Зміст

[User Story 1: Ініціалізація та запуск програми 2](#_Toc196759950)

[User Story 2: Моніторинг використання CPU 3](#_Toc196759951)

[User Story 3: Відображення використання CPU у GUI 4](#_Toc196759952)

[User Story 4: Моніторинг використання RAM 5](#_Toc196759953)

[User Story 5: Відображення використання RAM у GUI 6](#_Toc196759954)

[User Story 6: Моніторинг мережевої активності 7](#_Toc196759955)

[User Story 7: Відображення мережевої активності у GUI 8](#_Toc196759956)

[User Story 8: Управління процесами (завершення процесу) 9](#_Toc196759957)

[User Story 9: Логування даних та сповіщень 10](#_Toc196759958)

[User Story 10: Експорт даних у файл 11](#_Toc196759959)

# User Story 1: Ініціалізація та запуск програми

Опис:

Як користувач, я хочу, щоб програма ініціалізувала основні компоненти та запускала графічний інтерфейс, щоб я міг почати моніторинг системи.

Критерії приймання:

- Створюється екземпляр класу `SystemMonitor`, який ініціалізує `root`, `DataCollector`, `DisplayManager`, `ConfigManager`, і `Logger`.

- Викликається метод `setup\_gui()` для створення основного вікна з вкладками (Notebook).

- Викликається метод `start\_monitoring()` для запуску потоків моніторингу.

- Метод `run()` запускає головний цикл Tkinter (`mainloop`).

- Метод `on\_closing()` забезпечує коректне завершення програми (зупинка потоків, закриття GUI).

- Перевірка: Програма запускається, вікно відкривається, і при закритті не виникає помилок.

Оцінка часу:

- Написання коду: 3 години (реалізація ініціалізації, створення GUI, управління потоками).

- Відлагодження: 2 години (виправлення помилок із Tkinter та потоками).

- Тестування: 1 година (перевірка запуску, закриття, стабільності).

- Оновлення документації: 1 година (додавання опису ініціалізації до записки).

- Загальний час: 7 годин.

# User Story 2: Моніторинг використання CPU

Опис:

Як користувач, я хочу, щоб програма моніторила використання CPU в реальному часі, щоб я міг бачити, як завантажений мій процесор.

Критерії приймання:

- У класі `DataCollector` реалізовано метод `monitor\_cpu()`, який кожні 3 секунди збирає дані про використання CPU (загальне та по ядрах) за допомогою `psutil.cpu\_percent()`.

- Дані зберігаються в `cpu\_usage\_history`.

- Перевірка: Дані оновлюються кожні 3 секунди, `cpu\_usage\_history` містить коректні значення (0–100%).

Оцінка часу:

- Написання коду: 2 години (реалізація методу, використання `psutil`).

- Відлагодження: 1 година (перевірка коректності збору даних).

- Тестування: 1 година (перевірка, що дані оновлюються і зберігаються).

- Оновлення документації: 1 година (додавання опису моніторингу CPU).

- Загальний час: 5 годин.

# User Story 3: Відображення використання CPU у GUI

Опис:

Як користувач, я хочу бачити використання CPU у графічному інтерфейсі, щоб візуально оцінити завантаження процесора.

Критерії приймання:

- У класі `DisplayManager` реалізовано метод `update\_cpu\_display()`, який оновлює мітки та графік на вкладці CPU.

- Мітка показує загальне використання CPU (наприклад, "Total CPU Usage: 45.2%").

- Графік (Matplotlib) відображає використання CPU по ядрах за останні 60 секунд.

- Якщо використання CPU перевищує поріг (`cpu\_threshold`), викликається `show\_alert()`.

- Перевірка: Мітка та графік оновлюються кожні 3 секунди, сповіщення з’являється при перевищенні порогу.

Оцінка часу:

- Написання коду: 3 години (створення GUI-елементів, інтеграція з Matplotlib).

- Відлагодження: 2 години (виправлення проблем із оновленням GUI та графіками).

- Тестування: 1 година (перевірка оновлення GUI, коректність сповіщень).

- Оновлення документації: 1 година (додавання опису відображення CPU).

- \*\*Загальний час\*\*: 7 годин.

# User Story 4: Моніторинг використання RAM

Опис:

Як користувач, я хочу, щоб програма моніторила використання оперативної пам’яті, щоб я знав, скільки пам’яті використовується.

Критерії приймання:

- У класі `DataCollector` реалізовано метод `monitor\_ram()`, який кожні 3 секунди збирає дані про використання RAM за допомогою `psutil.virtual\_memory()`.

- Дані зберігаються в `ram\_usage\_history`.

- Перевірка: Дані оновлюються кожні 3 секунди, `ram\_usage\_history` містить коректні значення (0–100%).

Оцінка часу:

- Написання коду: 2 години (аналогічно моніторингу CPU).

- Відлагодження: 1 година (перевірка коректності збору даних).

- Тестування: 1 година (перевірка оновлення та збереження даних).

- Оновлення документації: 1 година (додавання опису моніторингу RAM).

- \*\*Загальний час\*\*: 5 годин.

# User Story 5: Відображення використання RAM у GUI

Опис:

Як користувач, я хочу бачити використання RAM у графічному інтерфейсі, щоб оцінити доступну пам’ять.

Критерії приймання:

- У класі `DisplayManager` реалізовано метод `update\_ram\_display()`, який оновлює мітку та графік на вкладці RAM.

- Мітка показує відсоток використання RAM та обсяг (наприклад, "RAM Usage: 65.3% (6.2/8.0 GB)").

- Графік (Matplotlib) відображає використання RAM за останні 60 секунд.

- Якщо використання RAM перевищує поріг (`ram\_threshold`), викликається `show\_alert()`.

- Перевірка: Мітка та графік оновлюються, сповіщення працює.

Оцінка часу:

- Написання коду: 3 години (аналогічно відображенню CPU).

- Відлагодження: 2 години (виправлення проблем із GUI).

- Тестування: 1 година (перевірка оновлення та сповіщень).

- Оновлення документації: 1 година (додавання опису відображення RAM).

- Загальний час: 7 годин.

# User Story 6: Моніторинг мережевої активності

Опис:

Як користувач, я хочу, щоб програма моніторила мережеву активність, щоб я знав, скільки даних передається.

Критерії приймання:

- У класі `DataCollector` реалізовано метод `monitor\_network()`, який кожні 3 секунди збирає дані про швидкість завантаження та відправлення за допомогою `psutil.net\_io\_counters()`.

- Дані зберігаються в `net\_download\_history` та `net\_upload\_history`.

- Перевірка: Дані оновлюються, значення швидкості коректні (Mbps).

Оцінка часу:

- Написання коду: 2 години (аналогічно моніторингу CPU).

- Відлагодження: 1 година (перевірка коректності даних).

- Тестування: 1 година (перевірка оновлення даних).

- Оновлення документації: 1 година (додавання опису моніторингу мережі).

- \*\*Загальний час\*\*: 5 годин.

# User Story 7: Відображення мережевої активності у GUI

Опис:

Як користувач, я хочу бачити мережеву активність у графічному інтерфейсі, щоб оцінити обсяг передачі даних.

Критерії приймання:

- У класі `DisplayManager` реалізовано метод `update\_network\_display()`, який оновлює мітку та графік на вкладці Network.

- Мітка показує швидкість завантаження та відправлення (наприклад, "Download: 5.2 Mbps | Upload: 2.1 Mbps").

- Графік (Matplotlib) відображає швидкість за останні 60 секунд.

- Якщо швидкість перевищує поріг (`net\_traffic\_threshold`), викликається `show\_alert()`.

- Перевірка: Мітка та графік оновлюються, сповіщення працює.

Оцінка часу:

- Написання коду: 3 години (аналогічно відображенню CPU).

- Відлагодження: 2 години (виправлення проблем із GUI).

- Тестування: 1 година (перевірка оновлення та сповіщень).

- Оновлення документації: 1 година (додавання опису відображення мережі).

- Загальний час: 7 годин.

# User Story 8: Управління процесами (завершення процесу)

Опис:

Як користувач, я хочу мати можливість завершувати процеси через інтерфейс, щоб зменшити навантаження на систему.

Критерії приймання:

- У класі `SystemMonitor` реалізовано метод `kill\_process()`, який завершує вибраний процес за PID після підтвердження користувача.

- У вкладці RAM відображається таблиця процесів (`process\_tree`) з колонками PID, Name, Memory, CPU.

- Кнопка "Kill Process" викликає `kill\_process()` для вибраного процесу.

- Перевірка: Процес завершується, таблиця оновлюється, помилки (наприклад, доступ заборонено) обробляються.

Оцінка часу:

- Написання коду: 3 години (реалізація таблиці, логіки завершення процесу).

- Відлагодження: 2 години (обробка помилок, перевірка коректності завершення).

- Тестування: 1 година (перевірка завершення процесу, оновлення таблиці).

- Оновлення документації: 1 година (додавання опису управління процесами).

- Загальний час: 7 годин.

# User Story 9: Логування даних та сповіщень

Опис:

Як розробник, я хочу, щоб програма логувала дані та сповіщення, щоб я міг діагностувати проблеми.

Критерії приймання:

- У класі `Logger` реалізовано методи `log\_data()` і `log\_alert()`.

- `log\_data()` записує діагностичні повідомлення у файл `monitor.log` (використовуючи `logging`).

- `log\_alert()` додає сповіщення в `alert\_log` і записує їх у лог.

- Перевірка: Файл `monitor.log` містить записи, `alert\_log` оновлюється.

Оцінка часу:

- Написання коду: 2 години (налаштування логування, реалізація методів).

- Відлагодження: 1 година (перевірка коректності запису).

- Тестування: 1 година (перевірка файлу та списку сповіщень).

- Оновлення документації: 1 година (додавання опису логування).

- Загальний час: 5 годин.

# User Story 10: Експорт даних у файл

Опис:

Як користувач, я хочу експортувати зібрані дані у файл, щоб зберегти звіт про стан системи.

Критерії приймання:

- У класі `Logger` реалізовано метод `export\_data()`, який створює текстовий файл із даними (CPU, RAM, мережа, сповіщення).

- Експорт викликається вручну через кнопку "Export Data".

- Файл має формат `system\_report\_<timestamp>.txt`.

- Перевірка: Файл створюється, містить коректні дані.

Оцінка часу:

- Написання коду: 2 години (реалізація експорту, створення кнопки).

- Відлагодження: 1 година (перевірка коректності файлу).

- Тестування: 1 година (перевірка вмісту файлу).

- Оновлення документації: 1 година (додавання опису експорту).

- Загальний час: 5 годин.