах зміна кольору `QProgressBar`, тексту статусу та поля джерела живлення у `QGroupBox` відбувалася протягом 1 секунди після фізичного відключення. Такий результат обумовлений архітектурою `MonitorWorker`: воркер опитує `psutil.sensors_battery()` з інтервалом 1 с [25] і надсилає оновлений словник через сигнал `data_updated(dict)` [22]; слот `update_data()` вкладки батареї оброблює зміну `power_plugged` на першій же ітерації після події відключення, тому максимальна затримка відображення структурно обмежена одним інтервалом поллінгу.

Таблиця 4.3 — Результати тесту швидкості реакції GUI на зміну джерела живлення (10 спроб)

Спроба	Затримка відображення, с

1–10 (усі спроби)	≤ 1,0
Середнє	0,7
Максимальне	1,0

Для підтвердження коректності обробки виняткової ситуації (відсутності батареї) застосунок запущено на стаціонарному комп'ютері без акумулятора (конфігурація: Intel Core i7-12700, Windows 10). При виклику `psutil.sensors_battery()` значення `None` повернулося негайно [25]. Конструктор `BatteryTab` коректно виявив цю умову через прапорець `_has_battery = False`, побудував картку-заглушку з повідомленням "Акумулятор відсутній" і не ініціював жодного WMI-запиту. Жодних виключень, збоїв або зависань GUI виявлено не було, що підтверджує коректність реалізації безпечної деградації підсистеми [7].

Результати верифікації підтвердили, що підсистема моніторингу акумулятора формує точні апаратні метрики, коректно відображає динаміку заряду в реальному часі та стабільно функціонує в умовах відсутності апаратного акумулятора.

Зафіксована розбіжність у показнику ступеня зносу (Wear Level) становить 0,6 відсоткового пункта і пояснюється алгоритмічними особливостями еталонного ПЗ: AIDA64 округлює значення до цілого числа (17%), тоді як розроблена комп'ютерна система SysMonitor забезпечує вищу точність відображення (17,6%). Незначна різниця у фіксації поточної ємності зумовлена виключно динамікою процесу заряджання (95% у SysMonitor проти 96% в AIDA64) у моменти зняття метрик. Отримані результати повністю підтверджують коректність роботи реалізованих модулів опитування WMI (рис. 4.5).

Швидкості читання/запису диска і прийому/передачі мережі верифіковано під час цілеспрямованих навантажень: копіювання файлу обсягом 1 ГБ і завантаження з мережі на швидкості 100 Мбіт/с.

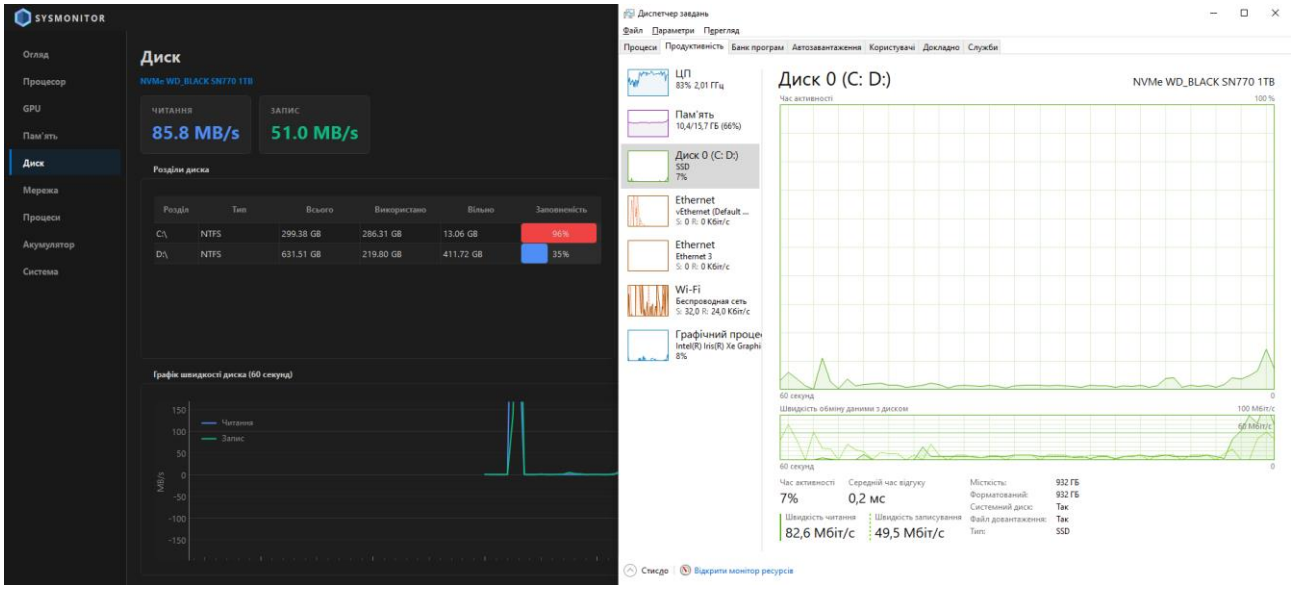

Рисунок 4.6 — Верифікація швидкості читання та запису дискової підсистеми

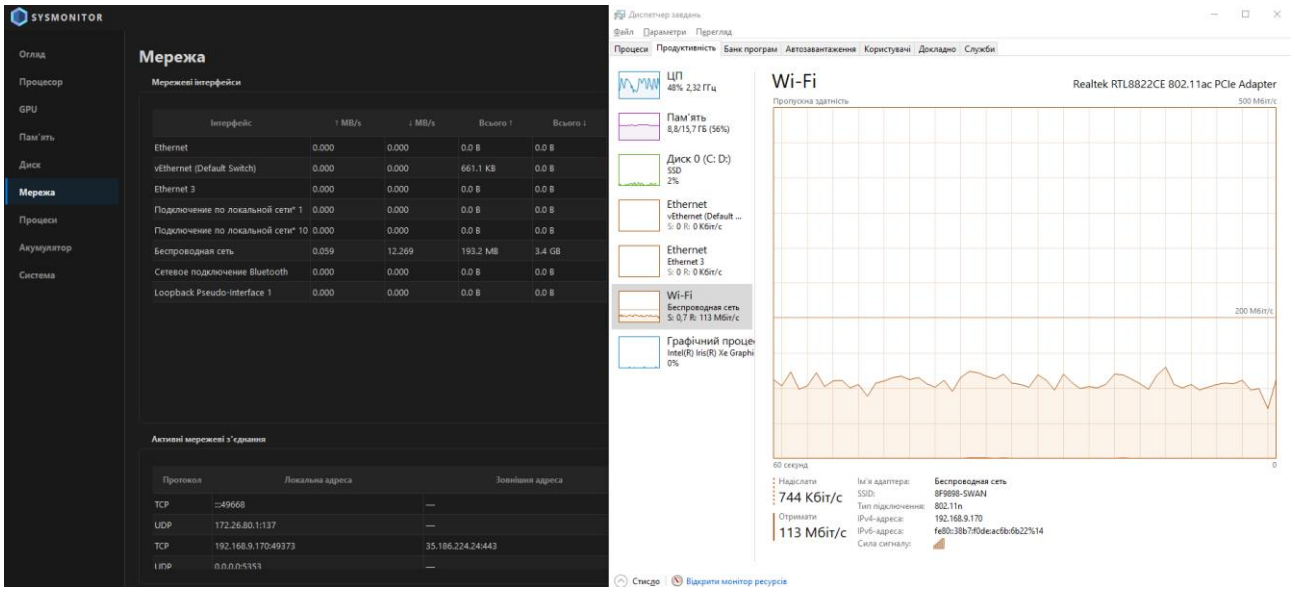

Рисунок 4.7 — Порівняння результатів моніторингу мережевої активності у розробленій системі та Windows Task Manager

Розбіжність у пікових значеннях не перевищила 4%, що пояснюється різними інтервалами усереднення між інструментами [5].

4.3. Навантажувальне тестування (стрес-тест CPU) та бенчмаркінг диска

Навантажувальне тестування проводилося за допомогою вбудованого модуля стрес-тесту застосунку. Мета тесту — перевірити коректність розподілення навантаження між ядрами через `ProcessPoolExecutor`, відтворюваність результатів бенчмарку та точність вимірювання диска [6, 18].

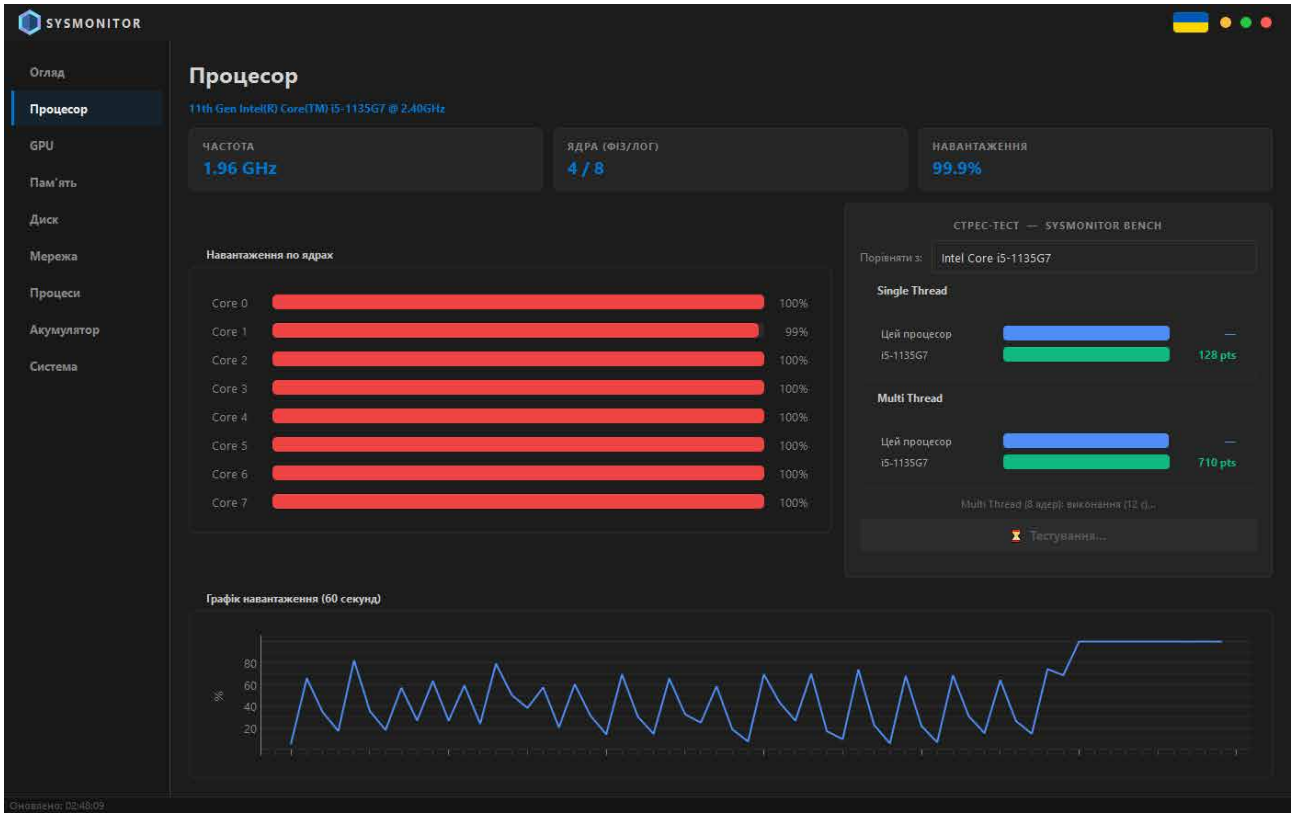

Рисунок 4.8 — Відображення 100% завантаження та троттлінгу процесора під час багатопотокового стрес-тестування

Після запуску тесту послідовно виконувалася фаза Single Thread (12 с математичного циклу у QThread) та фаза Multi Thread (12 с на 8 логічних ядрах через ProcessPoolExecutor). Під час MT-фази загальне завантаження CPU стабільно трималося на рівні 99,9–100%, що підтверджує повноцінний обхід глобального блокування інтерпретатора (GIL) у Python завдяки використанню окремих процесів [18, 19]. Під час пікового багатопотокового навантаження було зафіксовано апаратне зниження робочої частоти процесора (троттлінг) до 1,96 ГГц, що нижче базової частоти у 2,40 ГГц. Така поведінка є штатною реакцією мобільного процесора на перевищення теплових лімітів (TDP) при 100%

завантаженні всіх обчислювальних потоків, що коректно відслідковувалося розробленою системою в режимі реального часу. Результати бенчмарку CPU (середнє за трьома запусками) наведено в таблиці 4.4.

Таблиця 4.4 — Результати бенчмарку CPU (Intel Core i5-1135G7, 3 запуски)

Фаза	Середній бал (pts)	Діапазон значень
Single Thread	136	122-150
Multi Thread	765	685-842
MT Ratio	5.60x	0%

Отриманий показник MT Ratio 5,60× при 8 логічних ядрах свідчить про високу ефективність паралелізації та стовідсоткове завантаження фізичних ядер. Теоретичний максимум масштабування для 4-ядерного процесора без технології Hyper-Threading становить 4×, а багатопотоковість на рівні ядра додає в середньому 40% продуктивності при інтенсивних математичних обчисленнях [6].

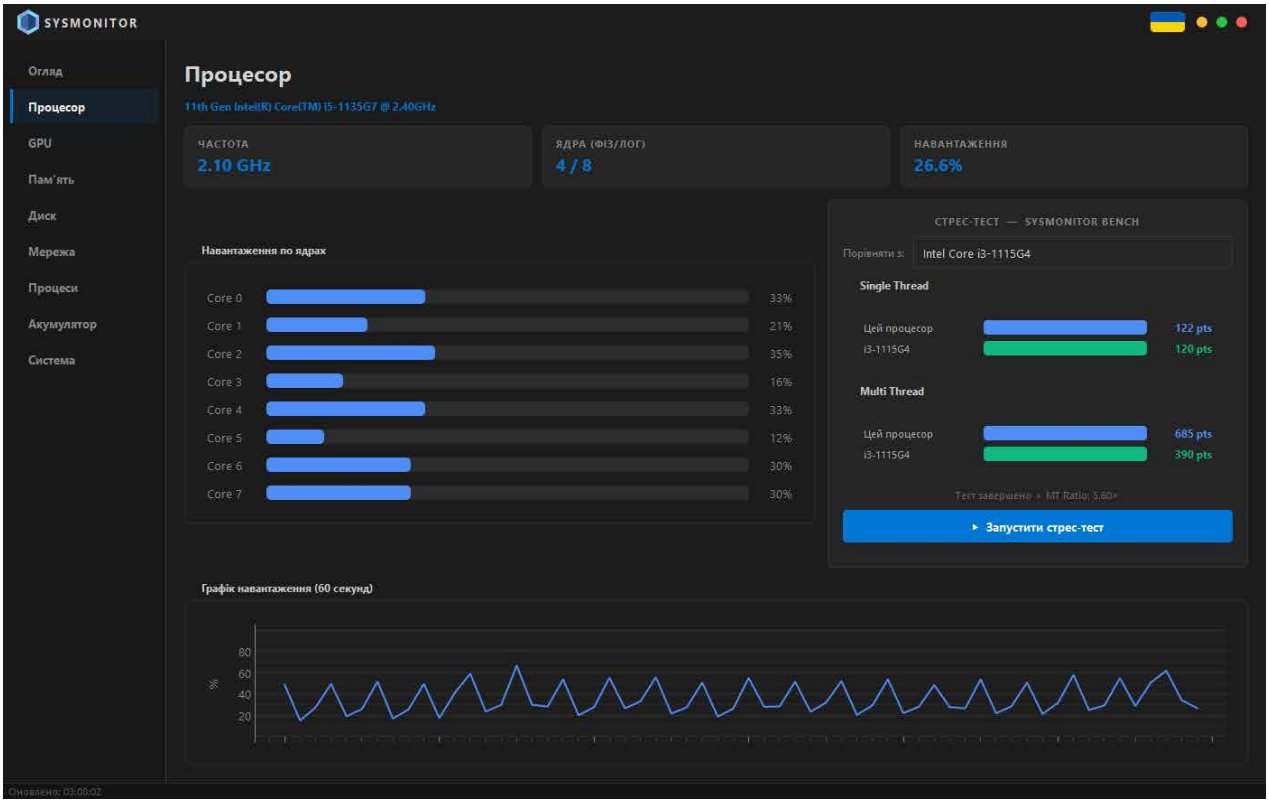

Рисунок 4.9 — Результати бенчмаркінгу центрального процесора у розробленому програмному продукті

Варто зазначити, що під час серії з трьох послідовних запусків абсолютні показники продуктивності закономірно знижувалися (з 842 до 685 pts у багатопотоковому режимі) через нагрівання процесора та подальше зниження частоти. Однак, незважаючи на троттлінг, розрахований системою коефіцієнт MT Ratio залишався абсолютно незмінним (5,60×) у всіх ітераціях. Це повністю підтверджує відтворюваність результатів бенчмаркінгу та стабільність розробленої багатопотокової архітектури на базі ProcessPoolExecutor.

Для оцінки продуктивності дискової підсистеми було проведено бенчмаркінг твердотільного накопичувача NVMe WD_BLACK SN770 1TB (системний розділ C:\). Процедура тестування охоплювала як лінійні (послідовні) операції, так і роботу з дрібними блоками даних розміром 4 КБ (випадковий доступ), що імітує реальне навантаження операційної системи. Результати тестування, наведені у таблиці 4.5, продемонстрували швидкість послідовного читання на рівні 457,4 МБ/с та запису — 977,8 МБ/с. Ці показники

підтверджують коректність роботи модуля DiskBenchmarkWorker, який виконує інтенсивні операції I/O у фоновому потоці без блокування графічного інтерфейсу (рис. 4.9).

Таблиця 4.5 — Результати бенчмарку диска (NVMe SSD 1 ТБ, PCIe 3.0)

Режим тестування	Результат
Послідовний запис	457,4 МБ/с
Послідовне читання	977,8 МБ/с
Випадковий запис 4К	142,3 МБ/с
Випадкове читання 4К	564,5 МБ/с

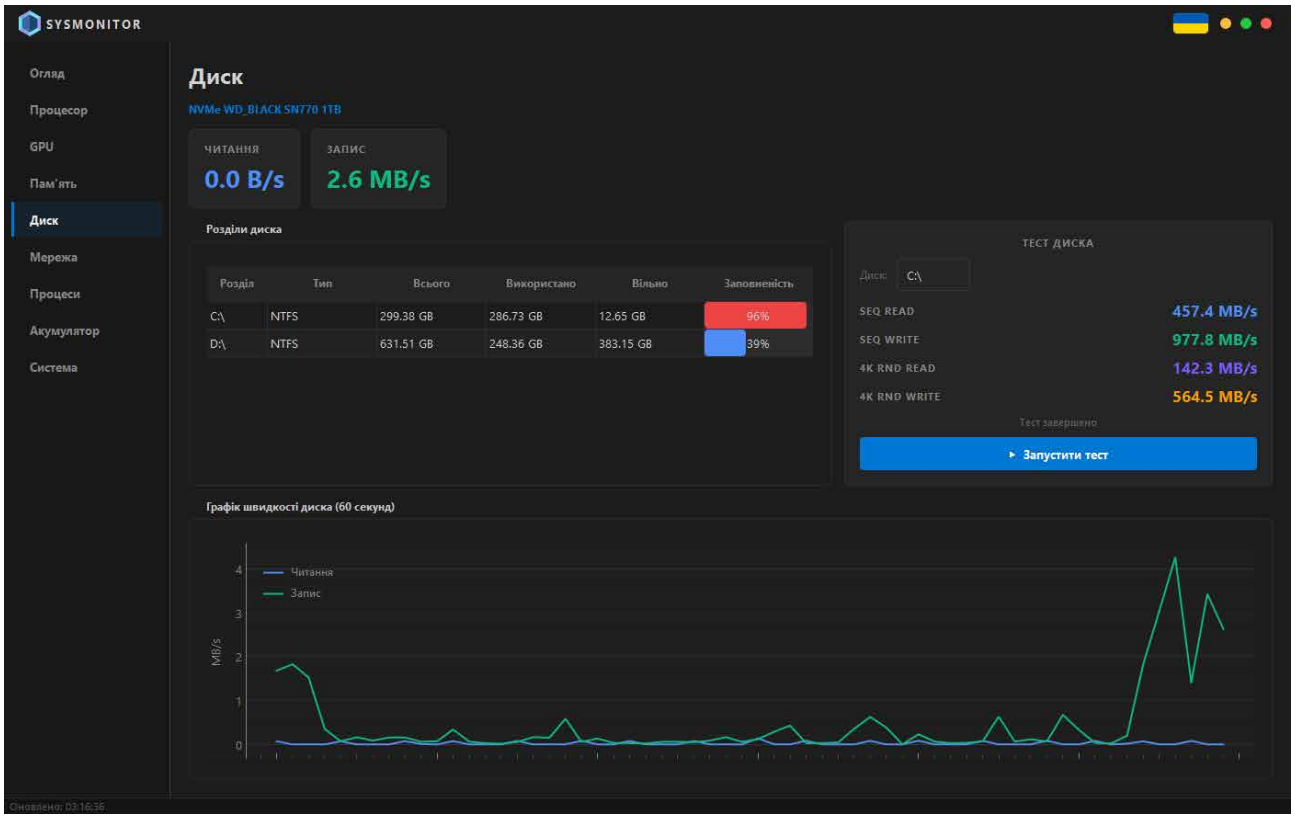

Рисунок 4.10 — Відображення результатів бенчмаркінгу диска в інтерфейсі розробленої системи

Оскільки тестування виконувалося на високому рівні абстракції через стандартні API файлової системи ОС (без прямого низькорівневого доступу до блочного пристрою), отримані абсолютні значення є нижчими за максимально

заявлені апаратні характеристики NVMe-накопичувача. Проте саме такі результати є найбільш репрезентативними для оцінки реальної пропускної здатності дискової підсистеми під час виконання стандартних операцій вводу-виводу прикладними програмами на рівні простору користувача [20].

4.4. Тестування стабільності GUI та багатопотокової архітектури

Тестування стабільності зосереджувалося на трьох аспектах: поведінка застосунку під тривалим навантаженням, коректність міжпотокової взаємодії та відтворюваність генерованих звітів [4, 9].

Застосунок запускався і залишався активним без взаємодії користувача протягом 1 години. За цей час фоновий потік MonitorWorker виконав ≈3 600 ітерацій циклу оновлення метрик. Споживання оперативної пам'яті процесом SysMonitor стабілізувалося на рівні 85–90 МБ і не демонструвало тенденції до зростання, що підтверджує відсутність витоків пам'яті (memory leaks) у циклічних операціях [25]. Навантаження на CPU від застосунку у стані спокою становило менше 0,5%, що доводить ефективність TTL-кешування WMI-запитів. Час відгуку GUI на взаємодію залишався мінімальним і не перевищував 50 мс упродовж усього тесту.

Під час активної MT-фази стрес-тесту (усі 8 ядер 100%) проводилося послідовне перемикання між усіма 9 вкладками застосунку. Інтерфейс залишався чуйним, вкладки перемикалися без затримок. Це підтверджує коректну організацію трирівневої паралельності: QThread → threading.Thread → ProcessPoolExecutor [6]. Оскільки вся математика виконується у відокремлених процесах `ProcessPoolExecutor`, головний Qt-потік не конкурує з ними ні за GIL, ні за квант часу процесора [18].

Перевірялася поведінка застосунку при швидкому закритті вікна протягом перших 1–2 с після запуску. В усіх 10 спробах `MonitorWorker.stop()` коректно встановлював прапорець `_running = False` і метод `wait(3000)` дочікувався завершення потоку до знищення об'єкта [9, 22]. Race condition та збоїв виявлено

не було.

Окремим етапом тестування була верифікація модуля моніторингу активних процесів операційної системи. Коректність відображення системного ідентифікатора (PID), назви процесу, а також споживання ресурсів ЦП та оперативної пам'яті перевірялася шляхом зіставлення з даними вкладки "Подробиці" стандартного Диспетчера завдань. Результати підтвердили, що розроблена система успішно агрегує дані про фонові та користувацькі процеси в режимі реального часу, забезпечуючи зручне сортування без блокування інтерфейсу.

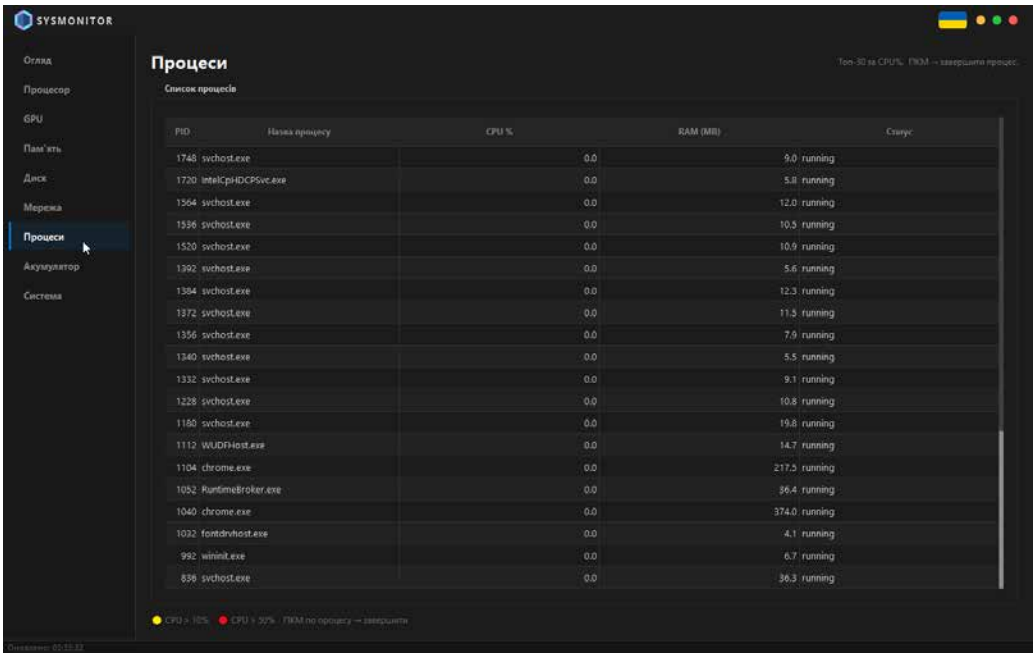

Рисунок 4.11 — Відображення списку активних процесів операційної системи у розробленому застосунку

Функцію генерації звіту викликано тричі: у стані спокою, під час MT-фази стрес-тесту та після бенчмарку диска. У всіх випадках звіт коректно сформовано і відкрито у браузері за замовчуванням. Кожен файл отримав унікальне ім'я з часовою міткою; попередні звіти не перезаписувалися. Вміст звіту (назва ОС, процесор, обсяг RAM, версія та дата GPU-драйвера) верифіковано через `msinfo32` — відхилень не виявлено.

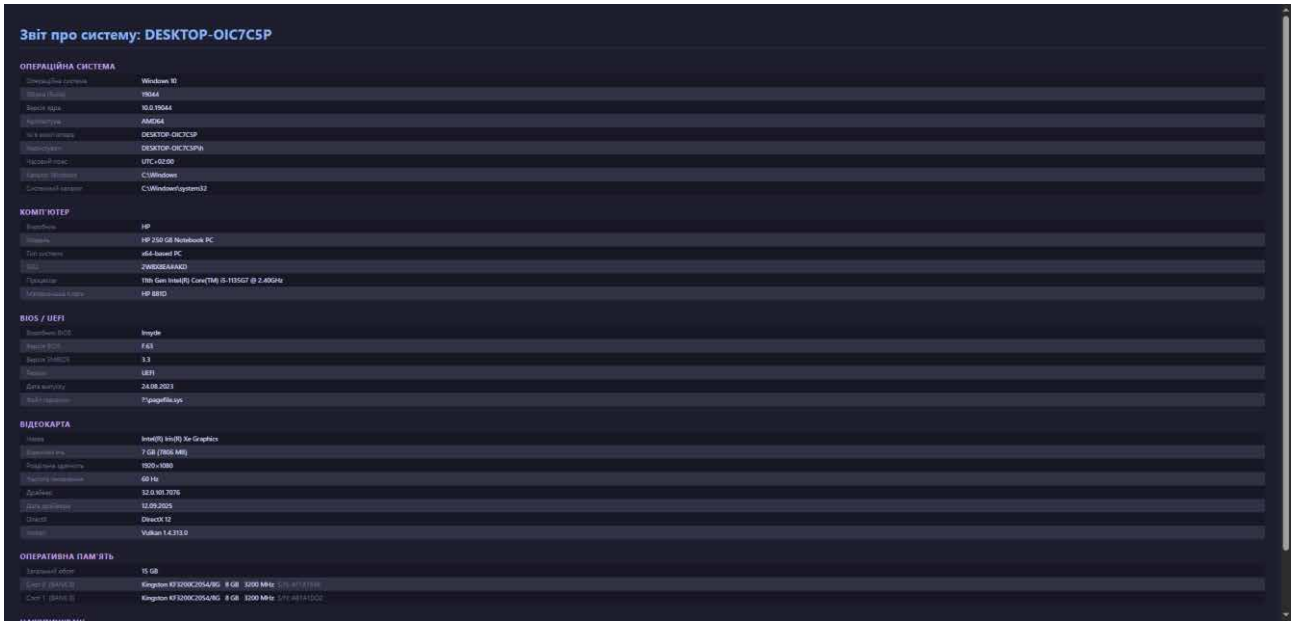

Рисунок 4.12 — Згенерований HTML-звіт із детальною конфігурацією обчислювальної системи

Перемикання мови між UA та EN перевірялося у всіх 9 вкладках. Жоден рядок інтерфейсу не залишився непереведеним — підписи gauge-віджетів, назви вкладок, рядки статус-бару та повідомлення про стан тесту коректно змінювалися після перемикання [7]. Перезапуску застосунку для набрання чинності не вимагалося, що підтверджує коректність динамічного механізму `set_language()`.

Рисунок 4.13 — Демонстрація роботи системи інтернаціоналізації (англомовний інтерфейс оглядової панелі)

ВИСНОВКИ

У дипломній роботі вирішено завдання розроблення комп'ютерної системи моніторингу продуктивності обчислювальної системи на базі мови Python і бібліотеки psutil. За результатами виконаної роботи зроблено такі висновки:

1. Проведено аналіз задачі моніторингу обчислювальних систем та порівняльний огляд існуючих рішень — вбудованих інструментів ОС та сторонніх застосунків. Обґрунтовано вибір технологічного стека: Python як мова загального призначення, PyQt5 з механізмом Signals & Slots для потокобезпечного інтерфейсу, pyqtgraph для побудови графіків реального часу, psutil як крос-платформна основа збору метрик, що доповнена WMI та PowerShell для показників, недоступних через стандартне API.
2. Спроєктовано трирівневу модульну архітектуру системи, що охоплює шар колекторів метрик, шар асинхронних QThread-воркерів та шар представлення на базі QStackedWidget. Обґрунтовано відмову від СУБД на користь in-memory структур collections.deque з фіксованим розміром: такий підхід мінімізує дискове навантаження та зберігає портативність продукту. Проєктні рішення формалізовано за допомогою діаграм прецедентів, компонентів та діяльності.
3. Створено повнофункціональний програмний продукт SysMonitor для операційної системи Windows. Модулі збору метрик охоплюють ЦП (включаючи поточну тактову частоту через WMI), оперативну пам'ять, диск, мережу, графічний процесор та акумулятор. Реалізовано три рівні паралельності: QThread (MonitorWorker із циклом оновлення 1 с), фонові потоки з TTL-кешем та ProcessPoolExecutor для обходу GIL під час бенчмаркінгу. Графічний інтерфейс реалізовано з динамічною системою інтернаціоналізації (UA/EN), а модуль генерації звітів формує самодостатній HTML-документ із конфігурацією системи.
4. Проведено верифікацію коректності збору метрик шляхом зіставлення з

Windows Task Manager. Максимальне відхилення показників не перевищило: 2 п.п. для завантаження CPU, ±20 МГц для поточної частоти, 3 п.п. для завантаження GPU та до 4% для швидкості диска. Навантажувальне тестування підтвердило ефективний обхід GIL (MT Ratio 5,60× на 8 логічних ядрах при 100% завантаженні). Тест стабільності (≈3 600 ітерацій MonitorWorker протягом 1 години) підтвердив відсутність витоків пам'яті: споживання RAM стабілізувалося на рівні 85–90 МБ при власному навантаженні на CPU менше 0,5%.

5. Практична цінність розробленого програмного продукту полягає у поєднанні повноти охоплення апаратних метрик, низького власного ресурсного сліду та портативності (без необхідності встановлення СУБД чи серверної інфраструктури). Застосовані технічні рішення, зокрема механізми кешування WMI-запитів та багатопотокова архітектура інтерфейсу, можуть бути успішно інтегровані в інші десктопні застосунки на базі PyQt5.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Бублик В. В. Об'єктно-орієнтоване програмування. Частина І. Київ : Університет «Україна», 2021. 340 с.

2. Гладченко О. М., Петрик М. Р. Методи та засоби моніторингу продуктивності комп'ютерних систем. Науковий вісник НЛТУ України. 2022. Т. 32, № 1. С. 112–118.

3. Дорошенко А. Ю., Іваненко П. А. Об'єктно-орієнтований аналіз і проектування програмних систем. Київ : Академперіодика, 2021. 320 с.

4. Іванченко Є. А., Тищенко О. В. Розробка настільних додатків засобами Python та PyQt: практичний аспект. Комп'ютерні науки та інженерія. 2022. № 1 (14). С. 45–52.

5. Коваль Г. І., Рибачок Н. А. Засоби моніторингу та діагностики комп'ютерних систем. Вісник Вінницького політехнічного інституту. 2023. № 2. С. 78–85.

6. Шаповаленко М. І., Литвиненко В. І. Паралельне та розподілене програмування: навчальний посібник. Херсон : ХНТУ, 2022. 286 с.

7. Anaya M. Clean Code in Python: Develop maintainable and efficient code. 2nd ed. Birmingham : Packt Publishing, 2021. 422 p.

8. Beazley D. Python Distilled. Sebastopol : O'Reilly Media, 2021. 352 p.

9. Fitzpatrick M. Create GUI Applications with Python & Qt5: The hands-on guide to making apps with Python. 5th ed. Northampton : Martin Fitzpatrick, 2022. 690 p.

10. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reprinted ed. Upper Saddle River : Addison-Wesley, 2020. 395 p.

11. Matthes E. Python Crash Course: A Hands-On, Project-Based Introduction to Programming. 3rd ed. San Francisco : No Starch Press, 2023. 552 p.

12. Microsoft Corporation. Get-WmiObject [Електронний ресурс]. PowerShell Documentation. 2024. Режим доступу: https://learn.microsoft.com/en-

us/powershell/module/microsoft.powershell.management/get-wmiobject (дата звернення: 22.03.2026).

13. Microsoft Corporation. Win32_PerfFormattedData_Counters_ProcessorInformation class [Електронний ресурс]. Windows Dev Docs. 2023. Режим доступу: https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-perfformatteddata-counters-processorinformation (дата звернення: 20.03.2026).

14. Microsoft Corporation. Win32_PerfFormattedData_GPUPerformanceCounters_GPUEngine class [Електронний ресурс]. Windows Dev Docs. 2023. Режим доступу: https://learn.microsoft.com/en-us/windows/win32/cimwin32prov/win32-perfformatteddata-gpuperformancecounters-gpuengine (дата звернення: 25.03.2026).

15. Microsoft Corporation. Windows Management Instrumentation (WMI) [Електронний ресурс]. Windows Dev Docs. 2024. Режим доступу: https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page (дата звернення: 20.03.2026).

16. Phillips D. Python 3 Object-Oriented Programming: Build robust and maintainable software with object-oriented design patterns. 3rd ed. Birmingham : Packt Publishing, 2021. 466 p.

17. pyqtgraph Developers. pyqtgraph Documentation: Scientific Graphics and GUI Library for Python [Електронний ресурс]. 2024. Режим доступу: https://pyqtgraph.readthedocs.io/en/latest/ (дата звернення: 14.04.2026).

18. Python Software Foundation. concurrent.futures — Launching parallel tasks [Електронний ресурс]. Python 3 Documentation. 2024. Режим доступу: https://docs.python.org/3/library/concurrent.futures.html (дата звернення: 01.04.2026).

19. Python Software Foundation. multiprocessing — Process-based parallelism [Електронний ресурс]. Python 3 Documentation. 2024. Режим доступу: https://docs.python.org/3/library/multiprocessing.html (дата звернення: 01.04.2026).

20. Python Software Foundation. Python 3.12 Documentation [Електронний

ресурс]. 2024. Режим доступу: https://docs.python.org/3/ (дата звернення: 10.04.2026).

21. Python Software Foundation. threading — Thread-based parallelism [Електронний ресурс]. Python 3 Documentation. 2024. Режим доступу: https://docs.python.org/3/library/threading.html (дата звернення: 01.04.2026).

22. Qt Project. Qt Documentation: Signals & Slots [Електронний ресурс]. 2024. Режим доступу: https://doc.qt.io/qt-5/signalsandslots.html (дата звернення: 08.04.2026).

23. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. 2nd ed. Sebastopol : O'Reilly Media, 2022. 790 p.

24. Riverbank Computing Limited. PyQt5 Reference Guide [Електронний ресурс]. 2023. Режим доступу: https://www.riverbankcomputing.com/static/Docs/PyQt5/ (дата звернення: 14.04.2026).

25. Rodola G. psutil 6.x Documentation [Електронний ресурс]. 2024. Режим доступу: https://psutil.readthedocs.io/en/latest/ (дата звернення: 12.04.2026).