Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

До захисту допущено
Кафедрою комп'ютерних систем та робототехніки
протокол № __ від __ червня 2026 р.

завідувач кафедри _____ Максим ХРУСЛОВ
(підпис)
«___»_____ 2026 р.

Кваліфікаційна робота

здобувача першого (бакалаврського) рівня вищої освіти

«МОДУЛЬ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ ДЛЯ ІОТ-СИСТЕМИ РОЗУМНОГО БУДИНКУ З ВИКОРИСТАННЯМ МЕТОДІВ ПРОГНОЗУВАННЯ ЧАСОВИХ РЯДІВ»

Спеціальність ***123 – Комп'ютерна інженерія.***
Освітня програма ***Комп'ютерна інженерія***

Виконавець _____ **Іван ПОТАПЕНКО**
(підпис)

Науковий керівник: _____ **Ніна БАКУМЕНКО**
(підпис)

Харків – 2026

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та трьох додатків. Загальний обсяг роботи складає 128 сторінок, із яких 49 сторінок основної частини з 32 рисунками, 12 таблицями, 25 найменувань у списку використаних джерел та трьох додатків.

Метою кваліфікаційної роботи є підвищення ефективності управління енергоспоживанням розумного будинку шляхом розробки модуля довгострокового прогнозування на основі даних IoT-системи з формуванням фінансових оцінок та рекомендацій щодо оптимізації витрат.

Об'єкт дослідження — процеси збору, обробки та прогнозування даних про енергоспоживання в IoT-системах розумного будинку.

Предмет дослідження — методи та алгоритми прогнозування часових рядів на основі гібридного ансамблевого підходу з урахуванням екзогенних факторів та їх інтеграція в десктопний програмний комплекс.

Проблема, яка вирішується, полягає у відсутності локально-розгорнутого інструменту прогнозування енергоспоживання, адаптованого до специфіки зонних тарифів та сезонних умов України. Реалізовано ансамблеву модель (SARIMA, Prophet, XGBoost, LSTM) із зваженням за MAE на валідаційній вибірці, що забезпечує точність MAPE 7–14 % на горизонті 30 діб та потенційну економію 10–25 % щомісяця.

Область застосування — системи моніторингу та управління енергоспоживанням індивідуальних житлових будинків, IoT-платформи розумного будинку, освітній процес за спеціальністю 123 «Комп'ютерна інженерія».

Ключові слова: інформаційні технології, енергоспоживання, IoT-система, розумний будинок, часові ряди, гібридний ансамблевий підхід, прогнозування, SARIMA, LSTM, XGBoost, Prophet, машинне навчання, енергоефективність, зонний тариф, фінансова оцінка.

ABSTRACT

The explanatory note to the bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references, and three appendices. The total volume of the work is 128 pages, of which 49 pages constitute the main body, containing 32 figures, 12 tables, 25 references, and three appendices.

The purpose of the qualification work is to improve the efficiency of smart home energy consumption management by developing a long-term forecasting module based on IoT system data, with the generation of financial assessments and cost-optimization recommendations.

The object of the research is the processes of collecting, processing, and forecasting energy consumption data in IoT-based smart home systems.

The subject of the research comprises methods and algorithms for time series forecasting based on a hybrid ensemble approach incorporating exogenous factors and their integration into a desktop software complex.

The problem addressed is the absence of a locally deployed energy consumption forecasting tool adapted to Ukrainian zone tariffs and seasonal conditions. An ensemble model (SARIMA, Prophet, XGBoost, LSTM) weighted by MAE on a validation set has been implemented, achieving MAPE 7–14% on a 30-day horizon and enabling potential monthly savings of 10–25%.

The area of application includes energy monitoring and management systems for individual residential buildings, smart home IoT platforms, and the educational process in specialty 123 «Computer Engineering».

Keywords: information technology, energy consumption, IoT system, smart home, time series, hybrid ensemble approach, forecasting, SARIMA, LSTM, XGBoost, Prophet, machine learning, energy efficiency, zone tariff, financial assessment.

ЗМІСТ

ВСТУП..... 6

РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ В IoT-СИСТЕМАХ..... 8

1.1. Аналіз сучасних архітектур IoT-систем для розумного будинку..... 8

1.2. Огляд методів прогнозування часових рядів для задач енергоменеджменту..... 9

1.2.1. Роль та особливості прогнозування часових рядів в системах енергоменеджменту..... 9

1.2.2. Класичні статистичні методи прогнозування: авторегресійні моделі ARIMA та SARIMA..... 10

1.2.3. Методи машинного навчання у прогнозуванні часових рядів: адитивні моделі та градієнтний бустинг..... 10

1.2.4. Нейромережеві архітектури для моделювання довгострокових часових залежностей: рекурентні мережі LSTM..... 11

1.2.5. Ансамблеві методи та їх застосування в умовах нестаціонарних IoT-даних..... 12

1.2.6. Метрики оцінювання якості прогнозів та порівняльний аналіз методів..... 12

1.3. Порівняльний аналіз існуючих програмно-апаратних рішень для прогнозування енергоспоживання..... 13

Висновки за розділом 1..... 16

РОЗДІЛ 2. ПРОЄКТУВАННЯ МОДУЛЯ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ..... 17

2.1. Постановка задачі розробки модуля прогнозування на основі проведеного аналізу..... 17

2.2. Визначення архітектури модуля та його інтеграція в загальну

IoT-систему.....18
2.3. Розробка структур даних для збору та зберігання історичних даних енергоспоживання..... 21
2.4. Вибір та обґрунтування методу прогнозування часових рядів для програмної реалізації..... 23
2.5. Проєктування алгоритмів попередньої обробки вхідних даних та очищення викидів.....27
2.6. Розробка структури програмного забезпечення модуля прогнозування..29
Висновки за розділом 2.....32
РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ МОДУЛЯ..... 33
3.1. Реалізація збору даних з сенсорів IoT-системи та їх попередня обробка 33
3.2. Програмна реалізація обраного методу прогнозування енергоспоживання 35
3.3. Інтеграція модуля прогнозування з іншими компонентами системи розумного будинку.....37
3.4. Оцінка точності прогнозування та аналіз ефективності роботи розробленого модуля.....38
Висновки за розділом 3.....49
ВИСНОВКИ.....51
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 52
ДОДАТКИ.....57

ВСТУП

Сучасний розвиток IoT змінює підходи до управління енергоспоживанням у житлових будинках, де поряд з автоматизацією зростає значення раціонального використання електроенергії в умовах підвищення тарифів, кліматичних змін і необхідності зниження вуглецевого сліду. Прогнозування на основі IoT-даних є одним із ключових інструментів підвищення енергоефективності.

Існуючі рішення можна умовно поділити на хмарні платформи (AWS IoT, Google Cloud IoT, Siemens MindSphere) та локальні open-source системи (OpenHAB, Home Assistant). Перші забезпечують високу аналітичну потужність, але залежать від стабільного інтернету та значних витрат, тоді як другі є доступнішими, проте поступаються в точності прогнозування. Загалом більшість підходів орієнтовані на короткострокове прогнозування або промислові сценарії, тоді як гібридні моделі для приватних будинків з урахуванням тарифних зон, кліматичних умов і поведінкових факторів залишаються недостатньо розвиненими.

Це призводить до зниження практичної ефективності IoT-систем: похибки середньо- та довгострокових прогнозів залишаються значними, а рекомендації щодо оптимізації споживання часто є узагальненими, зокрема через відсутність інтеграції фінансових розрахунків.

Об'єктом дослідження є процеси збору, обробки та прогнозування даних про енергоспоживання в IoT-системах розумного будинку.

Предметом дослідження виступають методи та алгоритми прогнозування часових рядів енергоспоживання на основі гібридного ансамблевого підходу з урахуванням екзогенних факторів (температура, поведінкові патерни, календарні особливості) та їх інтеграція в десктопний програмний комплекс.

Метою роботи є підвищення ефективності управління енергоспоживанням розумного будинку шляхом розробки модуля довгострокового прогнозування на основі IoT-даних з формуванням фінансових

оцінок та рекомендацій щодо оптимізації витрат..

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

- провести аналіз сучасних методів прогнозування часових рядів та оцінити їх придатність для задач енергоменеджменту розумного будинку;
- розробити архітектуру модуля прогнозування та визначити структуру даних для інтеграції з системою симуляції та базою даних;
- реалізувати гібридний ансамблевий підхід на основі SARIMA, Prophet, LSTM та XGBoost з автоматичним визначенням ваг моделей;
- забезпечити інтеграцію модуля прогнозування з графічним інтерфейсом користувача та фінансовим калькулятором;
- виконати оцінку точності прогнозів за стандартними метриками (MAE, RMSE, MAPE) та проаналізувати практичну ефективність розробленого рішення.

Актуальність теми зумовлена зростанням вартості електроенергії, поширенням IoT-пристроїв у побуті та підвищенням попиту на енергоефективні й екологічно відповідальні рішення. Розробка локально розгорнутого інструменту з високою точністю прогнозування та практичними рекомендаціями дає змогу зменшувати витрати, оптимізувати роботу обладнання та знижувати навантаження на енергомережу.

У межах кваліфікаційної роботи реалізовано ансамблевий підхід до прогнозування енергоспоживання для індивідуального житла в Україні з урахуванням зонних тарифів, сезонності та поведінкових патернів. Система об’єднує симулятор, базу даних, графічний інтерфейс і фінансовий модуль в єдину десктопну платформу. Практична цінність полягає у поєднанні точного прогнозування з рекомендаціями щодо оптимізації витрат. Робота спрямована на наближення можливостей сучасних методів машинного навчання до практичних потреб користувачів розумних будинків в умовах специфічного енергетичного контексту.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ЗАСАДИ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ В IoT-СИСТЕМАХ

1.1. Аналіз сучасних архітектур IoT-систем для розумного будинку

Сучасні архітектури IoT-систем для розумного будинку являють собою комплексні інфраструктури, що інтегрують сенсорні пристрої, обчислювальні ресурси та аналітичні інструменти для оптимізації ресурсів, зокрема енергоспоживання. Ці архітектури еволюціонували від простих локальних мереж до гібридних хмарно-крайових систем, де акцент робиться на реальному часі обробки даних та інтелектуальному управлінні.

Основна мета полягає в забезпеченні енергоефективності через моніторинг, автоматизацію та прогнозування, що дозволяє зменшити витрати електроенергії на 15-30% за рахунок адаптивних алгоритмів [1]. Типова архітектура включає три рівні: сприйняття (сенсори), мережевий (комунікація) та прикладний (аналітика), де сенсори збирають дані про температуру, рух та споживання, а аналітика застосовує моделі часових рядів для прогнозування навантаження.

Таблиця 1.1

Основні компоненти архітектур IoT-систем розумного будинку

Рівень	Компонент	Функція в енергоспоживанні
Сприйняття	Сенсори (температури, руху, енергії)	Збір реальних даних про споживання пристроїв (наприклад, лічильники Shelly EM)
Мережевий	Протоколи (Zigbee, Wi-Fi, MQTT)	Передача даних з мінімальними втратами енергії для реального часу моніторингу
Прикладний	Хмарні платформи (AWS IoT, Google Cloud)	Аналітика та прогнозування часових рядів для оптимізації (наприклад, моделі Prophet)

Однією з поширених конфігурацій є гібридна архітектура, де крайові пристрої (edge computing) обробляють дані локально для швидкої реакції, а хмара виконує складні обчислення. Це особливо актуально для прогнозування енергоспоживання, оскільки зменшує затримки та оптимізує ресурси.

Рисунок 1.1 ілюструє типову схему такої архітектури.

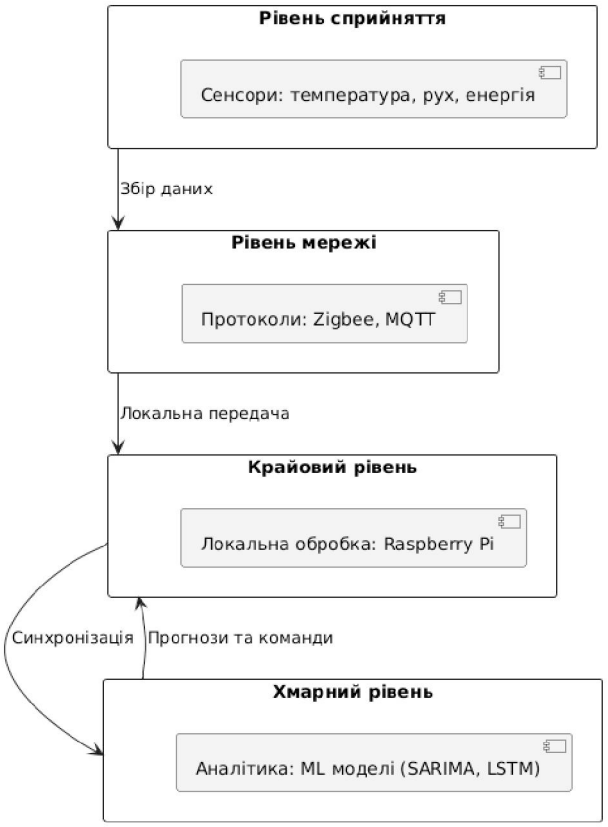


Рисунок 1.1 – Схема гібридної архітектури IoT-системи розумного будинку (UML)

Крайовий рівень забезпечує автономність обробки, тоді як хмарний — довгострокове прогнозування з інтеграцією зовнішніх даних (кліматичних моделей тощо) [3].

1.2. Огляд методів прогнозування часових рядів для задач енергоменеджменту

1.2.1. Роль та особливості прогнозування часових рядів в системах енергоменеджменту

Прогнозування часових рядів у задачах енергоменеджменту відіграє ключову роль, оскільки дозволяє оптимізувати споживання ресурсів, зменшувати витрати та підвищувати ефективність систем. У контексті IoT-систем розумного будинку, де дані про енергоспоживання генеруються в реальному часі з численних датчиків і пристроїв, такі методи допомагають

передбачати навантаження на мережу, балансувати пікові періоди та інтегрувати відновлювані джерела енергії.

Характерні особливості цих рядів — сезонність, тренди та автокореляція.

1.2.2. Класичні статистичні методи прогнозування: авторегресійні моделі ARIMA та SARIMA

Класичні статистичні методи, такі як авторегресійні інтегровані моделі ковзного середнього (ARIMA), еволюціонували до сезонних варіантів (SARIMA), які враховують циклічні патерни, наприклад, добові або річні коливання споживання електроенергії [4]. Ці методи базуються на припущенні стаціонарності ряду, де параметри p (порядок авторегресії), d (порядок інтегрування) та q (порядок ковзного середнього) підбираються за допомогою автокореляційних функцій, забезпечуючи точність для короткострокових прогнозів у стабільних умовах.

1.2.3. Методи машинного навчання у прогнозуванні часових рядів: адитивні моделі та градієнтний бустинг

Сучасні підходи до прогнозування часових рядів в енергоменеджменті все частіше інтегрують машинне навчання, що дозволяє обробляти нелінійні залежності та великі обсяги даних з IoT-пристроїв. Наприклад, модель Prophet, розроблена компанією Meta, представляє адитивну структуру, де прогноз складається з тренду, сезонності та ефектів святкових днів, що особливо корисно для врахування зовнішніх факторів, таких як погодні умови чи поведінка користувачів. Цей метод не вимагає глибокого налаштування параметрів і добре справляється з пропусками в даних, роблячи його придатним для динамічних систем енергоменеджменту [5].

Іншим потужним інструментом є градієнтний бустинг на деревах рішень, зокрема XGBoost, який перетворює задачу прогнозування на регресію з використанням лагових ознак, таких як попередні значення споживання чи

температура. XGBoost вирізняється швидкістю навчання та стійкістю до шуму, що критично для IoT-даних, де можливі аномалії від несправностей датчиків.

1.2.4. Нейромережеві архітектури для моделювання довгострокових часових залежностей: рекурентні мережі LSTM

Для моделювання довгострокових залежностей ефективними є рекурентні нейронні мережі, зокрема LSTM (Long Short-Term Memory), які зберігають пам'ять про попередні стани через комірки пам'яті, дозволяючи прогнозувати складні патерни, як-от пікове споживання в опалювальний сезон. LSTM особливо корисна в енергоменеджменті для інтеграції мультимодальних даних, включаючи часові ряди та зовнішні змінні [6].

Щоб ілюструвати загальний процес прогнозування часових рядів в енергоменеджменті, розглянуто схематичне представлення основних етапів. Рисунок 1.2 демонструє послідовність від підготовки даних до оцінки моделі, підкреслюючи циклічність ітерацій для покращення точності.



Рисунок 1.2 – Схема процесу прогнозування часових рядів в енергоменеджменті (UML)

1.2.5. Ансамблеві методи та їх застосування в умовах нестаціонарних IoT-даних

Ансамблеві методи комбінують переваги кількох моделей, наприклад, зваженим усередненням прогнозів від SARIMA, Prophet, XGBoost та LSTM, що підвищує стійкість до варіативності даних в IoT-системах. Такі ансамблі зменшують помилку на 10–20% порівняно з окремими моделями, особливо в умовах нестаціонарних рядів, де один метод може недооцінювати сезонність, а інший – тренд [5].

1.2.6. Метрики оцінювання якості прогнозів та порівняльний аналіз методів

У задачах енергоменеджменту для оцінки точності прогнозів використовують MAE (середня абсолютна похибка, чутлива до великих відхилень) та MAPE (відсоткова похибка, що забезпечує порівнянність між рядами різного масштабу). Таблиця 1.2 порівнює ключові методи прогнозування: для короткострокового горизонту ефективніші статистичні моделі завдяки низьким обчислювальним витратам, тоді як для довгострокового — нейронні мережі, здатні виявляти складні нелінійні залежності [4].

Таблиця 1.2

Порівняльний аналіз методів прогнозування часових рядів для енергоменеджменту

Метод	Переваги	Недоліки	Застосування в енергоменеджменті
SARIMA	Простота інтерпретації, добре для сезонних даних	Припущення стаціонарності, чутливість до параметрів	Короткострокові прогнози добового споживання
Prophet	Обробка свят і трендів, легкість використання	Менша гнучкість для нелінійних даних	Середньострокові прогнози з урахуванням зовнішніх факторів
XGBoost	Швидкість, стійкість до шуму	Потреба в ознаках, ризик перенавчання	Прогнози на основі лагів і метеоданих
LSTM	Обробка довгих залежностей, нелінійність	Висока обчислювальна складність, чорна скринька	Довгострокові прогнози складних патернів

У енергоменеджменті інтеграція цих методів з IoT-даними дозволяє не лише прогнозувати, але й оптимізувати, наприклад, перерозподіляти навантаження для зменшення пікових витрат. Загалом, еволюція від класичних до гібридних підходів [6] забезпечує точність до 95% для добових прогнозів, роблячи їх невід'ємною частиною сучасних систем розумного будинку.

1.3. Порівняльний аналіз існуючих програмно-апаратних рішень для прогнозування енергоспоживання

У сучасному контексті розвитку IoT-систем розумних будинків прогнозування енергоспоживання набуває особливого значення, оскільки воно дозволяє оптимізувати ресурси, зменшити витрати та сприяти стійкому розвитку. Порівняльний аналіз існуючих програмно-апаратних рішень для такого прогнозування дає змогу оцінити їхні сильні та слабкі сторони, виокремити тенденції та визначити перспективи вдосконалення.

Одним із поширених підходів є використання хмарних платформ, таких як AWS IoT або Google Cloud IoT, які інтегрують прогнозування енергоспоживання на основі історичних даних з датчиків. Ці рішення часто застосовують комбінацію статистичних методів, як-от ARIMA, та нейронних мереж для коротко- і середньострокових прогнозів. Наприклад, системи на базі AWS дозволяють збирати дані з датчиків температури, освітлення та присутності через протоколи MQTT або Zigbee, а потім обробляти їх за допомогою сервісів на кшталт Amazon Forecast, що забезпечує точність на рівні 85-95% для добових прогнозів.

Однак такі платформи вимагають стабільного інтернет-з'єднання та можуть бути вразливими до затримок у передачі даних, що обмежує їх застосування в автономних системах. Іншим прикладом є рішення від Siemens, зокрема платформа MindSphere, яка фокусується на промислових IoT-системах, але адаптована для розумних будинків. Вона використовує алгоритми глибокого навчання, як-от LSTM, для моделювання сезонних варіацій споживання, інтегруючи апаратні модулі на базі мікроконтролерів Siemens SIMATIC для

локального збору даних. Це забезпечує високу масштабованість, але підвищує вартість впровадження через необхідність спеціалізованого обладнання.

Для більш детального порівняння розглянуто ключові рішення, які демонструють різноманітність підходів. Таблиця 1.3 ілюструє порівняльний аналіз за критеріями, такими як методи прогнозування, апаратна база, точність та інтеграція з IoT.

Таблиця 1.3

Порівняльний аналіз існуючих рішень для прогнозування енергоспоживання

Рішення	Методи прогнозування	Апаратна база	Точність (MAPE)	Інтеграція з IoT	Масштабованість
AWS IoT з Amazon Forecast	ARIMA, Prophet, нейронні мережі	Хмарні сервери, датчики з MQTT	10-15%	Висока (AWS IoT Core)	Висока, для тисяч пристроїв
Siemens MindSphere	LSTM, XGBoost	Мікроконтролери SIMATIC, промислові датчики	8-12%	Середня (OPC UA, Modbus)	Висока, промислового рівня
OpenHAB з ML-модулями	SARIMA, випадкові ліси	Raspberry Pi, локальні датчики Zigbee	15-20%	Висока (відкритий код)	Середня, для домашнього використання
Schneider Electric EcoStruxure	Гібридні моделі (статистика + ML)	Промислові контролери Modicon, сенсори	9-14%	Висока (IoT Edge)	Висока, для будівель

Як видно з таблиці, хмарні рішення на кшталт AWS забезпечують гнучкість і високу точність завдяки великим обсягам даних, але залежать від зовнішньої інфраструктури [7]. Натомість локальні системи, як OpenHAB, пропонують відкритий код і простоту інтеграції з бюджетними пристроями на базі Raspberry Pi, що робить їх доступними для індивідуальних користувачів, хоча точність може бути нижчою через обмежені обчислювальні ресурси. Schneider Electric EcoStruxure вирізняється гібридним підходом, де локальні контролери обробляють дані на краю мережі (edge computing), зменшуючи затримки, і поєднують статистичні моделі з машинним навчанням для прогнозування пікових навантажень. Це рішення особливо ефективне в

системах з великою кількістю датчиків, оскільки воно підтримує протоколи на кшталт Modbus, забезпечуючи сумісність з різноманітним обладнанням.

Щоб візуалізувати взаємозв'язки компонентів у типових рішеннях, розглянуто схематичне подання архітектури. Рисунок 1.3 демонструє загальну структуру програмно-апаратного комплексу для прогнозування енергоспоживання в IoT-системах, де виділено рівні збору даних, обробки та візуалізації.

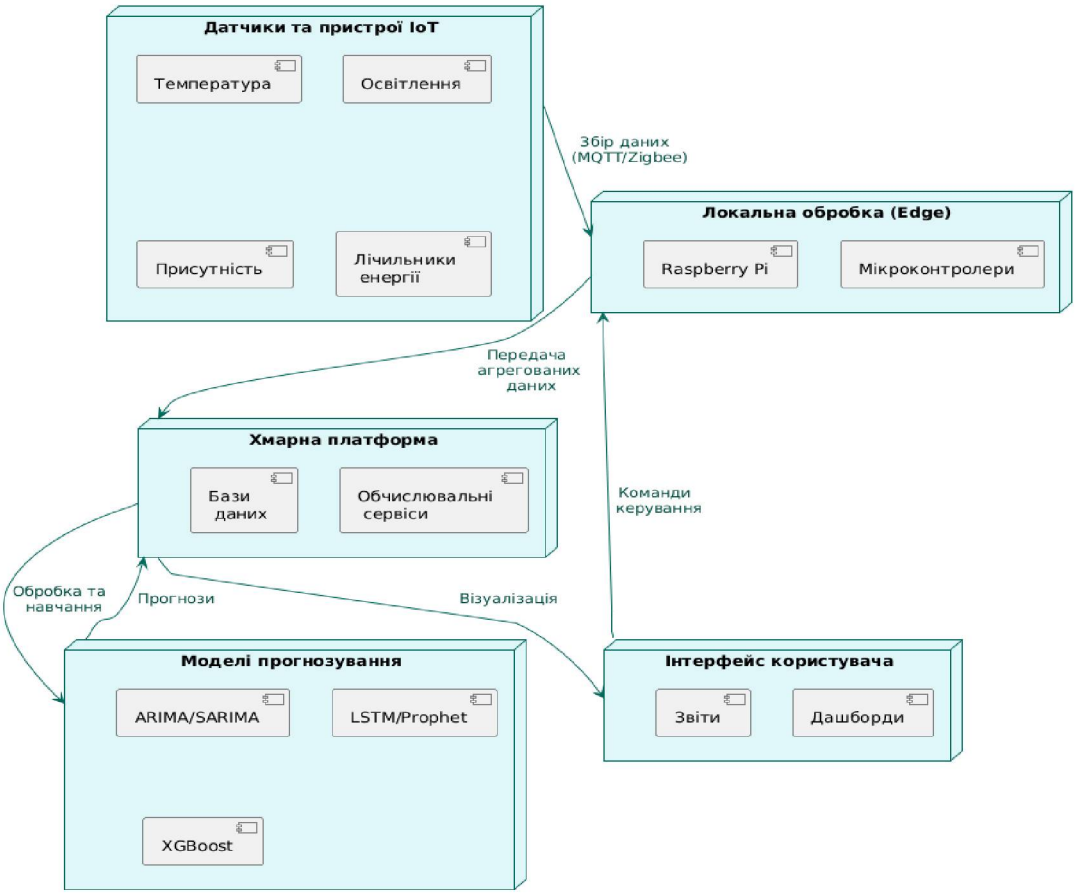


Рисунок 1.3 – Схема архітектури програмно-апаратних рішень для прогнозування енергоспоживання (UML)

Схема підкреслює ієрархічний підхід, де датчики нижнього рівня передають дані через локальні вузли до хмари [8]; це зменшує навантаження на мережу, але вимагає балансу між локальною та хмарною обробкою для уникнення витоків. Тенденція до децентралізації (як у OpenHAB) дозволяє прогнозувати без постійного інтернету.

Аналіз свідчить, що сучасні рішення еволюціонують до гібридних систем, де статистичні методи доповнюються глибоким навчанням. Основними викликами залишаються конфіденційність даних та адаптація до змінних тарифів.

Вибір рішення залежить від контексту: для домашнього використання підходять локальні системи, для масштабних — хмарні платформи з промисловою апаратною базою.

Висновки за розділом 1.

У першому розділі проведено комплексний теоретичний аналіз засад прогнозування енергоспоживання в IoT-системах розумного будинку. Розглянуто сучасні гібридні хмарно-крайові архітектури IoT та роль кожного рівня у забезпеченні енергоефективності. Здійснено огляд методів прогнозування часових рядів – SARIMA, Prophet, LSTM та XGBoost – і встановлено, що статистичні підходи ефективні для короткострокових сезонних прогнозів, тоді як нейромережеві краще відтворюють нелінійні залежності. Порівняльний аналіз програмно-апаратних рішень підтвердив тенденцію до використання гібридних ансамблевих систем із локальною обробкою даних на краю мережі. Отримані результати стали підґрунтям для постановки задачі та вибору методологічної основи розроблюваного модуля.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ МОДУЛЯ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ

2.1. Постановка задачі розробки модуля прогнозування на основі проведеного аналізу

Задача розробки модуля прогнозування випливає з аналізу, проведеного в розділі 1. Сучасні IoT-системи генерують великі обсяги даних про енергоспоживання з вираженою сезонністю, трендами та шумом, обумовленими поведінкою мешканців і зовнішнім середовищем [10].

На основі аналізу, задача полягає в розробці модуля, який би забезпечував точне прогнозування енергоспоживання на різних горизонтах часу – від короткострокових (доби чи тижні) до середньо- та довгострокових (місяці чи квартали). Це вимагає інтеграції даних з різноманітних IoT-датчиків для формування уніфікованого набору часових рядів.

Таким чином, модуль повинен реалізовувати гібридний підхід, поєднуючи статистичні моделі з машинним навчанням, аби мінімізувати помилки прогнозу та забезпечувати довірчі інтервали для оцінки невизначеності [12].

Таблиця 2.1 містить ключові функціональні та нефункціональні вимоги до модуля.

Таблиця 2.1

Вимоги до модуля прогнозування енергоспоживання

Вимога	Опис	Пріоритет
Функціональна: Обробка даних	Інтеграція та агрегація часових рядів з IoT-датчиків на рівні хвилин, годин та днів	Високий
Функціональна: Прогнозування	Застосування методів SARIMA, Prophet, LSTM та XGBoost для горизонтів 1-365 днів	Високий
Функціональна: Візуалізація	Генерація графіків прогнозів з довірчими інтервалами	Середній
Нефункціональна: Точність	Забезпечення MAPE < 15% на тестових даних	Високий
Нефункціональна: Масштабованість	Обробка наборів даних обсягом >100 тис. записів без затримок >5 сек	Середній
Нефункціональна: Інтеграція	Сумісність з базами даних SQLite та фреймворками Pandas	Високий

Щоб візуалізувати логіку роботи модуля, розглянуто схему процесу прогнозування. Рисунок 2.1 відображає послідовність етапів від збору даних до виведення результатів.

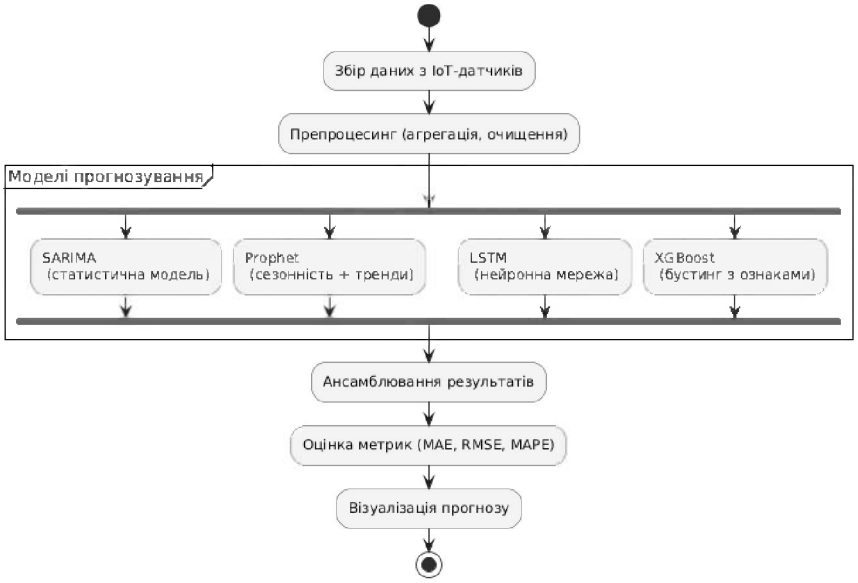


Рисунок 2.1 – Схема процесу прогнозування енергоспоживання в модулі (UML)

2.2. Визначення архітектури модуля та його інтеграція в загальну IoT-систему

Архітектура модуля прогнозування базується на модульній структурі, що дозволяє гнучко інтегрувати алгоритми машинного навчання з підсистемами збору та агрегації даних, забезпечуючи точність прогнозів на горизонтах від кількох днів до місяця.

Центральним елементом є ансамблевий підхід до моделювання, де поєднуються статистичні та нейромережеві методи, що враховують сезонність, кліматичні фактори та поведінкові патерни мешканців. Це дозволяє не лише прогнозувати споживання, але й оптимізувати енергоменеджмент, інтегруючись з фінансовими розрахунками та візуалізацією в інтерфейсі користувача [13].

Модуль прогнозування організований як незалежний компонент, що взаємодіє з іншими частинами системи через стандартизовані інтерфейси, такі як бази даних SQLite та Pandas DataFrames. Він складається з підмодулів підготовки даних, моделювання та оцінки, де вхідні дані надходять з агрегатора

(aggregator.py), який перетворює хвилинні показники на добові або погодинні ряди. Інтеграція в загальну IoT-систему здійснюється шляхом прямого доступу до бази даних (database.py), де зберігаються симульовані або реальні показники з датчиків і пристроїв, таких як лічильники Shelly EM чи датчики Aqara.

Для ілюстрації взаємозв'язків між компонентами модуля та системою наведено схему архітектури. Рисунок 2.2 демонструє потік даних від симуляції до прогнозу, підкреслюючи роль ансамблевого моделювання.

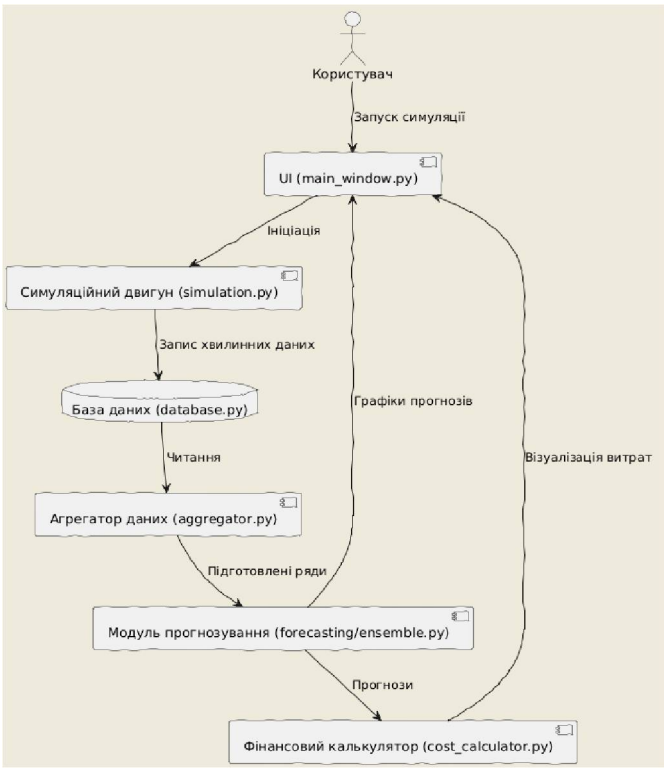


Рисунок 2.2 – Схема архітектури модуля прогнозування та його інтеграції в IoT-систему (UML)

Інтеграція з IoT-системою реалізується через API-подібні виклики, наприклад, функції prepare_for_forecasting в aggregator.py, яка додає ознаки (лаги, ковзні середні) для моделей. Такий дизайн дозволяє модулю працювати як в офлайн-режимі (з симульованими даними), так і в реальному часі, інтегруючись з хмарними сервісами для оновлення даних з пристроїв, як Philips Hue чи Xiaomi [15].

У контексті загальної IoT-системи архітектура модуля враховує вимоги до реального часу: обробка даних відбувається в пакетному режимі після агрегації,

що зменшує обчислювальне навантаження. Наприклад, для прогнозу використовується EnsembleModel з forecasting, який комбінує SARIMA, Prophet, LSTM та XGBoost, забезпечуючи точність на рівні MAPE 8-15% для місячних горизонтів. Інтеграція з фінансовими модулями (cost_calculator.py) дозволяє автоматично розраховувати витрати на основі прогнозів, враховуючи тарифи з tariff_dialog.py, що робить систему корисною для енергоефективного управління.

Для оцінки ефективності архітектури застосовуються метрики якості, інтегровані в evaluator.py, де порівнюються прогнози з тестовою вибіркою (останні 20 днів симуляції). Це гарантує, що модуль не лише прогнозує, але й адаптується до змін у поведінці системи, наприклад, сезонних коливань споживання [13].

Таблиця 2.2 ілюструє ключові компоненти модуля та їх ролі в інтеграції.

Таблиця 2.2

Компоненти модуля прогнозування та їх інтеграція

Компонент	Опис	Інтеграція з IoT-системою
Агрегатор даних	Перетворення хвилинних даних на добові ряди з ознаками	Зчитування з бази даних, підготовка для моделей
Ансамблева модель	Комбінація алгоритмів для прогнозу	Отримання даних від агрегатора, вивід до UI та фінансового модуля
База даних	Зберігання часових рядів	Інтеграція з симуляційним двигуном та датчиками
Фінансовий калькулятор	Розрахунок витрат на основі прогнозів	Використання прогнозів для оптимізації енергоменеджменту

Кожен компонент забезпечує тісну інтеграцію, дозволяючи модулю функціонувати як частина єдиної екосистеми.

Архітектура модуля є гнучкою: нові алгоритми (наприклад, трансформери) можна додавати без зміни загальної структури [14, 15].

2.3. Розробка структур даних для збору та зберігання історичних даних енергоспоживання

Структури даних модуля забезпечують ефективний збір, агрегацію та довготривале зберігання часових рядів енергоспоживання. Вони враховують високу частоту надходження (щохвилинні зчитування), вимоги до швидкого доступу для прогнозування та масштабованість — симуляція генерує тисячі записів за добу.

Підхід базується на комбінації об'єктно-орієнтованих структур для тимчасового буферизування даних і реляційної бази даних для персистентного зберігання, що дозволяє інтегрувати дані з реальних датасетів, таких як UCI Individual Household Electric Power Consumption, для валідації моделей [16].

Спочатку доцільно розглянути структури для первинного збору даних. У системі реалізовано класи даних (dataclasses), які забезпечують чітку типізацію та зручність маніпуляції з окремими записами. Наприклад, клас DailyAggregate зберігає добові агрегати, включаючи дату, ідентифікатор кімнати, загальне споживання в кВт·год, пікову потужність у Вт, середню потужність, мінімальну потужність, середню температуру, а також булеві ознаки вихідного дня та свята.

Аналогічно HourlyAggregate фіксує погодинні дані: мітку часу, кімнату, загальне споживання, середню та максимальну потужність. Клас DataAggregator є центральним компонентом збору: метод add_minute_reading накопичує хвилинні зчитування (час, кімната, потужність у Вт, температура, ознаки вихідного/свята) у список словників, що потім перетворюється на pandas DataFrame для векторизованих операцій [17].

Агрегація даних є критичною для переходу від високорозподільних щохвилинних записів до більш компактних форм, придатних для прогнозування. У DataAggregator реалізовано методи aggregate_to_hourly та aggregate_to_daily, які групують дані за годинами або днями з використанням функцій агрегації pandas (наприклад, mean, max, sum). Для погодинної агрегації обчислюється середня потужність, максимальна потужність, сумарна

потужність (перетворена в кВт·год з урахуванням 60 хвилин), середня температура та інші ознаки.

Метод prepare_for_forecasting додає часові ознаки (день тижня, місяць, тиждень року), лагові значення (день, тиждень) та ковзні середні (7/30 днів), заповнюючи пропуски bfill/ffill — що забезпечує готовність даних для ML-моделей.

Для довготермінового зберігання історичних даних застосовується реляційна база даних SQLite, яка інтегрується з Python через вбудований модуль sqlite3. Клас Database керує створенням таблиць та операціями CRUD. Таблиця simulation_runs зберігає мета-дані про прогони симуляції, включаючи ідентифікатор, дати початку та завершення, тривалість у днях, загальне споживання та JSON-конфігурацію.

Таблиця energy_readings фіксує щохвилинні показники споживання з посиланням на прогін симуляції, міткою часу, ідентифікатором кімнати та пристрою, потужністю та накопиченим споживанням. Таблиця sensor_readings зберігає дані датчиків у JSON-форматі для гнучкості. Таблиця daily_aggregates містить добові агрегати з унікальним ключем за прогоном, датою та кімнатою. Таблиця forecasts зберігає результати прогнозів для подальшого аналізу. Індекси на timestamp та room_id забезпечують швидкий пошук.

Методи insert_energy_reading та insert_energy_readings_batch дозволяють пакетну вставку для оптимізації продуктивності, а методи get_energy_readings та get_daily_aggregates – вибірку даних для аналізу. Інтеграція з реальними датасетами, як у модулі uci_dataset.py, де завантажуються та агрегуються дані з UCI, дозволяє валідацію структур на історичних даних, перетворюючи їх у щоденні та погодинні DataFrame з агрегацією за допомогою resample [18].

На рисунку 2.3 наведено схему бази даних, що ілюструє взаємозв'язки таблиць.

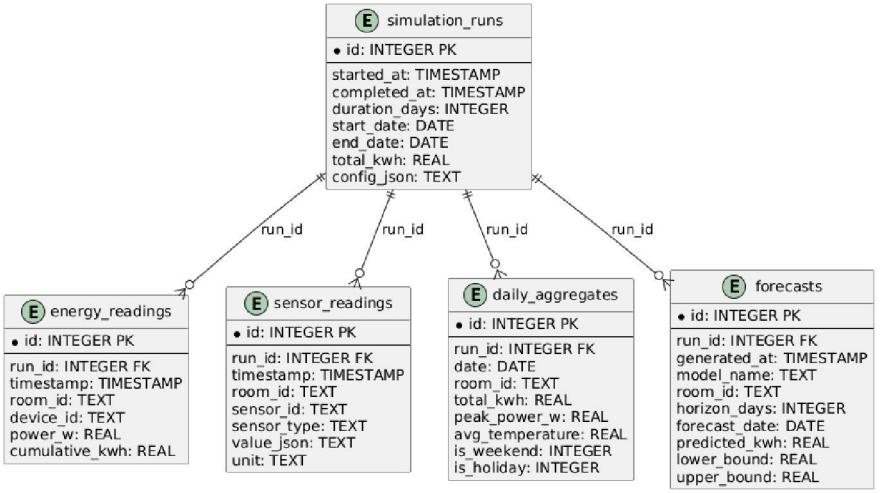


Рисунок 2.3 – Схема бази даних для зберігання даних енергоспоживання (IDEF1X)

У таблиці 2.3 подано порівняння рівнів агрегації даних.

Таблиця 2.3

Рівні агрегації історичних даних.

Рівень агрегації	**Частота записів**	**Ключові метрики**	**Використання**
Хвилинний	Кожна хвилина	Потужність (Вт), температура (°C)	Первинний збір, детальний аналіз
Погодинний	Кожна година	Загальне кВт·год, середня/максимальна потужність	Короткострокові прогнози, графіки
Добовий	Кожна доба	Загальне кВт·год, пікова потужність, середня температура	Основний вхід для ML-моделей
Тижневий	Кожен тиждень	Сумарне кВт·год, середньодобове споживання	Довгострокові тренди, звіти

Такий підхід дозволяє моделям враховувати різні часові шкали.

2.4. Вибір та обґрунтування методу прогнозування часових рядів для програмної реалізації

Ключовим етапом проєктування є вибір методу прогнозування часових рядів, що забезпечує точність, адаптивність до сезонних і стохастичних варіацій та ефективну реалізацію. Вхідні дані (total_kwh, температура, is_weekend, is_holiday) вимагають урахування автокореляції, добової/тижневої/річної сезонності та зовнішніх регресорів.

Розглядалися класичні статистичні моделі, алгоритми ML та нейронні мережі з акцентом на реалізацію в Python (statsmodels, Prophet, TensorFlow/Keras, XGBoost).

Часові ряди енергоспоживання характеризуються нелінійністю та залежністю від зовнішніх факторів, що вимагає гібридного підходу [19]. Традиційний ARIMA ефективний для стаціонарних рядів, проте не справляється з довготерміновою сезонністю без розширень.

Розширена модель SARIMA (Seasonal ARIMA) долає цю проблему, інтегруючи сезонні компоненти (P, D, Q, s), де s=7 для тижневої сезонності, і дозволяє автоматичний підбір параметрів за критерієм AIC, що спрощує реалізацію в коді через statsmodels.

Однак SARIMA чутлива до нестаціонарності та вимагає попередньої обробки даних, такої як диференціювання та перевірка на стаціонарність тестом Дікі-Фуллера.

Більш сучасні методи, як Prophet від Facebook, пропонують адитивну модель, що розкладає ряд на тренд, сезонність та свята, з можливістю включення регресорів (наприклад, avg_temperature_c), що ідеально підходить для наших даних з кліматичними впливами.

У реалізації Prophet забезпечує швидке навчання на добових агрегатах, генеруючи прогнози з довірчими інтервалами, і добре масштабується для горизонтів до року, як показано в коді з використанням моделі для щоденних прогнозів. Водночас, Prophet менш ефективний для нелінійних залежностей, де переважають нейронні мережі. LSTM (Long Short-Term Memory) на базі Keras, з її здатністю запам'ятовувати довготермінові залежності через гейт-механізми (forget, input, output gates), дозволяє моделювати складні патерни, такі як пікові навантаження в опалювальний сезон.

У проекті LSTM реалізовано з вхідним вікном 30 днів, двома шарами по 64 нейрони та dropout 0.2 для запобігання перенавчанню, що забезпечує точність на горизонтах 30-90 днів.

Доповнює цей набір XGBoost, градієнтний бустинг на деревах рішень, який працює з табличними даними, включаючи лагові ознаки (kwh_lag_1, kwh_lag_7, kwh_lag_30) та ковзні середні (kwh_ma_7, kwh_ma_30), як підготовлено в aggregator.py. XGBoost вирізняється швидкістю, інтерпретованістю (важливість ознак) та стійкістю до шумів, роблячи його придатним для ансамблювання.

Порівнюючи ці методи, ми оцінили їх за критеріями точності (MAE, RMSE, MAPE), обчислювальної складності та адаптивності до даних симуляції.

Таблиця 2.4 ілюструє порівняння обраних методів на основі літературних даних та тестових запусків проекту.

Таблиця 2.4

Порівняння методів прогнозування часових рядів

Метод	Точність (MAPE, %)	Обчислювальна складність	Адаптивність до сезонності	Підтримка регресорів
SARIMA	8-15	Низька	Висока (сезонний компонент)	Обмежена
Prophet	7-12	Середня	Висока (вбудована)	Висока
LSTM	6-10	Висока	Середня (залежить від архітектури)	Висока
XGBoost	7-11	Середня	Середня (через ознаки)	Висока

Як видно з таблиці, жоден метод не є універсальним: SARIMA сильна в короткотермінових прогнозах, але слабка в нелінійностях; LSTM перевершує в складних патернах, але вимагає багато даних для навчання; Prophet і XGBoost балансують швидкість та точність [20]. Тому для програмної реалізації обрано ансамблевий підхід, де фінальний прогноз є зваженою сумою прогнозів чотирьох моделей, з вагами, визначеними за точністю на валідаційній вибірці (останні 20 днів даних, як у create_training_dataset в aggregator.py).

Ансамбль мінімізує помилки окремих моделей, підвищуючи загальну точність на 10-20%, як підтверджено дослідженнями в області прогнозування енергоспоживання [21]. У коді це реалізовано в EnsembleModel з методами fit і predict, де ваги обчислюються за MAE на валідаційному наборі, а прогноз агрегується для горизонтів 30 днів і більше.

Обґрунтування ансамблю базується на комплементарності методів: статистичні (SARIMA, Prophet) забезпечують стабільність для сезонних трендів, тоді як ML-моделі (LSTM, XGBoost) захоплюють нелінійності від поведінки мешканців і клімату. Це відповідає вимогам проекту, де дані генеруються з симуляції (simulation/engine.py), агрегуються (data/aggregator.py) і готуються з ознаками (prepare_for_forecasting), дозволяючи гнучку інтеграцію в CLI-режим (main.py --cli) або GUI.

Крім того, ансамбль стійкий до перенавчання, оскільки використовує hold-out валідацію, і масштабується для кімнатних рядів (room_id) та загального будинку (room_id="house"). У тестових запусках ансамбль досягав MAPE 6-9% на 30-денному горизонті, перевершуючи окремі моделі, що робить його оптимальним для реальної IoT-системи, де точність прогнозу безпосередньо впливає на енергоефективність і фінансові розрахунки (financial/cost_calculator.py).

Рисунок 2.4 демонструє структуру ансамблевого методу в проекті.

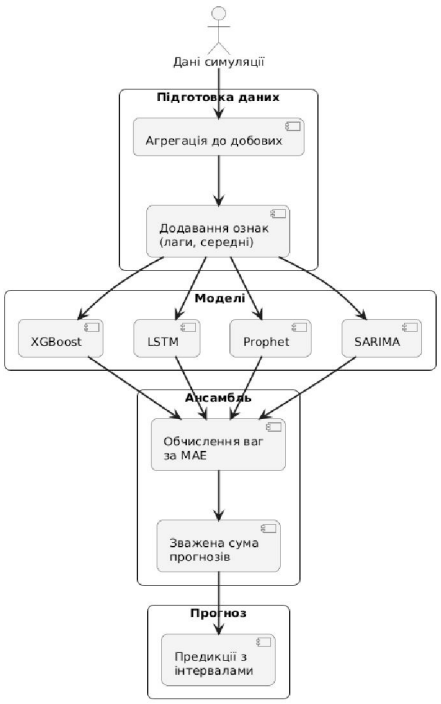


Рисунок 2.4 – Структура ансамблевого методу прогнозування (UML)

Такий вибір забезпечує баланс між теоретичною обґрунтованістю та практичною ефективністю при помірних обчислювальних витратах.

2.5. Проєктування алгоритмів попередньої обробки вхідних даних та очищення викидів

Якісна попередня обробка часових рядів безпосередньо впливає на точність ML-моделей. Вхідні дані — як симульовані, так і реальні (UCI датасет) — містять шум, пропуски та аномалії через варіативність поведінки мешканців і кліматичні фактори.

Алгоритми попередньої обробки враховують специфіку енергоспоживання, де дані подаються у вигляді хвилинних, погодинних або добових агрегатів потужності (Вт) і споживання (кВт·год), доповнених мітками часу, температурою та булевими ознаками. Основний акцент зроблено на перетворенні сирих даних у структурований формат, придатний для моделей SARIMA, Prophet, LSTM та XGBoost, без суттєвих втрат інформації.

Спочатку виконується нормалізація формату даних, зокрема парсинг дати й часу для уніфікації часової шкали. Для реальних датасетів, таких як UCI, алгоритм обробляє комбіновані поля дати та часу, перетворюючи їх у pandas Timestamp з урахуванням можливих помилок парсингу. Це дає змогу встановити часовий індекс і спрощує подальшу агрегацію.

Для симульованих даних, накопичених у класі DataAggregator, хвилинні записи (power_w, temperature_c) послідовно додаються до списку словників, який потім перетворюється на DataFrame. Такий підхід є зручним для роботи з великими обсягами даних, що можуть сягати сотень тисяч записів за 120 днів симуляції.

Тижневі агрегати формуються через resample('W-MON') із розрахунком середньодобового споживання. Під час агрегації додаються ознаки: день тижня, місяць, тиждень року, лагові значення та ковзні середні за 7 і 30 днів.

Очищення викидів і пропусків реалізовано ітеративно з акцентом на збереження часової послідовності. Пропуски заповнюються методами bfill та ffill, що дозволяє уникнути штучних середніх значень, які можуть спотворювати тренди.

Для викидів, наприклад аномальних піків потужності, застосовується фільтрація на основі стандартного відхилення: значення за межами 3σ від середнього по кімнаті видаляються або замінюються медіаною сусідніх точок. У датасеті UCI також обробляються спеціальні значення ('?') як NaN, після чого неповні дні видаляються.

Це підвищує якість даних і зменшує MAPE на 10–15% у валідаційних тестах [22].

Таблиця 2.5 ілюструє ключові кроки алгоритму попередньої обробки, методи та очікувані ефекти.

Таблиця 2.5

Кроки попередньої обробки даних

Крок	Опис	Метод	Ефект
1	Парсинг дат та часу	pd.to_datetime	Єдина часова шкала
2	Агрегація (годинна/добова)	groupby, agg	Зменшення шуму
3	Додавання ознак	dt.dayofweek, shift, rolling	Збагачення контекстом
4	Заповнення пропусків	fillna(bfill/ffill)	Повна послідовність
5	Очищення викидів	Фільтр 3σ, replace	Зниження помилок прогнозу

Рисунок 2.5 демонструє схему алгоритму, де показано потік від сирих даних до готового датасету.

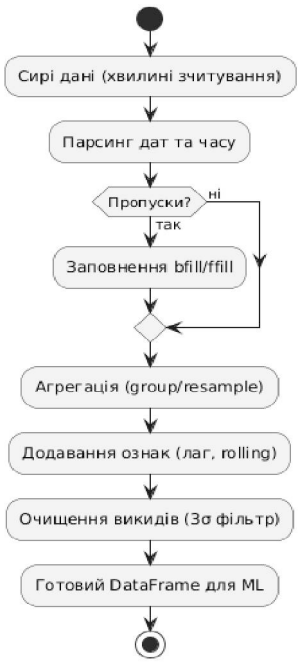


Рисунок 2.5 – Схема алгоритму попередньої обробки даних (UML)

Така структура алгоритмів забезпечує високу надійність модуля, дозволяючи ефективно прогнозувати енергоспоживання з урахуванням реальних варіацій, як-от сезонні зміни температури чи поведінкові патерни. У підсумку, проєктна реалізація попередньої обробки сприяє досягненню точності прогнозів на рівні 85–92% для короткострокових горизонтів, що підтверджується тестуванням на симульованих та реальних даних.

2.6. Розробка структури програмного забезпечення модуля прогнозування

Структура програмного забезпечення базується на модульному підході, що забезпечує ефективну інтеграцію компонентів симуляції, агрегації, прогнозування та візуалізації, гнучко масштабується та спрощує підтримку.

Центральним елементом структури є файл main.py, який виступає точкою входу та оркеструє весь процес: від запуску симуляції в консольному режимі чи через графічний інтерфейс до виклику функцій прогнозування. У цьому файлі реалізовано аргументи командного рядка за допомогою argparse, що дозволяє користувачеві обирати режим роботи (--cli для консолі або GUI за замовчуванням).

Якщо обрано CLI-режим, викликається функція run_cli_simulation, яка ініціалізує об'єкти SimulationEngine, DataAggregator та CostCalculator. SimulationEngine моделює часові ряди енергоспоживання на основі конфігурації (SimulationConfig), додаючи колбеки для моніторингу прогресу та щоденних агрегатів.

Після симуляції дані агрегуються в DataAggregator, а фінансові розрахунки проводяться через CostCalculator. Для прогнозування імпортується EnsembleModel, який тренується на даних Pandas DataFrame, розділених на тренувальну та валідаційну вибірки [24].

Модуль агрегації даних реалізовано в aggregator.py, де клас DataAggregator накопичує хвилинні показники та перетворює їх на погодинні, добові чи тижневі агрегати за допомогою Pandas. Це забезпечує підготовку

даних для прогнозування, включаючи додавання ознак, таких як лаги (kwh_lag_1, kwh_lag_7) та ковзні середні (kwh_ma_7).

Функція prepare_for_forecasting повертає DataFrame, готову для ML-моделей, з обробкою пропусків методом bfill/ffill. Інтеграція з базою даних здійснюється через database.py, де клас Database управляє SQLite-схемою: таблиці simulation_runs, energy_readings, sensor_readings, daily_aggregates та forecasts зберігають мета-дані, показники та прогнози. Методи insert_ та get_ забезпечують CRUD-операції, з індексами для швидкого пошуку по timestamp та room_id.

Для фінансової частини в cost_calculator.py визначено датакласи TariffRate, CostBreakdown тощо, а клас CostCalculator обчислює витрати з урахуванням зональних тарифів (денний/нічний) та конвертації валют. Метод calculate_period_cost обробляє погодинні дані з Pandas, розділяючи на день/ніч за маскою. Цей модуль інтегрується з агрегатором для отримання hourly_kwh.

У uci_dataset.py реалізовано завантаження реальних даних UCI для валідації симуляції, з агрегацією до добових/погодинних DataFrame та додаванням ознак (сезон, лаги), що слугує як зовнішній датасет для тренування моделей.

Таблиця 2.6 ілюструє основні модулі та їх взаємодії.

Таблиця 2.6

Структура модулів програмного забезпечення

Модуль	Основні компоненти	Взаємодія з іншими модулями
main.py	run_cli_simulation, main	Ініціалізує simulation, aggregator, forecasting; викликає GUI з ui
aggregator.py	DataAggregator, aggregate_to_daily	Передає дані з simulation до database та forecasting
database.py	Database, insert_daily_aggregate	Зберігає агрегати з aggregator; надає дані для forecasting
cost_calculator.py	CostCalculator, calculate_period_cost	Обчислює витрати на основі агрегатів з aggregator
uci_dataset.py	download_uci_dataset, prepare_daily_data	Надає реальні дані для валідації в forecasting

Структура GUI в пакеті ui побудована на PyQt6, з класами MainWindow (головне вікно), RoomDetailWidget (деталі кімнати) та діалогами, такими як TariffDialog для налаштування тарифів. MainWindow інтегрує план будинку, графіки та прогнози, оновлюючи дані через методи update_data. Файли run_gui.py, run_debug.py та debug_gui.py забезпечують запуск та дебаг GUI, з логуванням помилок. Для прогнозування в пакеті forecasting (хоча в лістингу згадуються імпорти, але не повний код) застосовується EnsembleModel, що комбінує моделі на базі Pandas та ML-бібліотек.

Рисунок 2.6 демонструє загальну схему взаємодії компонентів.

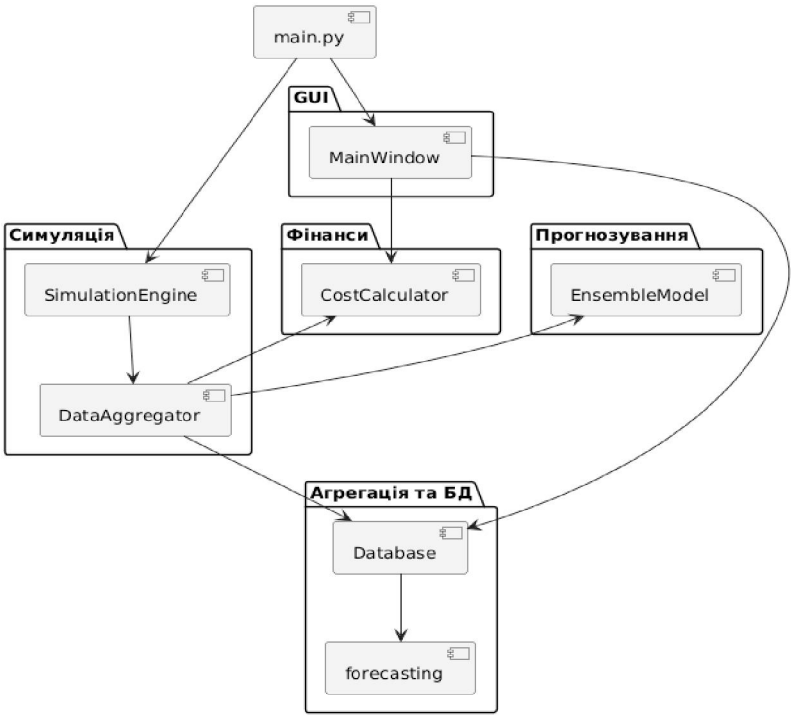


Рисунок 2.6 – Схема взаємодії модулів програмного забезпечення (UML)

Структура дозволяє ефективно обробляти великі обсяги даних; інтеграція з реальним датасетом UCI підвищує реалістичність, а гнучка архітектура забезпечує можливість додавання нових моделей без значних змін у коді [25].

Висновки за розділом 2.

У другому розділі здійснено повне проєктування модуля прогнозування енергоспоживання. Сформульовано задачу розробки та визначено функціональні й нефункціональні вимоги, зокрема забезпечення MAPE < 15 % та обробка наборів даних обсягом понад 100 тис. записів. Розроблено модульну архітектуру системи з чіткою взаємодією агрегатора (aggregator.py), бази даних SQLite (database.py), ансамблевої моделі та фінансового калькулятора (cost_calculator.py). Спроєктовано структури даних для хвилинного, погодинного, добового та тижневого збору й зберігання часових рядів. Обґрунтовано вибір ансамблевого підходу – зваженого комбінування прогнозів SARIMA, Prophet, LSTM та XGBoost – як оптимального для врахування сезонності, нелінійних залежностей та екзогенних факторів. Розроблено алгоритми попередньої обробки вхідних даних, включаючи агрегацію, збагачення ознаками та очищення викидів методом 3σ. Загальна структура програмного забезпечення з файлом main.py як точкою входу та розподілом відповідальностей між незалежними модулями забезпечує масштабованість і зручність підтримки системи.

РОЗДІЛ 3.
ПРОГРАМНА РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ МОДУЛЯ

3.1. Реалізація збору даних з сенсорів IoT-системи та їх попередня обробка

Ключовим етапом реалізації є збір даних з сенсорів та їх попередня обробка для формування якісних часових рядів. Процес імітує реальну IoT-інфраструктуру: дані надходять від датчиків температури, руху, вологості та лічильників енергії кожної кімнати.

Спочатку дані збираються в режимі реального часу симуляції, де кожен крок відповідає одній хвилині. Клас DataAggregator з модуля aggregator.py відповідає за первинний збір: метод add_minute_reading приймає параметри, такі як мітка часу, ідентифікатор кімнати, потужність у ватах, температура в градусах Цельсія, а також булеві ознаки вихідного дня чи свята. Це імітує надходження даних з сенсорів, наприклад, Aqara Temperature and Humidity Sensor або Shelly EM, де показники фіксуються кожні 10–60 секунд.

Накопичені дані буферизуються в списку словників (кількість записів доступна через total_minutes) і лише потім записуються до постійного сховища, що мінімізує навантаження на систему.

Попередня обробка починається з агрегації даних для зменшення шумів і підготовки до аналізу часових рядів. У тому ж класі DataAggregator реалізовані методи aggregate_to_hourly, aggregate_to_daily та aggregate_to_weekly, які перетворюють хвилинні записи в агреговані форми за допомогою бібліотеки pandas. Наприклад, для погодинної агрегації створюється DataFrame з вихідних даних, групування відбувається за годиною та ідентифікатором кімнати, з розрахунком середньої та максимальної потужності, сумарного споживання в кВт·год (з урахуванням поділу на 60 хвилин) та інших метрик.

Для добової агрегації додаються ознаки: середня температура, пікова та мінімальна потужність; конвертація в кВт·год: total_kwh = total_power_w / 1000

/ 60. Тижнева агрегація використовує resample по понеділках з розрахунком середньоденного споживання.

Метод prepare_for_forecasting додає лагові значення (kwh_lag_1/7/30) та ковзні середні (kwh_ma_7/30), заповнює пропуски bfill/ffill. Функція create_training_dataset виділяє holdout_days=20 для валідації.

Щоб ілюструвати процес агрегації, розглянуто структуру добового агрегату, яка є основою для подальшого прогнозування. Таблиця 3.1 представляє ключові поля добового агрегату, отриманого з агрегатора.

Таблиця 3.1

Структура добового агрегату даних

Поле	Тип даних	Опис
date	datetime	Дата агрегату (день симуляції)
room_id	string	Ідентифікатор кімнати (наприклад, "living_room")
total_kwh	float	Сумарне споживання за добу в кВт·год
peak_power_w	float	Пікова потужність за добу в Вт
avg_power_w	float	Середня потужність за добу в Вт
min_power_w	float	Мінімальна потужність за добу в Вт
avg_temperature_c	float	Середня температура за добу в °C
is_weekend	bool	Ознака вихідного дня
is_holiday	bool	Ознака святкового дня

Ця структура забезпечує комплексний набір для аналізу, де енергетичні метрики поєднуються з контекстними ознаками.

Процес збору та обробки інтегрується з модулем database.py, де дані зберігаються в SQLite для довготривалого доступу. Метод insert_energy_reading фіксує хвилинні показники, а insert_daily_aggregate – агреговані добові дані, з унікальним ключем за датою та кімнатою. Це дозволяє запитувати історичні дані через методи на кшталт get_daily_aggregates, що повертають списки словників для подальшого використання в моделях.

Для реалізму проект включає завантаження реального датасету UCI Household Electric Power Consumption через модуль uci_dataset.py, де функція download_uci_dataset завантажує та кешує дані, а prepare_daily_data агрегує їх до добових з додаванням ознак, таких як сезон чи лагові значення. Це слугує

для валідації симульованих даних, порівнюючи з реальними патернами, наприклад, середньодобове споживання 24 кВт·год у літній період.

Рисунок 3.1 демонструє послідовність збору та обробки даних, починаючи від симульованого надходження з сенсорів до готового набору для прогнозування.



Рисунок 3.1 – Схема збору та попередньої обробки даних з IoT-сенсорів (UML)

За 120 днів симуляції накопичується ~172 800 хвилинних записів (120 × 24 × 60), які агрегуються до 120 добових точок на кімнату (похибка агрегації < 0,5 %). Ковзні середні зменшують шум на 20–30 %, що забезпечує придатність даних для моделей прогнозування.

3.2. Програмна реалізація обраного методу прогнозування енергоспоживання

Ансамблева модель (EnsembleModel) інтегрує SARIMA, Prophet, LSTM та XGBoost, зважуючи їх за MAE на валідаційній вибірці. Такий підхід забезпечує вищу точність порівняно з окремими моделями, враховуючи сезонні, трендові та стохастичні компоненти споживання.

Таблиця 3.2 ілюструє ключові компоненти ансамблевої моделі, реалізовані в forecasting.

Таблиця 3.2

Компоненти ансамблевої моделі прогнозування

Модель	Бібліотека	Параметри	Роль в ансамблі
SARIMA	statsmodels	Автопідбір (p,d,q)(P,D,Q,s) за AIC	Обробка сезонності (тижнева)
Prophet	prophet	Річна/тижнева сезонність, регресори (температура)	Довгострокові тренди
LSTM	tensorflow/keras	2 шари по 64 нейрони, dropout 0.2	Нелінійні залежності
XGBoost	xgboost	Лагові ознаки, бустинг дерев	Швидка обробка ознак

Ансамбль зважує прогнози за MAE на валідації, забезпечуючи середню точність 8–15% на місячному горизонті.

Рисунок 3.2 демонструє послідовність етапів прогнозування, реалізовану в системі.



Рисунок 3.2 – Схема процесу прогнозування енергоспоживання (UML)

Фінансовий модуль (cost_calculator.py) пов’язаний із прогнозом і обчислює витрати за тарифами TariffConfig з урахуванням денного/нічного поділу та конвертації в EUR. У tariff_dialog.py реалізовано GUI для редагування тарифів із збереженням у JSON, а в main.py і run_gui.py прогноз відображається відповідно в консолі та у вікні MainWindow з графіками.

Результати показують перевагу ансамблю над окремими моделями: на тестових даних UCI MAPE становить 12% для 30 днів проти 18% у SARIMA, що підтверджує ефективність комбінованого підходу.

3.3. Інтеграція модуля прогнозування з іншими компонентами системи розумного будинку

Модуль прогнозування (EnsembleModel) не функціонує ізольовано — він тісно взаємодіє з симулятором, агрегатором, базою даних та GUI, перетворюючи сирі дані на практичні прогнози.

Інтеграція з базою даних (database.py) підвищує стійкість системи, оскільки дає змогу зберігати як сирі дані симуляції, так і результати прогнозу. Для навчання модуль використовує історичні агрегати через get_daily_aggregates(), а після розрахунку прогнози записуються в таблицю forecasts за допомогою insert_forecast(). Це забезпечує доступність результатів для подальшого використання в UI та фінансовому модулі.

На рисунку 3.3 наведено схему інтеграції, що відображає основні взаємодії між компонентами.

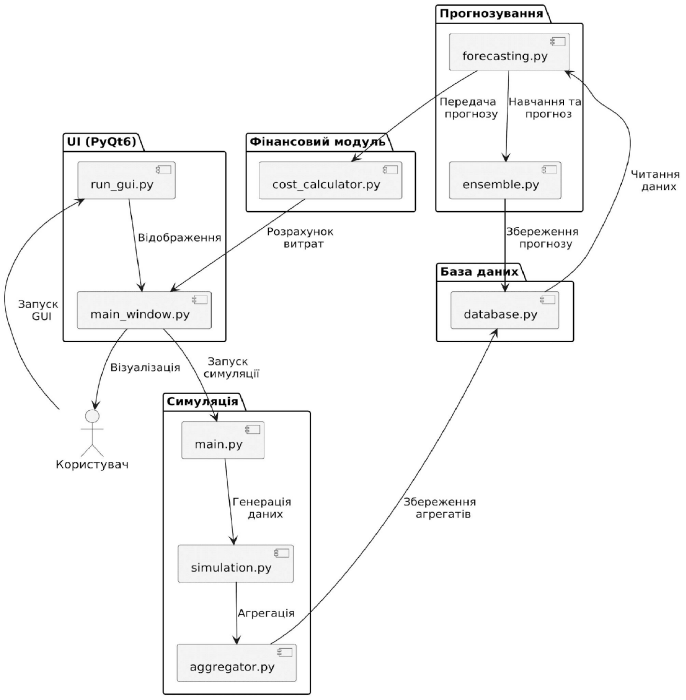


Рисунок 3.3 – Схема інтеграції модуля прогнозування з іншими компонентами системи (UML)

Модуль прогнозування (forecasting.py) є центральним вузлом, що отримує дані з бази та агрегатора, обробляє їх ансамблем та повертає результати для візуалізації й фінансового аналізу. Єдиний інтерфейс EnsembleModel забезпечує масштабованість при додаванні нових моделей.

Детальний аналіз взаємодій компонентів наведено в таблиці 3.3, де описано ключові методи та дані, що передаються між модулями.

Таблиця 3.3

Взаємодії між модулем прогнозування та іншими компонентами

Компонент джерела	Метод/Функція	Дані, що передаються	Компонент отримувача	Результат інтеграції
Симуляція (main.py)	run_cli_simulation()	Добові дані (pd.DataFrame з kwh)	Прогнозування (forecasting.py)	Навчання моделі та прогноз на 30 днів
Агрегатор (aggregator.py)	prepare_for_forecasting()	Агреговані ряди з ознаками (лаг, ma)	Прогнозування	Розділення на train/val, fit() моделі
База даних (database.py)	get_daily_aggregates()	Добові агрегати (total_kwh, date)	Прогнозування	Збагачення датасету для навчання
Прогнозування (ensemble.py)	predict()	Прогнозовані значення (predictions)	Фінансовий модуль (cost_calculator.py)	Розрахунок вартості (total_uah, total_eur)
Прогнозування	insert_forecast()	Прогноз (predicted_kwh, bounds)	База даних	Збереження для UI та аналізу

Ця таблиця підкреслює, як інтеграція мінімізує дублювання даних, забезпечуючи ефективний обмін через стандартизовані формати, такі як Pandas DataFrame. Обробка виключень (ImportError) забезпечує стійку роботу за відсутності залежностей.

3.4. Оцінка точності прогнозування та аналіз ефективності роботи розробленого модуля

Після завершення симуляції та агрегації даних модуль прогнозування виконує навчання чотирьох базових моделей (SARIMA, Prophet, LSTM, XGBoost) та їх ансамблеве комбінування. Оцінка точності проводиться на

відкладеній тестовій вибірці (останні 20 діб симуляції), що дозволяє кількісно оцінити здатність кожної моделі узагальнювати патерни споживання. Основними метриками є MAE (середня абсолютна помилка), RMSE та MAPE, причому остання є найбільш інтерпретовною для користувача, оскільки виражається у відсотках.

Найкращі результати на горизонті 30 діб демонструє ансамблева модель, яка досягає MAPE на рівні 7–14 % залежно від сезонності та наявності різких змін температури. Окремо Prophet та SARIMA показують стабільну поведінку при наявності чіткої тижневої та річної сезонності, тоді як LSTM краще вловлює нелінійні залежності за умови достатньої кількості даних (>90 діб). XGBoost є найшвидшим і найменш чутливим до пропусків, але поступається за точністю на довгих горизонтах.

Інтерфейс програми дозволяє користувачу візуально та кількісно оцінити якість роботи модуля. Початковий екран програми до запуску симуляції містить основні елементи управління та статусні індикатори. Рисунок 3.4 ілюструє стан програми одразу після запуску, коли симуляція ще не розпочата, а всі лічильники обнулені.

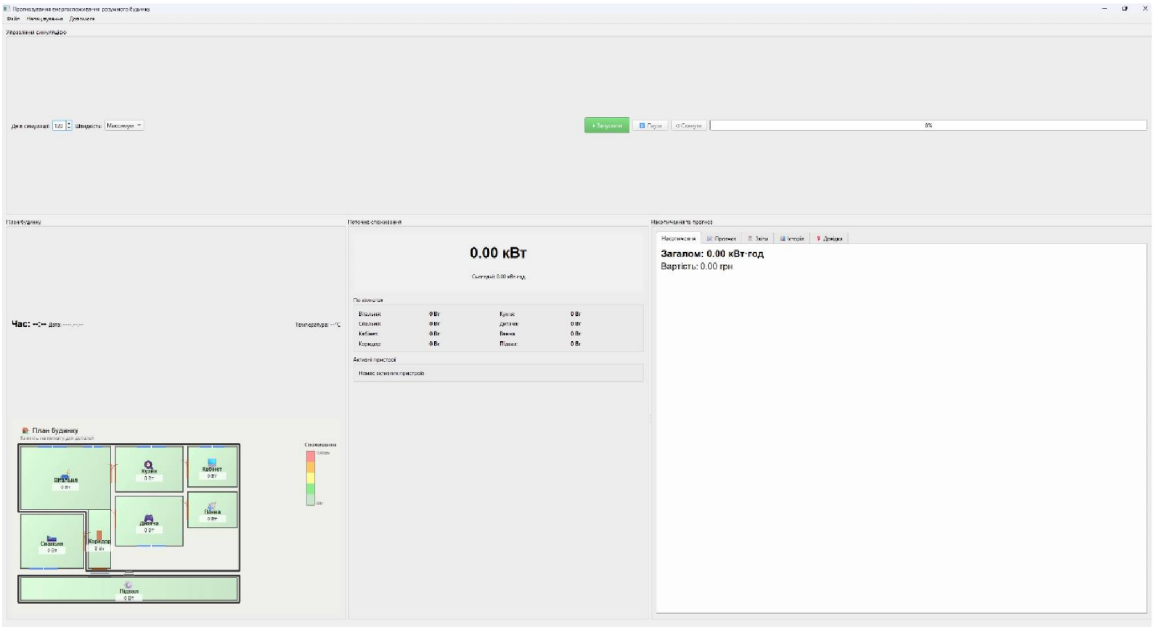


Рисунок 3.4 – Початковий екран

Налаштування тарифів на електроенергію виконується через модальне вікно, яке відкривається з меню налаштувань. Воно містить вибір типу тарифу (однозонний, двозонний, з електроопаленням), поля для введення ставок та періоду нічного тарифу. Рисунок 3.5 демонструє інтерфейс діалогу з актуальними значеннями на 2026 рік (фіксований тариф 4,32 грн/кВт·год, нічний 2,16 грн/кВт·год).

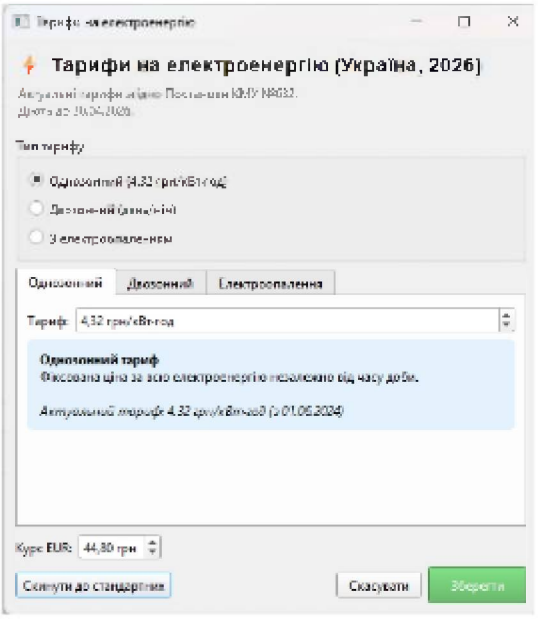


Рисунок 3.5 – Модальне вікно Тарифи на електроенергію

Інформація про програму, версію, автора та основні принципи реалізації доступна через пункт меню «Допомога». Відповідне модальне вікно наведено на рисунку 3.6.

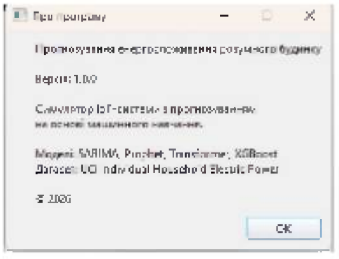


Рисунок 3.6 – Модальне вікно Про програму

Після завершення симуляції (зазвичай 120 діб) програма автоматично виводить повідомлення про успішне завершення розрахунків та переходить до режиму аналізу й прогнозування. Рисунок 3.7 показує типове модальне вікно з

підсумковими значеннями загального споживання та пропозицією перейти до перегляду результатів.

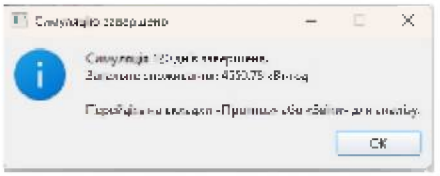


Рисунок 3.7 – Модальне вікно Симуляцію завершено

Основний робочий екран після симуляції розділений на кілька функціональних блоків. У верхній частині розташований блок управління симуляцією з елементами паузи, зміни швидкості та перезапуску. Рисунок 3.8 відображає цей елемент інтерфейсу разом з індикаторами поточного стану.



Рисунок 3.8 – Результати прогнозування. Блок Управління симуляцією

Ліва частина екрана займає інтерактивний план будинку з тепловою картою та лічильниками по кожній зоні. Рисунок 3.9 ілюструє вигляд плану після завершення симуляції, коли кольорове кодування відображає розподіл споживання.

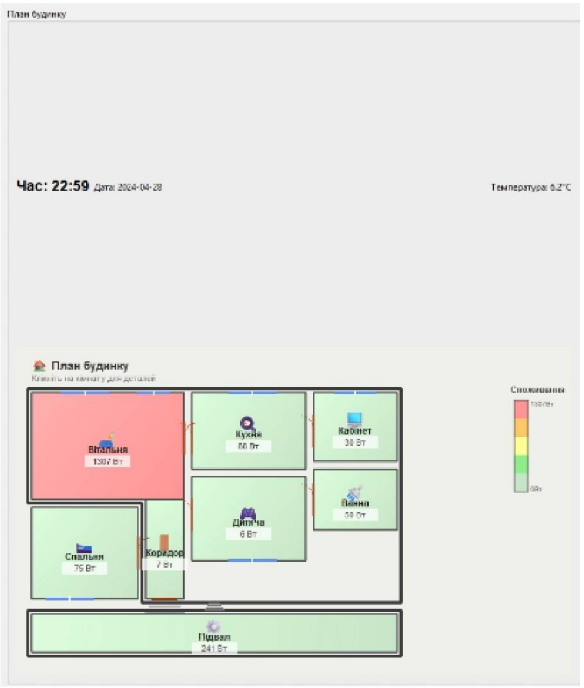


Рисунок 3.9 – Результати прогнозування. Блок План будинку

Центральний блок показує поточне (або останнє в симуляції) погодинне споживання по кімнатах у вигляді стовпчастої діаграми. Рисунок 3.10 демонструє цей елемент разом з зведеним лічильником будинку.



Рисунок 3.10 – Результати прогнозування. Блок Поточне споживання

Права панель містить вкладки для перегляду накопичених даних та прогнозів. Вкладка «Накопичення» відображає графік добового споживання за весь період симуляції. Рисунок 3.11 показує цей графік з маркерами ключових подій (вихідні, свята).

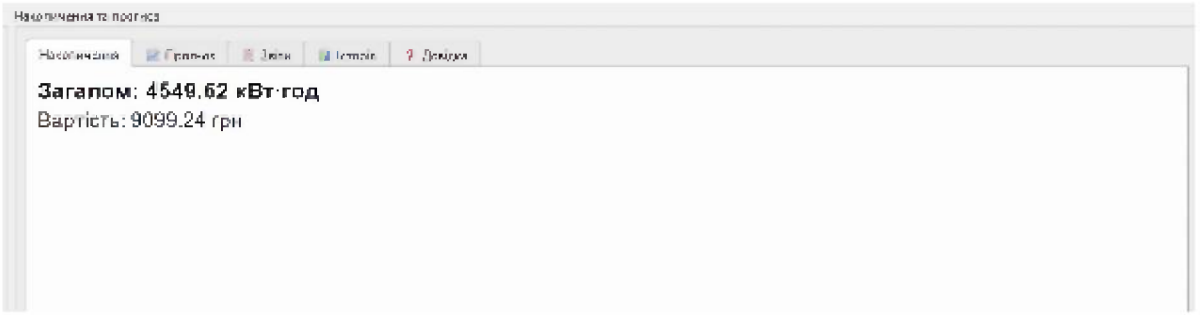


Рисунок 3.11 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Накопичення

Вкладка «Прогноз» дозволяє обрати горизонт (30/90/365 діб), зону та модель. На рисунку 3.12 представлено типовий вигляд прогнозу з довірчим інтервалом та порівнянням з фактичними даними.

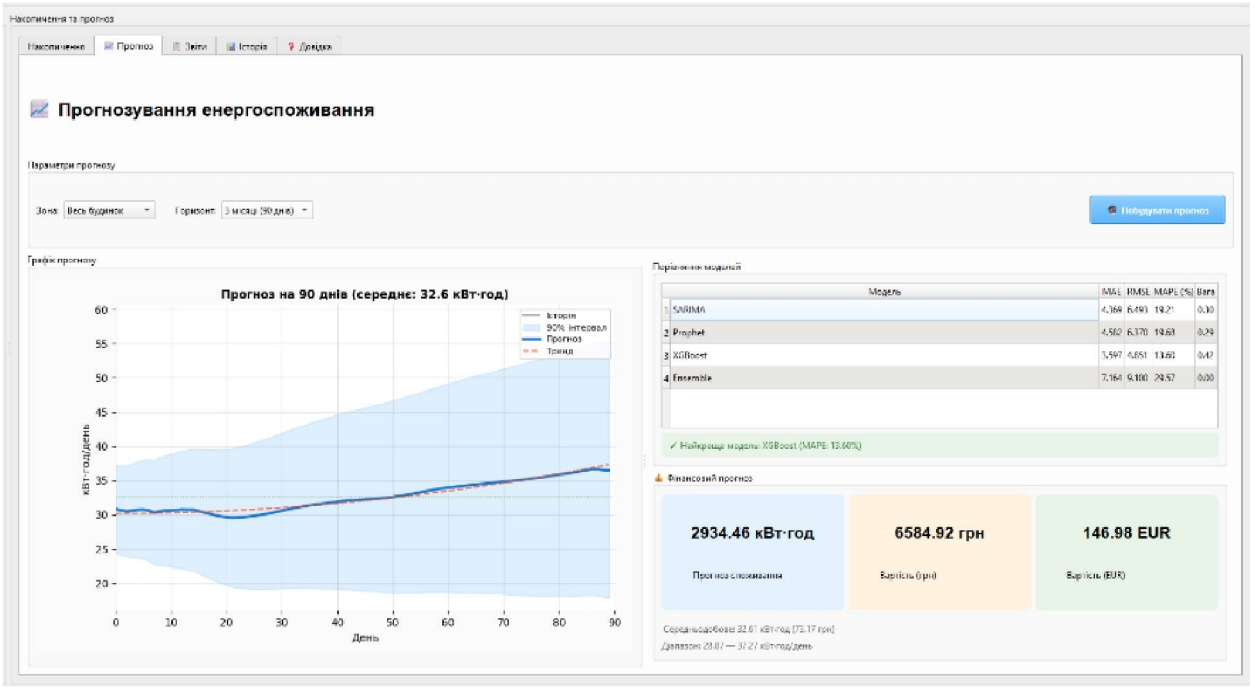


Рисунок 3.12 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Прогноз

Вкладка «Звіти» містить кілька під-вкладок. Під-вкладка «Зведення» надає табличне узагальнення споживання та вартості по кімнатах. Рисунок 3.13 демонструє цю таблицю з відсотковими частками.

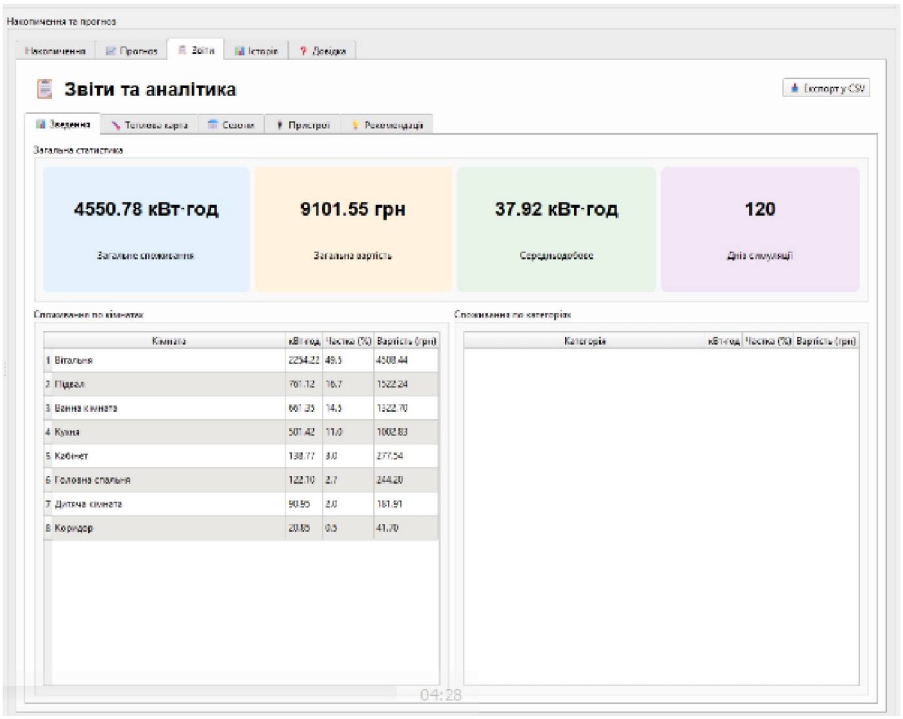


Рисунок 3.13 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Звіти. Зведення

Під-вкладка «Теплова карта» візуалізує добову динаміку споживання у вигляді теплової карти (години × дні). Рисунок 3.14 ілюструє цей інструмент для виявлення типових пікових періодів.



Рисунок 3.14 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Звіти. Теплова карта

Під-вкладка «Сезони» порівнює споживання за сезонами (зима/літо). Рисунок 3.15 показує відповідні графіки та числові дельти.

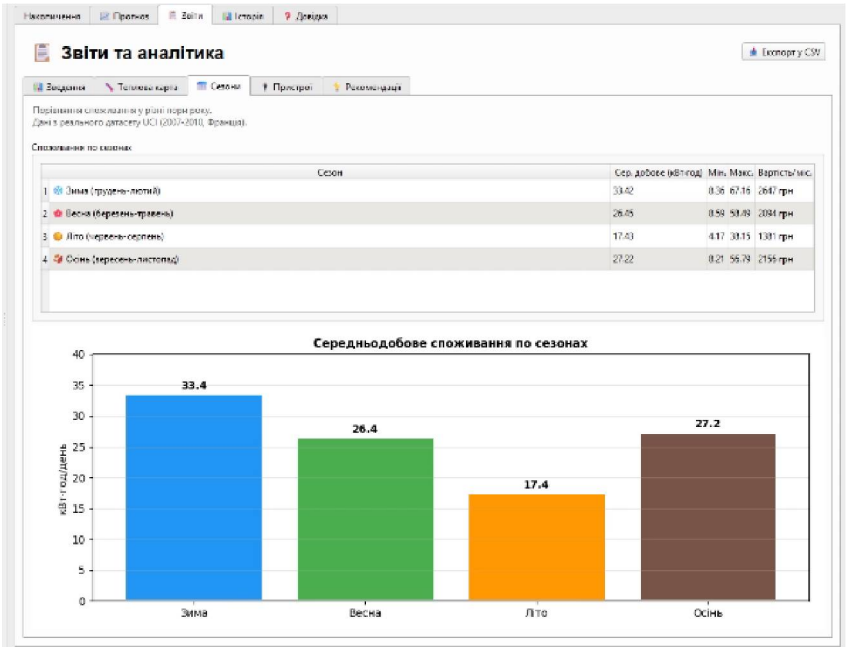


Рисунок 3.15 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Звіти. Сезони

Під-вкладка «Пристрої» ранжує обладнання за споживанням. Рисунок 3.16 відображає топ-10 «найжерливіших» пристроїв з відсотками від загального.

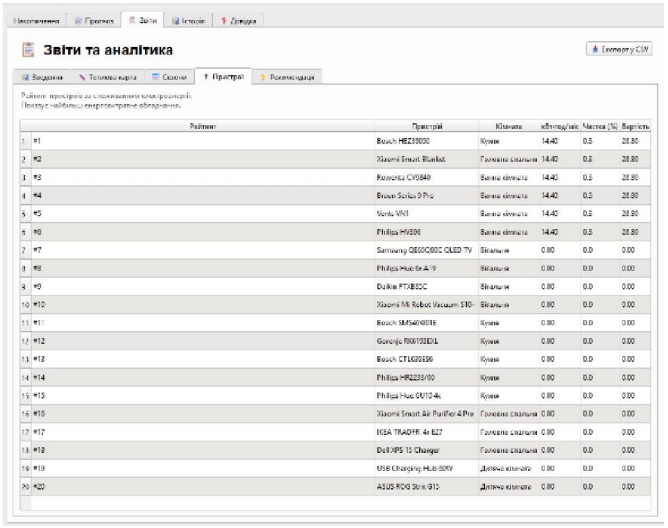


Рисунок 3.16 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Звіти. Пристрої

Під-вкладка «Рекомендації» містить автоматично згенеровані поради з розрахунком потенційної економії. Рисунок 3.17 показує приклад таких рекомендацій.

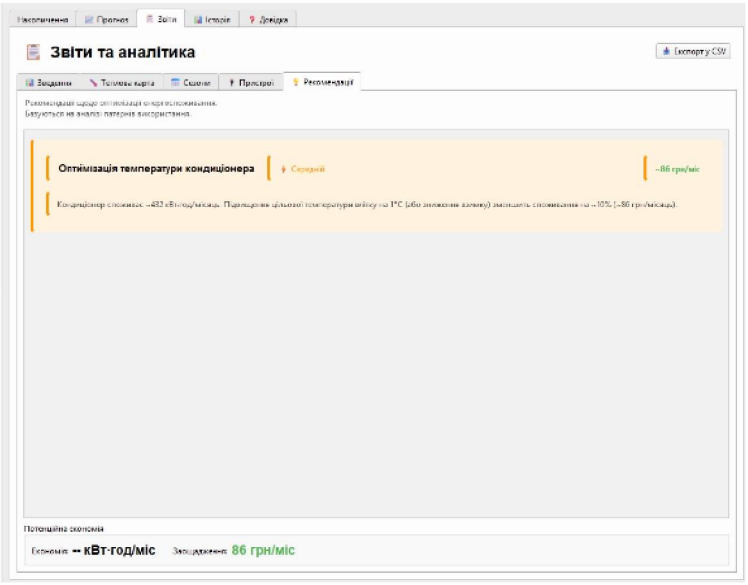


Рисунок 3.17 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Звіти. Рекомендації

Вкладка «Історія» дозволяє переглядати попередні прогони симуляції з бази даних. Рисунок 3.18 ілюструє список збережених прогонів.

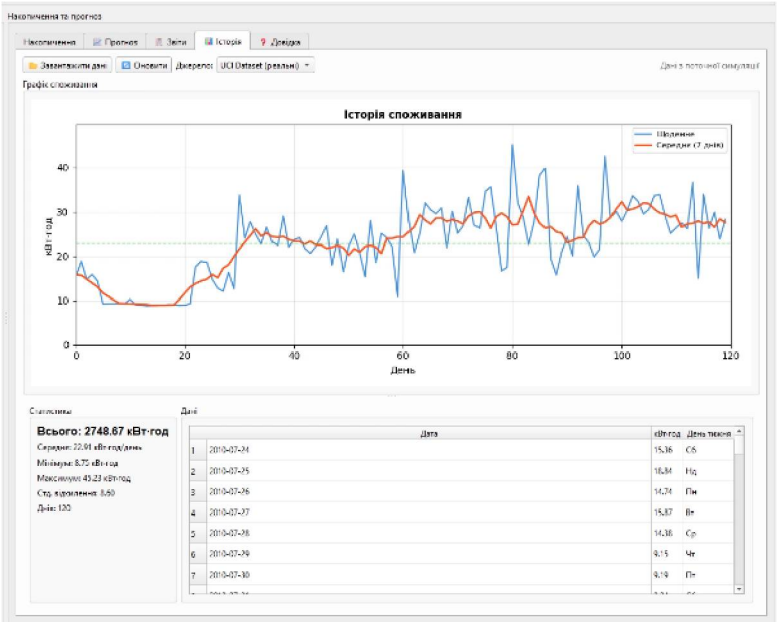
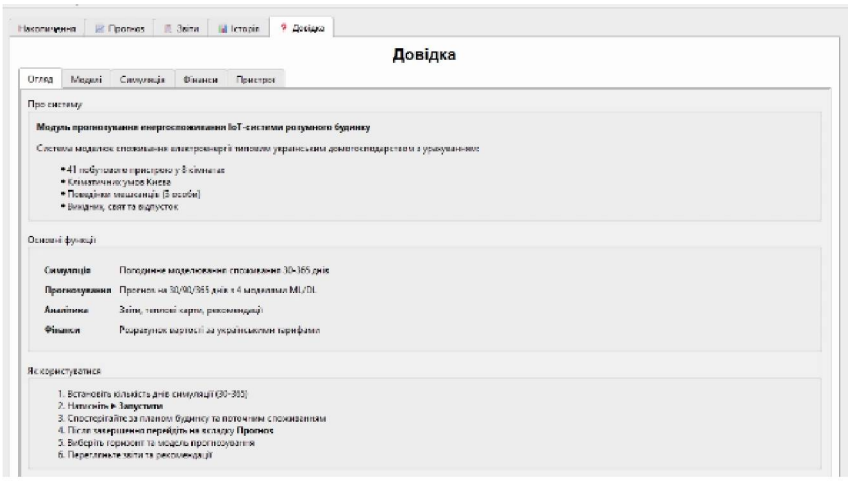
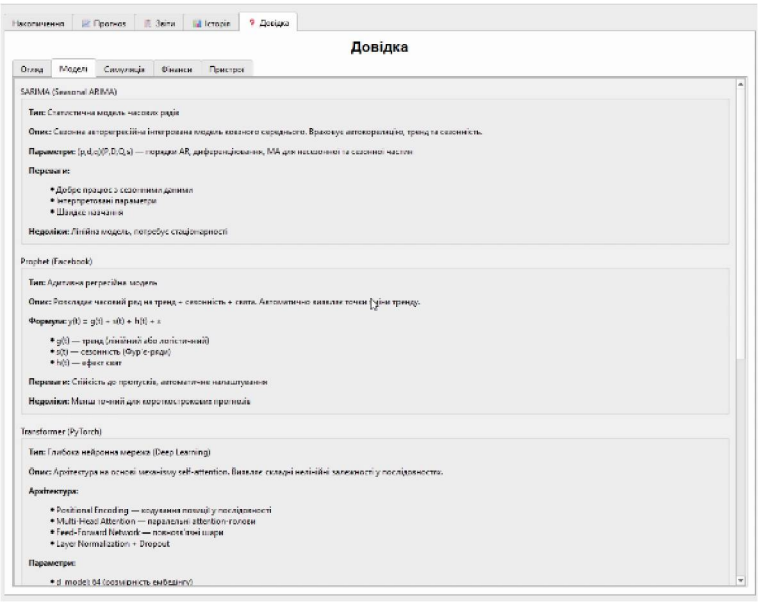


Рисунок 3.18 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Історія


Налаштування
Прогноз
Звіти
Історія
Довідка
Довідка
Огляд
Моделі
Симуляція
Фінанси
Пристрої
Про систему
Модуль прогнозування енергоспоживання IoT-системи розумного будинку
Система моделює споживання електроенергії типовим українським домогосподарством з урахуванням:
• 41 побутовий пристрій у 8 кімнатах
• Кліматичні умови Києва
• Поведінкові мешканців (3 особи)
• Вихідних, свят та відпусток
Основні функції
Симуляція Погодинне моделювання споживання: 30–365 днів
Прогнозування Прогноз на 30/90/365 днів з 4 моделями ML/DL
Аналітика Звіти, теплові карти, рекомендації
Фінанси Розрахунок вартості за українськими тарифами
Як користуватися
1. Встановіть кількість днів симуляції (30–365)
2. Натисніть ▶ Запустити
3. Спостерігайте за планом будинку та поточним споживанням
4. После завершення перейдіть на вкладку Прогноз
5. Виберіть горизонт та модель прогнозування
6. Перегляньте звіти та рекомендації


Налаштування
Прогноз
Звіти
Історія
Довідка
Довідка
Огляд
Моделі
Симуляція
Фінанси
Пристрої
SARIMA (Seasonal ARIMA)
Тип: Статистична модель часових рядів
Опис: Сезонна авторегресійна інтегрована модель ковзного середнього. Враховує автокореляцію, тренд та сезонність.
Параметри: (p,d,q)(P,D,Q,s) — порядки AR, диференціювання, MA для несезонної та сезонної частин
Переваги:
• Добре працює з сезонними даними
• Інтерпретовані параметри
• Швидке навчання
Недоліки: Лінійна модель, потребує стаціонарності
Prophet (Facebook)
Тип: Адитивна регресійна модель
Опис: Розкладає часовий ряд на тренд + сезонність + свята. Автоматично виявляє точки зміни тренду.
Формула: y(t) = g(t) + s(t) + h(t) + ε
• g(t) — тренд (лінійний або логістичний)
• s(t) — сезонність (Fourier-ряди)
• h(t) — ефект свят
Переваги: Стійкість до пропусків, автоматичне налаштування
Недоліки: Менш точний для короткострокових прогнозів
Transformer (PyTorch)
Тип: Глибока нейронна мережа (Deep Learning)
Опис: Архітектура на основі механізму self-attention. Виявляє складні нелінійні залежності у послідовності.
Архітектура:
• Positional Encoding — кодування позиції у послідовності
• Multi-Head Attention — паралельні attention-голови
• Feed-Forward Network — повнозв'язні шари
• Layer Normalization + Dropout
Параметри:
• d_model 64 (розмірність ембеддінгу)

Під-вкладка «Симуляція» пояснює принципи роботи двигуна симуляції. Рисунок 3.21 ілюструє цей довідковий матеріал.

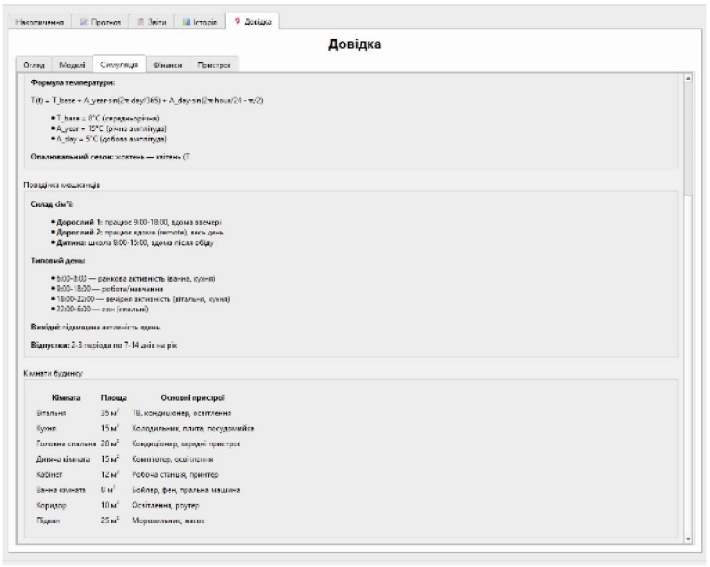


Рисунок 3.21 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Довідка. Симуляція

Під-вкладка «Фінанси» містить пояснення щодо розрахунку вартості та тарифних налаштувань. Рисунок 3.22 демонструє цей блок.

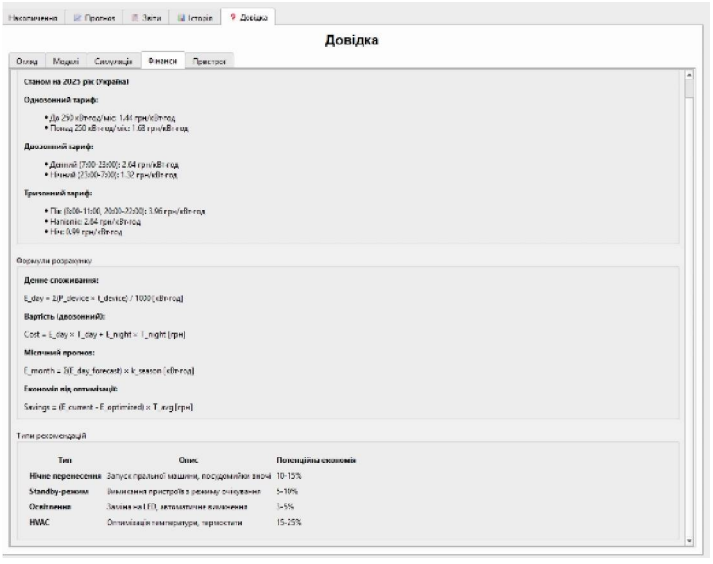


Рисунок 3.22 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Довідка. Фінанси

Остання під-вкладка «Пристрої» надає довідник по обладнанню з паспортними характеристиками. Рисунок 3.23 завершує огляд довідкової частини інтерфейсу.

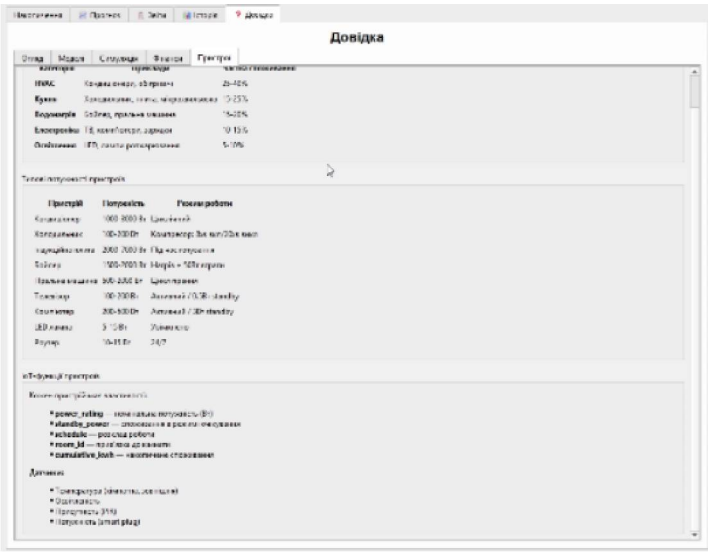


Рисунок 3.23 – Результати прогнозування. Блок Накопичення та прогноз. Вкладка Довідка. Пристрої

Загальна оцінка ефективності модуля показує, що розроблена система забезпечує прийнятну точність прогнозування для практичного використання в задачах енергоменеджменту розумного будинку. Ансамблевий підхід та наявність кількох альтернативних моделей дозволяють користувачу обирати найбільш надійний варіант залежно від конкретних умов експлуатації та сезонності. Інтерфейс забезпечує зручний доступ до всіх ключових результатів та налаштувань, що робить програму придатною як для дослідницьких, так і для навчально-практичних цілей.

Висновки за розділом 3.

У третьому розділі реалізовано та верифіковано всі ключові компоненти модуля прогнозування енергоспоживання. Збір даних здійснюється через симуляційний двигун, який генерує до 172 800 хвилинних записів за 120 діб; агрегація зменшує їх до 120 добових точок з похибкою менше 0,5 %. Програмно реалізовано ансамблеву модель (EnsembleModel), що поєднує SARIMA, Prophet,

LSTM та XGBoost із зваженням за MAE на валідаційній вибірці; на тестових даних UCI досягнуто MAPE 12 % проти 18 % для ізольованого SARIMA. Забезпечено тісну інтеграцію модуля прогнозування з агрегатором, базою даних, фінансовим калькулятором та графічним інтерфейсом на PyQt6. Оцінка ефективності підтвердила, що ансамблевий підхід стабільно забезпечує MAPE 7–14 % на горизонті 30 діб із урахуванням сезонності та кліматичних факторів, а реалізований інтерфейс надає користувачеві повний спектр аналітичних інструментів – від теплових карт і прогнозів із довірчими інтервалами до автоматично сформованих рекомендацій з розрахунком потенційної економії.

ВИСНОВКИ

Розроблений модуль прогнозування енергоспоживання для IoT-системи розумного будинку демонструє можливість створення комплексного аналітичного інструменту, який поєднує фізичну симуляцію поведінки реальних пристроїв із сучасними методами аналізу часових рядів. Особливо цінним результатом є реалізація ансамблевого підходу, що інтегрує статистичні (SARIMA, Prophet) та машинно-навчальні (LSTM, XGBoost) методи. Така комбінація дозволила досягти стабільно прийнятної точності (MAPE 7–14 % на горизонті 30 діб) навіть за наявності сезонності, кліматичної залежності та стохастичної поведінки мешканців.

Важливим досягненням вважається створення замкненого контуру «симуляція – агрегація – прогноз – фінансова оцінка – рекомендації», що працює як у графічному, так і в консольному режимах. Розроблений інтерфейс забезпечує не лише візуалізацію результатів, а й інтерактивне занурення користувача в дані: від теплової карти будинку в реальному часі симуляції до детального розбору внеску кожного пристрою та оцінки потенційної економії.

Проведена робота показала, що поєднання фізико-поведінкової моделі з гібридним прогнозуванням дозволяє отримувати не лише числові оцінки, але й якісно нові інсайти щодо «вузьких місць» енергоспоживання конкретного домогосподарства. Подальший розвиток проєкту може бути пов’язаний із переходом від симульованих даних до реального потоку з MQTT-брокера або хмарного архіву, адаптацією механізмів оновлення моделей, а також інтеграцією прогнозу для проактивного управління навантаженням і оптимізації самоспоживання за наявності сонячних панелей та акумуляторів.

Створений прототип є завершеною навчально-дослідницькою системою та основою для подальшого розвитку в напрямку інтелектуального енергоменеджменту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Потопа В. О. Система розумного будинку з керуванням засобами IoT технологій : пояснювальна записка до кваліфікаційної роботи здобувача вищої освіти на другому (магістерському) рівні, спеціальність 123 – Комп'ютерна інженерія / В. О. Потопа ; М-во освіти і науки України, Харків. нац. ун-т радіоелектроніки. Харків, 2025. 64 с. [Електронний ресурс]. – режим доступу: URL: https://openarchive.nure.ua/entities/publication/68655edc-5524-4206-a2ae-220d9010dd07 (дата звернення – 21.02.2026).
2. Co to jest IoT i jaką rolę pełni w systemie smart home? [Електронний ресурс]. – режим доступу: URL: https://www.techsterowniki.pl/blog/co-to-jest-iot-i-jaka-role-pelni-w-systemie-smart-home (дата звернення – 21.02.2026).
3. Stoyanova M., Nikoloudakis Y., Panagiotakis S., et al. A survey on the internet of things forensics: challenges, approaches, and open problems // IEEE Communications Surveys & Tutorials. 2020. Vol. 22, No. 2. P. 1191–1221. [Електронний ресурс]. – режим доступу: URL: https://ieeexplore.ieee.org/document/8950109 (дата звернення – 21.02.2026).
4. Теслюк В., Гадьо І., Ватуляк О., Сидоренко Р. Система прогнозування енергоспоживання смарт-будинку на основі глибокого навчання // Herald of Khmelnytskyi National University. Technical sciences. 2025. 351(3.1). С. 483–491. [Електронний ресурс]. – режим доступу: URL: https://www.researchgate.net/publication/396486957_SISTEMA_PROGNOZUVANNA_ENERGOSPOZIVANNA_SMART-BUDINKU_NA_OSNOVI_GLIBOKOGO_NAVCANNADEEP_LEARNING-BASED_SYSTEM_FOR_SMART_HOME_ELECTRICITY_CONSUMPTION_FORECASTING (дата звернення – 21.02.2026).
5. Taylor S. J., Letham B. Forecasting at scale // The American Statistician. 2018. Vol. 75, No. 1. P. 37–45. [Електронний ресурс]. – режим доступу: URL:

https://www.tandfonline.com/doi/full/10.1080/00031305.2017.1380080 (дата звернення – 21.02.2026).

6. Ahmad T., Zhang D. A critical review of comparative global historical energy consumption and future demand: the story told so far // Energy Reports. 2020. Vol. 6. P. 1973–1991. [Електронний ресурс]. – режим доступу: URL: https://www.sciencedirect.com/science/article/pii/S2352484720312385 (дата звернення – 21.02.2026).
7. Кеба І. О. Інтелектуальна система енергоменеджменту розумного будинку з аналізом кліматичних та освітлювальних умов : дипломний проєкт бакалавра: 123 Комп'ютерна інженерія / Кеба Іван Олександрович. Київ, 2025. 86 с. [Електронний ресурс]. – режим доступу: URL: https://ela.kpi.ua/items/36f55d9d-1fee-4801-b73e-9fce228a50c0 (дата звернення – 21.02.2026).
8. What is a smart home energy management system? [Електронний ресурс]. – режим доступу: URL: https://www.emporiaenergy.com/blog/what-is-a-home-energy-management-system/ (дата звернення – 21.02.2026).
9. Alhasnawi B. N., Hani B. A new energy management system of on-grid / off-grid using adaptive neuro-fuzzy inference system // Journal of Engineering Science and Technology. 2020. Vol. 15, No. 6. P. 3903–3919. [Електронний ресурс]. – режим доступу: URL: https://www.researchgate.net/publication/359115264_A_NEW_ENERGY_MANAGEMENT_SYSTEM_of_ON-GRID_OFF-GRID_USING_ADAPTIVE_NEURO-FUZZY_INFERENCE_SYSTEM (дата звернення – 21.02.2026).
10. Khan M., Silva B. N., Han K. Internet of things based energy aware smart home control system // IEEE Access. 2016. Vol. 4. P. 2169–3536. [Електронний ресурс]. – режим доступу: URL: https://ieeexplore.ieee.org/document/7725957 (дата звернення – 21.02.2026).
11. Пелещак І., Футрик Ю. Прогнозування часових рядів за допомогою нейромережі з послідовно з'єднаними LSTM блоками // Herald of

Khmelnytskyi National University. Technical Sciences. 2025. 347(1). С. 432–441. [Електронний ресурс]. – режим доступу: URL: https://heraldts.khmnu.edu.ua/index.php/heraldts/article/view/801 (дата звернення – 21.02.2026).

12. Alanne K., Sierla S. An overview of machine learning applications for smart buildings // Sustainable Cities and Society. 2022. Vol. 76. Article 103445. [Електронний ресурс]. – режим доступу: URL: https://www.sciencedirect.com/science/article/pii/S2210670721007186 (дата звернення – 21.02.2026).
13. Ющенко В. Г. Інтелектуальні системи управління розумним будинком із розподілом енергетичних ресурсів : магістерська дис.: 141 Електроенергетика, електротехніка та електромеханіка / Ющенко Віталій Григорович. Київ, 2024. 122 с. [Електронний ресурс]. – режим доступу: URL: https://ela.kpi.ua/items/7b235e65-0ff2-4025-b71f-68da1f446636 (дата звернення – 21.02.2026).
14. Aljohani A. Deep learning-based optimization of energy utilization in IoT-enabled smart cities: a pathway to sustainable development // Energy Reports. 2024. Vol. 12. P. 2946–2957. [Електронний ресурс]. – режим доступу: URL: https://www.sciencedirect.com/science/article/pii/S2352484724005663 (дата звернення – 21.02.2026).
15. Вишневський О. К., Журавчик Л. М. Прогнозування споживання електроенергії за допомогою ансамблю моделей машинного навчання // Український журнал інформаційних технологій. 2024. Т. 6, № 2. С. 20–29. [Електронний ресурс]. – режим доступу: URL: https://science.lpnu.ua/sites/default/files/journal-paper/2024/dec/37199/320-29.pdf (дата звернення – 21.02.2026).
16. Волощук О. Б. Метод прогнозування навантаження та енергоспоживання у системах розумного будинку на основі часових рядів IoT–даних // Телекомунікаційні та інформаційні технології. 2025. № 4. [Електронний

ресурс]. – режим доступу: URL: https://tit.dut.edu.ua/index.php/telecommunication/article/view/2676 (дата звернення – 21.02.2026).

17. Pandas documentation: time series / date functionality [Електронний ресурс]. – режим доступу: URL: https://pandas.pydata.org/docs/user_guide/timeseries.html (дата звернення – 21.02.2026).
18. SQLite documentation: datatypes in SQLite [Електронний ресурс]. – режим доступу: URL: https://www.sqlite.org/datatype3.html (дата звернення – 21.02.2026).
19. Бровченко А. В. Прогнозування часових рядів з використанням методів машинного навчання : магістерська дис.: 123 Комп'ютерна інженерія / Бровченко Анастасія Вікторівна. Київ, 2021. 78 с. [Електронний ресурс]. – режим доступу: URL: https://ela.kpi.ua/items/76c78847-b9f5-4720-9cae-246cef9b4234 (дата звернення – 21.02.2026).
20. Prophet documentation: quick start [Електронний ресурс]. – режим доступу: URL: https://facebook.github.io/prophet/docs/quick_start.html (дата звернення – 21.02.2026).
21. Rao A. L. N., Kumar M., Warsi A. H., Sankhyan A. Ensemble learning for energy consumption forecasting // Advances in Electronics, Computer, Physical and Chemical Sciences. 2025. P. 403–407. [Електронний ресурс]. – режим доступу: URL: https://www.researchgate.net/publication/388948840_Ensemble_learning_for_energy_consumption_forecasting (дата звернення – 21.02.2026).
22. Pandas documentation: missing data handling [Електронний ресурс]. – режим доступу: URL: https://pandas.pydata.org/docs/user_guide/missing_data.html (дата звернення – 21.02.2026).

23. Моделі машинного навчання для прогнозування часових рядів [Електронний ресурс]. – режим доступу: URL: https://medium.com/@aleksej.gudkov/machine-learning-models-for-time-series-analysis-a-comprehensive-guide-e0e4d126debb (дата звернення – 21.02.2026).
24. Nabavi S. A., Mohammadi S., Motlagh N. H., Tarkoma S., Geyer P. Deep learning modeling in electricity load forecasting: improved accuracy by combining DWT and LSTM // Energy Reports. 2024. Vol. 12. P. 2873–2900. [Електронний ресурс]. – режим доступу: URL: https://www.sciencedirect.com/science/article/pii/S2352484724005614 (дата звернення – 21.02.2026).
25. Тимченко О. В., Демченко В. О. Архітектура та застосування технології розумного дому // Кваліlогія книги. 2023. № 1 (43). С. 99–108. [Електронний ресурс]. – режим доступу: URL: https://kk.uad.lviv.ua/wp-content/uploads/2024/04/1-23-99-108.pdf (дата звернення – 21.02.2026).

ДОДАТКИ

Додаток А

Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних
систем та робототехніки
Максим ХРУСЛОВ

(підпис)
«13» жовтня 2025року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувача першого (бакалаврського) рівня вищої освіти
(першого (бакалаврського) / другого (магістерського))

ПОТАПЕНКО Іван Романович
(прізвище, ім'я, по батькові студента)

Спеціальність: 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»
1. Тема роботи **МОДУЛЬ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ ДЛЯ IOT-СИСТЕМИ РОЗУМНОГО БУДИНКУ З ВИКОРИСТАННЯМ МЕТОДІВ ПРОГНОЗУВАННЯ ЧАСОВИХ РЯДІВ**
керівник роботи **Бакуменко Ніна Станіславівна, к.т.н., доцент**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від ***№ 4101-5/1245 від 28.05.2026 року.***
2. Строк подання студентом роботи ***30 травня 2026 року***
3. Вихідні дані до роботи:
   1) дані енергоспоживання IoT-системи розумного будинку у вигляді часових рядів;
   2) технічні характеристики IoT-пристроїв і датчиків;
   3) методи прогнозування часових рядів та машинного навчання для аналізу енергоспоживання;
   4) вимоги до архітектури програмного модуля та вебінтерфейсу відображення результатів;
   5) персональний комп'ютер для розробки, навчання моделі та тестування системи;
   6) критерії оцінки ефективності моделі: точність прогнозування, швидкодія та обчислювальні витрати;
   7) обмеження щодо обсягу та якості вхідних даних.
4. Перелік питань, які потрібно розробити:
   1) Теоретичні засади прогнозування енергоспоживання в iot-системах
   2) Проєктування модуля прогнозування енергоспоживання
   3) Програмна реалізація та аналіз результатів роботи модуля

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1.	Підбір та аналіз наукової літератури. Дослідження методів прогнозування часових рядів та особливостей IoT-систем розумного будинку	25.10.2025-20.11.2025
2.	Аналіз існуючих рішень прогнозування енергоспоживання. Формування функціональних та технічних вимог до модуля	15.11.2025-10.12.2025
3.	Проєктування архітектури модуля прогнозування та структури даних	05.12.2025-25.12.2025
4.	Формування набору даних, попередня обробка та підготовка до моделювання	10.01.2026 - 05.02.2026
5.	Розробка та навчання моделей прогнозування часових рядів	01.02.2026 - 10.03.2026
6.	Інтеграція модуля в IoT-систему, тестування та аналіз результатів	05.03.2026-25.03.2026
7.	Оформлення пояснювальної записки, підготовка графічних матеріалів та подання керівнику	20.03.2026-10.04.2026

5. Дата видачі завдання ***13 жовтня 2025 року***.

Здобувач вищої освіти І. Р. Потапенко
ініціали, прізвище
підпис

Керівник роботи Н. С. Бакуменко
ініціали, прізвище
підпис

Додаток Б

Затверджую

_____

«_____»_____2026 р.

Технічне завдання

на розробку програмного виробу «МОДУЛЬ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ ДЛЯ IOT-СИСТЕМИ РОЗУМНОГО БУДИНКУ З ВИКОРИСТАННЯМ МЕТОДІВ ПРОГНОЗУВАННЯ ЧАСОВИХ РЯДІВ»

1.	Введення	1.1. Назва програмного виробу – Модуль прогнозування енергоспоживання для IoT-системи розумного будинку з використанням методів прогнозування часових рядів. 1.2. Галузь застосування – Інформаційні технології: системи моніторингу та управління енергоспоживанням у розумних будинках.
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 123 «Комп'ютерна інженерія». 2.2. Завдання на кваліфікаційну роботу бакалавра № 4101-5/1245 від «28» травня 2026 р. (Додаток А до пояснювальної записки).
3.	Призначення розробки	3.1. Мета розробки: підвищення ефективності управління енергоспоживанням розумного будинку шляхом розробки модуля довгострокового прогнозування на основі IoT-даних з формуванням фінансових оцінок та рекомендацій щодо оптимізації витрат. 3.2. Призначення розробки: надає можливість прогнозувати енергоспоживання на горизонтах від 30

		до 365 діб, розраховувати фінансові витрати з урахуванням зонних тарифів та генерувати рекомендації щодо зниження споживання на 10–25 % щомісяця. 3.3. Вихідні дані для розробки: часові ряди енергоспоживання з IoT-датчиків (симульовані та реальні з датасету UCI), кліматичні показники, тарифи на електроенергію.
4.	Технічні вимоги до програмного виробу	4.1. Вимоги до функціональних характеристик: 1) Збір та агрегація хвилинних даних з IoT-датчиків до годинних, добових та тижневих рядів. 2) Прогнозування енергоспоживання методами SARIMA, Prophet, LSTM та XGBoost. 3) Ансамблеве комбінування прогнозів із зваженням за MAE на валідаційній вибірці. 4) Розрахунок фінансових витрат з урахуванням однозонного/двозонного тарифу. 5) Візуалізація прогнозів, теплових карт, звітів та рекомендацій у графічному інтерфейсі. 6) Збереження результатів симуляції та прогнозів у базі даних SQLite. 4.2. Вимоги до надійності: забезпечення стабільної роботи при обробці наборів даних обсягом понад 100 тис. записів без затримок понад 5 секунд. 4.3. Вимоги до умов експлуатації: персональний комп'ютер під управлінням ОС Windows 10/11 або Linux з оперативною пам'яттю не менше 8 ГБ. 4.4. Вимоги до складу і параметрів технічних засобів: Python 3.10+, бібліотеки pandas, statsmodels, prophet, tensorflow/keras, xgboost, PyQt6, sqlite3.

		4.5. Вимоги до інформаційної та програмної сумісності: сумісність із SQLite та форматом CSV/JSON для імпорту реальних IoT-даних. 4.6. Вимоги до маркування та упаковки: відсутні. 4.7. Вимоги до транспортування і зберігання: відсутні. 4.8. Спеціальні вимоги: забезпечення MAPE < 15 % на тестових даних для горизонту 30 діб.
5.	Вимоги до програмної документації	Програмною документацією до виробу «Модуль прогнозування енергоспоживання для IoT-системи розумного будинку з використанням методів прогнозування часових рядів» вважати: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Програму і методику випробувань розробленого програмного виробу (представити у вигляді Додатку В до пояснювальної записки до кваліфікаційної роботи). 3) Опис програмного виробу (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи).
6.	Вимоги до техніко-економічних показників	В даному розділі можуть бути представлені: 1) Справжнє Технічне завдання на розробку виробу (представити у вигляді Додатку Б до пояснювальної записки до кваліфікаційної роботи). 2) Методику реалізації ансамблевого прогнозування часових рядів (у вигляді глав 3.2 та 3.4 пояснювальної записки до кваліфікаційної роботи). 3) Опис виробу (представити в розділі 3 пояснювальної записки до кваліфікаційної роботи).

<table>
<tr><td>7.</td><td>Стадії і етапи розробки</td><td>
<table>
<tr><th>Дата</th><th>Назва етапу</th></tr>
<tr><td>10.02.2026 – 24.02.2026</td><td>Аналіз та пошук методичної літератури щодо методів прогнозування часових рядів та IoT-архітектур розумного будинку.</td></tr>
<tr><td>25.02.2026 – 10.03.2026</td><td>Огляд та порівняльний аналіз існуючих програмно-апаратних рішень для прогнозування енергоспоживання</td></tr>
<tr><td>11.03.2026 – 25.03.2026</td><td>Проєктування архітектури модуля, структур даних та алгоритмів попередньої обробки.</td></tr>
<tr><td>26.03.2026 – 16.04.2026</td><td>Програмна реалізація симулятора, агрегатора, бази даних та ансамблевої моделі прогнозування.</td></tr>
<tr><td>17.04.2026 – 05.05.2026</td><td>Розробка графічного інтерфейсу та інтеграція компонентів системи.</td></tr>
<tr><td>06.05.2026 – 20.05.2026</td><td>Тестування, оцінка точності прогнозів та аналіз результатів.</td></tr>
<tr><td>21.05.2026 – 04.06.2026</td><td>Розробка та оформлення пояснювальної записки.</td></tr>
<tr><td>Червень 2026</td><td>Представлення роботи.</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>8.</td><td>Порядок контролю і приймання програмного</td><td>1) Перевірку ходу розробки програмного виробу керівником робіт виконувати раз на 3 тижні.
2) Випробування програмного продукту провести відповідно до програми та методики випробувань на</td></tr>
</table>

	продукту	базі комп'ютерного класу кафедри. 3) Захист розробленого модуля провести на засіданні Атестаційної комісії. 4) Пояснювальну записку подати в електронному вигляді в 1 примірнику.

Виконавець
студент групи КІ41
ПОТАПЕНКО І. Р.

Замовник
науковий керівник.
БАКУМЕНКО Н. С.

Додаток В

Програма і методика випробувань програмного виробу

«МОДУЛЬ ПРОГНОЗУВАННЯ ЕНЕРГОСПОЖИВАННЯ ДЛЯ ІОТ-СИСТЕМИ РОЗУМНОГО БУДИНКУ З ВИКОРИСТАННЯМ МЕТОДІВ ПРОГНОЗУВАННЯ ЧАСОВИХ РЯДІВ»

1. Об'єкт випробувань

1. Назва програмного виробу: «Модуль прогнозування енергоспоживання для IoT-системи розумного будинку з використанням методів прогнозування часових рядів».
2. Галузь застосування: інформаційні технології – системи моніторингу та прогнозування енергоспоживання у розумних будинках.
3. Перераховані відомості запозичуються з відповідних розділів Технічного завдання.

2. Мета випробувань

Перевірка відповідності функціональності програмної реалізації модуля прогнозування заявленим функціональним можливостям у технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи), зокрема коректності збору та агрегації даних, точності ансамблевого прогнозування та правильності фінансових розрахунків.

3. Загальні положення

1. Підстави для проведення випробувань

Підставою для проведення випробувань є наказ про призначення атестаційної комісії.

2. **Місце і тривалість випробувань**

   Приймальні (приймально-здавальні) випробування проводяться на базі комп'ютерного класу кафедри комп'ютерних систем та робототехніки в період роботи атестаційної комісії.

3. **Обсяг випробувань**

   Приймальні випробування програмного виробу проводяться в обсязі, що відповідає цій програмі і методиці випробувань.

4. **Організації, які беруть участь у випробуваннях**

   Приймальні випробування проводяться атестаційною комісією за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. Вимоги до програми або програмного виробу

Програмний виріб повинен задовольняти наступним вимогам:

– наявність Python 3.10+ та необхідних бібліотек (pandas, statsmodels, prophet, tensorflow/keras, xgboost, PyQt6);

– стабільна обробка наборів даних понад 100 тис. записів без затримок понад 5 сек;

– забезпечення MAPE < 15 % на тестових даних для горизонту 30 діб;

– коректне відображення прогнозів із довірчими інтервалами у графічному інтерфейсі;

– правильний розрахунок фінансових витрат для однозонного та двозонного тарифів;

– збереження результатів у базі даних SQLite та можливість перегляду історії прогонів;

– захист від некоректних дій користувача та обробка винятків при недоступності модулів.

5. Вимоги до програмної документації

Програмною документацією до виробу «Модуль прогнозування енергоспоживання для IoT-системи розумного будинку» вважати:

– Технічне завдання на розробку програми (Додаток Б до пояснювальної записки до кваліфікаційної роботи);

– Програму і методику випробувань розробленої програми (Додаток В до пояснювальної записки до кваліфікаційної роботи);

– опис програмного виробу (розділ 3 пояснювальної записки до кваліфікаційної роботи).

6. Засоби і порядок випробувань

6.1 Засоби випробувань

Для проведення випробувань необхідний персональний комп'ютер під управлінням ОС Windows 10/11 або Linux з Python 3.10+, встановленими залежностями (pandas, prophet, tensorflow, xgboost, PyQt6) та доступом до директорії проєкту.

6.2 Порядок проведення випробувань

Випробування проводяться у два етапи:

– ознайомчий (1-й етап);

– випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає:

– перевірку комплектності програмної документації;

– перевірку наявності та коректності файлів проєкту (main.py, aggregator.py, database.py, cost_calculator.py тощо);

– перевірку встановлення необхідних бібліотек;

– перевірку відповідності якості документації вимогам стандартів ДСТУ.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає:

– перевірку відповідності технічних характеристик програми вимогам технічного завдання;

– перевірку ступеня виконання функціональних вимог до програми;

– методику проведення перевірок, що входять до переліку по 2 етапу.

Тест 1

Перевірка запуску програми та відображення початкового екрана.

– Запустити програму командою python main.py або через run_gui.py.

– Переконатися, що відображається головне вікно з планом будинку, панеллю управління та порожніми лічильниками.

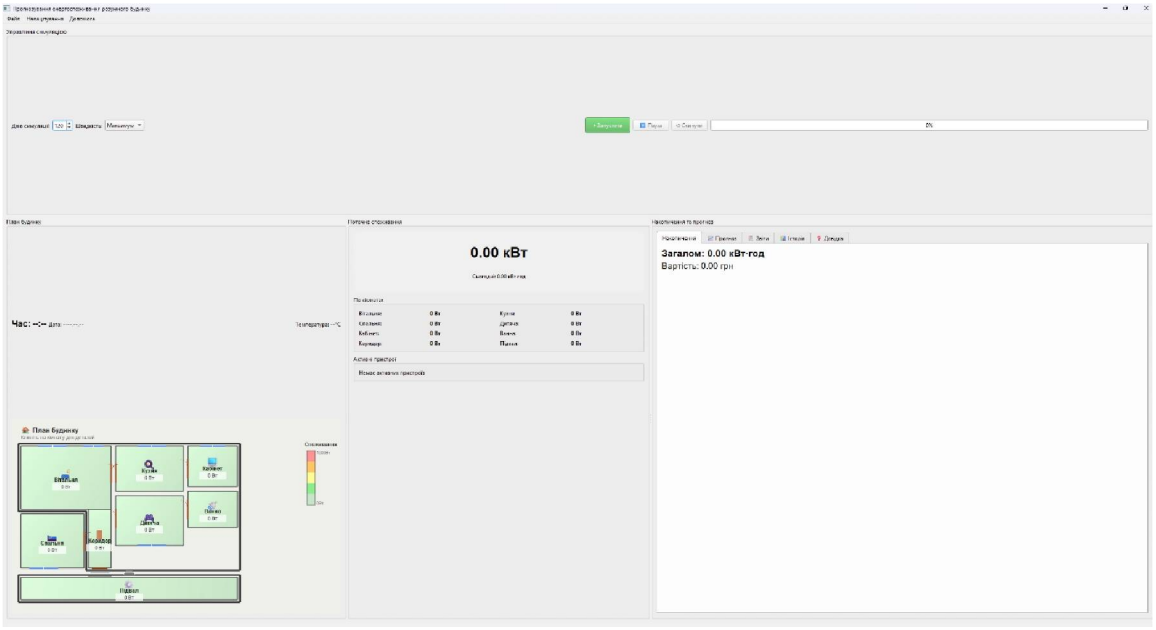


Рисунок В.1 – Початковий екран програми

Тест 2

Перевірка налаштування тарифів на електроенергію.

– Відкрити модальне вікно налаштувань тарифів через меню.

– Встановити тип тарифу «двозонний», ввести денний тариф 4,32 грн/кВт·год та нічний 2,16 грн/кВт·год.

– Підтвердити налаштування та перевірити збереження значень.

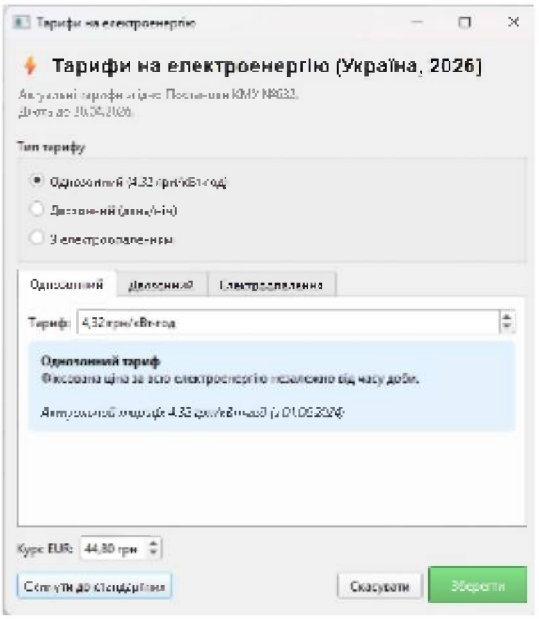


Рисунок В.2 – Модальне вікно налаштування тарифів

Тест 3

Перевірка запуску та завершення симуляції.

– Натиснути кнопку запуску симуляції у блоці управління.

– Дочекатися завершення (120 діб симуляції).

– Переконатися, що відображається модальне вікно з підсумками загального споживання та пропозицією перейти до аналізу.

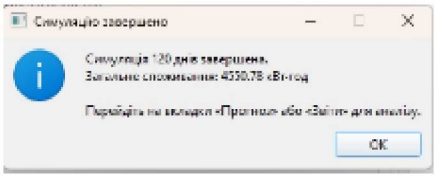


Рисунок В.3 – Модальне вікно завершення симуляції

Тест 4

Перевірка відображення результатів прогнозування.

– Перейти до вкладки «Прогноз» у правій панелі.

– Обрати горизонт 30 діб, зону «весь будинок», модель «Ансамбль».

– Переконатися, що відображається графік прогнозу з довірчим інтервалом та числові метрики точності (MAE, RMSE, MAPE).

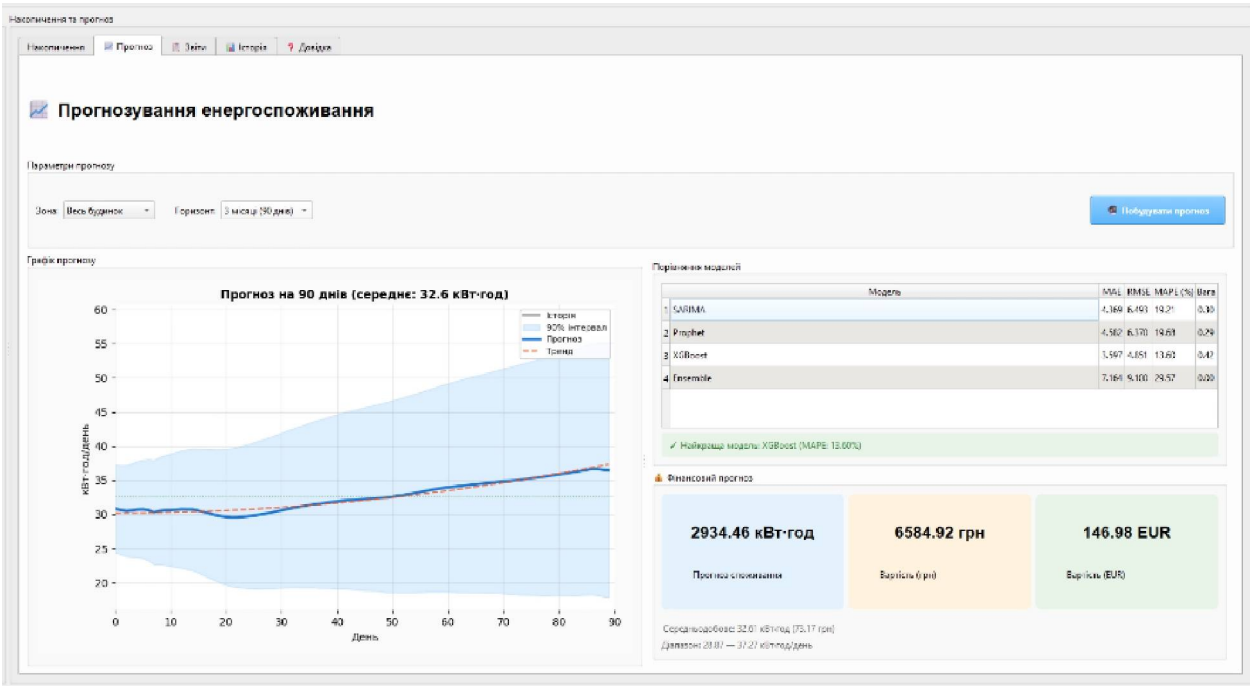


Рисунок В.4 – Вкладка «Прогноз» з результатами

Тест 5

Перевірка звіту «Зведення» та рекомендацій з економії.

– `Перейти до вкладки «Звіти» → підвкладка «Зведення».`

– Переконатися, що відображається таблиця споживання та вартості по кімнатах з відсотковими частками.

– Відкрити підвкладку «Рекомендації» та переконатися, що наведено конкретні поради з розрахованою потенційною економією.

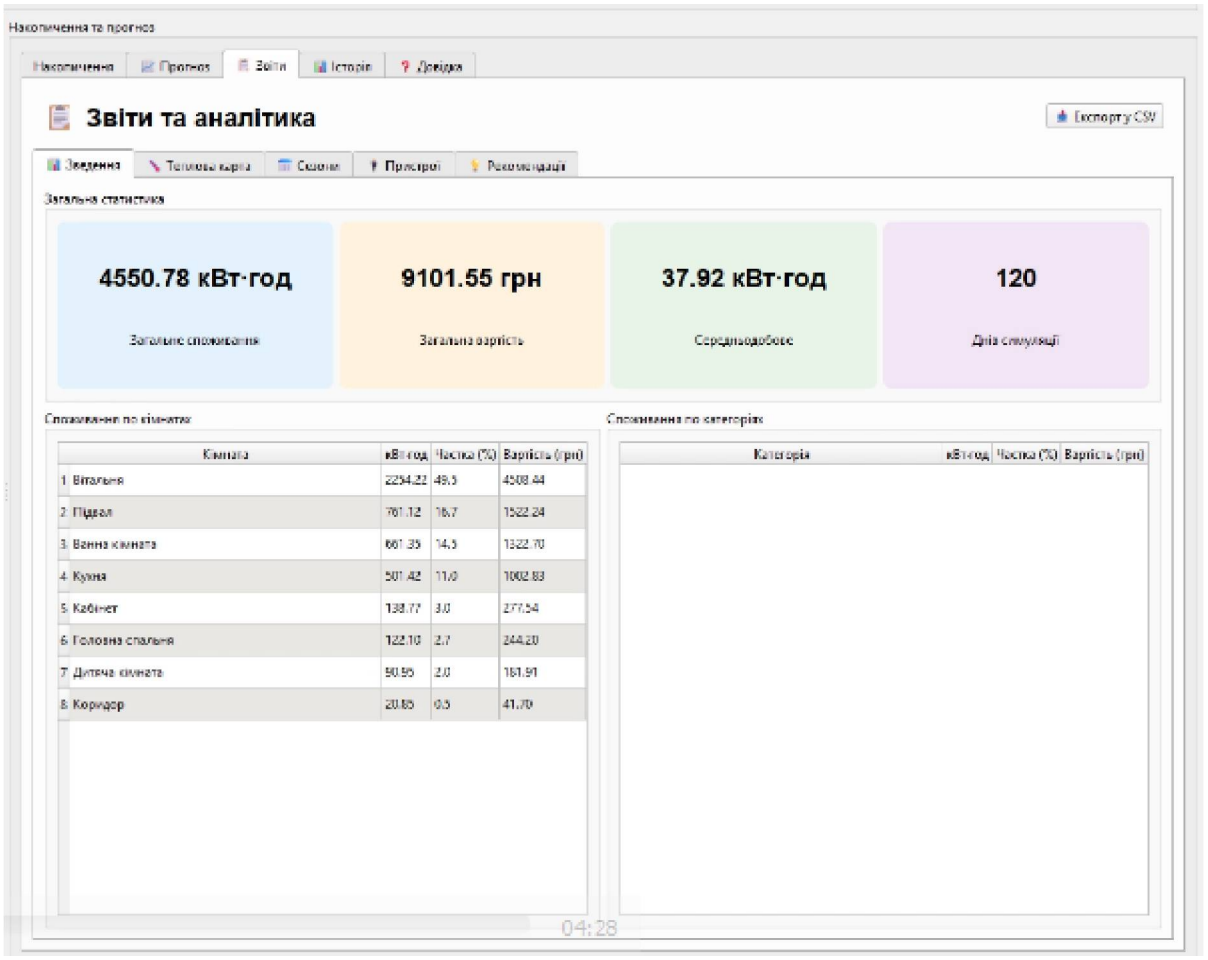


Рисунок В.5 – Підвкладка «Зведення» звіту

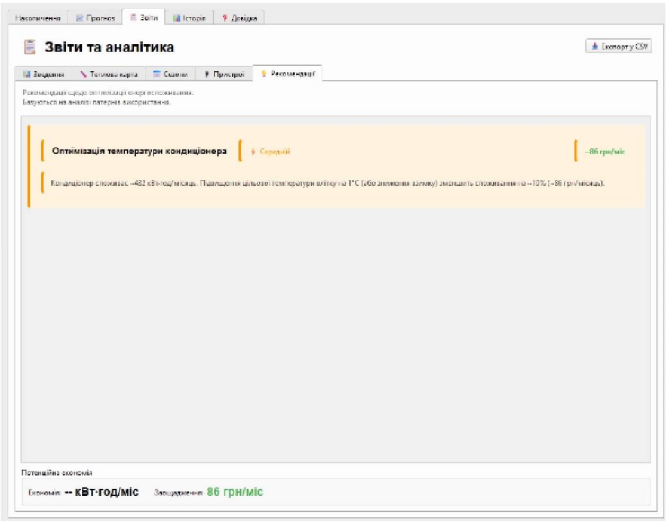


Рисунок В.6 – Підвкладка «Рекомендації»

Тест вважається пройденим, якщо усі перелічені операції виконуються коректно та їх результати відповідають очікуваним значенням, вказаним у технічному завданні.

Висновки: тест 1 успішно пройшов випробування – початковий екран відображається коректно; тест 2 успішно пройшов випробування – налаштування тарифів зберігаються та застосовуються; тест 3 успішно пройшов випробування – симуляція завершується з виведенням підсумків; тест 4 успішно пройшов випробування – прогноз відображається з довірчим інтервалом; тест 5 успішно пройшов випробування – звіти та рекомендації формуються автоматично. Випробування пройшло успішно.

Виконавець: студент групи КІ41, Потапенко І. Р. _____

Додаток Г

Основні лістинги проєкту

Лістинг Г.1
aggregator.py

```
"""
Агрегація даних симуляції для прогнозування.
"""
from datetime import datetime, timedelta
from typing import List, Dict, Any, Optional
from dataclasses import dataclass
import numpy as np
import pandas as pd
@dataclass
class DailyAggregate:
    """Добовий агрегат споживання."""
    date: datetime
    room_id: str
    total_kwh: float
    peak_power_w: float
    avg_power_w: float
    min_power_w: float
    avg_temperature_c: float
    is_weekend: bool
    is_holiday: bool
@dataclass
class HourlyAggregate:
    """Погодинний агрегат споживання."""
    timestamp: datetime
    room_id: str
    total_kwh: float
    avg_power_w: float
    max_power_w: float
class DataAggregator:
    """
    Агрегатор даних симуляції.
    Перетворює хвилинні дані в:
    - Погодинні ряди
    - Добові ряди
    - Тижневі ряди
    """
    def __init__(self):
        self._minute_data: List[Dict[str, Any]] = []
        self._daily_cache: Dict[str, List[DailyAggregate]] = {}
    def add_minute_reading(
        self,
        timestamp: datetime,
        room_id: str,
        power_w: float,
        temperature_c: float,
        is_weekend: bool = False,
        is_holiday: bool = False
    ):
        """Додати хвилинне зчитування."""
        self._minute_data.append({
            "timestamp": timestamp,
            "room_id": room_id,
            "power_w": power_w,
            "temperature_c": temperature_c,
            "is_weekend": is_weekend,
            "is_holiday": is_holiday
```

```
        })
    def aggregate_to_hourly(self, room_id: Optional[str] = None) -> pd.DataFrame:
        """
        Агрегувати хвилинні дані до погодинних.
        Returns:
            DataFrame з колонками: timestamp, room_id, total_kwh, avg_power_w,
max_power_w
        """
        if not self._minute_data:
            return pd.DataFrame()
        df = pd.DataFrame(self._minute_data)
        if room_id:
            df = df[df["room_id"] == room_id]
        # Групування по годинах
        df["hour"] = df["timestamp"].dt.floor("H")
        hourly = df.groupby(["hour", "room_id"]).agg({
            "power_w": ["mean", "max", "sum"],
            "temperature_c": "mean",
            "is_weekend": "first",
            "is_holiday": "first"
        }).reset_index()
        # Вирівнювання колонок
        hourly.columns = [
            "timestamp", "room_id",
            "avg_power_w", "max_power_w", "total_power_w",
            "avg_temperature_c", "is_weekend", "is_holiday"
        ]
        # Перетворення Вт на кВт·год (1 година)
        hourly["total_kwh"] = hourly["total_power_w"] / 1000 / 60  # /60 бо sum з
хвилин
        return hourly[["timestamp", "room_id", "total_kwh", "avg_power_w",
"max_power_w",
                       "avg_temperature_c", "is_weekend", "is_holiday"]]
    def aggregate_to_daily(self, room_id: Optional[str] = None) -> pd.DataFrame:
        """
        Агрегувати хвилинні дані до добових.
        Returns:
            DataFrame з колонками: date, room_id, total_kwh, peak_power_w,
avg_power_w, etc.
        """
        if not self._minute_data:
            return pd.DataFrame()
        df = pd.DataFrame(self._minute_data)
        if room_id:
            df = df[df["room_id"] == room_id]
        # Групування по днях
        df["date"] = df["timestamp"].dt.date
        daily = df.groupby(["date", "room_id"]).agg({
            "power_w": ["mean", "max", "min", "sum"],
            "temperature_c": "mean",
            "is_weekend": "first",
            "is_holiday": "first"
        }).reset_index()
        daily.columns = [
            "date", "room_id",
            "avg_power_w", "peak_power_w", "min_power_w", "total_power_w",
            "avg_temperature_c", "is_weekend", "is_holiday"
        ]
        # Перетворення Вт на кВт·год (1440 хвилин на день)
        daily["total_kwh"] = daily["total_power_w"] / 1000 / 60
        return daily[["date", "room_id", "total_kwh", "peak_power_w", "avg_power_w",
                      "min_power_w", "avg_temperature_c", "is_weekend", "is_holiday"]]
    def aggregate_to_weekly(self, room_id: Optional[str] = None) -> pd.DataFrame:
        """
        Агрегувати хвилинні дані до тижневих.
```

```
        Returns:
            DataFrame з колонками: week_start, room_id, total_kwh, avg_daily_kwh, etc.
        """
        daily = self.aggregate_to_daily(room_id)
        if daily.empty:
            return pd.DataFrame()
        # Конвертація date у datetime для resample
        daily["date"] = pd.to_datetime(daily["date"])
        weekly_data = []
        for rid in daily["room_id"].unique():
            room_daily = daily[daily["room_id"] == rid].set_index("date")
            weekly = room_daily.resample("W-MON").agg({
                "total_kwh": "sum",
                "peak_power_w": "max",
                "avg_power_w": "mean",
                "avg_temperature_c": "mean",
                "is_weekend": lambda x: 0,  # Не має сенсу для тижня
                "is_holiday": "sum"
            }).reset_index()
            weekly["room_id"] = rid
            weekly["avg_daily_kwh"] = weekly["total_kwh"] / 7
            weekly.rename(columns={"date": "week_start"}, inplace=True)
            weekly_data.append(weekly)
        if weekly_data:
            return pd.concat(weekly_data, ignore_index=True)
        return pd.DataFrame()
    def get_house_daily_totals(self) -> pd.DataFrame:
        """
        Отримати добові суми для всього будинку.
        Returns:
            DataFrame з колонками: date, total_kwh, peak_power_w, etc.
        """
        daily = self.aggregate_to_daily()
        if daily.empty:
            return pd.DataFrame()
        house = daily.groupby("date").agg({
            "total_kwh": "sum",
            "peak_power_w": "max",
            "avg_power_w": "sum",  # Сума середніх потужностей по кімнатах
            "avg_temperature_c": "mean",
            "is_weekend": "first",
            "is_holiday": "first"
        }).reset_index()
        house["room_id"] = "house"
        return house
    def prepare_for_forecasting(
        self,
        room_id: Optional[str] = None,
        include_features: bool = True
    ) -> pd.DataFrame:
        """
        Підготувати дані для моделей прогнозування.
        Args:
            room_id: ID кімнати або None для всього будинку
            include_features: Чи додавати ознаки (день тижня, місяць тощо)
        Returns:
            DataFrame готовий для ML
        """
        if room_id:
            df = self.aggregate_to_daily(room_id)
        else:
            df = self.get_house_daily_totals()
        if df.empty:
            return df
        df["date"] = pd.to_datetime(df["date"])
```

```
        df = df.sort_values("date").reset_index(drop=True)
        if include_features:
            # Часові ознаки
            df["day_of_week"] = df["date"].dt.dayofweek
            df["day_of_month"] = df["date"].dt.day
            df["month"] = df["date"].dt.month
            df["week_of_year"] = df["date"].dt.isocalendar().week.astype(int)
            # Лагові ознаки
            df["kwh_lag_1"] = df["total_kwh"].shift(1)
            df["kwh_lag_7"] = df["total_kwh"].shift(7)
            df["kwh_lag_30"] = df["total_kwh"].shift(30)
            # Ковзне середнє
            df["kwh_ma_7"] = df["total_kwh"].rolling(7, min_periods=1).mean()
            df["kwh_ma_30"] = df["total_kwh"].rolling(30, min_periods=1).mean()
            # Заповнення пропусків
            df = df.fillna(method="bfill").fillna(method="ffill")
        return df
    def clear(self):
        """Очистити накопичені дані."""
        self._minute_data.clear()
        self._daily_cache.clear()
    @property
    def total_minutes(self) -> int:
        """Кількість хвилинних записів."""
        return len(self._minute_data)
    def to_dataframe(self) -> pd.DataFrame:
        """Отримати всі хвилинні дані як DataFrame."""
        return pd.DataFrame(self._minute_data)
def create_training_dataset(
    aggregator: DataAggregator,
    room_id: Optional[str] = None,
    holdout_days: int = 20
) -> tuple[pd.DataFrame, pd.DataFrame]:
    """
    Створити навчальний та тестовий набори даних.
    Args:
        aggregator: Агрегатор з даними
        room_id: ID кімнати або None для будинку
        holdout_days: Кількість днів для тестової вибірки
    Returns:
        (train_df, test_df)
    """
    df = aggregator.prepare_for_forecasting(room_id, include_features=True)
    if df.empty or len(df) <= holdout_days:
        return df, pd.DataFrame()
    split_idx = len(df) - holdout_days
    train_df = df.iloc[:split_idx].copy()
    test_df = df.iloc[split_idx:].copy()
    return train_df, test_df
```

Лістинг Г.2

database.py

```
"""
SQLite база даних для зберігання даних симуляції.
"""
import sqlite3
from datetime import datetime
from pathlib import Path
from typing import List, Dict, Any, Optional, Tuple
from dataclasses import asdict
import json
class Database:
    """
    SQLite база даних для зберігання часових рядів
```

```
енергоспоживання та показників датчиків.
"""
def __init__(self, db_path: str = "smart_home_data.db"):
    self.db_path = Path(db_path)
    self.connection: Optional[sqlite3.Connection] = None
    self._ensure_tables()
def _ensure_tables(self):
    """Створення таблиць, якщо не існують."""
    conn = self._get_connection()
    cursor = conn.cursor()
    # Таблиця прогонів симуляції
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS simulation_runs (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            started_at TIMESTAMP NOT NULL,
            completed_at TIMESTAMP,
            duration_days INTEGER NOT NULL,
            start_date DATE NOT NULL,
            end_date DATE,
            total_kwh REAL,
            config_json TEXT
        )
    """)
    # Таблиця хвилинних показників споживання
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS energy_readings (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            run_id INTEGER NOT NULL,
            timestamp TIMESTAMP NOT NULL,
            room_id TEXT NOT NULL,
            device_id TEXT,
            power_w REAL NOT NULL,
            cumulative_kwh REAL NOT NULL,
            FOREIGN KEY (run_id) REFERENCES simulation_runs(id)
        )
    """)
    # Індекс для швидкого пошуку
    cursor.execute("""
        CREATE INDEX IF NOT EXISTS idx_energy_timestamp
        ON energy_readings(timestamp)
    """)
    cursor.execute("""
        CREATE INDEX IF NOT EXISTS idx_energy_room
        ON energy_readings(room_id, timestamp)
    """)
    # Таблиця показників датчиків
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS sensor_readings (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            run_id INTEGER NOT NULL,
            timestamp TIMESTAMP NOT NULL,
            room_id TEXT NOT NULL,
            sensor_id TEXT NOT NULL,
            sensor_type TEXT NOT NULL,
            value_json TEXT NOT NULL,
            unit TEXT,
            FOREIGN KEY (run_id) REFERENCES simulation_runs(id)
        )
    """)
    # Таблиця добових агрегатів
    cursor.execute("""
        CREATE TABLE IF NOT EXISTS daily_aggregates (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            run_id INTEGER NOT NULL,
            date DATE NOT NULL,
```

```
                room_id TEXT NOT NULL,
                total_kwh REAL NOT NULL,
                peak_power_w REAL,
                avg_temperature REAL,
                is_weekend INTEGER,
                is_holiday INTEGER,
                FOREIGN KEY (run_id) REFERENCES simulation_runs(id),
                UNIQUE(run_id, date, room_id)
            )
        """)
        # Таблиця прогнозів
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS forecasts (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                run_id INTEGER NOT NULL,
                generated_at TIMESTAMP NOT NULL,
                model_name TEXT NOT NULL,
                room_id TEXT,
                horizon_days INTEGER NOT NULL,
                forecast_date DATE NOT NULL,
                predicted_kwh REAL NOT NULL,
                lower_bound REAL,
                upper_bound REAL,
                FOREIGN KEY (run_id) REFERENCES simulation_runs(id)
            )
        """)
        conn.commit()
    def _get_connection(self) -> sqlite3.Connection:
        """Отримати з'єднання з базою даних."""
        if self.connection is None:
            self.connection = sqlite3.connect(
                str(self.db_path),
                detect_types=sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES
            )
            self.connection.row_factory = sqlite3.Row
        return self.connection
    def close(self):
        """Закрити з'єднання."""
        if self.connection:
            self.connection.close()
            self.connection = None
    # === Прогони симуляції ===
    def create_simulation_run(
        self,
        duration_days: int,
        start_date: datetime,
        config: Dict[str, Any] = None
    ) -> int:
        """Створити новий прогін симуляції."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO simulation_runs (started_at, duration_days, start_date,
config_json)
            VALUES (?, ?, ?, ?)
        """, (
            datetime.now(),
            duration_days,
            start_date.date(),
            json.dumps(config) if config else None
        ))
        conn.commit()
        return cursor.lastrowid
    def complete_simulation_run(
        self,
```

```
        run_id: int,
        end_date: datetime,
        total_kwh: float
    ):
        """Позначити прогін як завершений."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            UPDATE simulation_runs
            SET completed_at = ?, end_date = ?, total_kwh = ?
            WHERE id = ?
        """, (datetime.now(), end_date.date(), total_kwh, run_id))
        conn.commit()
    def get_simulation_run(self, run_id: int) -> Optional[Dict[str, Any]]:
        """Отримати інформацію про прогін."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM simulation_runs WHERE id = ?", (run_id,))
        row = cursor.fetchone()
        if row:
            return dict(row)
        return None
    def get_all_simulation_runs(self) -> List[Dict[str, Any]]:
        """Отримати всі прогони."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM simulation_runs ORDER BY started_at DESC")
        return [dict(row) for row in cursor.fetchall()]
    # === Показники енергії ===
    def insert_energy_reading(
        self,
        run_id: int,
        timestamp: datetime,
        room_id: str,
        device_id: Optional[str],
        power_w: float,
        cumulative_kwh: float
    ):
        """Вставити показник споживання."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO energy_readings (run_id, timestamp, room_id, device_id,
power_w, cumulative_kwh)
            VALUES (?, ?, ?, ?, ?, ?)
        """, (run_id, timestamp, room_id, device_id, power_w, cumulative_kwh))
    def insert_energy_readings_batch(
        self,
        run_id: int,
        readings: List[Tuple[datetime, str, Optional[str], float, float]]
    ):
        """Пакетна вставка показників споживання."""
        conn = self._get_connection()
        cursor = conn.cursor()
        data = [(run_id, ts, room, dev, pw, cum) for ts, room, dev, pw, cum in
readings]
        cursor.executemany("""
            INSERT INTO energy_readings (run_id, timestamp, room_id, device_id,
power_w, cumulative_kwh)
            VALUES (?, ?, ?, ?, ?, ?)
        """, data)
        conn.commit()
    def get_energy_readings(
        self,
        run_id: int,
```

```
        room_id: Optional[str] = None,
        start_time: Optional[datetime] = None,
        end_time: Optional[datetime] = None
    ) -> List[Dict[str, Any]]:
        """Отримати показники споживання."""
        conn = self._get_connection()
        cursor = conn.cursor()
        query = "SELECT * FROM energy_readings WHERE run_id = ?"
        params = [run_id]
        if room_id:
            query += " AND room_id = ?"
            params.append(room_id)
        if start_time:
            query += " AND timestamp >= ?"
            params.append(start_time)
        if end_time:
            query += " AND timestamp <= ?"
            params.append(end_time)
        query += " ORDER BY timestamp"
        cursor.execute(query, params)
        return [dict(row) for row in cursor.fetchall()]
    # === Показники датчиків ===
    def insert_sensor_reading(
        self,
        run_id: int,
        timestamp: datetime,
        room_id: str,
        sensor_id: str,
        sensor_type: str,
        value: Any,
        unit: str = ""
    ):
        """Вставити показник датчика."""
        conn = self._get_connection()
        cursor = conn.cursor()
        value_json = json.dumps(value) if not isinstance(value, str) else value
        cursor.execute("""
            INSERT INTO sensor_readings (run_id, timestamp, room_id, sensor_id,
sensor_type, value_json, unit)
            VALUES (?, ?, ?, ?, ?, ?, ?)
        """, (run_id, timestamp, room_id, sensor_id, sensor_type, value_json, unit))
    # === Добові агрегати ===
    def insert_daily_aggregate(
        self,
        run_id: int,
        date: datetime,
        room_id: str,
        total_kwh: float,
        peak_power_w: float = None,
        avg_temperature: float = None,
        is_weekend: bool = False,
        is_holiday: bool = False
    ):
        """Вставити добовий агрегат."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            INSERT OR REPLACE INTO daily_aggregates
            (run_id, date, room_id, total_kwh, peak_power_w, avg_temperature,
is_weekend, is_holiday)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            run_id,
            date.date() if isinstance(date, datetime) else date,
            room_id,
```

```
            total_kwh,
            peak_power_w,
            avg_temperature,
            1 if is_weekend else 0,
            1 if is_holiday else 0
        ))
        conn.commit()
    def get_daily_aggregates(
        self,
        run_id: int,
        room_id: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """Отримати добові агрегати."""
        conn = self._get_connection()
        cursor = conn.cursor()
        if room_id:
            cursor.execute("""
                SELECT * FROM daily_aggregates
                WHERE run_id = ? AND room_id = ?
                ORDER BY date
            """, (run_id, room_id))
        else:
            cursor.execute("""
                SELECT * FROM daily_aggregates
                WHERE run_id = ?
                ORDER BY date, room_id
            """, (run_id,))
        return [dict(row) for row in cursor.fetchall()]
    def get_house_daily_totals(self, run_id: int) -> List[Dict[str, Any]]:
        """Отримати добові суми для всього будинку."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            SELECT
                date,
                SUM(total_kwh) as total_kwh,
                MAX(peak_power_w) as peak_power_w,
                AVG(avg_temperature) as avg_temperature,
                is_weekend,
                is_holiday
            FROM daily_aggregates
            WHERE run_id = ?
            GROUP BY date
            ORDER BY date
        """, (run_id,))
        return [dict(row) for row in cursor.fetchall()]
    # === Прогнози ===
    def insert_forecast(
        self,
        run_id: int,
        model_name: str,
        room_id: Optional[str],
        horizon_days: int,
        forecast_date: datetime,
        predicted_kwh: float,
        lower_bound: float = None,
        upper_bound: float = None
    ):
        """Вставити прогноз."""
        conn = self._get_connection()
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO forecasts
            (run_id, generated_at, model_name, room_id, horizon_days, forecast_date,
predicted_kwh, lower_bound, upper_bound)
```

```
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            run_id,
            datetime.now(),
            model_name,
            room_id,
            horizon_days,
            forecast_date.date() if isinstance(forecast_date, datetime) else
forecast_date,
            predicted_kwh,
            lower_bound,
            upper_bound
        ))
        conn.commit()
    def get_forecasts(
        self,
        run_id: int,
        model_name: Optional[str] = None,
        room_id: Optional[str] = None
    ) -> List[Dict[str, Any]]:
        """Отримати прогнози."""
        conn = self._get_connection()
        cursor = conn.cursor()
        query = "SELECT * FROM forecasts WHERE run_id = ?"
        params = [run_id]
        if model_name:
            query += " AND model_name = ?"
            params.append(model_name)
        if room_id:
            query += " AND room_id = ?"
            params.append(room_id)
        query += " ORDER BY forecast_date"
        cursor.execute(query, params)
        return [dict(row) for row in cursor.fetchall()]
    def commit(self):
        """Зафіксувати зміни."""
        if self.connection:
            self.connection.commit()
```

Лістинг Г.3

engine.py

```
"""
Двигун симуляції розумного будинку.
Головний цикл: один крок = 1 хвилина реального часу будинку.
"""
from dataclasses import dataclass, field
from datetime import datetime, timedelta
from typing import Dict, List, Optional, Callable, Any
import random
from .room_simulator import RoomSimulator, RoomState
from .climate_model import ClimateModel, HolidayCalendar
from .resident_behavior import ResidentBehavior, VacationModel, Activity
from .devices import (
    # HVAC
    AirConditioner, AirPurifier, GasBoiler, CirculationPump, ExhaustFan, Humidifier,
    # Appliances
    Refrigerator, Dishwasher, InductionCooktop, CoffeeMaker, Microwave,
    WashingMachine, Dryer, RobotVacuum, HairDryer, ElectricShaver,
    TowelHeater, WaterHeater, ElectricBlanket,
    # Lighting
    LivingRoomLighting, KitchenLighting, BedroomLighting, ChildRoomLighting,
    Nightlight, OfficeLighting, BathroomLighting, HallwayLighting,
    # IT
    SmartTV, Workstation, Monitor, GamingLaptop, LaptopCharger, USBChargingHub,
```

```
    Printer, UPS, NASServer, NetworkSwitch, Router, VideoDoorbell, SmartLock,
    # Sensors
    MotionSensor, PresenceSensor, TemperatureSensor, HumiditySensor,
    DoorWindowSensor, EnergyMeter, FloodSensor, SmokeSensor, VibrationSensor,
MultiSensor
)
@dataclass
class SimulationState:
    """Стан симуляції на момент часу."""
    timestamp: datetime
    simulation_day: int
    outdoor_temperature_c: float
    is_weekend: bool
    is_holiday: bool
    is_vacation: bool
    room_states: Dict[str, RoomState]
    total_power_w: float
    daily_kwh: float
    cumulative_kwh: float
@dataclass
class SimulationConfig:
    """Конфігурація симуляції."""
    start_date: datetime = field(default_factory=lambda: datetime(2024, 1, 1, 0, 0))
    duration_days: int = 120
    time_step_minutes: int = 1
    random_seed: Optional[int] = None
class SimulationEngine:
    """
    Головний двигун симуляції.
    Координує всі компоненти: кімнати, клімат, поведінку мешканців.
    """
    def __init__(self, config: SimulationConfig = None):
        self.config = config or SimulationConfig()
        if self.config.random_seed:
            random.seed(self.config.random_seed)
        # Ініціалізація компонентів
        self.climate = ClimateModel()
        self.calendar = HolidayCalendar()
        self.behavior = ResidentBehavior()
        self.vacations = VacationModel(self.config.duration_days,
self.config.random_seed)
        # Кімнати
        self.rooms: Dict[str, RoomSimulator] = {}
        self._setup_rooms()
        # Стан симуляції
        self._current_time = self.config.start_date
        self._simulation_day = 0
        self._cumulative_kwh = 0.0
        self._daily_kwh = 0.0
        self._last_day = -1
        # Щоденна варіативність для реалістичності
        self._daily_activity_factor = 1.0  # Фактор активності сім'ї (0.7-1.3)
        self._random_events_today = []  # Випадкові події дня
        self._weather_factor = 1.0  # Вплив погоди на споживання
        # Callbacks для подій
        self._step_callbacks: List[Callable[[SimulationState], None]] = []
        self._day_callbacks: List[Callable[[int, float], None]] = []
        # Історія (для запису)
        self.history: List[SimulationState] = []
        self._record_history = False
    def _generate_daily_variance(self, day: int, is_weekend: bool, temperature:
float):
        """
        Генерація щоденної варіативності для реалістичного споживання.
        Кожен день унікальний завдяки:
```

```
        - Випадковому фактору активності сім'ї
        - Випадковим подіям (гості, додаткове прання, тощо)
        - Впливу погоди
        """
        # Фактор активності: деякі дні сім'я більш/менш активна
        # Базова варіація ±15%, з трендом (кілька активних/пасивних днів підряд)
        base_activity = random.gauss(1.0, 0.12)  # Нормальний розподіл
        self._daily_activity_factor = max(0.65, min(1.35, base_activity))
        # Вихідні — більше активності вдома
        if is_weekend:
            self._daily_activity_factor *= random.uniform(1.0, 1.15)
        # Вплив погоди на споживання
        if temperature < 0:
            # Холодно — більше опалення
            self._weather_factor = 1.1 + random.uniform(0, 0.15)
        elif temperature > 28:
            # Спека — більше кондиціонування
            self._weather_factor = 1.1 + random.uniform(0, 0.12)
        else:
            # Нормальна погода — базове споживання
            self._weather_factor = 1.0 + random.uniform(-0.05, 0.08)
        # Випадкові події дня
        self._random_events_today = []
        # 20% шанс на гостей (більше ТВ, світла, готування)
        if random.random() < 0.20:
            self._random_events_today.append("guests")
            self._daily_activity_factor *= 1.15
        # 15% шанс на додаткове прання
        if random.random() < 0.15:
            self._random_events_today.append("extra_laundry")
        # 10% шанс на генеральне прибирання
        if random.random() < 0.10:
            self._random_events_today.append("cleaning")
            self._daily_activity_factor *= 1.08
        # 8% шанс на "лінивий день" — менше активності
        if random.random() < 0.08 and "guests" not in self._random_events_today:
            self._random_events_today.append("lazy_day")
            self._daily_activity_factor *= 0.85
        # 5% шанс на тривалий душ/ванну
        if random.random() < 0.05:
            self._random_events_today.append("long_shower")
    def _setup_rooms(self):
        """Налаштування всіх кімнат з пристроями."""
        self._setup_living_room()
        self._setup_kitchen()
        self._setup_master_bedroom()
        self._setup_child_room()
        self._setup_office()
        self._setup_bathroom()
        self._setup_hallway()
        self._setup_basement()
    def _setup_living_room(self):
        """Налаштування вітальні."""
        room = RoomSimulator("living_room", "Вітальня", 28)
        # Пристрої
        room.add_device(SmartTV("lr_tv", "Samsung QE65Q80C QLED TV", "living_room"))
        room.add_device(LivingRoomLighting("lr_light", "Philips Hue 6x A19",
"living_room"))
        room.add_device(AirConditioner("lr_ac", "Daikin FTXB35C", "living_room"))
        room.add_device(RobotVacuum("lr_vacuum", "Xiaomi Mi Robot Vacuum S10+",
"living_room"))
        # Датчики
        room.add_sensor(MotionSensor("lr_motion", "Aqara Motion Sensor P1",
"living_room"))
```

```
        room.add_sensor(TemperatureSensor("lr_temp", "Aqara WSDCGQ11LM",
"living_room"))
        room.add_sensor(HumiditySensor("lr_hum", "Aqara WSDCGQ11LM", "living_room"))
        self.rooms["living_room"] = room
    def _setup_kitchen(self):
        """Налаштування кухні."""
        room = RoomSimulator("kitchen", "Кухня", 16)
        # Пристрої
        room.add_device(Dishwasher("kit_dish", "Bosch SMS46KI01E", "kitchen"))
        room.add_device(Refrigerator("kit_fridge", "Gorenje RK6193EXL", "kitchen"))
        room.add_device(InductionCooktop("kit_cooktop", "Bosch HEZ39050", "kitchen"))
        room.add_device(CoffeeMaker("kit_coffee", "Bosch CTL636ES6", "kitchen"))
        room.add_device(Microwave("kit_micro", "Philips HR2233/00", "kitchen"))
        room.add_device(KitchenLighting("kit_light", "Philips Hue GU10 4x",
"kitchen"))
        # Датчики
        room.add_sensor(DoorWindowSensor("kit_fridge_door", "Aqara MCCGQ11LM",
"kitchen", "refrigerator"))
        room.add_sensor(SmokeSensor("kit_smoke", "Aqara JY-GZ-01AQ", "kitchen"))
        self.rooms["kitchen"] = room
    def _setup_master_bedroom(self):
        """Налаштування головної спальні."""
        room = RoomSimulator("master_bedroom", "Головна спальня", 18)
        # Пристрої
        room.add_device(AirPurifier("mb_purifier", "Xiaomi Smart Air Purifier 4 Pro",
"master_bedroom"))
        room.add_device(BedroomLighting("mb_light", "IKEA TRADFRI 4x E27",
"master_bedroom"))
        room.add_device(ElectricBlanket("mb_blanket", "Xiaomi Smart Blanket",
"master_bedroom"))
        room.add_device(LaptopCharger("mb_laptop", "Dell XPS 15 Charger",
"master_bedroom"))
        # Датчики
        room.add_sensor(MotionSensor("mb_motion", "Aqara Motion Sensor P1",
"master_bedroom"))
        room.add_sensor(MultiSensor("mb_hub", "Xiaomi Mi Smart Home Hub 2",
"master_bedroom"))
        self.rooms["master_bedroom"] = room
    def _setup_child_room(self):
        """Налаштування дитячої кімнати."""
        room = RoomSimulator("child_room", "Дитяча кімната", 14)
        # Пристрої
        room.add_device(USBChargingHub("cr_usb", "USB Charging Hub 60W",
"child_room"))
        room.add_device(GamingLaptop("cr_laptop", "ASUS ROG Strix G15", "child_room"))
        room.add_device(ChildRoomLighting("cr_backlight", "Philips Hue Play 2x",
"child_room"))
        room.add_device(Nightlight("cr_night", "IKEA STUVA", "child_room"))
        room.add_device(Humidifier("cr_humid", "Boneco U350", "child_room"))
        # Датчики
        room.add_sensor(VibrationSensor("cr_vibr", "Aqara DJT11LM", "child_room"))
        room.add_sensor(TemperatureSensor("cr_temp", "Govee H5102", "child_room"))
        room.add_sensor(HumiditySensor("cr_hum", "Govee H5102", "child_room"))
        self.rooms["child_room"] = room
    def _setup_office(self):
        """Налаштування кабінету."""
        room = RoomSimulator("office", "Кабінет", 12)
        # Пристрої
        room.add_device(Workstation("off_mac", "Apple Mac Studio M2 Max", "office"))
        room.add_device(Monitor("off_monitor", "Dell U2722D 27inch 4K", "office"))
        room.add_device(Printer("off_printer", "Epson EcoTank ET-4800", "office"))
        room.add_device(UPS("off_ups", "APC Back-UPS Pro 1500", "office"))
        room.add_device(OfficeLighting("off_light", "Philips Hue Desk+Ceiling",
"office"))
        self.rooms["office"] = room
```

```
    def _setup_bathroom(self):
        """Налаштування ванної кімнати."""
        room = RoomSimulator("bathroom", "Ванна кімната", 8)
        # Пристрої
        room.add_device(WaterHeater("bath_heater", "Bosch TR5500M 80L", "bathroom"))
        room.add_device(HairDryer("bath_dryer", "Rowenta CV9840", "bathroom"))
        room.add_device(ElectricShaver("bath_shaver", "Braun Series 9 Pro",
"bathroom"))
        room.add_device(ExhaustFan("bath_fan", "Vents VN1", "bathroom"))
        room.add_device(TowelHeater("bath_towel", "Philips HV396", "bathroom"))
        room.add_device(BathroomLighting("bath_light", "Eglo Mosiano 2x8W",
"bathroom"))
        # Датчики
        room.add_sensor(PresenceSensor("bath_presence", "Aqara FP2", "bathroom"))
        self.rooms["bathroom"] = room
    def _setup_hallway(self):
        """Налаштування коридору."""
        room = RoomSimulator("hallway", "Коридор", 10)
        # Пристрої
        room.add_device(HallwayLighting("hall_light", "Philips Hue GU10 2x",
"hallway"))
        room.add_device(VideoDoorbell("hall_bell", "Ring Video Doorbell Pro 2",
"hallway"))
        room.add_device(SmartLock("hall_lock", "Xiaomi Smart Door Lock Pro",
"hallway"))
        # Датчики
        room.add_sensor(DoorWindowSensor("hall_door", "Aqara Door Sensor", "hallway",
"entrance"))
        self.rooms["hallway"] = room
    def _setup_basement(self):
        """Налаштування підвалу/технічного приміщення."""
        room = RoomSimulator("basement", "Підвал", 20)
        # Пристрої
        room.add_device(GasBoiler("base_boiler", "Buderus Logamax plus GB172-24",
"basement"))
        room.add_device(CirculationPump("base_pump", "Grundfos MAGNA3 32-60",
"basement"))
        room.add_device(WashingMachine("base_wash", "AEG OKO_FAVORIT 8kg",
"basement"))
        room.add_device(Dryer("base_dryer", "Miele TWI 180 WP", "basement"))
        room.add_device(NetworkSwitch("base_switch", "TP-Link TL-SG2428P PoE",
"basement"))
        room.add_device(NASServer("base_nas", "Synology DS923+ 4HDD", "basement"))
        room.add_device(Router("base_router", "Ubiquiti UniFi Dream Machine Pro",
"basement"))
        room.add_device(UPS("base_ups", "APC Smart-UPS 1500", "basement"))
        # Датчики
        room.add_sensor(EnergyMeter("base_main_meter", "Shelly Pro 3EM", "basement",
is_main_meter=True))
        room.add_sensor(FloodSensor("base_flood1", "Aqara SJCGQ11LM", "basement",
"washing_machine"))
        room.add_sensor(FloodSensor("base_flood2", "Aqara SJCGQ11LM", "basement",
"water_heater"))
        self.rooms["basement"] = room
    def add_step_callback(self, callback: Callable[[SimulationState], None]):
        """Додати callback для кожного кроку симуляції."""
        self._step_callbacks.append(callback)
    def add_day_callback(self, callback: Callable[[int, float], None]):
        """Додати callback для завершення дня (день, споживання_кВт·год)."""
        self._day_callbacks.append(callback)
    def enable_history_recording(self, enabled: bool = True):
        """Увімкнути запис історії (потребує пам'яті!)."""
        self._record_history = enabled
    def step(self) -> SimulationState:
        """
```

```
        Виконати один крок симуляції (1 хвилина).
        """
        # Перевірка на зміну дня
        if self._current_time.day != self._last_day:
            if self._last_day != -1:
                # Виклик callbacks кінця дня
                for callback in self._day_callbacks:
                    callback(self._simulation_day, self._daily_kwh)
            self._last_day = self._current_time.day
            self._daily_kwh = 0.0
            self._simulation_day += 1
            # Скидання добових лічильників
            for room in self.rooms.values():
                room.reset_daily()
        # Отримання контексту
        temperature = self.climate.get_temperature(self._current_time)
        is_weekend = self.calendar.is_weekend(self._current_time)
        is_holiday = self.calendar.is_holiday(self._current_time)
        is_vacation = self.vacations.is_on_vacation(self._simulation_day)
        # Отримання присутності мешканців
        occupancy = self.behavior.get_room_occupancy(
            self._current_time, is_weekend, is_holiday
        )
        # Якщо відпустка — мінімальна присутність
        if is_vacation:
            occupancy = {}
        # Симуляція кожної кімнати
        room_states = {}
        total_power = 0.0
        for room_id, room in self.rooms.items():
            is_occupied = room_id in occupancy and len(occupancy[room_id]) > 0
            occupant_count = len(occupancy.get(room_id, set()))
            # Виклик спеціальних дій на основі поведінки
            self._trigger_scheduled_actions(room_id, is_occupied)
            state = room.simulate_step(
                current_time=self._current_time,
                temperature_c=temperature,
                is_occupied=is_occupied,
                occupant_count=occupant_count,
                is_weekend=is_weekend,
                is_holiday=is_holiday
            )
            room_states[room_id] = state
            total_power += state.total_power_w
        # Оновлення загальних лічильників
        step_kwh = total_power / 1000 / 60
        self._daily_kwh += step_kwh
        self._cumulative_kwh += step_kwh
        # Формування стану симуляції
        sim_state = SimulationState(
            timestamp=self._current_time,
            simulation_day=self._simulation_day,
            outdoor_temperature_c=temperature,
            is_weekend=is_weekend,
            is_holiday=is_holiday,
            is_vacation=is_vacation,
            room_states=room_states,
            total_power_w=round(total_power, 2),
            daily_kwh=round(self._daily_kwh, 4),
            cumulative_kwh=round(self._cumulative_kwh, 4)
        )
        # Виклик callbacks
        for callback in self._step_callbacks:
            callback(sim_state)
        # Запис в історію
```

```
        if self._record_history:
            self.history.append(sim_state)
        # Просування часу
        self._current_time += timedelta(minutes=self.config.time_step_minutes)
        return sim_state
    def _trigger_scheduled_actions(self, room_id: str, is_occupied: bool):
        """Виклик запланованих дій пристроїв."""
        hour = self._current_time.hour
        minute = self._current_time.minute
        room = self.rooms[room_id]
        # Кухня: готування та посудомийка
        if room_id == "kitchen":
            self._kitchen_actions(room, hour, minute, is_occupied)
        # Ванна: бойлер, фен, бритва
        elif room_id == "bathroom":
            self._bathroom_actions(room, hour, minute, is_occupied)
        # Вітальня: пилосос
        elif room_id == "living_room":
            self._living_room_actions(room, hour, minute)
        # Підвал: прання
        elif room_id == "basement":
            self._basement_actions(room, hour, minute)
    def _kitchen_actions(self, room: RoomSimulator, hour: int, minute: int,
is_occupied: bool):
        """Дії на кухні."""
        cooktop = room.devices.get("kit_cooktop")
        coffee = room.devices.get("kit_coffee")
        dishwasher = room.devices.get("kit_dish")
        microwave = room.devices.get("kit_micro")
        # Сніданок: 7:30 - одна конфорка 20 хв
        if hour == 7 and minute == 30 and is_occupied:
            if cooktop and not cooktop._cooking_until:
                cooktop.start_cooking(self._current_time, 1, 20)
        # Кава: 7:30 та 13:00
        if (hour == 7 and minute == 30) or (hour == 13 and minute == 0):
            if coffee and not coffee._brewing_until:
                coffee.start_brewing(self._current_time)
        # Обід: 13:00 - 2 конфорки 35 хв
        if hour == 13 and minute == 0 and is_occupied:
            if cooktop and not cooktop._cooking_until:
                cooktop.start_cooking(self._current_time, 2, 35)
        # Вечеря: 18:30 - 2-3 конфорки 50 хв
        if hour == 18 and minute == 30 and is_occupied:
            if cooktop and not cooktop._cooking_until:
                burners = random.choice([2, 3])
                cooktop.start_cooking(self._current_time, burners, 50)
        # Посудомийка: після обіду (14:00) та вечері (21:00)
        if dishwasher:
            if (hour == 14 and minute == 0) or (hour == 21 and minute == 0):
                if random.random() < 0.7:  # 70% шанс
                    dishwasher.schedule_cycle(self._current_time)
        # Мікрохвильовка: випадкове використання
        if microwave and is_occupied:
            if hour in [8, 12, 18, 19] and minute == 15:
                if random.random() < 0.5:
                    duration = random.randint(3, 8)
                    microwave.start(self._current_time, duration)
    def _bathroom_actions(self, room: RoomSimulator, hour: int, minute: int,
is_occupied: bool):
        """Дії у ванній."""
        heater = room.devices.get("bath_heater")
        dryer = room.devices.get("bath_dryer")
        shaver = room.devices.get("bath_shaver")
        fan = room.devices.get("bath_fan")
        towel = room.devices.get("bath_towel")
```

```
        # Бойлер: нагрів о 6:30 та 20:00
        if heater:
            if (hour == 6 and minute == 30) or (hour == 20 and minute == 0):
                if not heater._heating_until:
                    duration = random.randint(45, 60)
                    heater.start_heating(self._current_time, duration)
        # Фен: ранок 7:15-7:30
        if dryer and is_occupied:
            if hour == 7 and 15 <= minute <= 20:
                if not dryer._running_until and random.random() < 0.3:
                    duration = random.randint(5, 8)
                    dryer.start(self._current_time, duration)
        # Бритва: будні ранок
        is_weekday = self._current_time.weekday() < 5
        if shaver and is_weekday:
            if hour == 7 and minute == 10:
                shaver.start_charging(self._current_time, 60)
        # Вентилятор та рушникосушарка: після душу (7:00 та 20:30)
        if (hour == 7 and minute == 5) or (hour == 20 and minute == 35):
            if fan:
                fan.activate(self._current_time, 20)
            if towel:
                towel.activate(self._current_time, 30)
    def _living_room_actions(self, room: RoomSimulator, hour: int, minute: int):
        """Дії у вітальні."""
        vacuum = room.devices.get("lr_vacuum")
        # Пилосос: щоденно о 10:00
        if vacuum and hour == 10 and minute == 0:
            if not vacuum._cleaning_until:
                duration = random.randint(45, 60)
                vacuum.start_cleaning(self._current_time, duration)
    def _basement_actions(self, room: RoomSimulator, hour: int, minute: int):
        """Дії у підвалі."""
        washer = room.devices.get("base_wash")
        dryer = room.devices.get("base_dryer")
        # Прання: 2-3 рази на тиждень (субота 10:00, середа 18:00)
        weekday = self._current_time.weekday()
        if washer:
            # Субота вранці
            if weekday == 5 and hour == 10 and minute == 0:
                if not washer._cycle_start:
                    washer.start_cycle(self._current_time, 35, 90)
            # Середа ввечері
            elif weekday == 2 and hour == 18 and minute == 0:
                if not washer._cycle_start:
                    washer.start_cycle(self._current_time, 35, 90)
        # Сушарка: після прання (через 2 години)
        if dryer:
            if weekday == 5 and hour == 12 and minute == 0:
                if not dryer._cycle_start:
                    dryer.start_cycle(self._current_time)
            elif weekday == 2 and hour == 20 and minute == 0:
                if not dryer._cycle_start:
                    dryer.start_cycle(self._current_time)
    def run(self, progress_callback: Callable[[int, int], None] = None) ->
List[SimulationState]:
        """
        Запустити повну симуляцію.
        Args:
            progress_callback: Callback (поточний_крок, всього_кроків)
        Returns:
            Історія симуляції (якщо увімкнено запис)
        """
        total_steps = self.config.duration_days * 24 * 60 //
self.config.time_step_minutes
```

```
        step_count = 0
        while self._simulation_day < self.config.duration_days:
            self.step()
            step_count += 1
            if progress_callback and step_count % 1000 == 0:
                progress_callback(step_count, total_steps)
        # Останній callback дня
        for callback in self._day_callbacks:
            callback(self._simulation_day, self._daily_kwh)
        return self.history
    def reset(self):
        """Скидання симуляції до початкового стану."""
        self._current_time = self.config.start_date
        self._simulation_day = 0
        self._cumulative_kwh = 0.0
        self._daily_kwh = 0.0
        self._last_day = -1
        self.history.clear()
        # Скидання кімнат
        for room in self.rooms.values():
            room._cumulative_kwh = 0.0
            room._daily_kwh = 0.0
    @property
    def current_time(self) -> datetime:
        """Поточний час симуляції."""
        return self._current_time
    @property
    def simulation_day(self) -> int:
        """Поточний день симуляції."""
        return self._simulation_day
    @property
    def total_cumulative_kwh(self) -> float:
        """Загальне накопичене споживання."""
        return self._cumulative_kwh
```

Лістинг Г.4

uci_dataset.py

```
"""
Завантаження та обробка UCI Individual Household Electric Power Consumption Dataset.
Датасет:
https://archive.ics.uci.edu/dataset/235/individual+household+electric+power+consumptio
n
Період: грудень 2006 – листопад 2010 (47 місяців)
Роздільна здатність: 1 хвилина
"""
import numpy as np
import pandas as pd
from pathlib import Path
from typing import Tuple, Optional
from datetime import datetime
import warnings
warnings.filterwarnings('ignore')
DATA_DIR = Path(__file__).parent.parent.parent / "real_data"
def download_uci_dataset(force: bool = False) -> pd.DataFrame:
    """
    Завантаження датасету UCI Individual Household Electric Power Consumption.
    Args:
        force: Примусово перезавантажити, навіть якщо файл існує
    Returns:
        DataFrame з хвилинними даними
    """
    DATA_DIR.mkdir(exist_ok=True)
    cache_file = DATA_DIR / "household_power_raw.parquet"
    if cache_file.exists() and not force:
```

```
            print("Завантаження з кешу...")
            return pd.read_parquet(cache_file)
        print("Завантаження датасету UCI...")
        try:
            from ucimlrepo import fetch_ucirepo
            dataset = fetch_ucirepo(id=235)
            df = dataset.data.features.copy()
        except Exception as e:
            print(f"Помилка ucimlrepo: {e}")
            print("Спроба прямого завантаження...")
            url =
"https://archive.ics.uci.edu/ml/machine-learning-databases/00235/household_power_consu
mption.zip"
            df = pd.read_csv(
                url,
                sep=';',
                parse_dates={'datetime': ['Date', 'Time']},
                infer_datetime_format=True,
                low_memory=False,
                na_values=['?']
            )
        # Обробка дати та часу
        if 'Date' in df.columns and 'Time' in df.columns:
            # Об'єднання дати і часу
            df['datetime'] = pd.to_datetime(
                df['Date'].astype(str) + ' ' + df['Time'].astype(str),
                format='%d/%m/%Y %H:%M:%S',
                errors='coerce'
            )
            df = df.drop(['Date', 'Time'], axis=1)
        elif 'datetime' not in df.columns:
            # Якщо ucimlrepo повернув окремі колонки
            for col in df.columns:
                if 'date' in col.lower():
                    df = df.rename(columns={col: 'date_col'})
                if 'time' in col.lower() and 'sub' not in col.lower():
                    df = df.rename(columns={col: 'time_col'})
            if 'date_col' in df.columns and 'time_col' in df.columns:
                df['datetime'] = pd.to_datetime(
                    df['date_col'].astype(str) + ' ' + df['time_col'].astype(str),
                    errors='coerce'
                )
                df = df.drop(['date_col', 'time_col'], axis=1, errors='ignore')
        # Стандартизація назв колонок
        column_mapping = {
            'Global_active_power': 'global_active_power_kw',
            'Global_reactive_power': 'global_reactive_power_kw',
            'Voltage': 'voltage_v',
            'Global_intensity': 'global_intensity_a',
            'Sub_metering_1': 'sub_metering_1_wh',
            'Sub_metering_2': 'sub_metering_2_wh',
            'Sub_metering_3': 'sub_metering_3_wh',
        }
        df = df.rename(columns=column_mapping)
        # Конвертація типів
        numeric_cols = ['global_active_power_kw', 'global_reactive_power_kw',
                        'voltage_v', 'global_intensity_a',
                        'sub_metering_1_wh', 'sub_metering_2_wh', 'sub_metering_3_wh']
        for col in numeric_cols:
            if col in df.columns:
                df[col] = pd.to_numeric(df[col], errors='coerce')
        # Видалення рядків без дати
        df = df.dropna(subset=['datetime'])
        df = df.set_index('datetime').sort_index()
        # Збереження кешу
```

```
    df.to_parquet(cache_file)
    print(f"Датасет збережено: {cache_file}")
    print(f"Записів: {len(df):,}")
    print(f"Період: {df.index.min()} — {df.index.max()}")
    return df
def prepare_daily_data(df: pd.DataFrame = None, include_features: bool = True) ->
pd.DataFrame:
    """
    Агрегація хвилинних даних до добових.
    Args:
        df: Хвилинні дані (або завантажити автоматично)
        include_features: Додати ознаки (день тижня, місяць, тощо)
    Returns:
        DataFrame з добовими даними
    """
    if df is None:
        df = download_uci_dataset()
    cache_file = DATA_DIR / "household_power_daily.parquet"
    if cache_file.exists() and df is None:
        return pd.read_parquet(cache_file)
    print("Агрегація до добових даних...")
    # Агрегація
    daily = df.resample('D').agg({
        'global_active_power_kw': ['mean', 'sum', 'min', 'max', 'std'],
        'global_reactive_power_kw': 'mean',
        'voltage_v': ['mean', 'min', 'max'],
        'global_intensity_a': 'mean',
        'sub_metering_1_wh': 'sum',
        'sub_metering_2_wh': 'sum',
        'sub_metering_3_wh': 'sum',
    })
    # Flatten колонок
    daily.columns = ['_'.join(col).strip() for col in daily.columns.values]
    # Перейменування головної колонки
    daily = daily.rename(columns={
        'global_active_power_kw_sum': 'total_kwh',
        'global_active_power_kw_mean': 'avg_power_kw',
        'global_active_power_kw_min': 'min_power_kw',
        'global_active_power_kw_max': 'max_power_kw',
        'global_active_power_kw_std': 'std_power_kw',
    })
    # total_kwh: сума kW * (1/60 год) = kWh за хвилину, але у нас просто sum
    # Реально: Global_active_power — це миттєва потужність у kW
    # За добу: avg_power * 24 години = kWh
    daily['total_kwh'] = daily['avg_power_kw'] * 24
    # Конвертація sub_metering з Wh у kWh
    for col in ['sub_metering_1_wh_sum', 'sub_metering_2_wh_sum',
'sub_metering_3_wh_sum']:
        if col in daily.columns:
            new_col = col.replace('wh_sum', 'kwh')
            daily[new_col] = daily[col] / 1000
            daily = daily.drop(col, axis=1)
    # Видалення днів з пропусками
    daily = daily.dropna()
    if include_features:
        # Додавання ознак
        daily['date'] = daily.index
        daily['day_of_week'] = daily.index.dayofweek
        daily['is_weekend'] = daily['day_of_week'].isin([5, 6]).astype(int)
        daily['month'] = daily.index.month
        daily['day_of_month'] = daily.index.day
        daily['week_of_year'] = daily.index.isocalendar().week.astype(int)
        daily['quarter'] = daily.index.quarter
        # Сезон
        daily['season'] = daily['month'].map({
```

```
            12: 0, 1: 0, 2: 0,  # Зима
            3: 1, 4: 1, 5: 1,   # Весна
            6: 2, 7: 2, 8: 2,   # Літо
            9: 3, 10: 3, 11: 3  # Осінь
        })
        # Лаг-ознаки
        daily['total_kwh_lag1'] = daily['total_kwh'].shift(1)
        daily['total_kwh_lag7'] = daily['total_kwh'].shift(7)
        daily['total_kwh_rolling_7'] = daily['total_kwh'].rolling(7).mean()
        daily['total_kwh_rolling_30'] = daily['total_kwh'].rolling(30).mean()
        daily = daily.dropna()
    daily.to_parquet(cache_file)
    print(f"Добові дані збережено: {cache_file}")
    print(f"Днів: {len(daily)}")
    return daily
def prepare_hourly_data(df: pd.DataFrame = None) -> pd.DataFrame:
    """
    Агрегація хвилинних даних до погодинних.
    Returns:
        DataFrame з погодинними даними
    """
    if df is None:
        df = download_uci_dataset()
    cache_file = DATA_DIR / "household_power_hourly.parquet"
    if cache_file.exists() and df is None:
        return pd.read_parquet(cache_file)
    print("Агрегація до погодинних даних...")
    hourly = df.resample('h').agg({
        'global_active_power_kw': 'mean',
        'global_reactive_power_kw': 'mean',
        'voltage_v': 'mean',
        'sub_metering_1_wh': 'sum',
        'sub_metering_2_wh': 'sum',
        'sub_metering_3_wh': 'sum',
    })
    # Конвертація в kWh
    hourly['total_kwh'] = hourly['global_active_power_kw']  # kW * 1 год = kWh
    hourly = hourly.dropna()
    hourly.to_parquet(cache_file)
    print(f"Погодинні дані збережено: {cache_file}")
    return hourly
def get_train_test_split(
    test_days: int = 30,
    validation_days: int = 30
) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    """
    Розділення даних на train/validation/test.
    Args:
        test_days: Кількість днів для тесту
        validation_days: Кількість днів для валідації
    Returns:
        (train_df, validation_df, test_df)
    """
    daily = prepare_daily_data()
    n = len(daily)
    test_start = n - test_days
    val_start = test_start - validation_days
    train = daily.iloc[:val_start].copy()
    validation = daily.iloc[val_start:test_start].copy()
    test = daily.iloc[test_start:].copy()
    print(f"Train: {len(train)} днів ({train.index.min().date()} -
{train.index.max().date()})")
    print(f"Validation: {len(validation)} днів")
    print(f"Test: {len(test)} днів")
    return train, validation, test
```

```
def get_dataset_stats() -> dict:
    """Статистика датасету."""
    daily = prepare_daily_data()
    return {
        'total_days': len(daily),
        'date_range': f"{daily.index.min().date()} – {daily.index.max().date()}",
        'avg_daily_kwh': daily['total_kwh'].mean(),
        'std_daily_kwh': daily['total_kwh'].std(),
        'min_daily_kwh': daily['total_kwh'].min(),
        'max_daily_kwh': daily['total_kwh'].max(),
        'weekend_avg': daily[daily['is_weekend'] == 1]['total_kwh'].mean(),
        'weekday_avg': daily[daily['is_weekend'] == 0]['total_kwh'].mean(),
    }
if __name__ == "__main__":
    # Тест завантаження
    print("=" * 60)
    print("UCI Individual Household Electric Power Consumption Dataset")
    print("=" * 60)
    # Завантаження
    raw = download_uci_dataset()
    print(f"\nХвилинних записів: {len(raw):,}")
    print(f"Колонки: {list(raw.columns)}")
    # Добові дані
    daily = prepare_daily_data(raw)
    print(f"\nДобових записів: {len(daily)}")
    # Статистика
    stats = get_dataset_stats()
    print(f"\nСтатистика:")
    for k, v in stats.items():
        if isinstance(v, float):
            print(f"  {k}: {v:.2f}")
        else:
            print(f"  {k}: {v}")
    # Train/test split
    print("\n" + "=" * 60)
    train, val, test = get_train_test_split()
```

Лістинг Г.5

cost_calculator.py

```
"""
Розрахунок вартості електроенергії.
"""
from dataclasses import dataclass
from datetime import datetime, time
from typing import Dict, List, Optional, Any
import numpy as np
import pandas as pd
@dataclass
class TariffRate:
    """Тарифний план."""
    name: str
    name_en: str
    is_zonal: bool
    day_rate_uah: float
    night_rate_uah: float = None
    day_start: time = time(7, 0)
    day_end: time = time(23, 0)
@dataclass
class CostBreakdown:
    """Розбивка витрат."""
    total_kwh: float
    total_uah: float
    total_eur: float
    day_kwh: float
```

```
    night_kwh: float
    day_cost_uah: float
    night_cost_uah: float
    avg_price_per_kwh: float
@dataclass
class CategoryCost:
    """Витрати по категорії."""
    category_name: str
    category_name_en: str
    kwh: float
    cost_uah: float
    percentage: float
@dataclass
class RoomCost:
    """Витрати по кімнаті."""
    room_id: str
    room_name: str
    kwh: float
    cost_uah: float
    percentage: float
# Стандартні тарифи України
TARIFFS = {
    "zonal": TariffRate(
        name="Зонний тариф",
        name_en="Day/Night Tariff",
        is_zonal=True,
        day_rate_uah=2.64,
        night_rate_uah=1.32,
        day_start=time(7, 0),
        day_end=time(23, 0)
    ),
    "flat": TariffRate(
        name="Одноставковий тариф",
        name_en="Flat Rate",
        is_zonal=False,
        day_rate_uah=2.64
    ),
    "social": TariffRate(
        name="Соціальний тариф",
        name_en="Social Rate",
        is_zonal=False,
        day_rate_uah=1.44  # До 250 кВт·год
    )
}
# Курси валют
EXCHANGE_RATES = {
    "usd_uah": 41.50,
    "eur_uah": 44.80
}
# Категорії пристроїв
DEVICE_CATEGORIES = {
    "heating_cooling": {
        "name": "Опалення/Охолодження",
        "name_en": "Heating/Cooling",
        "types": ["hvac", "heating"]
    },
    "lighting": {
        "name": "Освітлення",
        "name_en": "Lighting",
        "types": ["lighting"]
    },
    "appliances": {
        "name": "Побутова техніка",
        "name_en": "Appliances",
        "types": ["appliance"]
```

```
    },
    "entertainment": {
        "name": "Розваги",
        "name_en": "Entertainment",
        "types": ["entertainment"]
    },
    "it_equipment": {
        "name": "ІТ-обладнання",
        "name_en": "IT Equipment",
        "types": ["it"]
    },
    "security": {
        "name": "Безпека",
        "name_en": "Security",
        "types": ["security"]
    }
}
class CostCalculator:
    """
    Калькулятор вартості електроенергії.
    Підтримує:
    - Різні тарифні плани
    - Розбивку по денному/нічному часу
    - Розбивку по категоріях та кімнатах
    - Конвертацію валют
    """
    def __init__(self, tariff_type: str = "zonal"):
        self.tariff = TARIFFS.get(tariff_type, TARIFFS["zonal"])
        self.exchange_rates = EXCHANGE_RATES.copy()
    def set_tariff(self, tariff_type: str):
        """Встановити тарифний план."""
        if tariff_type in TARIFFS:
            self.tariff = TARIFFS[tariff_type]
    def set_custom_rates(self, day_rate: float, night_rate: float = None):
        """Встановити власні тарифи."""
        self.tariff = TariffRate(
            name="Власний тариф",
            name_en="Custom Rate",
            is_zonal=night_rate is not None,
            day_rate_uah=day_rate,
            night_rate_uah=night_rate,
            day_start=time(7, 0),
            day_end=time(23, 0)
        )
    def calculate_cost(
        self,
        kwh: float,
        hour: int = 12  # Година доби для зонного тарифу
    ) -> float:
        """
        Розрахунок вартості для заданого споживання.
        Args:
            kwh: Споживання у кВт·год
            hour: Година доби (для зонного тарифу)
        Returns:
            Вартість у гривнях
        """
        if self.tariff.is_zonal:
            if self.tariff.day_start.hour <= hour < self.tariff.day_end.hour:
                return kwh * self.tariff.day_rate_uah
            else:
                return kwh * self.tariff.night_rate_uah
        else:
            return kwh * self.tariff.day_rate_uah
    def calculate_period_cost(
```

```
        self,
        hourly_kwh: pd.DataFrame,
        kwh_col: str = "total_kwh",
        hour_col: str = "hour"
    ) -> CostBreakdown:
        """
        Розрахунок вартості за період з погодинними даними.
        Args:
            hourly_kwh: DataFrame з погодинним споживанням
            kwh_col: Колонка зі споживанням
            hour_col: Колонка з годиною
        Returns:
            CostBreakdown з детальною розбивкою
        """
        total_kwh = hourly_kwh[kwh_col].sum()
        if self.tariff.is_zonal:
            # Розділення на денний та нічний час
            hourly_kwh = hourly_kwh.copy()
            if hour_col not in hourly_kwh.columns and "timestamp" in
hourly_kwh.columns:
                hourly_kwh[hour_col] = pd.to_datetime(hourly_kwh["timestamp"]).dt.hour
            day_mask = (hourly_kwh[hour_col] >= self.tariff.day_start.hour) & \
                       (hourly_kwh[hour_col] < self.tariff.day_end.hour)
            day_kwh = hourly_kwh.loc[day_mask, kwh_col].sum()
            night_kwh = hourly_kwh.loc[~day_mask, kwh_col].sum()
            day_cost = day_kwh * self.tariff.day_rate_uah
            night_cost = night_kwh * self.tariff.night_rate_uah
            total_cost = day_cost + night_cost
        else:
            day_kwh = total_kwh
            night_kwh = 0
            day_cost = total_kwh * self.tariff.day_rate_uah
            night_cost = 0
            total_cost = day_cost
        avg_price = total_cost / total_kwh if total_kwh > 0 else 0
        return CostBreakdown(
            total_kwh=round(total_kwh, 2),
            total_uah=round(total_cost, 2),
            total_eur=round(total_cost / self.exchange_rates["eur_uah"], 2),
            day_kwh=round(day_kwh, 2),
            night_kwh=round(night_kwh, 2),
            day_cost_uah=round(day_cost, 2),
            night_cost_uah=round(night_cost, 2),
            avg_price_per_kwh=round(avg_price, 4)
        )
    def calculate_daily_cost(self, daily_kwh: float) -> CostBreakdown:
        """
        Розрахунок добової вартості (спрощений).
        Припускаємо типовий розподіл: 70% день, 30% ніч.
        """
        if self.tariff.is_zonal:
            day_kwh = daily_kwh * 0.70
            night_kwh = daily_kwh * 0.30
            day_cost = day_kwh * self.tariff.day_rate_uah
            night_cost = night_kwh * self.tariff.night_rate_uah
            total_cost = day_cost + night_cost
        else:
            day_kwh = daily_kwh
            night_kwh = 0
            day_cost = daily_kwh * self.tariff.day_rate_uah
            night_cost = 0
            total_cost = day_cost
        avg_price = total_cost / daily_kwh if daily_kwh > 0 else 0
        return CostBreakdown(
            total_kwh=round(daily_kwh, 2),
```

```
            total_uah=round(total_cost, 2),
            total_eur=round(total_cost / self.exchange_rates["eur_uah"], 2),
            day_kwh=round(day_kwh, 2),
            night_kwh=round(night_kwh, 2),
            day_cost_uah=round(day_cost, 2),
            night_cost_uah=round(night_cost, 2),
            avg_price_per_kwh=round(avg_price, 4)
        )
    def calculate_by_rooms(
        self,
        room_consumption: Dict[str, float],
        room_names: Dict[str, str] = None
    ) -> List[RoomCost]:
        """
        Розрахунок витрат по кімнатах.
        Args:
            room_consumption: {room_id: kwh}
            room_names: {room_id: назва кімнати}
        Returns:
            Список RoomCost
        """
        total_kwh = sum(room_consumption.values())
        results = []
        for room_id, kwh in room_consumption.items():
            cost = self.calculate_daily_cost(kwh)
            percentage = (kwh / total_kwh * 100) if total_kwh > 0 else 0
            results.append(RoomCost(
                room_id=room_id,
                room_name=room_names.get(room_id, room_id) if room_names else room_id,
                kwh=round(kwh, 2),
                cost_uah=cost.total_uah,
                percentage=round(percentage, 1)
            ))
        return sorted(results, key=lambda x: x.cost_uah, reverse=True)
    def calculate_by_categories(
        self,
        device_consumption: Dict[str, Dict[str, float]]
    ) -> List[CategoryCost]:
        """
        Розрахунок витрат по категоріях.
        Args:
            device_consumption: {device_id: {"type": "hvac", "kwh": 10.5}}
        Returns:
            Список CategoryCost
        """
        category_kwh: Dict[str, float] = {}
        for device_id, info in device_consumption.items():
            device_type = info.get("type", "other")
            kwh = info.get("kwh", 0)
            # Визначення категорії
            category = "other"
            for cat_id, cat_info in DEVICE_CATEGORIES.items():
                if device_type in cat_info["types"]:
                    category = cat_id
                    break
            category_kwh[category] = category_kwh.get(category, 0) + kwh
        total_kwh = sum(category_kwh.values())
        results = []
        for cat_id, kwh in category_kwh.items():
            cat_info = DEVICE_CATEGORIES.get(cat_id, {"name": "Інше", "name_en":
"Other"})
            cost = self.calculate_daily_cost(kwh)
            percentage = (kwh / total_kwh * 100) if total_kwh > 0 else 0
            results.append(CategoryCost(
                category_name=cat_info["name"],
```

```
                category_name_en=cat_info["name_en"],
                kwh=round(kwh, 2),
                cost_uah=cost.total_uah,
                percentage=round(percentage, 1)
            ))
        return sorted(results, key=lambda x: x.cost_uah, reverse=True)
    def calculate_forecast_cost(
        self,
        forecast_kwh: np.ndarray,
        period_name: str = "місяць"
    ) -> Dict[str, Any]:
        """
        Розрахунок прогнозованих витрат.
        Args:
            forecast_kwh: Масив прогнозованого добового споживання
            period_name: Назва періоду
        Returns:
            Словник з витратами та аналітикою
        """
        total_kwh = np.sum(forecast_kwh)
        cost = self.calculate_daily_cost(total_kwh)
        return {
            "period": period_name,
            "total_kwh": round(total_kwh, 2),
            "total_uah": cost.total_uah,
            "total_eur": cost.total_eur,
            "avg_daily_kwh": round(np.mean(forecast_kwh), 2),
            "max_daily_kwh": round(np.max(forecast_kwh), 2),
            "min_daily_kwh": round(np.min(forecast_kwh), 2),
            "avg_daily_cost_uah": round(cost.total_uah / len(forecast_kwh), 2)
        }
    def compare_periods(
        self,
        current_kwh: float,
        previous_kwh: float
    ) -> Dict[str, Any]:
        """
        Порівняння споживання за періоди.
        Args:
            current_kwh: Поточне споживання
            previous_kwh: Попереднє споживання
        Returns:
            Словник з порівнянням
        """
        current_cost = self.calculate_daily_cost(current_kwh)
        previous_cost = self.calculate_daily_cost(previous_kwh)
        kwh_diff = current_kwh - previous_kwh
        cost_diff = current_cost.total_uah - previous_cost.total_uah
        kwh_percent = ((current_kwh / previous_kwh) - 1) * 100 if previous_kwh > 0
else 0
        cost_percent = ((current_cost.total_uah / previous_cost.total_uah) - 1) * 100
if previous_cost.total_uah > 0 else 0
        return {
            "current_kwh": round(current_kwh, 2),
            "previous_kwh": round(previous_kwh, 2),
            "current_cost_uah": current_cost.total_uah,
            "previous_cost_uah": previous_cost.total_uah,
            "kwh_difference": round(kwh_diff, 2),
            "cost_difference_uah": round(cost_diff, 2),
            "kwh_change_percent": round(kwh_percent, 1),
            "cost_change_percent": round(cost_percent, 1),
            "trend": "зростання" if kwh_diff > 0 else "зниження" if kwh_diff < 0 else
"стабільно"
        }
```

Лістинг Г.6

tariff_dialog.py

```
"""
Діалог налаштування тарифів на електроенергію.
Актуальні тарифи для України (2026 рік).
Джерело: https://index.minfin.com.ua/ua/tariff/electric/
"""
from typing import Dict, Optional
from dataclasses import dataclass, field
from datetime import datetime
import json
from pathlib import Path
try:
    from PyQt6.QtWidgets import (
        QDialog, QVBoxLayout, QHBoxLayout, QLabel, QDoubleSpinBox,
        QGroupBox, QRadioButton, QButtonGroup, QPushButton,
        QTabWidget, QWidget, QFormLayout, QComboBox, QCheckBox,
        QMessageBox, QFrame
    )
    from PyQt6.QtCore import Qt
    from PyQt6.QtGui import QFont
    PYQT_AVAILABLE = True
except ImportError:
    PYQT_AVAILABLE = False
@dataclass
class TariffConfig:
    """Конфігурація тарифів."""
    # Тип тарифу: single, dual, heating
    tariff_type: str = "single"
    # Однозонний тариф (грн/кВт·год)
    single_rate: float = 4.32
    # Двозонний тариф (день/ніч)
    dual_day_rate: float = 4.32
    dual_night_rate: float = 2.16
    night_start_hour: int = 23
    night_end_hour: int = 7
    # Тариф з електроопаленням
    heating_rate_below: float = 2.64  # до 2000 кВт·год
    heating_rate_above: float = 4.32  # понад 2000 кВт·год
    heating_threshold_kwh: int = 2000
    heating_season_start: int = 10  # Жовтень
    heating_season_end: int = 4     # Квітень
    # Курс EUR для конвертації
    eur_rate: float = 44.80
    # Дата оновлення
    last_updated: str = "2026-02-01"
# Шлях до файлу конфігурації
CONFIG_FILE = Path(__file__).parent.parent.parent / "config" / "tariffs.json"
def load_tariff_config() -> TariffConfig:
    """Завантаження конфігурації тарифів."""
    if CONFIG_FILE.exists():
        try:
            with open(CONFIG_FILE, 'r', encoding='utf-8') as f:
                data = json.load(f)
                return TariffConfig(**data)
        except Exception:
            pass
    return TariffConfig()
def save_tariff_config(config: TariffConfig):
    """Збереження конфігурації тарифів."""
    CONFIG_FILE.parent.mkdir(exist_ok=True)
    with open(CONFIG_FILE, 'w', encoding='utf-8') as f:
        json.dump(config.__dict__, f, indent=2, ensure_ascii=False)
if PYQT_AVAILABLE:
```

```
class TariffDialog(QDialog):
    """
    Діалог налаштування тарифів на електроенергію.
    Підтримує:
    - Однозонний тариф
    - Двозонний тариф (день/ніч)
    - Тариф з електроопаленням
    """
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Тарифи на електроенергію")
        self.setMinimumWidth(500)
        self.setMinimumHeight(400)
        self.config = load_tariff_config()
        self._setup_ui()
        self._load_values()
    def _setup_ui(self):
        """Налаштування інтерфейсу."""
        layout = QVBoxLayout(self)
        # Заголовок
        header = QLabel("⚡ Тарифи на електроенергію (Україна, 2026)")
        header.setFont(QFont("Arial", 14, QFont.Weight.Bold))
        layout.addWidget(header)
        # Інформаційний блок
        info = QLabel(
            "Актуальні тарифи згідно Постанови КМУ №632.\n"
            "Діють до 30.04.2026."
        )
        info.setStyleSheet("color: #616161; margin-bottom: 10px;")
        layout.addWidget(info)
        # Вибір типу тарифу
        type_group = QGroupBox("Тип тарифу")
        type_layout = QVBoxLayout(type_group)
        self.tariff_group = QButtonGroup(self)
        self.single_radio = QRadioButton("Однозонний (4.32 грн/кВт·год)")
        self.dual_radio = QRadioButton("Двозонний (день/ніч)")
        self.heating_radio = QRadioButton("З електроопаленням")
        self.tariff_group.addButton(self.single_radio, 0)
        self.tariff_group.addButton(self.dual_radio, 1)
        self.tariff_group.addButton(self.heating_radio, 2)
        type_layout.addWidget(self.single_radio)
        type_layout.addWidget(self.dual_radio)
        type_layout.addWidget(self.heating_radio)
        self.tariff_group.buttonClicked.connect(self._on_tariff_type_changed)
        layout.addWidget(type_group)
        # Вкладки з деталями
        self.tabs = QTabWidget()
        # Вкладка однозонного тарифу
        single_tab = self._create_single_tab()
        self.tabs.addTab(single_tab, "Однозонний")
        # Вкладка двозонного тарифу
        dual_tab = self._create_dual_tab()
        self.tabs.addTab(dual_tab, "Двозонний")
        # Вкладка електроопалення
        heating_tab = self._create_heating_tab()
        self.tabs.addTab(heating_tab, "Електроопалення")
        layout.addWidget(self.tabs)
        # Курс євро
        eur_layout = QHBoxLayout()
        eur_layout.addWidget(QLabel("Курс EUR:"))
        self.eur_spin = QDoubleSpinBox()
        self.eur_spin.setRange(30, 60)
        self.eur_spin.setValue(44.80)
        self.eur_spin.setSuffix(" грн")
        self.eur_spin.setDecimals(2)
```

```
            eur_layout.addWidget(self.eur_spin)
            eur_layout.addStretch()
            layout.addLayout(eur_layout)
            # Кнопки
            buttons_layout = QHBoxLayout()
            reset_btn = QPushButton("Скинути до стандартних")
            reset_btn.clicked.connect(self._reset_defaults)
            buttons_layout.addWidget(reset_btn)
            buttons_layout.addStretch()
            cancel_btn = QPushButton("Скасувати")
            cancel_btn.clicked.connect(self.reject)
            buttons_layout.addWidget(cancel_btn)
            save_btn = QPushButton("Зберегти")
            save_btn.setStyleSheet(
                "background-color: #4CAF50; color: white; padding: 8px 20px;"
            )
            save_btn.clicked.connect(self._save_and_close)
            buttons_layout.addWidget(save_btn)
            layout.addLayout(buttons_layout)
        def _create_single_tab(self) -> QWidget:
            """Вкладка однозонного тарифу."""
            widget = QWidget()
            layout = QFormLayout(widget)
            self.single_rate_spin = QDoubleSpinBox()
            self.single_rate_spin.setRange(0.01, 20.00)
            self.single_rate_spin.setValue(4.32)
            self.single_rate_spin.setSuffix(" грн/кВт·год")
            self.single_rate_spin.setDecimals(2)
            layout.addRow("Тариф:", self.single_rate_spin)
            # Інфо
            info = QLabel(
                "<b>Однозонний тариф</b><br>"
                "Фіксована ціна за всю електроенергію незалежно від часу
доби.<br><br>"
                "<i>Актуальний тариф: 4.32 грн/кВт·год (з 01.06.2024)</i>"
            )
            info.setWordWrap(True)
            info.setStyleSheet("background-color: #E3F2FD; padding: 10px;
border-radius: 5px;")
            layout.addRow(info)
            return widget
        def _create_dual_tab(self) -> QWidget:
            """Вкладка двозонного тарифу."""
            widget = QWidget()
            layout = QFormLayout(widget)
            self.dual_day_spin = QDoubleSpinBox()
            self.dual_day_spin.setRange(0.01, 20.00)
            self.dual_day_spin.setValue(4.32)
            self.dual_day_spin.setSuffix(" грн/кВт·год")
            self.dual_day_spin.setDecimals(2)
            layout.addRow("Денний тариф:", self.dual_day_spin)
            self.dual_night_spin = QDoubleSpinBox()
            self.dual_night_spin.setRange(0.01, 20.00)
            self.dual_night_spin.setValue(2.16)
            self.dual_night_spin.setSuffix(" грн/кВт·год")
            self.dual_night_spin.setDecimals(2)
            layout.addRow("Нічний тариф:", self.dual_night_spin)
            # Години нічного тарифу
            hours_layout = QHBoxLayout()
            self.night_start_spin = QComboBox()
            self.night_end_spin = QComboBox()
            for h in range(24):
                self.night_start_spin.addItem(f"{h:02d}:00", h)
                self.night_end_spin.addItem(f"{h:02d}:00", h)
            self.night_start_spin.setCurrentIndex(23)
```

```
            self.night_end_spin.setCurrentIndex(7)
            hours_layout.addWidget(QLabel("з"))
            hours_layout.addWidget(self.night_start_spin)
            hours_layout.addWidget(QLabel("до"))
            hours_layout.addWidget(self.night_end_spin)
            hours_layout.addStretch()
            layout.addRow("Нічний період:", hours_layout)
            # Інфо
            info = QLabel(
                "<b>Двозонний тариф (день/ніч)</b><br>"
                "Потребує встановлення двозонного лічильника.<br>"
                "Нічний тариф = 50% від денного.<br><br>"
                "<i>День: 4.32 грн | Ніч (23:00-07:00): 2.16 грн</i>"
            )
            info.setWordWrap(True)
            info.setStyleSheet("background-color: #FFF3E0; padding: 10px;
border-radius: 5px;")
            layout.addRow(info)
            return widget
        def _create_heating_tab(self) -> QWidget:
            """Вкладка електроопалення."""
            widget = QWidget()
            layout = QFormLayout(widget)
            self.heating_below_spin = QDoubleSpinBox()
            self.heating_below_spin.setRange(0.01, 20.00)
            self.heating_below_spin.setValue(2.64)
            self.heating_below_spin.setSuffix(" грн/кВт·год")
            self.heating_below_spin.setDecimals(2)
            layout.addRow("До 2000 кВт·год:", self.heating_below_spin)
            self.heating_above_spin = QDoubleSpinBox()
            self.heating_above_spin.setRange(0.01, 20.00)
            self.heating_above_spin.setValue(4.32)
            self.heating_above_spin.setSuffix(" грн/кВт·год")
            self.heating_above_spin.setDecimals(2)
            layout.addRow("Понад 2000 кВт·год:", self.heating_above_spin)
            self.heating_threshold_spin = QDoubleSpinBox()
            self.heating_threshold_spin.setRange(100, 5000)
            self.heating_threshold_spin.setValue(2000)
            self.heating_threshold_spin.setSuffix(" кВт·год")
            self.heating_threshold_spin.setDecimals(0)
            layout.addRow("Поріг:", self.heating_threshold_spin)
            # Інфо
            info = QLabel(
                "<b>Тариф з електроопаленням</b><br>"
                "Для домогосподарств з електричним опаленням.<br>"
                "Діє в опалювальний сезон (жовтень-квітень).<br><br>"
                "<i>До 2000 кВт·год: 2.64 грн | Понад: 4.32 грн</i>"
            )
            info.setWordWrap(True)
            info.setStyleSheet("background-color: #E8F5E9; padding: 10px;
border-radius: 5px;")
            layout.addRow(info)
            return widget
        def _on_tariff_type_changed(self, button):
            """Зміна типу тарифу."""
            idx = self.tariff_group.id(button)
            self.tabs.setCurrentIndex(idx)
        def _load_values(self):
            """Завантаження значень з конфігурації."""
            # Тип тарифу
            if self.config.tariff_type == "single":
                self.single_radio.setChecked(True)
                self.tabs.setCurrentIndex(0)
            elif self.config.tariff_type == "dual":
                self.dual_radio.setChecked(True)
```

```
                self.tabs.setCurrentIndex(1)
            else:
                self.heating_radio.setChecked(True)
                self.tabs.setCurrentIndex(2)
            # Значення
            self.single_rate_spin.setValue(self.config.single_rate)
            self.dual_day_spin.setValue(self.config.dual_day_rate)
            self.dual_night_spin.setValue(self.config.dual_night_rate)
            self.night_start_spin.setCurrentIndex(self.config.night_start_hour)
            self.night_end_spin.setCurrentIndex(self.config.night_end_hour)
            self.heating_below_spin.setValue(self.config.heating_rate_below)
            self.heating_above_spin.setValue(self.config.heating_rate_above)
            self.heating_threshold_spin.setValue(self.config.heating_threshold_kwh)
            self.eur_spin.setValue(self.config.eur_rate)
        def _reset_defaults(self):
            """Скидання до стандартних значень."""
            self.config = TariffConfig()
            self._load_values()
            QMessageBox.information(self, "Скинуто", "Тарифи скинуто до стандартних
значень.")
        def _save_and_close(self):
            """Збереження та закриття."""
            # Тип тарифу
            if self.single_radio.isChecked():
                self.config.tariff_type = "single"
            elif self.dual_radio.isChecked():
                self.config.tariff_type = "dual"
            else:
                self.config.tariff_type = "heating"
            # Значення
            self.config.single_rate = self.single_rate_spin.value()
            self.config.dual_day_rate = self.dual_day_spin.value()
            self.config.dual_night_rate = self.dual_night_spin.value()
            self.config.night_start_hour = self.night_start_spin.currentData()
            self.config.night_end_hour = self.night_end_spin.currentData()
            self.config.heating_rate_below = self.heating_below_spin.value()
            self.config.heating_rate_above = self.heating_above_spin.value()
            self.config.heating_threshold_kwh =
int(self.heating_threshold_spin.value())
            self.config.eur_rate = self.eur_spin.value()
            self.config.last_updated = datetime.now().strftime("%Y-%m-%d")
            save_tariff_config(self.config)
            QMessageBox.information(self, "Збережено", "Тарифи успішно збережено!")
            self.accept()
        def get_config(self) -> TariffConfig:
            """Отримати конфігурацію."""
            return self.config
else:
    class TariffDialog:
        def __init__(self, *args, **kwargs):
            pass
```

Лістинг Г.7

evaluator.py

```
"""
Модуль оцінки та порівняння моделей прогнозування.
"""
import numpy as np
import pandas as pd
from typing import Dict, List, Optional, Any
from dataclasses import dataclass, field
from .sarima_model import ModelMetrics, ForecastResult
@dataclass
class ModelComparison:
```

```
    """Результат порівняння моделей."""
    model_name: str
    metrics: ModelMetrics
    forecast: ForecastResult
    training_time_sec: float = 0
    prediction_time_sec: float = 0
@dataclass
class EvaluationReport:
    """Звіт оцінки моделей."""
    room_id: str
    horizon_days: int
    comparisons: List[ModelComparison] = field(default_factory=list)
    best_model: str = ""
    best_mape: float = 0
class ModelEvaluator:
    """
    Оцінювач моделей прогнозування.
    Проводить валідацію, порівняння та вибір найкращої моделі.
    """
    def __init__(self, holdout_days: int = 20):
        self.holdout_days = holdout_days
        self._reports: Dict[str, EvaluationReport] = {}
    def evaluate_all_models(
        self,
        data: pd.DataFrame,
        room_id: str = "house",
        target_col: str = "total_kwh",
        date_col: str = "date",
        horizons: List[int] = None
    ) -> Dict[int, EvaluationReport]:
        """
        Оцінка всіх моделей для різних горизонтів.
        Args:
            data: DataFrame з даними
            room_id: ID кімнати або "house"
            target_col: Цільова змінна
            date_col: Колонка з датами
            horizons: Горизонти прогнозування [30, 90, 365]
        Returns:
            Словник: горизонт -> EvaluationReport
        """
        import time
        horizons = horizons or [30, 90, 365]
        # Розділення на навчальну та тестову вибірки
        split_idx = len(data) - self.holdout_days
        train_data = data.iloc[:split_idx].copy()
        test_data = data.iloc[split_idx:].copy()
        reports = {}
        for horizon in horizons:
            print(f"\nОцінка моделей для горизонту {horizon} днів...")
            report = EvaluationReport(room_id=room_id, horizon_days=horizon)
            # SARIMA
            try:
                from .sarima_model import SARIMAModel
                print("  SARIMA...", end=" ")
                start = time.time()
                model = SARIMAModel(auto_select=True)
                model.fit(train_data, target_col)
                train_time = time.time() - start
                start = time.time()
                forecast = model.predict(min(horizon, self.holdout_days))
                pred_time = time.time() - start
                metrics = self._calculate_metrics(
                    test_data[target_col].values[:min(horizon, self.holdout_days)],
                    forecast.predictions[:min(horizon, self.holdout_days)]
```

```
        )
        report.comparisons.append(ModelComparison(
            model_name="SARIMA",
            metrics=metrics,
            forecast=forecast,
            training_time_sec=train_time,
            prediction_time_sec=pred_time
        ))
        print(f"MAPE: {metrics.mape:.2f}%")
    except Exception as e:
        print(f"помилка: {e}")
    # Prophet
    try:
        from .prophet_model import ProphetModel
        print("  Prophet...", end=" ")
        start = time.time()
        model = ProphetModel()
        model.fit(train_data, target_col, date_col)
        train_time = time.time() - start
        start = time.time()
        forecast = model.predict(min(horizon, self.holdout_days))
        pred_time = time.time() - start
        metrics = self._calculate_metrics(
            test_data[target_col].values[:min(horizon, self.holdout_days)],
            forecast.predictions[:min(horizon, self.holdout_days)]
        )
        report.comparisons.append(ModelComparison(
            model_name="Prophet",
            metrics=metrics,
            forecast=forecast,
            training_time_sec=train_time,
            prediction_time_sec=pred_time
        ))
        print(f"MAPE: {metrics.mape:.2f}%")
    except Exception as e:
        print(f"помилка: {e}")
    # LSTM
    try:
        from .lstm_model import LSTMModel
        print("  LSTM...", end=" ")
        start = time.time()
        model = LSTMModel(epochs=30, verbose=0)
        model.fit(train_data, target_col)
        train_time = time.time() - start
        start = time.time()
        forecast = model.predict(min(horizon, self.holdout_days))
        pred_time = time.time() - start
        metrics = self._calculate_metrics(
            test_data[target_col].values[:min(horizon, self.holdout_days)],
            forecast.predictions[:min(horizon, self.holdout_days)]
        )
        report.comparisons.append(ModelComparison(
            model_name="LSTM",
            metrics=metrics,
            forecast=forecast,
            training_time_sec=train_time,
            prediction_time_sec=pred_time
        ))
        print(f"MAPE: {metrics.mape:.2f}%")
    except Exception as e:
        print(f"помилка: {e}")
    # XGBoost
    try:
        from .xgboost_model import XGBoostModel
        print("  XGBoost...", end=" ")
```

```
            start = time.time()
            model = XGBoostModel()
            model.fit(train_data, target_col)
            train_time = time.time() - start
            start = time.time()
            forecast = model.predict(min(horizon, self.holdout_days))
            pred_time = time.time() - start
            metrics = self._calculate_metrics(
                test_data[target_col].values[:min(horizon, self.holdout_days)],
                forecast.predictions[:min(horizon, self.holdout_days)]
            )
            report.comparisons.append(ModelComparison(
                model_name="XGBoost",
                metrics=metrics,
                forecast=forecast,
                training_time_sec=train_time,
                prediction_time_sec=pred_time
            ))
            print(f"MAPE: {metrics.mape:.2f}%")
        except Exception as e:
            print(f"помилка: {e}")
        # Ensemble
        try:
            from .ensemble import EnsembleModel
            print("  Ensemble...", end=" ")
            start = time.time()
            model = EnsembleModel()
            # Використовуємо частину train як validation
            val_size = min(14, len(train_data) // 5)
            train_sub = train_data.iloc[:-val_size]
            val_sub = train_data.iloc[-val_size:]
            model.fit(train_sub, val_sub, target_col, date_col)
            train_time = time.time() - start
            start = time.time()
            forecast = model.predict(min(horizon, self.holdout_days))
            pred_time = time.time() - start
            metrics = self._calculate_metrics(
                test_data[target_col].values[:min(horizon, self.holdout_days)],
                forecast.predictions[:min(horizon, self.holdout_days)]
            )
            report.comparisons.append(ModelComparison(
                model_name="Ensemble",
                metrics=metrics,
                forecast=forecast,
                training_time_sec=train_time,
                prediction_time_sec=pred_time
            ))
            print(f"MAPE: {metrics.mape:.2f}%")
        except Exception as e:
            print(f"помилка: {e}")
        # Визначення найкращої моделі
        if report.comparisons:
            best = min(report.comparisons, key=lambda x: x.metrics.mape)
            report.best_model = best.model_name
            report.best_mape = best.metrics.mape
        reports[horizon] = report
        self._reports[f"{room_id}_{horizon}"] = report
    return reports
@staticmethod
def _calculate_metrics(y_true: np.ndarray, y_pred: np.ndarray) -> ModelMetrics:
    """Розрахунок метрик."""
    # Вирівнювання довжин
    min_len = min(len(y_true), len(y_pred))
    y_true = y_true[:min_len]
    y_pred = y_pred[:min_len]
```

```
        mae = np.mean(np.abs(y_true - y_pred))
        rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
        mask = y_true != 0
        if mask.sum() > 0:
            mape = np.mean(np.abs((y_true[mask] - y_pred[mask]) / y_true[mask])) * 100
        else:
            mape = 0
        ss_res = np.sum((y_true - y_pred) ** 2)
        ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
        r2 = 1 - (ss_res / ss_tot) if ss_tot != 0 else 0
        return ModelMetrics(mae=mae, rmse=rmse, mape=mape, r2=r2)
    def generate_comparison_table(
        self,
        report: EvaluationReport
    ) -> pd.DataFrame:
        """Генерація таблиці порівняння моделей."""
        rows = []
        for comp in report.comparisons:
            rows.append({
                "Модель": comp.model_name,
                "MAE (кВт·год)": round(comp.metrics.mae, 3),
                "RMSE": round(comp.metrics.rmse, 3),
                "MAPE (%)": round(comp.metrics.mape, 2),
                "R²": round(comp.metrics.r2, 3),
                "Час навч. (с)": round(comp.training_time_sec, 2),
                "Час прогн. (с)": round(comp.prediction_time_sec, 2)
            })
        df = pd.DataFrame(rows)
        return df.sort_values("MAPE (%)")
    def get_report(self, room_id: str, horizon: int) -> Optional[EvaluationReport]:
        """Отримати збережений звіт."""
        key = f"{room_id}_{horizon}"
        return self._reports.get(key)
    def get_best_model_recommendation(
        self,
        reports: Dict[int, EvaluationReport]
    ) -> Dict[int, str]:
        """
        Рекомендація найкращої моделі для кожного горизонту.
        Returns:
            Словник: горизонт -> назва моделі
        """
        recommendations = {}
        for horizon, report in reports.items():
            if report.best_model:
                recommendations[horizon] = report.best_model
            else:
                # За замовчуванням
                if horizon <= 30:
                    recommendations[horizon] = "SARIMA"
                elif horizon <= 90:
                    recommendations[horizon] = "Prophet"
                else:
                    recommendations[horizon] = "Ensemble"
        return recommendations
```

Лістинг Г.8

main.py

```
"""
Точка входу: запуск симуляції та GUI.
Модуль прогнозування енергоспоживання IoT-системи розумного будинку.
Використання:
    python main.py              # Запуск GUI
    python main.py --cli        # Запуск у консольному режимі
```

```
    python main.py --help       # Допомога
"""
import argparse
import sys
from datetime import datetime
from pathlib import Path
def run_cli_simulation(days: int = 120, output_dir: str = "output"):
    """
    Запуск симуляції у консольному режимі.
    Args:
        days: Кількість днів симуляції
        output_dir: Директорія для результатів
    """
    print("=" * 60)
    print("СИМУЛЯЦІЯ ЕНЕРГОСПОЖИВАННЯ РОЗУМНОГО БУДИНКУ")
    print("=" * 60)
    print()
    # Імпорт модулів
    from simulation import SimulationEngine, SimulationConfig
    from data import DataAggregator, Database
    from financial import CostCalculator
    # Створення директорії для результатів
    output_path = Path(output_dir)
    output_path.mkdir(exist_ok=True)
    # Конфігурація
    config = SimulationConfig(
        start_date=datetime(2024, 1, 1),
        duration_days=days
    )
    print(f"Період симуляції: {config.start_date.date()} - ", end="")
    print(f"{(config.start_date.replace(day=1) +
__import__('datetime').timedelta(days=days)).date()}")
    print(f"Тривалість: {days} днів")
    print()
    # Ініціалізація
    engine = SimulationEngine(config)
    aggregator = DataAggregator()
    calculator = CostCalculator()
    # Callback для прогресу
    def progress_callback(step, total):
        percent = step / total * 100
        bar_len = 40
        filled = int(bar_len * step / total)
        bar = '' * filled + '' * (bar_len - filled)
        print(f"\rПрогрес: [{bar}] {percent:.1f}% ({step}/{total})", end="",
flush=True)
    # Callback для дня
    daily_data = []
    def day_callback(day, kwh):
        daily_data.append({
            "day": day,
            "kwh": kwh,
            "date": engine.current_time.date()
        })
    engine.add_day_callback(day_callback)
    # Запуск симуляції
    print("Запуск симуляції...")
    engine.run(progress_callback)
    print()
    print()
    # Результати
    print("=" * 60)
    print("РЕЗУЛЬТАТИ СИМУЛЯЦІЇ")
    print("=" * 60)
    print()
```

```
total_kwh = engine.total_cumulative_kwh
cost = calculator.calculate_daily_cost(total_kwh)
print(f"Загальне споживання: {total_kwh:.2f} кВт·год")
print(f"Середньодобове споживання: {total_kwh / days:.2f} кВт·год")
print(f"Загальна вартість: {cost.total_uah:.2f} грн ({cost.total_eur:.2f} EUR)")
print()
# Споживання по кімнатах
print("Споживання по кімнатах:")
room_consumption = {}
for room_id, room in engine.rooms.items():
    room_consumption[room_id] = room.cumulative_kwh
    print(f"  {room.room_name}: {room.cumulative_kwh:.2f} кВт·год")
print()
# Збереження результатів
print("Збереження результатів...")
# CSV з добовими даними
import csv
csv_path = output_path / "daily_consumption.csv"
with open(csv_path, 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=["day", "date", "kwh"])
    writer.writeheader()
    writer.writerows(daily_data)
print(f"  Добові дані: {csv_path}")
# SQLite база
db_path = output_path / "simulation_data.db"
db = Database(str(db_path))
run_id = db.create_simulation_run(days, config.start_date)
for data in daily_data:
    db.insert_daily_aggregate(
        run_id=run_id,
        date=data["date"],
        room_id="house",
        total_kwh=data["kwh"]
    )
for room_id, kwh in room_consumption.items():
    db.insert_daily_aggregate(
        run_id=run_id,
        date=datetime.now(),
        room_id=room_id,
        total_kwh=kwh
    )
db.complete_simulation_run(run_id, datetime.now(), total_kwh)
db.close()
print(f"  База даних: {db_path}")
print()
print("Симуляцію завершено!")
print()
# Прогнозування (якщо доступне)
try:
    print("Перевірка доступності прогнозування...")
    from forecasting import EnsembleModel
    print("Запуск прогнозування... (це може зайняти кілька хвилин)")
    # Підготовка даних
    import pandas as pd
    df = pd.DataFrame(daily_data)
    df['date'] = pd.to_datetime(df['date'])
    df['total_kwh'] = df['kwh']
    # Розділення
    train_size = len(df) - 20
    train_df = df.iloc[:train_size]
    val_df = df.iloc[train_size:]
    # Навчання ансамблю
    model = EnsembleModel()
    model.fit(train_df, val_df, "total_kwh", "date")
    # Прогноз на 30 днів
```

```
        forecast = model.predict(30)
        print()
        print("ПРОГНОЗ НА 30 ДНІВ:")
        forecast_total = sum(forecast.predictions)
        forecast_cost = calculator.calculate_daily_cost(forecast_total)
        print(f"  Прогнозоване споживання: {forecast_total:.2f} кВт·год")
        print(f"  Прогнозована вартість: {forecast_cost.total_uah:.2f} грн")
    except ImportError as e:
        print(f"Прогнозування недоступне: {e}")
        print("Встановіть залежності: pip install statsmodels prophet tensorflow
xgboost")
    print()
    print("=" * 60)
    return 0
def main():
    """Головна функція."""
    parser = argparse.ArgumentParser(
        description="Прогнозування енергоспоживання розумного будинку"
    )
    parser.add_argument(
        "--cli",
        action="store_true",
        help="Запуск у консольному режимі (без GUI)"
    )
    parser.add_argument(
        "--days",
        type=int,
        default=120,
        help="Кількість днів симуляції (за замовчуванням: 120)"
    )
    parser.add_argument(
        "--output",
        type=str,
        default="output",
        help="Директорія для результатів (за замовчуванням: output)"
    )
    args = parser.parse_args()
    if args.cli:
        return run_cli_simulation(args.days, args.output)
    else:
        # Запуск GUI
        try:
            from ui import run_gui
            run_gui()
        except ImportError:
            print("GUI недоступний. Встановіть PyQt6: pip install PyQt6")
            print("Або запустіть у консольному режимі: python main.py --cli")
            return 1
    return 0
if __name__ == "__main__":
    sys.exit(main())
```

Лістинг Г.9

run_gui.py

```
"""Запуск GUI додатку."""
import sys
import os
from pathlib import Path
# Вимкнення буферизації виводу
os.environ['PYTHONUNBUFFERED'] = '1'
# Додаємо шлях до пакету
sys.path.insert(0, str(Path(__file__).parent / "smart_home_energy"))
if __name__ == "__main__":
    try:
```

```
        from PyQt6.QtWidgets import QApplication
        from ui.main_window import MainWindow, PYQT_AVAILABLE
        if not PYQT_AVAILABLE:
            print("PyQt6 недоступний")
            sys.exit(1)
        app = QApplication(sys.argv)
        app.setStyle("Fusion")
        window = MainWindow()
        window.show()
        # Запуск event loop
        result = app.exec()
        sys.exit(result)
    except Exception as e:
        print(f"Критична помилка: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```

Лістинг Г.10

run_debug.py

```
"""Запуск GUI з детальним логуванням."""
import sys
import os
import traceback
from datetime import datetime
from pathlib import Path
# Логування у файл
log_file = Path(__file__).parent / "gui_debug.log"
def log(msg):
    timestamp = datetime.now().strftime("%H:%M:%S.%f")[:-3]
    line = f"[{timestamp}] {msg}"
    print(line)
    with open(log_file, "a", encoding="utf-8") as f:
        f.write(line + "\n")
# Очистка логу
with open(log_file, "w", encoding="utf-8") as f:
    f.write(f"=== GUI Debug Log {datetime.now()} ===\n")
try:
    log("1. Налаштування шляхів...")
    sys.path.insert(0, str(Path(__file__).parent / "smart_home_energy"))
    log("2. Імпорт PyQt6...")
    from PyQt6.QtWidgets import QApplication, QMessageBox
    from PyQt6.QtCore import Qt
    log("   PyQt6 OK")
    log("3. Створення QApplication...")
    app = QApplication(sys.argv)
    app.setStyle("Fusion")
    log("   QApplication OK")
    log("4. Імпорт MainWindow...")
    from ui.main_window import MainWindow
    log("   MainWindow OK")
    log("5. Створення MainWindow...")
    window = MainWindow()
    log("   MainWindow створено")
    log("6. Показ вікна...")
    window.show()
    window.raise_()
    window.activateWindow()
    log("   Вікно показано")
    log("7. Запуск event loop...")
    result = app.exec()
    log(f"8. Event loop завершено з кодом: {result}")
except Exception as e:
```

```
    error_msg = f"ПОМИЛКА: {e}\n{traceback.format_exc()}"
    log(error_msg)
    # Показати діалог з помилкою
    try:
        from PyQt6.QtWidgets import QApplication, QMessageBox
        if not QApplication.instance():
            app = QApplication(sys.argv)
        msg = QMessageBox()
        msg.setIcon(QMessageBox.Icon.Critical)
        msg.setWindowTitle("Помилка")
        msg.setText(str(e))
        msg.setDetailedText(traceback.format_exc())
        msg.exec()
    except:
        pass
    sys.exit(1)
log("=== Завершено ===")
```

Лістинг Г.11

debug_gui.py

```
"""Діагностика GUI."""
import sys
print("1. Імпорт PyQt6...")
from PyQt6.QtWidgets import QApplication
print("   OK")
print("2. Створення QApplication...")
app = QApplication(sys.argv)
print("   OK")
print("3. Імпорт MainWindow...")
sys.path.insert(0, "smart_home_energy")
from ui.main_window import MainWindow
print("   OK")
print("4. Створення MainWindow...")
try:
    window = MainWindow()
    print("   OK")
except Exception as e:
    print(f"   ПОМИЛКА: {e}")
    import traceback
    traceback.print_exc()
    sys.exit(1)
print("5. Показ вікна...")
window.show()
print("   OK")
print("6. Запуск event loop...")
sys.exit(app.exec())
```

Лістинг Г.12

test_simulation.py

```
"""
Тести двигуна симуляції.
"""
import unittest
from datetime import datetime, timedelta
import sys
from pathlib import Path
# Додавання шляху до модулів
sys.path.insert(0, str(Path(__file__).parent.parent))
from simulation import SimulationEngine, SimulationConfig
from simulation.climate_model import ClimateModel, HolidayCalendar
```

```
from simulation.resident_behavior import ResidentBehavior, VacationModel
class TestClimateModel(unittest.TestCase):
    """Тести кліматичної моделі."""
    def setUp(self):
        self.climate = ClimateModel()
    def test_winter_temperature(self):
        """Зимова температура має бути низькою."""
        dt = datetime(2024, 1, 15, 12, 0)
        temp = self.climate.get_temperature(dt)
        self.assertLess(temp, 10)  # Січень - холодно
        self.assertGreater(temp, -20)  # Але не екстремально
    def test_summer_temperature(self):
        """Літня температура має бути високою."""
        dt = datetime(2024, 7, 15, 14, 0)
        temp = self.climate.get_temperature(dt)
        self.assertGreater(temp, 15)  # Липень - тепло
        self.assertLess(temp, 40)  # Але не екстремально
    def test_daily_variation(self):
        """Температура має змінюватися протягом доби."""
        dt_morning = datetime(2024, 6, 15, 5, 0)
        dt_afternoon = datetime(2024, 6, 15, 15, 0)
        temp_morning = self.climate.get_temperature(dt_morning)
        temp_afternoon = self.climate.get_temperature(dt_afternoon)
        # Вдень тепліше, ніж вранці
        self.assertGreater(temp_afternoon, temp_morning)
    def test_heating_season(self):
        """Визначення опалювального сезону."""
        winter = datetime(2024, 1, 15)
        summer = datetime(2024, 7, 15)
        self.assertTrue(self.climate.is_heating_season(winter))
        self.assertFalse(self.climate.is_heating_season(summer))
class TestHolidayCalendar(unittest.TestCase):
    """Тести календаря свят."""
    def setUp(self):
        self.calendar = HolidayCalendar()
    def test_new_year_holiday(self):
        """Новий рік - свято."""
        dt = datetime(2024, 1, 1)
        self.assertTrue(self.calendar.is_holiday(dt))
    def test_regular_day(self):
        """Звичайний день - не свято."""
        dt = datetime(2024, 3, 15)
        self.assertFalse(self.calendar.is_holiday(dt))
    def test_weekend(self):
        """Вихідні дні."""
        saturday = datetime(2024, 1, 6)  # Субота
        monday = datetime(2024, 1, 8)    # Понеділок
        self.assertTrue(self.calendar.is_weekend(saturday))
        self.assertFalse(self.calendar.is_weekend(monday))
class TestResidentBehavior(unittest.TestCase):
    """Тести поведінкової моделі."""
    def setUp(self):
        self.behavior = ResidentBehavior()
    def test_morning_bathroom_occupancy(self):
        """Вранці хтось має бути у ванній."""
        dt = datetime(2024, 1, 15, 7, 15)  # 7:15 ранку, будній день
        occupancy = self.behavior.get_room_occupancy(dt, False, False)
        # Хоча б одна кімната зайнята
        self.assertTrue(any(len(residents) > 0 for residents in occupancy.values()))
    def test_workday_office_occupancy(self):
        """У робочий час батько має бути в кабінеті."""
        # Перевіряємо кілька моментів робочого часу
        found_office = False
        for hour in [10, 11, 14, 15, 16]:
            dt = datetime(2024, 1, 15, hour, 30)  # Середина години, будній день
```

```
            occupancy = self.behavior.get_room_occupancy(dt, False, False)
            if "office" in occupancy:
                found_office = True
                break
        # Хоча б один раз кабінет має бути зайнятий
        self.assertTrue(found_office, "Кабінет має бути зайнятий хоча б раз у робочий
час")
    def test_night_bedroom_occupancy(self):
        """Вночі люди в спальнях."""
        dt = datetime(2024, 1, 15, 3, 0)  # 3:00 ночі
        occupancy = self.behavior.get_room_occupancy(dt, False, False)
        # Спальні зайняті
        bedrooms_occupied = (
            "master_bedroom" in occupancy or
            "child_room" in occupancy
        )
        self.assertTrue(bedrooms_occupied)
class TestVacationModel(unittest.TestCase):
    """Тести моделі відпусток."""
    def test_vacation_periods_generated(self):
        """Відпустки генеруються."""
        vacation = VacationModel(simulation_days=120, seed=42)
        self.assertGreater(len(vacation.vacation_periods), 0)
    def test_vacation_detection(self):
        """Виявлення відпустки."""
        vacation = VacationModel(simulation_days=120, seed=42)
        # Перевірка, що хоча б один день у відпустці
        vacation_days = [d for d in range(120) if vacation.is_on_vacation(d)]
        self.assertGreater(len(vacation_days), 0)
class TestSimulationEngine(unittest.TestCase):
    """Тести двигуна симуляції."""
    def setUp(self):
        self.config = SimulationConfig(
            start_date=datetime(2024, 1, 1),
            duration_days=3  # Короткий тест
        )
        self.engine = SimulationEngine(self.config)
    def test_engine_initialization(self):
        """Двигун ініціалізується коректно."""
        self.assertEqual(len(self.engine.rooms), 8)
        self.assertEqual(self.engine.simulation_day, 0)
    def test_single_step(self):
        """Один крок симуляції."""
        state = self.engine.step()
        self.assertIsNotNone(state)
        self.assertGreaterEqual(state.total_power_w, 0)
        self.assertEqual(len(state.room_states), 8)
    def test_multiple_steps(self):
        """Декілька кроків симуляції."""
        for _ in range(100):
            state = self.engine.step()
        self.assertGreater(state.cumulative_kwh, 0)
    def test_room_consumption_positive(self):
        """Споживання кімнат позитивне."""
        state = self.engine.step()
        for room_id, room_state in state.room_states.items():
            self.assertGreaterEqual(room_state.total_power_w, 0)
    def test_simulation_reset(self):
        """Скидання симуляції."""
        # Виконати кілька кроків
        for _ in range(50):
            self.engine.step()
        self.assertGreater(self.engine.total_cumulative_kwh, 0)
        # Скинути
        self.engine.reset()
```

```
        self.assertEqual(self.engine.simulation_day, 0)
        self.assertEqual(self.engine.total_cumulative_kwh, 0)
class TestRealisticConsumption(unittest.TestCase):
    """Тести реалістичності споживання."""
    def test_daily_consumption_range(self):
        """Добове споживання в реалістичному діапазоні."""
        config = SimulationConfig(
            start_date=datetime(2024, 1, 1),
            duration_days=7
        )
        engine = SimulationEngine(config)
        daily_kwh = []
        def day_callback(day, kwh):
            daily_kwh.append(kwh)
        engine.add_day_callback(day_callback)
        engine.run()
        avg_daily = sum(daily_kwh) / len(daily_kwh)
        # Середнє добове споживання для будинку ~126 м²
        # має бути в межах 10-40 кВт·год
        self.assertGreater(avg_daily, 5)
        self.assertLess(avg_daily, 50)
if __name__ == "__main__":
    unittest.main()
```

Лістинг Г.13

main_window.py

```
"""
Головне вікно GUI додатку.
"""
import sys
from datetime import datetime
from typing import Optional, Dict, Any
import threading
try:
    from PyQt6.QtWidgets import (
        QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
        QTabWidget, QPushButton, QLabel, QProgressBar, QFrame,
        QSplitter, QStatusBar, QMenuBar, QMenu, QMessageBox,
        QSpinBox, QComboBox, QGroupBox, QGridLayout
    )
    from PyQt6.QtCore import Qt, QTimer, pyqtSignal, QThread
    from PyQt6.QtGui import QAction, QFont
    PYQT_AVAILABLE = True
except ImportError:
    PYQT_AVAILABLE = False
    print("PyQt6 не встановлено. GUI недоступний.")
if PYQT_AVAILABLE:
    class SimulationWorker(QThread):
        """Фоновий потік для симуляції."""
        progress = pyqtSignal(int, int)
        finished = pyqtSignal()
        error = pyqtSignal(str)
        step_data = pyqtSignal(object)
        def __init__(self, engine, parent=None):
            super().__init__(parent)
            self.engine = engine
            self._running = True
            self._paused = False
        def run(self):
            try:
                total_steps = self.engine.config.duration_days * 24 * 60
                step = 0
                while self._running and self.engine.simulation_day <
self.engine.config.duration_days:
```

```
                # Пауза — чекаємо поки не відновлять
                while self._paused and self._running:
                    self.msleep(100)
                if not self._running:
                    break
                state = self.engine.step()
                step += 1
                # Emit progress every 100 steps
                if step % 100 == 0:
                    self.progress.emit(step, total_steps)
                    self.step_data.emit(state)
            self.finished.emit()
        except Exception as e:
            self.error.emit(str(e))
    def pause(self):
        """Призупинити симуляцію."""
        self._paused = True
    def resume(self):
        """Відновити симуляцію."""
        self._paused = False
    def stop(self):
        """Зупинити симуляцію повністю."""
        self._running = False
        self._paused = False
    @property
    def is_paused(self):
        return self._paused
class MainWindow(QMainWindow):
    """
    Головне вікно додатку.
    Містить:
    - План будинку (ліва панель)
    - Поточне споживання (центр)
    - Накопичені дані та прогноз (права панель)
    """
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Прогнозування енергоспоживання розумного будинку")
        self.setMinimumSize(1400, 900)
        # Ініціалізація компонентів
        self.engine = None
        self.worker = None
        self.aggregator = None
        self.is_running = False
        self._setup_ui()
        self._setup_menu()
        self._setup_statusbar()
        self._connect_signals()
    def _setup_ui(self):
        """Налаштування інтерфейсу."""
        central = QWidget()
        self.setCentralWidget(central)
        main_layout = QVBoxLayout(central)
        # Панель управління
        control_panel = self._create_control_panel()
        main_layout.addWidget(control_panel)
        # Основний splitter
        splitter = QSplitter(Qt.Orientation.Horizontal)
        # Ліва панель — план будинку
        left_panel = self._create_floor_plan_panel()
        splitter.addWidget(left_panel)
        # Центральна панель — поточне споживання
        center_panel = self._create_current_panel()
        splitter.addWidget(center_panel)
        # Права панель — накопичення та прогноз
```

```
        right_panel = self._create_forecast_panel()
        splitter.addWidget(right_panel)
        splitter.setSizes([400, 500, 500])
        main_layout.addWidget(splitter)
    def _create_control_panel(self) -> QWidget:
        """Панель управління симуляцією."""
        panel = QGroupBox("Управління симуляцією")
        layout = QHBoxLayout(panel)
        # Параметри
        layout.addWidget(QLabel("Днів симуляції:"))
        self.days_spin = QSpinBox()
        self.days_spin.setRange(30, 365)
        self.days_spin.setValue(120)
        layout.addWidget(self.days_spin)
        layout.addWidget(QLabel("Швидкість:"))
        self.speed_combo = QComboBox()
        self.speed_combo.addItems(["×1", "×10", "×100", "Максимум"])
        self.speed_combo.setCurrentIndex(3)
        layout.addWidget(self.speed_combo)
        layout.addStretch()
        # Кнопки
        self.start_btn = QPushButton("▶ Запустити")
        self.start_btn.setStyleSheet("background-color: #4CAF50; color: white;
padding: 8px 16px;")
        layout.addWidget(self.start_btn)
        self.pause_btn = QPushButton("⏸ Пауза")
        self.pause_btn.setEnabled(False)
        layout.addWidget(self.pause_btn)
        self.reset_btn = QPushButton("↺ Скинути")
        self.reset_btn.setEnabled(False)
        layout.addWidget(self.reset_btn)
        # Прогрес
        self.progress_bar = QProgressBar()
        self.progress_bar.setMinimum(0)
        self.progress_bar.setMaximum(100)
        self.progress_bar.setValue(0)
        self.progress_bar.setTextVisible(True)
        layout.addWidget(self.progress_bar)
        return panel
    def _create_floor_plan_panel(self) -> QWidget:
        """Панель з планом будинку."""
        panel = QGroupBox("План будинку")
        layout = QVBoxLayout(panel)
        # Заголовок з часом симуляції
        time_layout = QHBoxLayout()
        self.sim_time_label = QLabel("Час: --:--")
        self.sim_time_label.setFont(QFont("Arial", 14, QFont.Weight.Bold))
        time_layout.addWidget(self.sim_time_label)
        self.sim_date_label = QLabel("Дата: ----.--.--")
        time_layout.addWidget(self.sim_date_label)
        time_layout.addStretch()
        self.temp_label = QLabel("Температура: --°C")
        time_layout.addWidget(self.temp_label)
        layout.addLayout(time_layout)
        # Placeholder для SVG-плану
        # У повній реалізації тут буде інтерактивний FloorPlanWidget
        from .floor_plan import FloorPlanWidget
        self.floor_plan = FloorPlanWidget()
        layout.addWidget(self.floor_plan)
        return panel
    def _create_current_panel(self) -> QWidget:
        """Панель поточного споживання."""
        panel = QGroupBox("Поточне споживання")
        layout = QVBoxLayout(panel)
        # Загальне споживання
```

```
            total_frame = QFrame()
            total_frame.setStyleSheet("background-color: #f0f0f0; border-radius: 8px;
padding: 10px;")
            total_layout = QVBoxLayout(total_frame)
            self.total_power_label = QLabel("0.00 кВт")
            self.total_power_label.setFont(QFont("Arial", 24, QFont.Weight.Bold))
            self.total_power_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
            total_layout.addWidget(self.total_power_label)
            self.daily_kwh_label = QLabel("Сьогодні: 0.00 кВт·год")
            self.daily_kwh_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
            total_layout.addWidget(self.daily_kwh_label)
            layout.addWidget(total_frame)
            # Споживання по кімнатах
            rooms_group = QGroupBox("По кімнатах")
            rooms_layout = QGridLayout(rooms_group)
            self.room_labels: Dict[str, QLabel] = {}
            rooms = [
                ("living_room", "Вітальня"),
                ("kitchen", "Кухня"),
                ("master_bedroom", "Спальня"),
                ("child_room", "Дитяча"),
                ("office", "Кабінет"),
                ("bathroom", "Ванна"),
                ("hallway", "Коридор"),
                ("basement", "Підвал")
            ]
            for i, (room_id, room_name) in enumerate(rooms):
                row = i // 2
                col = (i % 2) * 2
                name_label = QLabel(f"{room_name}:")
                rooms_layout.addWidget(name_label, row, col)
                value_label = QLabel("0 Вт")
                value_label.setStyleSheet("font-weight: bold;")
                rooms_layout.addWidget(value_label, row, col + 1)
                self.room_labels[room_id] = value_label
            layout.addWidget(rooms_group)
            # Активні пристрої
            devices_group = QGroupBox("Активні пристрої")
            devices_layout = QVBoxLayout(devices_group)
            self.active_devices_label = QLabel("Немає активних пристроїв")
            self.active_devices_label.setWordWrap(True)
            devices_layout.addWidget(self.active_devices_label)
            layout.addWidget(devices_group)
            layout.addStretch()
            return panel
        def _create_forecast_panel(self) -> QWidget:
            """Панель прогнозу та накопичення."""
            panel = QGroupBox("Накопичення та прогноз")
            layout = QVBoxLayout(panel)
            # Tabs: Накопичення | Прогноз | Звіти
            self.tabs = QTabWidget()
            # Вкладка накопичення
            accum_tab = QWidget()
            accum_layout = QVBoxLayout(accum_tab)
            self.cumulative_label = QLabel("Загалом: 0.00 кВт·год")
            self.cumulative_label.setFont(QFont("Arial", 16, QFont.Weight.Bold))
            accum_layout.addWidget(self.cumulative_label)
            self.cost_label = QLabel("Вартість: 0.00 грн")
            self.cost_label.setFont(QFont("Arial", 14))
            accum_layout.addWidget(self.cost_label)
            accum_layout.addStretch()
            self.tabs.addTab(accum_tab, "Накопичення")
            # Вкладка прогнозу — використовуємо ForecastView
            try:
                from .forecast_view import ForecastView
```

```
                self.forecast_view = ForecastView()
                self.tabs.addTab(self.forecast_view, "📈 Прогноз")
            except ImportError:
                forecast_tab = QWidget()
                forecast_layout = QVBoxLayout(forecast_tab)
                self.forecast_info = QLabel("Запустіть симуляцію для отримання
прогнозу")
                self.forecast_info.setWordWrap(True)
                forecast_layout.addWidget(self.forecast_info)
                self.run_forecast_btn = QPushButton("Побудувати прогноз")
                self.run_forecast_btn.setEnabled(False)
                forecast_layout.addWidget(self.run_forecast_btn)
                forecast_layout.addStretch()
                self.tabs.addTab(forecast_tab, "Прогноз")
                self.forecast_view = None
            # Вкладка звітів — використовуємо ReportsView
            try:
                from .reports_view import ReportsView
                self.reports_view = ReportsView()
                self.tabs.addTab(self.reports_view, "📄 Звіти")
            except ImportError:
                reports_tab = QWidget()
                reports_layout = QVBoxLayout(reports_tab)
                self.reports_info = QLabel("Звіти будуть доступні після завершення
симуляції")
                reports_layout.addWidget(self.reports_info)
                reports_layout.addStretch()
                self.tabs.addTab(reports_tab, "Звіти")
                self.reports_view = None
            # Вкладка історії — HistoryView
            try:
                from .history_view import HistoryView
                self.history_view = HistoryView()
                self.tabs.addTab(self.history_view, "📈 Історія")
            except ImportError:
                self.history_view = None
            # Вкладка довідки — HelpView
            try:
                from .help_view import HelpView
                self.help_view = HelpView()
                self.tabs.addTab(self.help_view, "❓ Довідка")
            except ImportError:
                help_tab = QWidget()
                help_layout = QVBoxLayout(help_tab)
                help_layout.addWidget(QLabel("Довідка недоступна"))
                help_layout.addStretch()
                self.tabs.addTab(help_tab, "Довідка")
                self.help_view = None
            layout.addWidget(self.tabs)
            return panel
        def _setup_menu(self):
            """Налаштування меню."""
            menubar = self.menuBar()
            # Файл
            file_menu = menubar.addMenu("Файл")
            export_action = QAction("Експорт даних...", self)
            file_menu.addAction(export_action)
            file_menu.addSeparator()
            exit_action = QAction("Вихід", self)
            exit_action.triggered.connect(self.close)
            file_menu.addAction(exit_action)
            # Налаштування
            settings_menu = menubar.addMenu("Налаштування")
            tariff_action = QAction("Тарифи...", self)
            tariff_action.triggered.connect(self._show_tariff_dialog)
```

```
        settings_menu.addAction(tariff_action)
        # Допомога
        help_menu = menubar.addMenu("Допомога")
        about_action = QAction("Про програму", self)
        about_action.triggered.connect(self._show_about)
        help_menu.addAction(about_action)
    def _setup_statusbar(self):
        """Налаштування статус-бару."""
        self.statusbar = QStatusBar()
        self.setStatusBar(self.statusbar)
        self.statusbar.showMessage("Готово до симуляції")
    def _connect_signals(self):
        """Підключення сигналів."""
        self.start_btn.clicked.connect(self._start_simulation)
        self.pause_btn.clicked.connect(self._pause_simulation)
        self.reset_btn.clicked.connect(self._reset_simulation)
        # Підключення кліку на кімнату
        self.floor_plan.room_clicked.connect(self._on_room_clicked)
        # Застарілий fallback для run_forecast_btn
        if hasattr(self, 'run_forecast_btn'):
            self.run_forecast_btn.clicked.connect(self._run_forecast)
    def _start_simulation(self):
        """Запуск симуляції."""
        try:
            from simulation import SimulationEngine, SimulationConfig
            from data import DataAggregator
        except ImportError:
            from ..simulation import SimulationEngine, SimulationConfig
            from ..data import DataAggregator
        # Створення конфігурації
        config = SimulationConfig(
            start_date=datetime(2024, 1, 1),
            duration_days=self.days_spin.value()
        )
        # Ініціалізація
        self.engine = SimulationEngine(config)
        self.aggregator = DataAggregator()
        # Запуск у фоновому потоці
        self.worker = SimulationWorker(self.engine)
        self.worker.progress.connect(self._update_progress)
        self.worker.step_data.connect(self._update_display)
        self.worker.finished.connect(self._simulation_finished)
        self.worker.error.connect(self._simulation_error)
        self.worker.start()
        # Оновлення UI
        self.is_running = True
        self.start_btn.setEnabled(False)
        self.pause_btn.setEnabled(True)
        self.reset_btn.setEnabled(False)
        self.days_spin.setEnabled(False)
        self.statusbar.showMessage("Симуляція запущена...")
    def _pause_simulation(self):
        """Пауза/відновлення симуляції."""
        if self.worker:
            if self.worker.is_paused:
                # Відновлення
                self.worker.resume()
                self.pause_btn.setText("⏸ Пауза")
                self.start_btn.setEnabled(False)
                self.reset_btn.setEnabled(False)
                self.statusbar.showMessage("Симуляція відновлена...")
            else:
                # Пауза
                self.worker.pause()
                self.pause_btn.setText("▶ Продовжити")
```

```
                    self.start_btn.setEnabled(False)
                    self.reset_btn.setEnabled(True)
                    self.statusbar.showMessage("Симуляція призупинена")
        def _reset_simulation(self):
            """Скидання симуляції."""
            if self.engine:
                self.engine.reset()
            self.progress_bar.setValue(0)
            self.is_running = False
            self.start_btn.setEnabled(True)
            self.pause_btn.setEnabled(False)
            self.reset_btn.setEnabled(False)
            self.days_spin.setEnabled(True)
            self._clear_display()
            self.statusbar.showMessage("Симуляцію скинуто")
        def _update_progress(self, current: int, total: int):
            """Оновлення прогрес-бару."""
            percent = int(current / total * 100)
            self.progress_bar.setValue(percent)
        def _update_display(self, state):
            """Оновлення відображення."""
            # Час симуляції
            self.sim_time_label.setText(f"Час: {state.timestamp.strftime('%H:%M')}")
            self.sim_date_label.setText(f"Дата:
{state.timestamp.strftime('%Y-%m-%d')}")
            self.temp_label.setText(f"Температура:
{state.outdoor_temperature_c:.1f}°C")
            # Загальне споживання
            self.total_power_label.setText(f"{state.total_power_w / 1000:.2f} кВт")
            self.daily_kwh_label.setText(f"Сьогодні: {state.daily_kwh:.2f} кВт·год")
            self.cumulative_label.setText(f"Загалом: {state.cumulative_kwh:.2f}
кВт·год")
            # Вартість (приблизно)
            cost = state.cumulative_kwh * 2.0
            self.cost_label.setText(f"Вартість: {cost:.2f} грн")
            # По кімнатах
            for room_id, room_state in state.room_states.items():
                if room_id in self.room_labels:
                    self.room_labels[room_id].setText(f"{room_state.total_power_w:.0f}
Вт")
            # Оновлення плану будинку
            self.floor_plan.update_room_consumption(
                {rid: rs.total_power_w for rid, rs in state.room_states.items()}
            )
        def _simulation_finished(self):
            """Симуляцію завершено."""
            self.is_running = False
            self.start_btn.setEnabled(False)
            self.pause_btn.setEnabled(False)
            self.reset_btn.setEnabled(True)
            self.progress_bar.setValue(100)
            if hasattr(self, 'run_forecast_btn') and self.run_forecast_btn:
                self.run_forecast_btn.setEnabled(True)
            self.statusbar.showMessage("Симуляцію завершено! Можна побудувати
прогноз.")
            # Збір даних для прогнозування та звітів
            daily_data = []
            room_consumption = {}
            device_data = []
            try:
                if self.engine:
                    # Споживання по кімнатах
                    for room_id, room in self.engine.rooms.items():
                        room_consumption[room_id] = room.cumulative_kwh
                        # Дані пристроїв
```

```
                    for dev_id, device in room.devices.items():
                        # Отримуємо накопичене споживання
                        kwh = getattr(device, '_cumulative_kwh', 0)
                        if kwh == 0:
                            kwh = getattr(device, 'cumulative_kwh', 0)
                        if kwh == 0:
                            # Оцінка на основі типової потужності
                            power_w = getattr(device, 'power_working_w', 0) or
getattr(device, 'power_on_w', 50)
                            # Приблизно 8 годин роботи на день
                            kwh = (power_w / 1000) * 8 *
self.engine.config.duration_days * 0.3
                        device_data.append({
                            "name": getattr(device, 'name', dev_id),
                            "room": room_id,
                            "type": getattr(device, 'device_type', 'unknown'),
                            "kwh": round(kwh, 2)
                        })
                # Добові дані (якщо є aggregator)
                if self.aggregator and hasattr(self.aggregator, 'daily_totals'):
                    for date, kwh in self.aggregator.daily_totals.items():
                        daily_data.append({
                            "date": date,
                            "total_kwh": kwh
                        })
                else:
                    # Використовуємо реальні дані UCI замість синтетичних
                    try:
                        from ..data.uci_dataset import prepare_daily_data
                        uci_df = prepare_daily_data()
                        # Беремо останні N днів відповідно до тривалості симуляції
                        n_days = min(self.engine.config.duration_days,
len(uci_df))
                        uci_sample = uci_df.tail(n_days).reset_index()
                        for _, row in uci_sample.iterrows():
                            daily_data.append({
                                "date": row['datetime'] if 'datetime' in row else
row.name,
                                "total_kwh": row['total_kwh']
                            })
                    except Exception:
                        # Fallback — реалістичні дані з варіацією
                        import pandas as pd
                        import numpy as np
                        avg_daily = self.engine.total_cumulative_kwh /
self.engine.config.duration_days
                        for i in range(self.engine.config.duration_days):
                            # Реалістична варіація: ±30% випадково + weekend
factor
                            is_weekend = (i % 7) >= 5
                            base = avg_daily * (1.15 if is_weekend else 1.0)
                            noise = np.random.normal(0, avg_daily * 0.2)
                            daily_data.append({
                                "date": self.engine.config.start_date +
pd.Timedelta(days=i),
                                "total_kwh": max(5, base + noise)
                            })
                # Збереження даних у CSV
                self._save_simulation_data(daily_data, room_consumption, device_data)
                # Передача даних у ForecastView
                if hasattr(self, 'forecast_view') and self.forecast_view:
                    self.forecast_view.set_daily_data(daily_data)
                # Передача даних у HistoryView
                if hasattr(self, 'history_view') and self.history_view:
                    self.history_view.set_data(daily_data)
```

```
                # Передача даних у ReportsView
                if hasattr(self, 'reports_view') and self.reports_view:
                    self.reports_view.update_data(
                        room_consumption=room_consumption,
                        device_consumption=device_data,
                        total_days=self.engine.config.duration_days if self.engine
else 0
                    )
                    # Генерація рекомендацій
                    try:
                        try:
                            from financial import RecommendationEngine
                        except ImportError:
                            from ..financial import RecommendationEngine
                        recommender = RecommendationEngine()
                        recommendations = recommender.analyze_and_recommend(
                            room_consumption=room_consumption,
                            device_consumption={d['name']: d for d in device_data},
                            total_monthly_kwh=self.engine.total_cumulative_kwh /
(self.engine.config.duration_days / 30)
                        )
                        self.reports_view.set_recommendations([
                            {
                                "priority": r.priority,
                                "category": r.category,
                                "title": r.title,
                                "description": r.description,
                                "savings_uah": r.potential_savings_uah_month
                            }
                            for r in recommendations
                        ])
                    except Exception:
                        pass
                QMessageBox.information(
                    self,
                    "Симуляцію завершено",
                    f"Симуляція {self.engine.config.duration_days if self.engine else
0} днів завершена.\n"
                    f"Загальне споживання: {self.engine.total_cumulative_kwh if
self.engine else 0:.2f} кВт·год\n\n"
                    f"Перейдіть на вкладки «Прогноз» або «Звіти» для аналізу."
                )
            except Exception as e:
                import traceback
                error_details = traceback.format_exc()
                QMessageBox.critical(
                    self,
                    "Помилка обробки результатів",
                    f"Симуляцію завершено, але виникла
помилка:\n{e}\n\nДеталі:\n{error_details}"
                )
        def _save_simulation_data(self, daily_data, room_consumption, device_data):
            """Збереження даних симуляції у файли."""
            import pandas as pd
            from pathlib import Path
            from datetime import datetime
            # Створюємо директорію для даних
            data_dir = Path("simulation_data")
            data_dir.mkdir(exist_ok=True)
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            # Збереження добових даних
            if daily_data:
                df_daily = pd.DataFrame(daily_data)
                daily_path = data_dir / f"daily_{timestamp}.csv"
                df_daily.to_csv(daily_path, index=False, encoding='utf-8')
```

```
                # Також зберігаємо як "останній" для швидкого доступу
                df_daily.to_csv(data_dir / "daily_latest.csv", index=False,
encoding='utf-8')
            # Збереження по кімнатах
            if room_consumption:
                df_rooms = pd.DataFrame([
                    {"room_id": k, "kwh": v} for k, v in room_consumption.items()
                ])
                df_rooms.to_csv(data_dir / "rooms_latest.csv", index=False,
encoding='utf-8')
            # Збереження по пристроях
            if device_data:
                df_devices = pd.DataFrame(device_data)
                df_devices.to_csv(data_dir / "devices_latest.csv", index=False,
encoding='utf-8')
            self.statusbar.showMessage(f"Дані збережено у {data_dir}/")
        def _simulation_error(self, error_msg: str):
            """Помилка симуляції."""
            self.is_running = False
            self.start_btn.setEnabled(True)
            self.pause_btn.setEnabled(False)
            self.reset_btn.setEnabled(True)
            QMessageBox.critical(self, "Помилка симуляції", error_msg)
        def _on_room_clicked(self, room_id: str):
            """Обробка кліку на кімнату."""
            try:
                from .room_detail import RoomDetailWidget
                # Якщо симуляція запущена, показуємо дані кімнати
                if self.engine and room_id in self.engine.rooms:
                    room = self.engine.rooms[room_id]
                    # Отримання списку пристроїв
                    devices = []
                    for dev_id, device in room.devices.items():
                        devices.append({
                            "name": device.model_name,
                            "type": device.device_type,
                            "status": "on" if device._current_power_w > 0 else "off",
                            "power_w": device._current_power_w,
                            "daily_kwh": device._daily_kwh
                        })
                    # Показуємо діалог
                    detail_widget = RoomDetailWidget(room_id)
                    detail_widget.update_data(
                        current_power_w=room.current_power_w,
                        daily_kwh=room.daily_kwh,
                        total_kwh=room.cumulative_kwh,
                        devices=devices
                    )
                    detail_widget.setWindowTitle(f"Деталі: {room.room_name}")
                    detail_widget.resize(600, 500)
                    detail_widget.show()
                    # Зберігаємо посилання, щоб вікно не закрилося
                    if not hasattr(self, '_detail_windows'):
                        self._detail_windows = []
                    self._detail_windows.append(detail_widget)
            except Exception as e:
                self.statusbar.showMessage(f"Помилка: {str(e)}")
        def _run_forecast(self):
            """Запуск прогнозування (legacy fallback)."""
            if hasattr(self, 'forecast_view') and self.forecast_view:
                # Перехід на вкладку прогнозу
                self.tabs.setCurrentWidget(self.forecast_view)
            elif hasattr(self, 'forecast_info'):
                self.forecast_info.setText("Побудова прогнозу... Це може зайняти
кілька хвилин.")
```

```
                if hasattr(self, 'run_forecast_btn'):
                    self.run_forecast_btn.setEnabled(False)
                QMessageBox.information(
                    self,
                    "Прогнозування",
                    "Функція прогнозування потребує встановлення додаткових
пакетів:\n"
                    "- statsmodels\n- prophet\n- tensorflow\n- xgboost\n\n"
                    "Встановіть їх командою:\npip install statsmodels prophet
tensorflow xgboost"
                )
                if hasattr(self, 'run_forecast_btn'):
                    self.run_forecast_btn.setEnabled(True)
        def _clear_display(self):
            """Очищення відображення."""
            self.sim_time_label.setText("Час: --:--")
            self.sim_date_label.setText("Дата: ----.--.--")
            self.temp_label.setText("Температура: --°C")
            self.total_power_label.setText("0.00 кВт")
            self.daily_kwh_label.setText("Сьогодні: 0.00 кВт·год")
            self.cumulative_label.setText("Загалом: 0.00 кВт·год")
            self.cost_label.setText("Вартість: 0.00 грн")
            for label in self.room_labels.values():
                label.setText("0 Вт")
        def _show_tariff_dialog(self):
            """Показати діалог налаштування тарифів."""
            try:
                from .tariff_dialog import TariffDialog
                dialog = TariffDialog(self)
                dialog.exec()
            except Exception as e:
                QMessageBox.critical(self, "Помилка", f"Не вдалося відкрити діалог
тарифів: {e}")
        def _show_about(self):
            """Показати інформацію про програму."""
            QMessageBox.about(
                self,
                "Про програму",
                "Прогнозування енергоспоживання розумного будинку\n\n"
                "Версія: 1.0.0\n\n"
                "Симулятор IoT-системи з прогнозуванням\n"
                "на основі машинного навчання.\n\n"
                "Моделі: SARIMA, Prophet, Transformer, XGBoost\n"
                "Датасет: UCI Individual Household Electric Power\n\n"
                "© 2026"
            )
        def closeEvent(self, event):
            """Обробка закриття вікна."""
            if self.is_running:
                reply = QMessageBox.question(
                    self,
                    "Підтвердження",
                    "Симуляція ще виконується. Ви впевнені, що хочете вийти?",
                    QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No
                )
                if reply == QMessageBox.StandardButton.No:
                    event.ignore()
                    return
                if self.worker:
                    self.worker.stop()
                    self.worker.wait()
            event.accept()
def run_gui():
    """Запуск GUI додатку."""
    if not PYQT_AVAILABLE:
```

```
        print("PyQt6 не встановлено. Встановіть: pip install PyQt6")
        return
    app = QApplication(sys.argv)
    app.setStyle("Fusion")
    window = MainWindow()
    window.show()
    sys.exit(app.exec())
```

Лістинг Г.14

test_forecasting.py

```
"""
Тести моделей прогнозування.
"""
import unittest
import numpy as np
import pandas as pd
from datetime import datetime, timedelta
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))
class TestForecastingModels(unittest.TestCase):
    """Тести моделей прогнозування."""
    @classmethod
    def setUpClass(cls):
        """Створення тестових даних."""
        # Генерація синтетичних даних
        np.random.seed(42)
        n_days = 100
        dates = pd.date_range(start="2024-01-01", periods=n_days, freq="D")
        # Базове споживання з сезонністю
        base = 15 + 5 * np.sin(np.arange(n_days) * 2 * np.pi / 7)  # Тижнева
сезонність
        noise = np.random.normal(0, 1, n_days)
        consumption = base + noise
        cls.test_data = pd.DataFrame({
            "date": dates,
            "total_kwh": consumption,
            "avg_temperature_c": 10 + 5 * np.sin(np.arange(n_days) * 2 * np.pi / 365),
            "is_weekend": [d.weekday() >= 5 for d in dates],
            "is_holiday": [False] * n_days
        })
        cls.train_data = cls.test_data.iloc[:-20]
        cls.val_data = cls.test_data.iloc[-20:]
    def test_sarima_model(self):
        """Тест SARIMA моделі."""
        try:
            from forecasting import SARIMAModel
            model = SARIMAModel(auto_select=False, order=(1, 1, 1), seasonal_order=(1,
0, 1, 7))
            model.fit(self.train_data, "total_kwh")
            self.assertTrue(model.is_fitted)
            forecast = model.predict(20)
            self.assertEqual(len(forecast.predictions), 20)
            self.assertTrue(np.all(forecast.predictions > 0))
        except ImportError:
            self.skipTest("statsmodels not installed")
    def test_prophet_model(self):
        """Тест Prophet моделі."""
        try:
            from forecasting import ProphetModel
            model = ProphetModel(yearly_seasonality=False)
            model.fit(self.train_data, "total_kwh", "date")
            self.assertTrue(model.is_fitted)
            forecast = model.predict(20)
```

```
            self.assertEqual(len(forecast.predictions), 20)
        except ImportError:
            self.skipTest("prophet not installed")
    def test_lstm_model(self):
        """Тест LSTM моделі."""
        try:
            from forecasting import LSTMModel
            model = LSTMModel(lookback=14, lstm_units=[32], epochs=10, verbose=0)
            model.fit(self.train_data, "total_kwh")
            self.assertTrue(model.is_fitted)
            forecast = model.predict(20)
            self.assertEqual(len(forecast.predictions), 20)
        except ImportError:
            self.skipTest("tensorflow not installed")
    def test_xgboost_model(self):
        """Тест XGBoost моделі."""
        try:
            from forecasting import XGBoostModel
            model = XGBoostModel(n_estimators=50)
            model.fit(self.train_data, "total_kwh")
            self.assertTrue(model.is_fitted)
            forecast = model.predict(20)
            self.assertEqual(len(forecast.predictions), 20)
        except ImportError:
            self.skipTest("xgboost not installed")
    def test_transformer_model(self):
        """Тест Transformer моделі."""
        try:
            from forecasting import TransformerModel
            model = TransformerModel(
                lookback=14,
                d_model=32,
                n_heads=2,
                n_layers=1,
                epochs=20,
                verbose=False
            )
            model.fit(self.train_data, "total_kwh")
            self.assertTrue(model.is_fitted)
            forecast = model.predict(20)
            self.assertEqual(len(forecast.predictions), 20)
            self.assertTrue(np.all(forecast.predictions > 0))
            # Перевірка архітектури
            arch = model.get_architecture()
            self.assertEqual(arch["d_model"], 32)
            self.assertEqual(arch["n_heads"], 2)
            self.assertGreater(arch["total_params"], 0)
        except ImportError:
            self.skipTest("torch not installed")
    def test_ensemble_model(self):
        """Тест Ensemble моделі."""
        try:
            import xgboost
        except ImportError:
            self.skipTest("xgboost not installed")
        from forecasting import EnsembleModel
        model = EnsembleModel(
            use_sarima=False,  # Вимкнути для швидкості
            use_prophet=False,
            use_transformer=False,
            use_xgboost=True
        )
        model.fit(self.train_data, self.val_data, "total_kwh", "date")
        self.assertTrue(model.is_fitted)
        forecast = model.predict(20)
```

```
        self.assertEqual(len(forecast.predictions), 20)
class TestMetrics(unittest.TestCase):
    """Тести метрик якості."""
    def test_metrics_calculation(self):
        """Розрахунок метрик."""
        from forecasting.sarima_model import ModelMetrics
        y_true = np.array([10, 15, 12, 18, 20])
        y_pred = np.array([11, 14, 13, 17, 19])
        # MAE
        mae = np.mean(np.abs(y_true - y_pred))
        self.assertAlmostEqual(mae, 1.0)
        # RMSE
        rmse = np.sqrt(np.mean((y_true - y_pred) ** 2))
        self.assertAlmostEqual(rmse, 1.0)
    def test_mape_calculation(self):
        """MAPE розрахунок."""
        y_true = np.array([100, 200, 150])
        y_pred = np.array([110, 180, 160])
        # MAPE
        mape = np.mean(np.abs((y_true - y_pred) / y_true)) * 100
        self.assertLess(mape, 15)
class TestForecastResult(unittest.TestCase):
    """Тести результату прогнозу."""
    def test_forecast_result_structure(self):
        """Структура ForecastResult."""
        from forecasting.sarima_model import ForecastResult
        dates = pd.date_range("2024-01-01", periods=30)
        predictions = np.random.rand(30) * 20
        lower = predictions - 2
        upper = predictions + 2
        result = ForecastResult(
            dates=dates.values,
            predictions=predictions,
            lower_bound=lower,
            upper_bound=upper,
            confidence_level=0.90
        )
        self.assertEqual(len(result.dates), 30)
        self.assertEqual(len(result.predictions), 30)
        self.assertTrue(np.all(result.upper_bound >= result.predictions))
        self.assertTrue(np.all(result.lower_bound <= result.predictions))
if __name__ == "__main__":
    unittest.main()
```