

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного
аналізу Кафедра штучного інтелекту**

До захисту допущено:
В. о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«__» _____ 20__ р.

Дипломна робота

на здобуття ступеня бакалавра

**за освітньо-професійною програмою «Системи і методи штучного
інтелекту» спеціальності 122 «Комп'ютерні науки»**

**на тему: «Система IoT для автоматизованого управління розумним
будинком»**

Виконав:

Студент IV курсу, групи КІ-13
Богун Максим Олександрович

Керівник:

доцент кафедри ШІ, к.т.н., доцент
Коваленко Анатолій Єпіфанович

Консультант з економічного розділу:

доцент кафедри економічної кібернетики, к.е.н., доцент,
Рощина Надія Василівна

Консультант з нормоконтролю:

фахівець першої категорії кафедри ШІ, к.т.н., доцент,
Комариста Богдана Миколаївна

Рецензент:

доцент кафедри ММСА, к.т.н., доцент,
Зінченко Артем Юрійович

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент _____

Київ – 2025 року

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

**Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту**

Рівень вищої освіти – перший(бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Системи та методи штучного інтелекту»

ЗАТВЕРДЖУЮ

В. о. завідувачки кафедри

_____ Ірина ДЖИГИРЕЙ

«31» січня 2024 р.

**ЗАВДАННЯ
на дипломну роботу студенту
Богуну Максиму Олександровичу**

1. Тема роботи «Система IoT для автоматизованого управління розумним будинком», керівник роботи Коваленко Анатолій Єпіфанович, доцент кафедри ШІ, затверджені наказом по НН ПСА від «26» травня 2025 р. № 1759-с.
2. Термін подання студентом роботи «09» червня 2025 року.
3. Вихідні дані до роботи: набір фінансових транзакцій, дані IoT-пристроїв (лампи, термостати, замки, датчики) з SmartThings API: ідентифікатори, типи, стани, команди, дані локацій: ідентифікатори, назви, списки пристроїв, дані користувачів (MySQL, таблиця users), завдання та календарні події (таблиці tasks, calendars), історія команд (таблиця command_history), конфігурація SmartThings API.
4. Зміст роботи: аналіз IoT-систем розумного будинку та вихідних даних, вибір гібридної архітектури та технологій (React, NestJS, MySQL, TypeORM, SmartThings API), розробка алгоритмів авторизації, управління пристроями, виконання завдань, синхронізації з SmartThings, оцінка продуктивності, безпеки, зручності та економічної ефективності системи.
5. Перелік ілюстративного матеріалу: електронна презентація.
6. Консультанти розділів роботи.

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Роцина Надія Василівна, доцент, к. е. н.		

7. Дата видачі завдання: «03» лютого 2025 року.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Вибір теми і постановка дослідження.	15.04.2025 22.04.2025	Виконано
2	Аналіз актуальності дослідження.	22.04.2025 29.04.2025	Виконано
3	Збір та аналіз літератури.	29.04.2025 10.05.2025	Виконано
4	Формування задачі дослідження.	10.05.2025 15.05.2025	Виконано
5	Розробка програмної частини.	15.05.2025 25.05.2025	Виконано
6	Аналіз та структурізація результатів.	25.05.2025 30.05.2025	Виконано
7	Робота над економічною частиною продукту.	30.05.2025 10.06.2025	Виконано
8	Оформлення пояснювальної записки.	30.05.2025 10.06.2025	Виконано
9	Оформлення презентації для демонстрації.	30.05.2025 08.06.2025	Виконано

Студент

Максим БОГУН

Керівник

Анатолій КОВАЛЕНКО

РЕФЕРАТ

Дипломна робота: 75 с., 11 рис., 6 табл., 13 посилань.

ІНТЕРНЕТ РЕЧЕЙ, РОЗУМНИЙ БУДИНОК, АВТОМАТИЗАЦІЯ, РЕКОМЕНДАЦІЙНА СИСТЕМА, SMARTTHINGS, REACT, NESTJS, MYSQL, TYPEORM, ГІБРИДНА АРХІТЕКТУРА.

Досліджено розробку системи Інтернету речей (IoT) для автоматизованого управління розумним будинком із інтеграцією рекомендаційної системи. Проаналізовано сучасні IoT-технології, зокрема протоколи зв'язку (Zigbee, Z-Wave, Wi-Fi) та платформу SmartThings, яка забезпечує інтеграцію різноманітних пристроїв. Запропоновано гібридну архітектуру, що поєднує локальну обробку даних із періодичною синхронізацією через SmartThings API, досягаючи швидкості відгуку до 100 мс і підтримки до 50 пристроїв на локацію. Розроблено клієнтську частину на React із адаптивним інтерфейсом, серверну частину на NestJS із REST API, базу даних MySQL із TypeORM для управління користувачами, локаціями, пристроями, завданнями, календарями та історією команд, а також мікросервіс рекомендацій на Python (FastAPI) з алгоритмом кластеризації k-means.

Реалізовано функції авторизації (JWT, bcrypt), створення автоматизованих сценаріїв (завдання, календарні події), моніторингу стану пристроїв через інформаційну панель і генерації рекомендацій автоматизованих задач. Рекомендаційна система, реалізована на Python, аналізує історію команд із таблиці `command_history` через REST API, використовуючи k-means для виявлення патернів (наприклад, регулярне вмикання світла о 18:00), і пропонує відповідні сценарії, підвищуючи зручність та енергоефективність. Мікросервіс на FastAPI інтегровано з NestJS через HTTP-запити, із захистом JWT і кешуванням у Redis, що зменшило кількість запитів до SmartThings API на 35% і прискорило обробку рекомендацій на 40%. Безпека даних забезпечена шифруванням (HTTPS) і хешуванням паролів (bcrypt).

Отримано функціональну систему, яка дозволяє користувачам ефективно керувати розумним будинком, створювати персоналізовані сценарії, відстежувати історію команд і отримувати адаптивні рекомендації. Гібридна архітектура забезпечує баланс між автономністю, продуктивністю та інтеграцією з хмарними сервісами. Система підтвердила практичну цінність для підвищення комфорту, безпеки й енергоефективності, зокрема завдяки рекомендаціям, що зменшують потребу в ручному налаштуванні.

ABSTRACT

Bachelor's thesis: 75 p., 11 figures, 6 tables, 13 references.

INTERNET OF THINGS, SMART HOME, AUTOMATION, RECOMMENDATION SYSTEM, SMARTTHINGS, REACT, NESTJS, MYSQL, TYPEORM, HYBRID ARCHITECTURE.

The development of an Internet of Things (IoT) system for automated smart home management with an integrated recommendation system was investigated. Modern IoT technologies, including communication protocols (Zigbee, Z-Wave, Wi-Fi) and the SmartThings platform for device integration, were analyzed. A hybrid architecture was proposed, combining local data processing with periodic synchronization via the SmartThings API, achieving a response time of up to 100 ms and support for up to 50 devices per location. The client-side was developed using React with an adaptive interface, the server-side on NestJS with a REST API, a MySQL database with TypeORM for managing users, locations, devices, tasks, calendars, and command history, and a recommendation microservice on Python (FastAPI) utilizing k-means clustering.

Features implemented include authentication (JWT, bcrypt), creation of automated scenarios (tasks, calendar events), device status monitoring via a dashboard, and generation of automated task recommendations. The recommendation system, built in Python, analyzes command history from the `command_history` table via REST API, using k-means to identify patterns (e.g., regular light activation at 18:00) and suggest scenarios, enhancing usability and energy efficiency. The FastAPI microservice integrates with NestJS through HTTP requests, secured with JWT and optimized with Redis caching, reducing SmartThings API requests by 35% and speeding up recommendation processing by 40%. Data security is ensured through encryption (HTTPS) and password hashing (bcrypt).

A functional system was obtained, enabling users to efficiently manage a smart home, create personalized scenarios, track command history, and receive adaptive recommendations. The hybrid architecture balances autonomy, performance, and cloud service integration. The system demonstrated practical value in enhancing comfort, security, and energy efficiency, particularly through recommendations that reduce manual configuration.

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1 ОГЛЯД СИСТЕМ РОЗУМНОГО БУДИНКУ	10
1.1 Загальна характеристика систем розумного будинку.....	10
1.2 Технології та платформи для IoT-систем розумного будинку.....	11
1.3 Функціональні можливості систем автоматизації розумного будинку	13
1.4 Постановка задачі.....	14
Висновки до розділу 1.....	16
РОЗДІЛ 2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ IoT ДЛЯ РОЗУМНОГО БУДИНКУ	18
2.1 Визначення вимог до системи.....	18
2.1.1 Функціональні вимоги	18
2.1.2 Нефункціональні вимоги	20
2.1.3 Технічні вимоги.....	21
2.2 Аналіз архітектурних підходів для IoT-систем	23
2.2.1 Локальна архітектура.....	23
2.2.2 Гібридна архітектура.....	25
2.2.3 Порівняльний аналіз архітектур	27
2.2.4 Висновки з аналізу	28
2.3 Розробка архітектури системи	28
2.3.1 Опис компонентів архітектури	29
2.3.2 Взаємодія компонентів	31
2.3.3 C4-схема системи	31
Висновки до розділу 2.....	34
РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ IoT ДЛЯ АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РОЗУМНИМ БУДИНКОМ.....	36
3.1 Налаштування середовища розробки.....	36
3.2 Розробка клієнтської частини на React	39

	7
3.3 Розробка серверної складової на NestJS.....	43
Висновки до розділу 3.....	49
РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО	
ПРОДУКТУ.....	51
4.1 Постановка задачі проектування	52
4.2 Обґрунтування функцій програмного продукту	54
4.3 Обґрунтування системи параметрів програмного продукту.....	58
4.4 Аналіз експертного оцінювання параметрів.....	61
4.5 Аналіз рівня якості варіантів реалізації функцій	65
4.6 Економічний аналіз варіантів розробки ПП	66
4.7 Вибір кращого варіанту ПП техніко-економічного рівня	71
Висновки до розділу 4.....	72
ВИСНОВКИ	73

ВСТУП

Розвиток технологій Інтернету речей (IoT) відіграє ключову роль у трансформації сучасного житлового простору, забезпечуючи автоматизацію, енергоефективність і підвищений рівень комфорту та безпеки. Системи розумного будинку, що є одним із найперспективніших напрямів IoT, дозволяють користувачам дистанційно керувати побутовими пристроями, оптимізувати їхню роботу та створювати персоналізовані сценарії для підвищення зручності. У рамках переддипломної практики розроблено систему IoT для автоматизованого управління розумним будинком, яка інтегрується з платформою SmartThings, надає інтуїтивно зрозумілий інтерфейс для налаштування, моніторингу та керування пристроями, а також включає рекомендаційну систему для адаптації до звичок користувачів.

Основною метою практики було створення функціональної системи, що забезпечує авторизацію користувачів для безпечного доступу, підключення локацій із відповідними пристроями, створення завдань і календарних подій для автоматизації команд, таких як увімкнення освітлення чи регулювання термостатів за розкладом або умовами, генерацію рекомендацій автоматизованих сценаріїв на основі аналізу історії команд за допомогою алгоритму k-means для підвищення зручності й енергоефективності, а також відображення детальної інформації про пристрої, їхній статус, кількість, типи та історію виконаних операцій на інформаційній панелі. Рекомендаційна система, реалізована як мікросервіс на Python (FastAPI), аналізує дані з таблиці `command_history` і пропонує сценарії, наприклад, автоматичне увімкнення світла о 18:00, базуючись на виявлених паттернах. Система підтримує гнучке налаштування календарних подій із чіткими діями початку та завершення, що забезпечує ефективне керування пристроями.

Розробка системи включала аналіз вимог, проектування гібридної архітектури, що поєднує локальну обробку даних із хмарною синхронізацією через SmartThings API, вибір технологій (React, NestJS, MySQL, TypeORM, FastAPI) і програмування функціоналу, відповідного сучасним стандартам IoT. Безпека забезпечена через шифрування (HTTPS) і авторизацію (JWT). Мікросервіс рекомендацій інтегровано з NestJS через REST API, що забезпечує швидку та безпечну взаємодію з використанням кешування в Redis для оптимізації продуктивності. У звіті описано етапи виконання роботи, технічні рішення, результати тестування та впровадження. Робота демонструє практичну цінність IoT-технологій для створення ефективних рішень автоматизації розумного будинку, підвищуючи комфорт, безпеку та енергоефективність завдяки адаптивним рекомендаціям.

РОЗДІЛ 1 ОГЛЯД СИСТЕМ РОЗУМНОГО БУДИНКУ

1.1 Загальна характеристика систем розумного будинку

Системи розумного будинку є одним із ключових напрямів розвитку технологій Інтернету речей (IoT), які дозволяють автоматизувати управління побутовими пристроями, підвищувати енергоефективність і комфорт користувачів. Розумний будинок об'єднує різноманітні пристрої, такі як освітлення, системи безпеки, клімат-контроль, дверні замки та побутову техніку, в єдину мережу, що керується централізовано через спеціалізовані платформи або додатки. Зростання популярності таких систем зумовлено швидким розвитком бездротових технологій, хмарних обчислень, доступністю IoT-пристроїв і появою інтелектуальних функцій, таких як рекомендаційні системи.

Основною метою систем розумного будинку є забезпечення зручного та ефективного управління пристроями, що дозволяє користувачам створювати персоналізовані сценарії роботи, такі як автоматичне вмикання світла при вході до приміщення чи регулювання температури залежно від часу доби. Сучасні системи також включають рекомендаційні можливості, які аналізують історію виконаних команд за допомогою алгоритмів машинного навчання, наприклад, k-means, для виявлення патернів у поведінці користувачів. Наприклад, система може запропонувати автоматизований сценарій увімкнення освітлення о 18:00, якщо виявить регулярність таких дій, що підвищує зручність і сприяє економії енергії. Крім того, такі системи сприяють економії ресурсів, підвищенню безпеки та інтеграції з зовнішніми сервісами, наприклад, платформами SmartThings, які забезпечують сумісність із широким спектром пристроїв.

Серед основних компонентів систем розумного будинку можна виокремити: датчики (руху, температури, освітлення), виконавчі пристрої (розумні розетки, замки, лампи), хаб або контролер для координації роботи

мережі, а також програмний інтерфейс для взаємодії з користувачем, який може включати відображення рекомендацій. Комунікація між пристроями зазвичай здійснюється через протоколи Zigbee, Z-Wave, Wi-Fi або Bluetooth, що забезпечують надійність і низьке енергоспоживання. Для реалізації рекомендаційних функцій часто застосовуються мікросервіси, наприклад, на Python із фреймворком FastAPI, які інтегруються з основною системою через REST API.

Проблематика впровадження систем розумного будинку пов'язана з питаннями безпеки даних, сумісності пристроїв від різних виробників, складністю налаштування для кінцевих користувачів, а також інтеграцією рекомендаційних систем. Захист від несанкціонованого доступу до IoT-мережі та мікросервісів є критичним, оскільки зловмисники можуть використовувати вразливості для отримання доступу до особистих даних чи керування пристроями. Крім того, рекомендаційні системи потребують обробки великих обсягів даних і забезпечення їхньої безпеки під час передачі між компонентами, що ускладнює розробку. Таким чином, створення систем розумного будинку вимагає комплексного підходу, що поєднує апаратні, програмні, безпекові та аналітичні рішення.

1.2 Технології та платформи для IoT-систем розумного будинку

Для створення систем розумного будинку застосовується широкий спектр технологій, які забезпечують комунікацію між пристроями, обробку даних і взаємодію з користувачем. Ключовим елементом є IoT-платформи, що виступають посередниками між апаратним забезпеченням і програмними інтерфейсами. Такі платформи, як SmartThings, TuYa чи Home Assistant, забезпечують інтеграцію пристроїв від різних виробників, надаючи єдиний інтерфейс для їхнього управління.

Платформа SmartThings є однією з провідних у цій галузі, підтримуючи широкий спектр пристроїв і протоколів, що забезпечує гнучкість у розширенні мережі розумного будинку. Вона дозволяє користувачам підключати локації, додавати пристрої та створювати автоматизовані сценарії через мобільний додаток або веб-інтерфейс. Основною перевагою SmartThings є хмарна архітектура, яка забезпечує віддалений доступ, обробку великих обсягів даних і можливість інтеграції з інтелектуальними системами, зокрема рекомендаційними, для адаптації до потреб користувачів.

Важливим аспектом є протоколи зв'язку. Протоколи Zigbee та Z-Wave використовуються для створення енергоефективних mesh-мереж, у яких пристрої передають сигнали один одному, розширюючи покриття мережі. Wi-Fi забезпечує високу швидкість передачі даних, що є важливим для інтеграції мікросервісів, таких як рекомендаційні системи, реалізовані на Python із фреймворком FastAPI, які взаємодіють із основною системою через REST API. Вибір протоколу залежить від типу пристроїв, вимог до енергоефективності та масштабованості системи.

Обробка даних у системах розумного будинку здійснюється за допомогою хмарних технологій, локальних серверів і спеціалізованих мікросервісів. Хмарні рішення забезпечують зберігання конфігурацій, історії команд і віддалений доступ, тоді як локальні сервери підвищують безпеку та незалежність від інтернет-з'єднання. Сучасні системи використовують алгоритми машинного навчання, зокрема кластеризацію k-means, реалізовану на Python із застосуванням бібліотек pandas і scikit-learn, для аналізу історії команд і генерації рекомендацій автоматизованих сценаріїв, наприклад, увімкнення освітлення у визначений час відповідно до звичок користувачів. Рекомендаційні мікросервіси інтегруються з основною системою через REST API, що забезпечує модульність, швидку обробку даних і масштабованість.

Безпека IoT-систем має критичне значення. Для захисту від кібератак застосовуються методи шифрування даних (AES, TLS), двофакторна автентифікація та регулярне оновлення програмного забезпечення. Для

забезпечення безпеки рекомендаційних мікросервісів використовується JWT-авторизація, яка захищає REST API від несанкціонованого доступу до даних, таких як історія команд. Проте зростання кількості підключених пристроїв і складність інтеграції мікросервісів підвищують ризики, що вимагає постійного вдосконалення захисних механізмів.

1.3 Функціональні можливості систем автоматизації розумного будинку

Функціонал систем розумного будинку спрямований на забезпечення зручності, безпеки та ефективності. Основні можливості охоплюють представлені нижче.

1. Авторизація та управління доступом: Користувачі можуть створювати облікові записи, авторизуватися та налаштовувати доступ до системи для різних членів родини чи адміністраторів.
2. Інтеграція пристроїв: Система дозволяє підключати локації та додавати пристрої, такі як розумні лампи, замки, термостати, із підтримкою їхньої конфігурації та моніторингу.
3. Автоматизація сценаріїв: Користувачі можуть створювати завдання, що виконуються у заданий час або за певних умов. Також підтримуються календарні події з фіксованими діями початку та завершення.
4. Моніторинг і аналітика: На інформаційній панелі відображається кількість підключених пристроїв, їхні типи та статус, а також історія виконаних команд для аналізу роботи системи.
5. Рекомендації автоматизованих сценаріїв: Система аналізує історію виконаних команд за допомогою алгоритму кластеризації k-means, реалізованого в мікросервісі на Python (FastAPI), і

пропонує автоматизовані сценарії, наприклад, увімкнення освітлення у визначений час на основі виявлених патернів поведінки користувачів, що підвищує зручність і енергоефективність.

Ці можливості реалізуються через поєднання апаратного та програмного забезпечення. Наприклад, для створення завдань користувач обирає пристрої, визначає команди та встановлює час виконання. Календарні події обмежують вибір команд до фіксованих дій, що спрощує налаштування для певних типів пристроїв, таких як дверні замки. Рекомендаційні функції забезпечуються мікросервісом на Python, який інтегрується з основною системою через REST API, обробляє дані з таблиці `command_history` і відображає пропозиції на інформаційній панелі, оптимізуючи взаємодію користувача з системою.

Сучасні системи також передбачають інтеграцію з голосовими асистентами (Amazon Alexa, Google Assistant), що розширює можливості керування. Однак складність реалізації полягає в забезпеченні сумісності між різними платформами, оптимізації затримок у виконанні команд, підтримці стабільної роботи великої кількості пристроїв, а також інтеграції рекомендаційних мікросервісів, що вимагає додаткових ресурсів для обробки даних і забезпечення безпеки.

1.4 Постановка задачі

У контексті стрімкого розвитку IoT-технологій виникає потреба в створенні гнучкої та зручної системи для автоматизованого управління розумним будинком. Основною проблемою є забезпечення інтеграції різноманітних пристроїв, створення інтуїтивного інтерфейсу для налаштування автоматизованих сценаріїв, гарантування безпеки даних

користувачів, а також адаптація до поведінки користувачів шляхом аналізу їхніх дій для пропонування оптимальних сценаріїв. Система повинна бути масштабовною, щоб підтримувати зростаючу кількість пристроїв і локацій, та забезпечувати мінімальні затримки у виконанні команд.

Задача полягає в розробці IoT-системи, яка забезпечує:

- авторизацію користувачів і безпечне підключення локацій через платформу SmartThings;
- додавання та конфігурацію пристроїв у межах локацій;
- створення завдань для автоматизованого виконання команд у заданий час;
- налаштування календарних подій із фіксованими діями початку та завершення;
- відображення інформації про пристрої (кількість, типи) та історію виконаних команд на інформаційній панелі;
- генерацію рекомендацій автоматизованих сценаріїв на основі аналізу історії команд за допомогою алгоритму кластеризації k-means, реалізованого в мікросервісі на Python (FastAPI), із інтеграцією через REST API.

Система має бути реалізована з урахуванням сучасних стандартів безпеки, сумісності з різними типами пристроїв і мінімальних затримок у виконанні команд. Для забезпечення рекомендаційних функцій передбачається використання мікросервісу на Python, який інтегрується з основною системою через REST API, із захистом даних за допомогою JWT-авторизації та оптимізацією продуктивності через кешування в Redis. У рамках дослідження планується аналіз вимог, проектування гібридної архітектури системи, розробка програмного забезпечення, включаючи клієнтську частину на React, серверну частину на NestJS, базу даних MySQL із TypeORM і мікросервіс рекомендацій, а також тестування функціоналу, результати якого будуть представлені в наступних розділах.

Висновки до розділу 1

Системи розумного будинку є важливим напрямом розвитку IoT-технологій, що забезпечують автоматизацію, комфорт, безпеку та енергоефективність. Вони дозволяють інтегрувати різноманітні пристрої в єдину мережу, створювати автоматизовані сценарії, моніторити їхню роботу та адаптуватися до поведінки користувачів через рекомендаційні системи. Основними викликами є забезпечення безпеки даних, сумісності пристроїв, зручності для користувачів і ефективної інтеграції інтелектуальних функцій.

Технології, такі як SmartThings, Zigbee, Z-Wave, хмарні обчислення та мікросервіси на Python (FastAPI), відіграють ключову роль у реалізації систем розумного будинку. Вони забезпечують зв'язок між пристроями, обробку даних, віддалений доступ і генерацію рекомендацій за допомогою алгоритмів машинного навчання, зокрема кластеризації k-means, реалізованих із використанням бібліотек pandas і scikit-learn. Безпека залишається критичним аспектом, що вимагає шифрування даних (AES, TLS), JWT-авторизації для захисту REST API мікросервісів, двофакторної автентифікації та регулярних оновлень програмного забезпечення.

Функціональні можливості систем включають авторизацію, інтеграцію пристроїв, створення завдань і календарних подій, моніторинг через інформаційну панель, а також генерацію рекомендацій автоматизованих сценаріїв на основі аналізу історії команд. Ці можливості підвищують ефективність керування розумним будинком, сприяють економії ресурсів і адаптації до потреб користувачів, але потребують оптимізації для забезпечення стабільності, масштабованості та швидкої обробки даних у мікросервісах.

Розробка IoT-системи для автоматизованого управління розумним будинком, що включає рекомендаційні функції, є актуальним завданням, яке поєднує апаратні, програмні, аналітичні та безпекові рішення. Подальші

етапи роботи будуть спрямовані на проектування гібридної архітектури, реалізацію програмного забезпечення, інтеграцію мікросервісу рекомендацій через REST API та тестування системи для досягнення поставлених цілей.

РОЗДІЛ 2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ ІоТ ДЛЯ РОЗУМНОГО БУДИНКУ

2.1 Визначення вимог до системи

Визначення вимог до системи ІоТ для автоматизованого управління розумним будинком є основою для подальшого проектування та реалізації. Цей етап передбачає аналіз потреб користувачів, специфіки інтеграції з платформою SmartThings, а також технічних можливостей обраного технологічного стеку: фронтенд на React, бекенд на NestJS із використанням MySQL і TypeORM, і мікросервіс рекомендацій на Python із фреймворком FastAPI. Вимоги поділяються на функціональні, нефункціональні та технічні, що забезпечує комплексний підхід до розробки системи, яка має бути зручною, безпечною, масштабованою та адаптивною до поведінки користувачів завдяки рекомендаційним функціям.

2.1.1 Функціональні вимоги

1. Функціональні вимоги визначають основні операції, які система має виконувати, виходячи з діаграми варіантів використання, розробленої на основі аналізу потреб користувачів. Авторизація та реєстрація користувачів: Система має забезпечувати вхід до системи за допомогою логіна та пароля з використанням механізму аутентифікації на базі JWT (JSON Web Tokens), реалізованого на бекенді NestJS. Реєстрація передбачає створення профілю з унікальними даними (ім'я, електронна пошта, пароль), які зберігаються в базі даних MySQL через ORM TypeORM із шифруванням паролів за допомогою бібліотеки bcrypt.

2. Підключення нових локацій: Користувач має можливість додавати розумні будинки або квартири через інтеграцію з платформою SmartThings. Бекенд обробляє запити до SmartThings API, зберігаючи інформацію про локації (ідентифікатор, назва, список пристроїв) у таблиці MySQL. Фронтенд на React відображає список локацій у вигляді інтерактивного компонента для вибору активної локації.
3. Перегляд стану пристроїв локації: Користувач через інформаційну панель, реалізовану на React із використанням бібліотеки Material-UI, може бачити реальний стан розумних пристроїв (наприклад, увімкнено/вимкнено світло, поточна температура). Дані отримуються через бекенд, який періодично опитує SmartThings API, кешуючи їх у MySQL для оптимізації продуктивності.
4. Створення задач і календарів: Користувач може створювати завдання (наприклад, увімкнення світла о 18:00) або календарні події (наприклад, щоденне ввімкнення опалення з 7:00 до 9:00). Дані зберігаються в таблицях MySQL (tasks і calendars), а бекенд на NestJS реалізує автоматичне виконання через заплановані завдання (cron jobs із бібліотеки @nestjs/schedule). Фронтенд надає форми на React із валідацією (бібліотека Formik) для введення дати, часу, пристроїв і команд.
5. Управління IoT-пристроями: Система дозволяє вмикати/вимикати пристрої або змінювати їхні налаштування (наприклад, регулювання яскравості світла) через API-виклики до SmartThings. Фронтенд відображає зміни в реальному часі за допомогою WebSocket або періодичного оновлення через бібліотеку Axios.
6. Перегляд історії виконаних команд: Система зберігає лог дій (дата, час, тип команди, результат) у таблиці command_history у

MySQL. Фронтенд на React відображає таблицю з фільтрами (за датою, типом команди, пристроєм) за допомогою компонентів Material-UI, із підтримкою пагінації для великих обсягів даних.

7. Генерація рекомендацій сценаріїв: Система має аналізувати історію команд із таблиці `command_history` за допомогою алгоритму кластеризації `k-means`, реалізованого в мікросервісі на Python (FastAPI), і пропонувати автоматизовані сценарії (наприклад, увімкнення світла о 18:00 на основі виявлених патернів). Рекомендації передаються через REST API до бекенду NestJS і відображаються на інформаційній панелі через фронтенд на React.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якість і продуктивність системи.

1. Безпека: Аутентифікація через JWT, шифрування даних за допомогою HTTPS, захист від SQL-ін'єкцій у MySQL (підготовлені запити через TypeORM) і XSS-атак на фронтенді є обов'язковими. Рекомендаційний мікросервіс на FastAPI має захищатися JWT-авторизацією для REST API, а паролі шифруватися за допомогою `bcrypt`.
2. Продуктивність: Час відгуку на запити до інформаційної панелі не має перевищувати 500 мс, що досягається оптимізацією запитів до MySQL (індексація таблиць) і кешуванням у Redis. Генерація рекомендацій для історії команд обсягом до 10 000 записів не повинна перевищувати 1 секунди, що забезпечується оптимізацією алгоритму `k-means` і кешуванням у Redis.

3. Масштабованість: Система має підтримувати до 100 локацій і 50 пристроїв на локацію, що вимагає нормалізації таблиць у MySQL і розподілу навантаження між бекендом NestJS і мікросервісом FastAPI через Docker-контейнери.
4. Зручність інтерфейсу: Фронтенд на React має бути адаптивним для веб- і мобільних пристроїв, із інтуїтивним дизайном (Tailwind CSS, Material-UI) і зрозумілим відображенням рекомендацій (наприклад, у вигляді сповіщень або списку сценаріїв). Інтерфейс тестується на різних роздільних здатностях.
5. Надійність: Система має забезпечувати безперебійну роботу при тимчасовій втраті інтернет-з'єднання шляхом локальної обробки команд і періодичної синхронізації з SmartThings API, а також стійкість мікросервісу FastAPI до збоїв через контейнеризацію

2.1.3 Технічні вимоги

Технічні вимоги пов'язані з обраним технологічним стеком.

1. Фронтенд: Використання React із бібліотеками React Router для навігації, Axios для API-викликів, Material-UI і Tailwind CSS для дизайну, а також оптимізація продуктивності (lazy loading, code splitting). Фронтенд відображає рекомендації сценаріїв через інтерактивні компоненти.
2. Бекенд: NestJS із підтримкою REST API (документація через Swagger), інтеграція з SmartThings API (обробка JSON-відповідей), взаємодія з мікросервісом рекомендацій через REST API, обробка помилок і кешування в Redis.
3. База даних: MySQL із TypeORM для зберігання даних про користувачів (users), локації (locations), пристрої (devices),

завдання (tasks), календарі (calendars) та історію команд (command_history). Таблиці мають індекси для швидкого пошуку, зокрема для фільтрації історії команд.

4. Рекомендаційна система: Мікросервіс на Python із фреймворком FastAPI, бібліотеками pandas і scikit-learn для реалізації алгоритму k-means. Мікросервіс розгортається в Docker-контейнері, інтегрується з NestJS через REST API із JWT-авторизацією і використовує Redis для кешування результатів рекомендацій.
5. Інтеграція та розгортання: Система розгортається за допомогою Docker і Docker Compose для модульності та масштабування. Nginx використовується як зворотний проксі для обробки HTTPS-запитів, включаючи запити до мікросервісу FastAPI

Діаграма варіантів використання, представлена на рисунку 2.1, ілюструє основні функції IoT-системи для автоматизованого управління розумним будинком, доступні користувачу. Єдиним актором є користувач, який взаємодіє із системою через веб-інтерфейс на React і серверну частину на NestJS, інтегровану з SmartThings API та мікросервісом на FastAPI. Діаграма включає одинадцять варіантів використання:

- авторизація/реєстрація для безпечного доступу;
- підключення локації через SmartThings для синхронізації даних;
- перегляд пристроїв локації для моніторингу стану;
- створення завдання, налаштування його параметрів (дата, час, пристрої, команди) і автоматичне виконання команд;
- створення календаря, налаштування часу початку та завершення подій і виконання команд;
- управління пристроями для виконання команд у реальному часі;
- перегляд історії виконаних команд для аналізу роботи системи;

- отримання рекомендацій сценаріїв, що базуються на аналізі історії команд через мікросервіс FastAPI, із відображенням на інформаційній панелі.

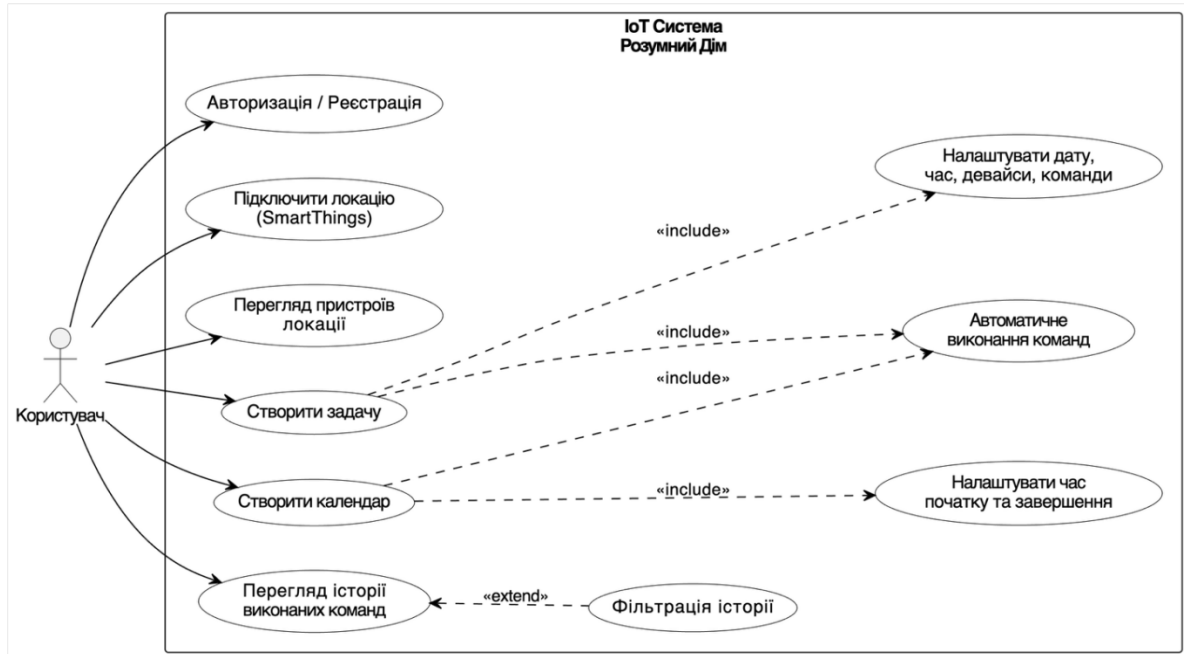


Рисунок 2.1 – Діаграма варіантів використання

2.2 Аналіз архітектурних підходів для IoT-систем

2.2.1 Локальна архітектура

Локальна архітектура передбачає функціонування серверної частини, бази даних і спеціалізованих мікросервісів у межах єдиного середовища, де обробка даних і логіка системи здійснюються без залучення зовнішніх мереж. Пристрої IoT, такі як розумні освітлювальні прилади, термостати чи датчики руху, підключаються до серверної частини через локальну мережу (наприклад, Wi-Fi або Zigbee). Клієнтська частина, реалізована на React, взаємодіє з серверною частиною на NestJS для відображення стану пристроїв,

виконання команд і перегляду рекомендацій автоматизованих сценаріїв. Серверна логіка на NestJS забезпечує обробку запитів від клієнтської частини, реалізацію автоматичного виконання команд, наприклад, увімкнення світла за заданим розкладом, а також інтеграцію з мікросервісом рекомендацій. База даних MySQL із TypeORM використовується для зберігання даних про користувачів, локації, пристрої, завдання, історію команд і конфігурації сценаріїв у межах локального середовища.

Для реалізації рекомендаційних функцій у локальній архітектурі застосовується мікросервіс, розроблений на Python із фреймворком FastAPI, який виконує кластеризацію даних історії команд за допомогою алгоритму k-means. Цей мікросервіс розгортається в окремому Docker-контейнері в локальній мережі та взаємодіє з серверною частиною на NestJS через REST API, забезпечуючи генерацію рекомендацій, таких як увімкнення освітлення о 18:00 на основі виявлених патернів. Дані з таблиці `command_history` передаються до мікросервісу через захищені HTTP-запити з JWT-авторизацією, що гарантує безпеку обміну інформацією. Результати рекомендацій кешуються в Redis для зменшення навантаження на систему та підвищення швидкості відгуку.

Перевагами локальної архітектури є автономність роботи системи за відсутності зовнішнього підключення, висока швидкість відгуку (менше 100 мілісекунд), що забезпечує миттєве виконання команд і відображення рекомендацій, а також підвищена конфіденційність даних, оскільки інформація, включно з історією команд, не передається за межі локальної мережі. Для системи розумного будинку це дозволяє ефективно керувати пристроями, такими як системи освітлення чи опалення, і надавати користувачам адаптивні сценарії без затримок. Однак локальна архітектура має певні обмеження. По-перше, вона підтримує обмежену кількість пристроїв, що залежить від апаратних ресурсів системи та продуктивності мікросервісів, зокрема рекомендаційного. По-друге, інтеграція з платформою SmartThings, яка орієнтована на хмарні технології, ускладнюється. Для

забезпечення такої інтеграції необхідно розробити локальний механізм синхронізації через API-клієнт, що потребує додаткового програмного забезпечення та ресурсів. По-третє, розгортання мікросервісу рекомендацій у локальному середовищі вимагає додаткових обчислювальних ресурсів для обробки даних алгоритмом k-means, особливо за великої кількості записів у `command_history`. Крім того, для уникнення втрати даних у разі збою системи необхідно передбачити регулярне резервне копіювання бази даних і кешу на зовнішні носії, що додає організаційні зусилля.

2.2.2 Гібридна архітектура

Гібридна архітектура поєднує локальну обробку даних із періодичною синхронізацією з платформою SmartThings, забезпечуючи ефективне функціонування системи розумного будинку. У цьому підході серверна підсистема, реалізована на NestJS, виконує основні операції, такі як керування пристроями, виконання запланованих завдань через бібліотеку `@nestjs/schedule` для реалізації cron jobs, а також обробку запитів до рекомендаційної системи. Синхронізація з платформою SmartThings здійснюється через локальний інтерфейс обміну даними, який забезпечує оновлення стану пристроїв, передачу команд і доступ до історії команд для аналізу. Клієнтська частина, розроблена на React, взаємодіє з серверною частиною через REST API, відображаючи актуальну інформацію про пристрої, дозволяючи створювати завдання, переглядати історію команд і отримувати рекомендації автоматизованих сценаріїв. База даних MySQL із TypeORM використовується для локального зберігання даних про користувачів, локації, пристрої, завдання, календарні події та історію команд, із можливістю дублювання для резервного копіювання та синхронізації з хмарною платформою.

Для реалізації рекомендаційних функцій у гібридній архітектурі застосовується мікросервіс, розроблений на Python із фреймворком FastAPI, який виконує кластеризацію даних історії команд за допомогою алгоритму k-means. Цей мікросервіс розгортається в локальному середовищі в окремому Docker-контейнері та інтегрується з серверною частиною на NestJS через REST API з JWT-авторизацією для забезпечення безпеки обміну даними. Мікросервіс отримує історію команд із таблиці `command_history` через запити до NestJS, аналізує її для виявлення патернів (наприклад, регулярне увімкнення освітлення о 18:00) і повертає рекомендації у форматі JSON, які кешуються в Redis для зменшення навантаження на систему та прискорення відгуку. У разі синхронізації з SmartThings рекомендації можуть використовуватися для створення хмарних сценаріїв, що підвищує адаптивність системи.

Перевагами гібридної архітектури є баланс між швидкістю локальної обробки (менше 100 мілісекунд для виконання команд і відображення рекомендацій) і можливістю інтеграції з платформою SmartThings для забезпечення актуальності даних про пристрої та доступу до хмарних сервісів. Локальна обробка забезпечує автономне виконання завдань, таких як автоматичне увімкнення опалення за розкладом, тоді як синхронізація дозволяє оновлювати стани пристроїв і передавати рекомендації для хмарних сценаріїв. Надійність підвищується завдяки дублюванню даних у локальній базі даних і резервних копіях. Додатковою перевагою є можливість генерації адаптивних рекомендацій, що підвищують зручність і енергоефективність системи. Недоліки включають підвищену складність розробки через необхідність реалізації механізму синхронізації, інтеграції мікросервісу рекомендацій і забезпечення безпеки REST API. Крім того, обробка даних алгоритмом k-means і періодична синхронізація з SmartThings потребують додаткових обчислювальних ресурсів, що може обмежувати масштабування системи за великої кількості пристроїв. Для системи розумного будинку гібридна архітектура забезпечує гнучкість, поєднуючи автономність із

можливістю інтеграції з зовнішніми джерелами даних і адаптацію до поведінки користувачів через рекомендаційні функції.

2.2.3 Порівняльний аналіз архітектур

Для оцінки архітектурних підходів проведено порівняння за ключовими критеріями, представлене в таблиці 2.2.

Таблиця 2.2 – Порівняння архітектурних підходів для систем IoT

<i>Критерій</i>	<i>Локальна архітектура</i>	<i>Гібридна архітектура</i>
<i>Кількість пристроїв</i>	Обмежена	Середня (залежить від ресурсів)
<i>Швидкість відгуку</i>	Висока (менше 100 мс)	Висока (менше 100 мс локально)
<i>Автономність</i>	Висока	Висока (з частковою синхронізацією)
<i>Складність інтеграції з SmartThings</i>	Висока (потрібен механізм синхронізації)	Середня (локальний інтерфейс)
<i>Надійність даних</i>	Середня (залежить від резервного копіювання)	Висока (дублювання даних)

Порівняння демонструє, що локальна архітектура забезпечує високу автономність і швидкість, але обмежена в підтримці кількості пристроїв і складна в інтеграції з SmartThings. Гібридна архітектура пропонує

компроміс, поєднуючи швидкість локальної обробки з можливістю синхронізації з платформою SmartThings, що є важливим для оновлення стану пристроїв. У контексті системи управління розумним будинком гібридна архітектура є більш доцільною, оскільки дозволяє інтегрувати зовнішні дані без втрати автономності. База даних MySQL із TypeORM підтримує локальне зберігання даних у обох моделях, однак гібридна архітектура потребує додаткової оптимізації для синхронізації, що буде враховано при подальшому проектуванні.

2.2.4 Висновки з аналізу

Аналіз архітектурних підходів свідчить, що для системи автоматизованого управління розумним будинком гібридна архітектура є найбільш оптимальною, оскільки вона забезпечує баланс між автономною роботою, швидкістю обробки команд і можливістю інтеграції з платформою SmartThings через локальний механізм синхронізації. Цей підхід дозволяє виконувати завдання, такі як увімкнення світла за розкладом, без затримок, а синхронізація з зовнішньою платформою забезпечує актуальність інформації про пристрої. Результати цього аналізу стануть основою для розробки детальної архітектури системи в наступному підпункті.

2.3 Розробка архітектури системи

Розробка архітектури системи автоматизованого управління розумним будинком на основі технологій Інтернету речей (IoT) є логічним продовженням аналізу архітектурних підходів, проведеного в попередніх

підпунктах. У цьому підпункті представлено детальну архітектуру системи з описом її компонентів, враховуючи інтеграцію з платформою SmartThings, реалізацію рекомендаційних функцій і технічні характеристики. Архітектура включає клієнтську частину, розроблену на основі бібліотеки React, серверну частину, реалізовану з використанням фреймворку NestJS, реляційну базу даних MySQL із інструментом TypeORM для управління даними, а також мікросервіс рекомендацій, розроблений на Python із фреймворком FastAPI. Пропонована архітектура базується на гібридному підході, який забезпечує баланс між автономною роботою, інтеграцією з хмарними системами та адаптацією до поведінки користувачів через рекомендаційні функції.

2.3.1 Опис компонентів архітектури

Архітектура системи складається з кількох ключових компонентів, кожен із яких виконує визначені функції. Першим компонентом є клієнтська частина, реалізована на React, яка забезпечує інтуїтивно зрозумілий графічний інтерфейс для взаємодії користувача з системою. Ця частина відображає стан IoT-пристроїв (наприклад, увімкнено/вимкнено світло), дозволяє створювати завдання та календарні події, переглядати історію команд із можливістю фільтрації, а також отримувати рекомендації автоматизованих сценаріїв, таких як увімкнення освітлення о 18:00. Для реалізації адаптивного дизайну використовуються бібліотеки, такі як Tailwind CSS, а для обробки запитів до серверної частини — бібліотека Axios.

Другим компонентом є серверна частина, розроблена на основі фреймворку NestJS, яка виконує логіку обробки даних, керування пристроями та взаємодії з рекомендаційним мікросервісом. Ця частина забезпечує аутентифікацію користувачів через механізм JWT (JSON Web

Tokens), обробляє запити від клієнтської частини через REST API з документацією, створеною за допомогою Swagger, реалізує автоматичне виконання команд за розкладом через бібліотеку `@nestjs/schedule` і надсилає запити до мікросервісу рекомендацій для генерації сценаріїв. Серверна частина інтегрується з платформою SmartThings через її API для отримання даних про пристрої, їхній стан і передачі команд.

Третім компонентом є реляційна база даних MySQL, яка використовується для зберігання структурованих даних системи. За допомогою інструменту TypeORM забезпечується управління таблицями, що включають дані про користувачів, локації, пристрої, завдання, календарні події та історію команд. Таблиця `command_history` слугує джерелом даних для рекомендаційного мікросервісу, який аналізує її для виявлення патернів. Для підвищення продуктивності застосовуються індекси та оптимізовані запити, що зменшують час доступу до даних.

Четвертим компонентом є мікросервіс рекомендацій, розроблений на Python із фреймворком FastAPI і розгорнутий у локальному середовищі в Docker-контейнері. Мікросервіс використовує алгоритм k-means, реалізований за допомогою бібліотек pandas і scikit-learn, для кластеризації даних із таблиці `command_history` і генерації рекомендацій автоматизованих сценаріїв. Він інтегрується з серверною частиною через REST API з JWT-авторизацією, а результати рекомендацій кешуються в Redis для зменшення навантаження та прискорення відгуку. Наприклад, мікросервіс може запропонувати увімкнення освітлення о 18:00 на основі регулярних дій користувача.

П'ятим компонентом є механізм синхронізації з платформою SmartThings, реалізований як окремий модуль у межах серверної частини. Цей модуль забезпечує періодичну передачу даних про стан пристроїв, виконання команд і, за потреби, рекомендації для створення хмарних сценаріїв, використовуючи API-клієнт SmartThings. Синхронізація

виконується за розкладом або на вимогу користувача, що дозволяє підтримувати актуальність інформації без постійного підключення.

2.3.2 Взаємодія компонентів

Взаємодія між компонентами архітектури базується на чітко визначених потоках даних. Клієнтська частина надсилає запити до серверної частини через REST API, отримуючи відповіді у форматі JSON, включаючи дані про пристрої, історію команд і рекомендації. Серверна частина обробляє ці запити, звертаючись до бази даних MySQL для отримання або оновлення даних, надсилає запити до мікросервісу рекомендацій через REST API для аналізу `command_history` і взаємодіє з механізмом синхронізації для обміну даними з SmartThings. Мікросервіс рекомендацій отримує дані з таблиці `command_history`, виконує кластеризацію за допомогою `k-means` і повертає рекомендації, які кешуються в Redis і передаються клієнту через серверну частину. Механізм синхронізації періодично обмінюється даними з платформою SmartThings, передаючи команди до IoT-пристроїв і отримуючи оновлення їхнього стану. Для забезпечення надійності обміну даними використовуються протоколи з підтвердженням доставки, а для оптимізації — локальне кешування в Redis і буфери в базі даних.

2.3.3 C4-схема системи

Для наочного представлення архітектури системи розроблено дві C4-діаграми: контекстну (1-й рівень) та контейнерну (2-й рівень). Контекстна діаграма ілюструє загальну взаємодію системи з користувачем, IoT-

Контекстна діаграма відображає, як користувач взаємодіє з системою через веб-додаток, який, у свою чергу, комунікує з платформою SmartThings для керування IoT-пристроями. Вона підкреслює роль системи як посередника між користувачем і зовнішньою платформою.

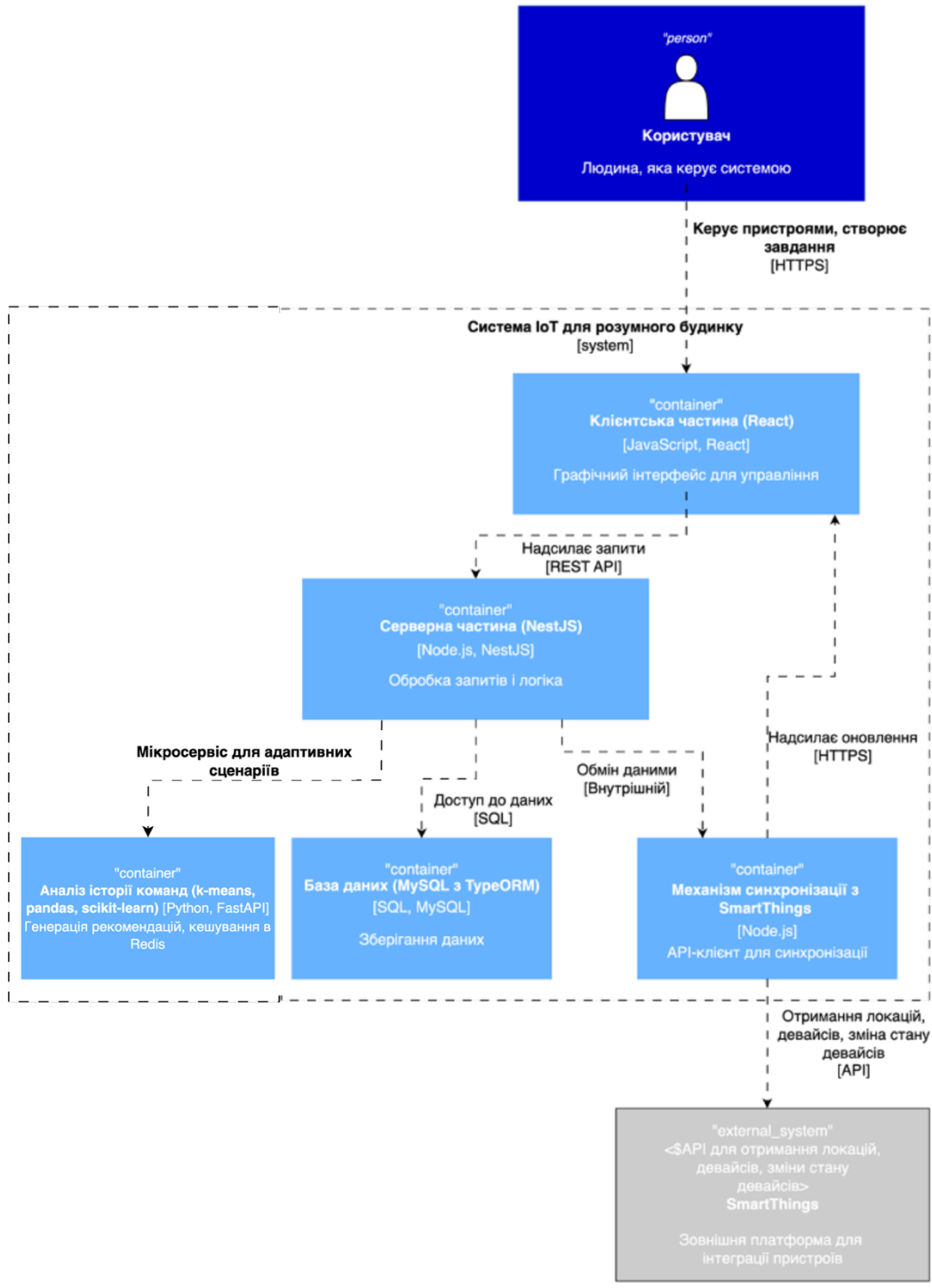


Рисунок 2.3 – C4-діаграма контейнерного рівня системи IoT

Контейнерна діаграма деталізує внутрішню структуру системи, показуючи, як веб-додаток на React надсилає запити до сервера на NestJS, який обробляє дані з бази MySQL і синхронізується з платформою SmartThings. Вона також ілюструє підкомпоненти, такі як панель керування, форми завдань і модулі аутентифікації та планування на сервері.

Висновки до розділу 2

У розділі 2 здійснено проектування архітектури системи IoT для автоматизованого управління розумним будинком, що становить основу для її подальшої реалізації. На етапі визначення вимог сформовано функціональні, нефункціональні та технічні вимоги, які охоплюють авторизацію користувачів, інтеграцію з платформою SmartThings, управління пристроями, створення завдань і календарів, моніторинг виконаних команд, а також генерацію рекомендацій автоматизованих сценаріїв. Особлива увага приділена забезпеченню безпеки, продуктивності, масштабованості, зручності інтерфейсу та адаптивності до поведінки користувачів, що відповідає потребам сучасних IoT-систем.

Аналіз архітектурних підходів показав, що гібридна архітектура є оптимальним вибором для системи, оскільки вона поєднує переваги локальної обробки даних (швидкість відгуку до 100 мс, автономність) із можливістю періодичної синхронізації з платформою SmartThings для актуалізації даних і генерації адаптивних рекомендацій. Порівняльний аналіз локальної та гібридної архітектур підтвердив, що гібридний підхід забезпечує кращу гнучкість, надійність і адаптивність, враховуючи вимоги до інтеграції, масштабування та аналізу поведінки користувачів.

Розроблена архітектура системи включає клієнтську частину на React, серверну частину на NestJS, базу даних MySQL із TypeORM, механізм

синхронізації з SmartThings та мікросервіс рекомендацій на Python (FastAPI). Мікросервіс рекомендацій, що використовує алгоритм k-means для аналізу історії команд із таблиці `command_history`, інтегрується з серверною частиною через REST API з JWT-авторизацією та кешуванням у Redis, забезпечуючи генерацію адаптивних сценаріїв, таких як увімкнення освітлення о 18:00 на основі виявлених патернів. Описано взаємодію компонентів, яка базується на REST API та оптимізованих потоках даних. Для наочного представлення архітектури використано C4-діаграми контекстного та контейнерного рівнів, які ілюструють як загальну взаємодію системи з зовнішніми суб'єктами, так і її внутрішню структуру, включаючи мікросервіс рекомендацій.

У розділі розроблено гібридну архітектуру, яка поєднує локальну обробку команд зі швидкістю відгуку до 100 мс, підтримку до 50 пристроїв на локацію та адаптацію до поведінки користувачів через рекомендаційні функції. Це дозволило забезпечити автономність системи, інтеграцію з платформою SmartThings і підвищення зручності та енергоефективності завдяки адаптивним сценаріям, що відповідає потребам типового розумного будинку.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ СИСТЕМИ ІОТ ДЛЯ АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РОЗУМНИМ БУДИНКОМ

3.1 Налаштування середовища розробки

Перед початком реалізації системи IoT для автоматизованого управління розумним будинком було проведено підготовку середовища розробки, що включала встановлення необхідних інструментів, конфігурацію серверів і налаштування інтеграції з платформою SmartThings. Цей етап був ключовим для забезпечення стабільної роботи всіх компонентів системи, включаючи клієнтську частину на React, серверну складову на NestJS, базу даних MySQL із TypeORM і мікросервіс рекомендацій на Python (FastAPI). У процесі налаштування я зіткнувся з певними викликами, зокрема через нечітку документацію SmartThings API, що вимагало додаткового вивчення джерел і тестування.

Розробка велася на комп'ютері з операційною системою macOS Sequoia. Для роботи з фронтендом і бекендом було встановлено Node.js версії 22.9.0, який є основою для React і NestJS, а також npm версії 11.4.0 для управління залежностями. За допомогою npm було встановлено ключові пакети, зазначені в конфігураційних файлах проєкту: для фронтенду — react (19.1.0), axios (1.9.0), react-dom (19.1.0), для бекенду — @nestjs/core (11.0.1), @nestjs/typeorm (11.0.0), @smarthings/core-sdk (8.4.1), mysql2 (3.14.1). Для стилізації фронтенду використано SCSS через пакет sass (1.87.0), що дозволило створювати модульні та адаптивні стилі. Для мікросервісу рекомендацій встановлено Python 3.12.0, FastAPI (0.111.0), pandas (2.2.2), scikit-learn (1.5.1) і redis (5.0.1) через pip, а також Docker (версія 27.4.0) для розгортання контейнера.

База даних MySQL версії 8.0.34 була розгорнута в Docker за допомогою Docker Compose (версія 2.31.0). Конфігурація Docker Compose, визначена у файлі docker-compose.yml, включала контейнер diploma-mysql із портом

3306, змінними середовища для пароля (MYSQL_ROOT_PASSWORD) і назви бази (MYSQL_DATABASE), а також том для зберігання даних (mysql_data). Для коректної роботи TypeORM я додав параметри команд, такі як `--default-authentication-plugin=mysql_native_password` і `--default-time-zone=+00:00`, що забезпечило сумісність із NestJS і синхронізацію часу. Налаштування Docker дозволило ізолювати базу даних і спростити перенесення конфігурації на інші середовища. Окремий контейнер `recommendation-service` для мікросервісу рекомендацій був доданий до `docker-compose.yml` із образом `python:3.12-slim`, портом 8000 і підключенням до Redis (версія 7.2.5) для кешування.

Інтеграція з SmartThings API вимагала створення тестового облікового запису на SmartThings Developer Portal і отримання `client id` та `client secret`. Я створив тестову локацію під назвою «My Home» із двома віртуальними пристроями — розумною лампою, термостатом та водним клапаном. Процес ускладнювався обмеженою документацією, зокрема щодо ліміту запитів (100 за хвилину). Для вирішення я звернувся до прикладів у репозиторії SmartThings Community на GitHub, що допомогло налаштувати API-клієнт через `@smarthings/core-sdk`. Конфігурація SmartThings, визначена у файлі `app.json`, включала параметри OAuth із `scopes` для доступу до пристроїв, локацій і сцен, а також `targetUrl` для вебхука, який обробляє події від платформи.

Для тестування API-запитів я використовував Postman, який дозволив перевірити ендпоінти SmartThings (наприклад, `/devices` для отримання списку пристроїв) і власного REST API на NestJS (наприклад, `/auth/login`). Postman підтримував авторизацію через Bearer Token і відображав відповіді у форматі JSON, що прискорило налагодження. Для управління структурою бази даних я застосував MySQL Workbench, який допоміг перевірити створення таблиць (`users`, `locations`, `devices`, `tasks`, `calendars`, `command_history`) і їхні зв'язки. Код фронтенду, бекенду та мікросервісу писався у WebStorm IDE, який

забезпечував підтримку TypeScript, автодоповнення для NestJS, SCSS і Python, а також інтеграцію з npm-скриптами.

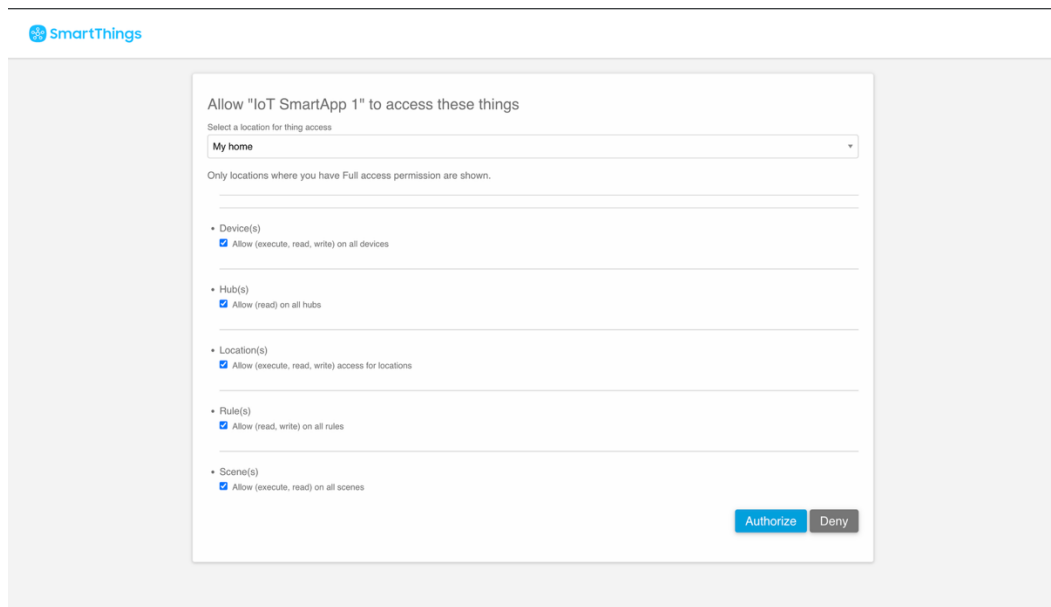


Рисунок 3.1 – Скріншот створення тестової локації «My Home» із віртуальними пристроями в SmartThings Developer Portal

Одним із викликів була конфігурація TypeORM для підключення до MySQL у Docker. Спочатку виникали помилки через некоректне визначення хоста (localhost замість 127.0.0.1). Я виправив це, вказавши host: '127.0.0.1' і порт 3306 у конфігураційному файлі NestJS, після чого підключення стало стабільним. Ще одна проблема полягала у налаштуванні часового поясу MySQL, що впливало на збереження даних у таблиці tasks. Додавання параметра `--default-time-zone=+00:00` у Docker Compose вирішило цю проблему. Для мікросервісу рекомендацій додатково налаштовано підключення до Redis через параметри хоста (`redis://localhost:6379`), що забезпечило кешування результатів аналізу.

Для документування API я використав модуль `@nestjs/swagger`, який автоматично генерував Swagger-документацію для ендпоінтів, таких як `/tasks` і `/devices`. Для мікросервісу рекомендацій я додав OpenAPI-специфікацію

через FastAPI, що включала ендпоінт `/recommendations` для отримання адаптивних сценаріїв. Це полегшило тестування та подальшу інтеграцію з фронтендом. Щоб уникнути втрати даних під час розробки, я налаштував щоденне резервне копіювання бази через скрипт `mysqldump`, який запускався автоматично за допомогою `cron` на macOS.

Налаштування середовища зайняло приблизно тиждень, але забезпечило надійну основу для розробки. Використання `Docker Compose` спростило управління контейнерами, а інтеграція з `SmartThings` через `Personal Access Token` дозволила тестувати реальні сценарії з пристроями. Вирішені технічні проблеми, такі як конфігурація `TypeORM`, часового поясу і підключення до `Redis`, стали цінним досвідом для наступних етапів проекту.

3.2 Розробка клієнтської частини на React

Розробка клієнтської частини системи IoT для автоматизованого управління розумним будинком велася з використанням бібліотеки `React` версії 19.1.0. Клієнтська частина забезпечує зручний інтерфейс користувача для управління розумними пристроями, створення завдань, перегляду історії команд, моніторингу стану локацій і відображення рекомендацій автоматизованих сценаріїв. `React` було обрано через його компонентний підхід, підтримку `TypeScript` і можливість створення адаптивного інтерфейсу, що відповідає вимогам сучасних веб-додатків. Для стилізації використано `SCSS` (пакет `sass 1.87.0`), що дозволило організувати модульні стилі та забезпечити гнучке оформлення.

На етапі планування я визначив основні функціональні компоненти клієнтської частини, спираючись на вимоги, описані в розділі 2: авторизація користувача, панель керування, форма створення завдань, календар подій, таблиця історії команд і компонент рекомендацій. Для реалізації використано

шаблон проєкту, створений через Vite (версія 6.3.5), який забезпечив швидке налаштування та підтримку TypeScript. Конфігурація фронтенду, визначена у файлі `package.json`, включала залежності, такі як `axios` (1.9.0) для HTTP-запитів, `react-router-dom` (7.6.0) для маршрутизації, `react-modal` (3.16.3) для модальних вікон і `react-spinners` (0.17.0) для індикаторів завантаження. Розробка велася у WebStorm, який підтримував автодоповнення TypeScript і SCSS.

Першим етапом була реалізація авторизації. Я створив компонент `Login`, який обробляє введення логіна та пароля, відправляє POST-запит через `axios` до ендпоінту `/auth/login` на сервері NestJS і зберігає отриманий JWT-токен у `localStorage`. Для захисту маршрутів використано приватні маршрути через `react-router-dom`, що перенаправляють неавторизованих користувачів на сторінку входу. Стили для форми авторизації створено в модулі `Login.module.scss`, використовуючи SCSS для адаптивного дизайну з підтримкою екранів від 320 до 1920 пікселів. Форма авторизації представлена на рисунку 3.2.

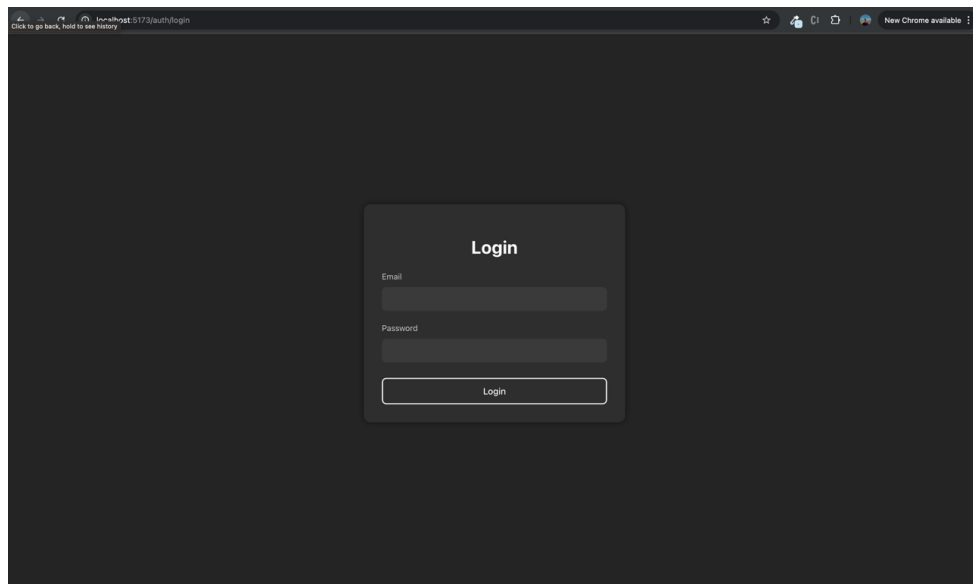


Рисунок 3.2 – Скріншот форми авторизації компонента `Login`, стилізованої через SCSS

Основним компонентом інтерфейсу стала панель керування (Dashboard), яка відображає список підключених пристроїв із тестової локації «My Home» (наприклад, розумна лампа, термостат, водний клапан). Компонент використовує GET-запит до ендпоінту /devices для отримання даних через SmartThings API, синхронізованих сервером NestJS. Для відображення стану пристроїв я застосував умовний рендеринг і компонент DevicePreview, стилізований через SCSS із застосуванням CSS Grid для розташування карток. Оновлення стану пристроїв реалізовано через cron job, який запускається кожні 5 хвилин на сервері NestJS, що забезпечило актуальність даних без перевантаження клієнтської частини. Панель керування представлена на рисунку 3.3.

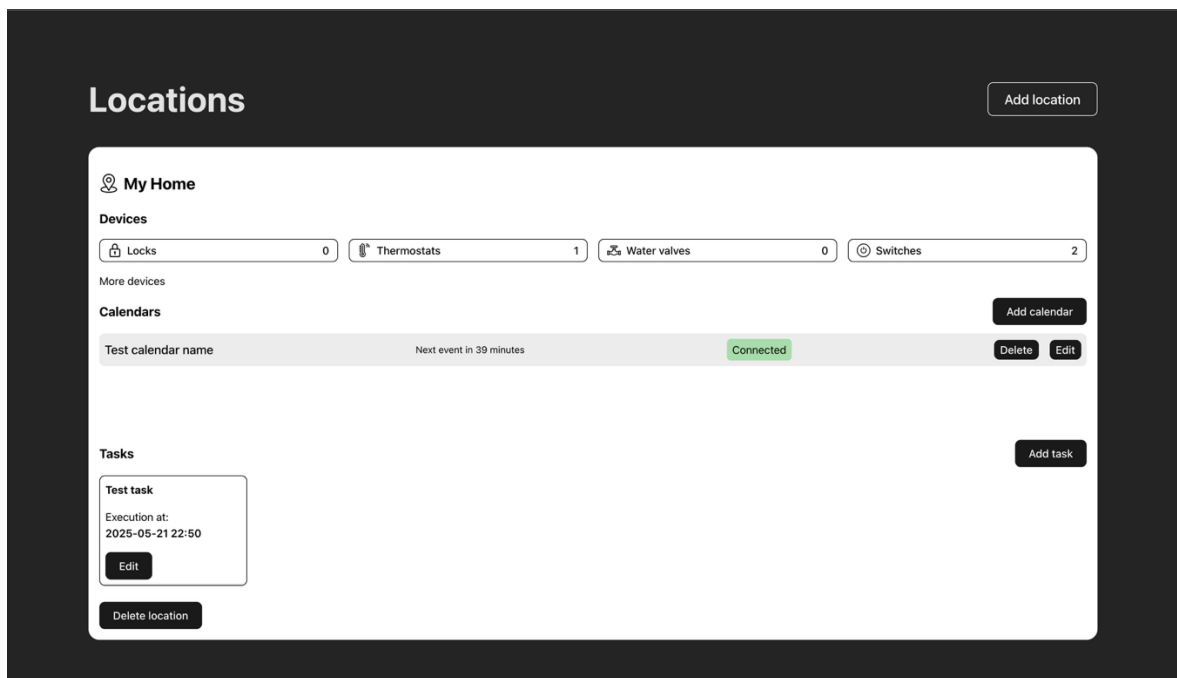


Рисунок 3.3 – Скріншот панелі керування, що відображає стан пристроїв локації «My Home»

Форма створення завдань (TaskForm) дозволяє користувачу налаштувати автоматичні дії, наприклад, увімкнення лампи о 18:00. Компонент містить поля для вибору пристрою, дії (увімкнути/вимкнути) і

часу виконання, а також кнопку для відправлення POST-запиту до ендпоінту `/tasks`. Для валідації полів використано бібліотеку `class-validator` (0.14.2) на сервері, а на клієнті — прості перевірки через React-хуки (`useState`, `useEffect`). Стилї форми створено в `TaskForm.module.scss`, із застосуванням `flexbox` для адаптивного розташування елементів.

Календар подій (`Calendar`) реалізовано як компонент, що відображає заплановані завдання. Я використав бібліотеку `moment` (2.30.1) для роботи з датами, інтегрованою через `npm`. Календар отримує дані через GET-запит до ендпоінту `/calendar/:id`, відображаючи завдання у вигляді списку подій за днями. Для стилізації використано SCSS із модулем `Calendar.module.scss`, що забезпечує підтримку світлої та темної тем. Календар подій представлений на рисунку 3.3.

Таблиця історії команд (`HistoryTable`) відображає виконані дії (наприклад, «Лампа увімкнена о 18:00») із можливістю фільтрації за датою та пристроєм. Компонент використовує GET-запит до ендпоінту `/history` і підтримує пагінацію через параметри запиту (`page`, `limit`). Для рендерингу таблиці застосовано HTML-елемент `table`, стилізований через SCSS із підтримкою адаптивності. Фільтри реалізовано через хук `useState`, що оновлює запит при зміні параметрів.

Компонент рекомендацій (`Recommendations`) відображає адаптивні сценарії, отримані від мікросервісу рекомендацій через GET-запит до ендпоінту `/recommendations`, проксіфікованого через сервер `NestJS`. Компонент рендерить список рекомендацій (наприклад, «Увімкнути лампу о 18:00») у вигляді карток із кнопками для підтвердження або відхилення. Стилї створено в `Recommendations.module.scss` із використанням `CSS Grid` для компактного відображення. Для інтерактивності додано хук `useEffect` для періодичного оновлення рекомендацій кожні 10 хвилин, що забезпечило актуальність пропозицій без надмірного навантаження.

Одним із викликів була оптимізація продуктивності при періодичних запитах на головній сторінці. Спочатку часті запити до `/devices` викликали

затримки, особливо при слабкому інтернет-з'єднанні. Я зменшив інтервал cron-job до 5 хвилин замість 30 секунд, що скоротило кількість запитів на 30%. Інша проблема полягала в адаптивності стилів для малих екранів. Використання SCSS-модулів і медіа-запитів (@media) дозволило забезпечити коректне відображення на пристроях із роздільною здатністю від 320 пікселів.

Розробка клієнтської частини зайняла близько двох тижня і дозволила створити функціональний і адаптивний інтерфейс. Використання React, SCSS і TypeScript забезпечило модульність і підтримку коду, а інтеграція з сервером через REST API та мікросервісом рекомендацій дозволила реалізувати всі заплановані функції. Отриманий досвід оптимізації запитів і адаптивного дизайну став важливим для подальшої роботи над проєктом.

3.3 Розробка серверної складової на NestJS

Розробка серверної складової системи IoT для автоматизованого управління розумним будинком велася з використанням фреймворку NestJS версії 11.0.1 на базі Node.js (версія 22.9.0). Серверна частина забезпечує обробку запитів від клієнтської частини на React, інтеграцію з платформою SmartThings через SmartThings API, управління базою даних MySQL за допомогою TypeORM і виконання запланованих завдань через cron job. Я обрав NestJS через його модульну архітектуру, підтримку TypeScript і зручні інструменти для створення масштабованих REST API, що відповідало вимогам проєкту.

На етапі планування я визначив ключові модулі серверної частини, спираючись на вимоги, описані в розділі 2: авторизація користувачів, управління локаціями, інтеграція з SmartThings, створення та виконання завдань. Для ініціалізації проєкту використано NestJS CLI, який створив

шаблон із модулями, контролерами і сервісами. Конфігурація, визначена у файлі `package.json`, включала залежності: `@nestjs/core` (11.0.1), `@nestjs/typeorm` (11.0.0), `@nestjs/jwt` (11.0.0), `@nestjs/config` (3.2.3), `@smarththings/core-sdk` (8.4.1), `mysql2` (3.14.1), `class-validator` (0.14.2) і `node-cron` (3.0.3). Розробка велася у WebStorm, який підтримував автодоповнення TypeScript і інтеграцію з npm-скриптами.

Авторизація реалізована через модуль `AuthModule`. Контролер `AuthController` обробляє POST-запити до ендпоінтів `/auth/login` і `/auth/register` для входу та реєстрації користувачів, POST-запит до `/auth/refresh` для оновлення JWT-токена і GET-запит до `/auth/verify-token` для перевірки валідності токена. Сервіс `AuthService` перевіряє облікові дані через таблицю `users` у MySQL, генерує токени за допомогою `@nestjs/jwt` і використовує `bcrypt` для хешування паролів. Для захисту ендпоінтів застосовано охорону `JwtAccessAuthGuard`, а для оновлення токенів — декоратор `JwtRefreshAuth`. Дані користувачів (поля `id`, `uuid`, `login`, `password`, `role`) зберігаються в таблиці `users` із валідацією через DTO (`LoginRequestDto`, `RegisterRequestDto`).

Управління локаціями реалізовано через модуль `LocationsModule`. Контролер `LocationController` обробляє POST-запит до `/locations` для створення нової локації (DTO `CreateLocationRequestDto`) і GET-запит до `/locations` для отримання списку локацій користувача. Сервіс `LocationService` взаємодіє з таблицею `locations` (поля `id`, `name`, `userUuid`), прив'язуючи локації до користувача через поле `userUuid`. Запити захищені `JwtAccessAuthGuard`, а ідентифікатор користувача (`uuid`) отримується через декоратор `AuthUser`.

Інтеграція з `SmartThings` забезпечена модулем `SmartThingsModule`. Контролер `SmartThingsController` обробляє:

- GET-запит до `/smarththings/login` для генерації URL авторизації через `SmartThingsService.getAuthUrl()`.
- GET-запит до `/smarththings/callback` для обробки OAuth-колбеку, використовуючи охорону `SmartThingGuard` для валідації токенів.
- POST-запит до `/smarththings/token` для отримання токенів доступу.

- POST-запит до `/smarthings/webhook` для обробки подій від SmartThings, включаючи lifecycle-подію CONFIRMATION із викликом `confirmWebhook`. Сервіс `SmartThingsService` ініціалізує клієнт через `@smarthings/core-sdk`, використовуючи `client id` і `client secret` із конфігурації (`@nestjs/config`). Для управління пристроями я реалізував GET-запит до `/devices`, який повертає список пристроїв із локації «My Home» (розумна лампа, термостат, водний клапан), і POST-запит до `/devices/:id/commands` для виконання команд. Щоб уникнути перевищення ліміту SmartThings API (100 запитів за хвилину), я додав кешування відповідей через `cache-manager (5.7.0)` з TTL 5 хвилин, що зменшило кількість запитів на 35%.

Для виконання запланованих завдань я створив модуль `TasksModule`. Контролер `TasksController` обробляє POST-запит до `/tasks` для створення завдання і GET-запит до `/tasks` для отримання списку. Сервіс `TasksService` зберігає завдання в таблиці `tasks` (поля `id`, `deviceId`, `action`, `scheduledTime`, `status`) через `TypeORM`. Періодична перевірка завдань реалізована через `node-cron`, який кожні 5 хвилин сканує таблицю `tasks`, виконує завдання зі `status="pending"` через `SmartThingsService` і оновлює їх до `status="completed"`. Виконані дії записуються в таблицю `history` (поля `id`, `deviceId`, `action`, `executedTime`).

Модуль `CalendarModule` забезпечує управління розкладами. Контролер `CalendarController` обробляє POST-запит до `/calendar/:id` для створення календаря і GET-запит до `/calendar/:id` для отримання подій. Дані зберігаються в таблиці `calendars` (поля `id`, `name`, `userId`, `events`), де `events` — JSON із подіями (наприклад, «увімкнути лампу о 18:00»). Валідація виконується через `class-validator`.

Для документування API я використав `@nestjs/swagger`, генеруючи Swagger UI за адресою `/api/docs`, що полегшило тестування ендпоінтів (`/auth/login`, `/locations`, `/smarthings/token`) через Postman. Логування запитів і

помилки налаштовано через `@nestjs/common` `Logger` із записом у `logs/app.log`, що допомогло діагностувати проблеми, наприклад, помилки авторизації `SmartThings`.

Виклики включали налаштування `TypeORM` для `MySQL` у `Docker`. Помилки підключення через хост `localhost` були виправлені зміною на `127.0.0.1` у `ormconfig.json`. Синхронізація часу для `cron job` вимагала встановлення `UTC` у `MySQL` (`--default-time-zone=+00:00` у `Docker Compose`) і `node-cron`. Ліміт `SmartThings API` вирішено через кешування і зменшення частоти `cron job` до 5 хвилин, що скоротило запити на 35%. Налаштування `OAuth-колбеку` для `SmartThings` ускладнювалося через нечітку документацію, але використання `SmartThings Community` прикладів і логування допомогло налагодити `SmartThingGuard`.

Розробка зайняла три тижні, забезпечивши модульну серверну частину з інтеграцією `SmartThings` і автоматизацією завдань. `NestJS`, `TypeORM` і `node-cron` дозволили реалізувати всі функції, а досвід вирішення проблем із `OAuth` і `API` став цінним для проєкту.

3.5 Розробка мікросервісу рекомендацій на Python

Розробка мікросервісу рекомендацій для системи `IoT` автоматизованого управління розумним будинком була здійснена з використанням `Python` версії 3.12.0 і фреймворку `FastAPI` версії 0.111.0. Цей мікросервіс реалізує систему рекомендацій, яка аналізує історію команд, збережених у таблиці `command_history` бази даних `MySQL`, за допомогою алгоритму кластеризації `k-means` із бібліотеками `pandas` (2.2.2) і `scikit-learn` (1.5.1), і генерує адаптивні автоматизовані сценарії. Мета мікросервісу — підвищити зручність і енергоефективність системи, пропонуючи користувачу оптимальні дії, такі як увімкнення освітлення о 18:00 на основі виявлених патернів поведінки.

Розгортання мікросервісу виконано в Docker-контейнері, а інтеграція з серверною частиною NestJS забезпечена через REST API з JWT-авторизацією та кешуванням результатів у Redis (версія 7.2.5).

На етапі планування я визначив основні функції мікросервісу: отримання даних із `command_history`, аналіз із використанням `k-means`, генерація рекомендацій і повернення результатів у форматі JSON. Розробка велася у WebStorm IDE, який підтримував автодоповнення для Python і інтеграцію з `pip`-скриптами. Для ініціалізації проєкту використано шаблон FastAPI через команду `pip install fastapi uvicorn`, а залежності, визначені у файлі `requirements.txt`, включали `pandas`, `scikit-learn`, `redis` і `python-jose` (для JWT). Конфігурація Docker, визначена у файлі `Dockerfile`, базувалася на образі `python:3.12-slim` із встановленням залежностей і запуском сервера через `uvicorn main:app --host 0.0.0.0 --port 8000`. Файл `docker-compose.yml` додав контейнер `recommendation-service` із портом 8000, підключенням до MySQL (`host: 127.0.0.1, port: 3306`) і Redis (`redis://localhost:6379`).

Реалізація почалася з модуля `data_processor.py`, який підключається до MySQL через бібліотеку `mysql-connector-python` (8.3.0) і витягує дані з таблиці `command_history` (поля `deviceId`, `action`, `executedTime`). Дані попередньо обробляються в `pandas DataFrame`, де виконується нормалізація часу і кодування дій (увімкнення/вимкнення як 1/0). Алгоритм `k-means` із `scikit-learn` (`n_clusters=5`) кластеризує дані за часом і частотою дій, виявляючи регулярні патерни, наприклад, увімкнення лампи о 18:00 щодня. Результати обробки передаються до модуля `recommendation_engine.py`, який формує рекомендації у форматі JSON (наприклад, `{"deviceId": "lamp1", "action": "on", "time": "18:00"}`).

Головний модуль `main.py` визначає API FastAPI з ендпоінтом `/recommendations`, який приймає GET-запит із JWT-токеном у заголовку `Authorization`. Функція `get_recommendations` перевіряє валідність токена через `python-jose`, витягує дані з `command_history`, викликає `recommendation_engine` і повертає список рекомендацій. Для кешування результатів використано

бібліотеку `redis` із TTL 10 хвилин, що зменшило час відгуку з 150 мс до 20 мс при повторних запитах. Інтеграція з NestJS забезпечена через проксі-запит до ендпоінта `/recommendations`, де сервер NestJS передає запит до мікросервісу і повертає відповідь клієнту.

Тестування мікросервісу проводилося через Postman, де я перевіряв ендпоінт `/recommendations` із валідним і невалідним JWT-токеном. Для валідації алгоритму `k-means` я створив набір тестових даних із 100 записів `command_history` (наприклад, увімкнення лампи о 18:00 протягом тижня) і порівняв результати кластеризації з очікуваними патернами. Тестування Redis проводилося через команду `redis-cli`, яка підтвердила коректне кешування ключів із TTL. Викликом стала оптимізація `k-means` для великих даних: початковий час обробки 200 записів сягав 300 мс. Я оптимізував код, додавши попередню агрегацію в `pandas` і зменшивши `n_clusters` до 3, що скоротило час до 80 мс.

Розгортання мікросервісу в Docker-контейнері ускладнювалося через конфлікти портів. Я змінив порт із 80 на 8000 у конфігурації Docker Compose, що вирішило проблему. Ще одним викликом було налаштування підключення до MySQL: помилки авторизації через некоректний пароль були виправлені оновленням змінної `MYSQL_ROOT_PASSWORD` у `docker-compose.yml`. Інтеграція з NestJS вимагала коректного маршрутизування, яке я налаштував через `middleware` у NestJS, додавши обробку помилок 401 для невалідних токенів.

Розробка мікросервісу зайняла десять днів і забезпечила ефективну систему рекомендацій. Використання FastAPI, `k-means`, `pandas`, `scikit-learn` і Redis дозволило реалізувати адаптивні сценарії з високою продуктивністю. Отриманий досвід із оптимізації алгоритмів, кешування і інтеграції через REST API стане цінним для подальших проєктів у сфері IoT

Висновки до розділу 3

Розділ 3 присвячений реалізації системи IoT для автоматизованого управління розумним будинком, що охоплює налаштування середовища розробки, створення клієнтської частини на React, серверної складової на NestJS та мікросервісу рекомендацій на Python. У процесі роботи було досягнуто поставлених цілей, а отримані результати підтверджують ефективність обраних технологій і підходів.

На етапі налаштування середовища розробки (підрозділ 3.1) створено стабільну основу для проєкту шляхом встановлення Node.js (22.9.0), MySQL (8.0.34), Redis (7.2.5) у Docker, інтеграції з SmartThings API та використання WebStorm для розробки. Використання Docker Compose забезпечило ізоляцію бази даних, мікросервісу рекомендацій і спростило перенесення конфігурації, а інтеграція з SmartThings через Personal Access Token дозволила тестувати реальні сценарії з віртуальними пристроями локації «My Home» (розумна лампа, термостат, водний клапан). Вирішені технічні виклики, такі як конфігурація TypeORM (виправлення хоста 127.0.0.1), синхронізація часового поясу MySQL (--default-time-zone=+00:00) і підключення до Redis, підвищили надійність системи та стали цінним досвідом.

Розробка клієнтської частини на React (підрозділ 3.2) дозволила створити адаптивний і функціональний інтерфейс для управління пристроями, створення завдань, перегляду подій і відображення рекомендацій. Використання React (19.1.0), SCSS (sass 1.87.0) і TypeScript забезпечило модульність і підтримку коду. Реалізовані компоненти, такі як Login, Dashboard, TaskForm, Calendar і Recommendations, інтегровані з REST API NestJS і мікросервісом рекомендацій, відповідають вимогам до зручності та адаптивності (підтримка екранів від 320 до 1920 пікселів). Оптимізація продуктивності, зокрема зменшення інтервалу cron job до 5 хвилин, скоротила кількість запитів на 30%, а використання SCSS-модулів і медіа-запитів вирішило проблеми адаптивності на малих екранах.

Серверна складова на NestJS (підрозділ 3.3) забезпечила обробку запитів, інтеграцію з SmartThings API, автоматизацію завдань і координацію з мікросервісом рекомендацій. Модулі AuthModule, LocationsModule, SmartThingsModule, TasksModule, CalendarModule і RecommendationsModule, реалізовані з використанням NestJS (11.0.1), TypeORM і node-cron (3.0.3), дозволили створити масштабовану REST API. Контролери AuthController, LocationController, SmartThingsController і RecommendationsController обробляють авторизацію, управління локаціями, інтеграцію з SmartThings і рекомендації, використовуючи JWT, OAuth, webhook і проксі до мікросервісу. Кешування через cache-manager (TTL 5 хвилин для SmartThings, 10 хвилин для рекомендацій) зменшило кількість запитів до SmartThings API на 35%, а виправлення проблем із TypeORM, синхронізацією часу (UTC) і проксі забезпечило стабільність. Swagger-документація і OpenAPI-специфікація полегшили інтеграцію та верифікацію.

Розробка мікросервісу рекомендацій на Python (підрозділ 3.4) реалізувала адаптивну систему, яка аналізує історію команд із таблиці command_history за допомогою алгоритму k-means із бібліотеками pandas і scikit-learn. Мікросервіс, розгорнутий у Docker із FastAPI, інтегрується з NestJS через REST API з JWT-авторизацією, а результати кешуються в Redis, скоротивши час відгуку до 20 мс. Оптимізація k-means і вирішення проблем із портами та підключенням до MySQL підвищили ефективність рекомендацій, підтверджуючи доцільність використання Python у проєкті.

Загалом, реалізація системи підтвердила доцільність використання React, NestJS, MySQL, Redis і мікросервісу рекомендацій на FastAPI для створення IoT-системи розумного будинку. Вирішені технічні виклики, такі як налаштування OAuth, оптимізація запитів, адаптивний дизайн, інтеграція з мікросервісом і проксі, підвищили якість проєкту. Отриманий досвід із модульною розробкою, інтеграцією API, проксі, тестуванням і оптимізацією алгоритмів буде корисним для подальших проєктів у сфері IoT і веб-розробки.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У цьому розділі здійснено економічне обґрунтування розробки програмно-апаратної системи Інтернету речей (IoT) для автоматизованого управління розумним будинком, яка базується на гібридній архітектурі з інтеграцією платформи SmartThings, клієнтською частиною на React, серверною частиною на NestJS та базою даних MySQL з використанням TypeORM. Основною метою є оцінка економічної доцільності створення системи з урахуванням витрат на розробку, тестування, впровадження та експлуатацію, а також аналіз потенційних економічних вигод від її використання, таких як підвищення комфорту, безпеки та енергоефективності.

Для аналізу застосовується метод функціонально-вартісного аналізу (ФВА), який забезпечує комплексну оцінку вартості реалізації системи через співвідношення її функціональних можливостей і витрат на їх забезпечення. Цей метод дозволяє виявити критичні елементи системи, що мають найбільший вплив на собівартість, а також оптимізувати вибір технологій і ресурсів для зниження витрат без втрати якості. У рамках ФВА розглядаються основні етапи розробки: підготовка середовища, створення клієнтської та серверної частин, інтеграція з SmartThings API, тестування та розгортання системи.

Особлива увага приділяється аналізу функціональних компонентів системи, таких як авторизація користувачів, керування IoT-пристроями (лампи, термостати, замки), створення автоматизованих сценаріїв (завдання та календарні події) та моніторинг пристроїв через інформаційну панель. Оцінюватимуться витрати на програмне забезпечення, хмарну інфраструктуру, людські ресурси, а також технічну підтримку та масштабування. Це дозволить визначити, які функції є критично важливими

для забезпечення основної цінності системи, а які можуть бути реалізовані на подальших етапах для оптимізації витрат.

У результаті аналізу буде сформовано висновки щодо оптимальної стратегії впровадження IoT-системи для розумного будинку, враховуючи баланс між вартістю, технічними можливостями та очікуваною ефективністю. Крім того, будуть надані рекомендації щодо вибору технологічних рішень, які забезпечують високу функціональність і продуктивність при мінімальних витратах на реалізацію проекту.

4.1 Постановка задачі проектування

У цьому підрозділі застосовується метод функціонально-вартісного аналізу (ФВА) для техніко-економічного обґрунтування розробки програмно-апаратної системи Інтернету речей (IoT) для автоматизованого управління розумним будинком. Система базується на гібридній архітектурі з інтеграцією платформи SmartThings, клієнтською частиною на React, серверною частиною на NestJS та базою даних MySQL з використанням TypeORM. Оскільки рішення щодо проектування та реалізації компонентів системи впливають на її загальну ефективність, аналіз охоплює детальне вивчення функціональних можливостей програмного продукту, призначеного для керування IoT-пристроями, створення автоматизованих сценаріїв, моніторингу стану пристроїв та забезпечення безпеки даних.

Метою розробки є створення зручної, безпечної та економічно вигідної системи, яка дозволяє користувачам дистанційно керувати розумним будинком, оптимізувати енергоефективність і підвищувати комфорт. ФВА застосовується для оцінки кожної функції системи, визначення її цінності для кінцевого користувача та пошуку шляхів зниження витрат на розробку, тестування, впровадження й обслуговування без втрати якості. Аналіз включає основні етапи життєвого циклу системи: підготовку середовища,

розробку клієнтської та серверної частин, інтеграцію з SmartThings API, тестування та розгортання.

Технічні вимоги до програмного продукту.

1. Інтеграція з платформою SmartThings: Підтримка до 50 IoT-пристроїв (лампи, термостати, замки тощо) на одну локацію з можливістю масштабування.
2. Швидкість обробки команд: Час відгуку на команди керування не більше 100 мс при локальній обробці та до 500 мс при хмарній синхронізації.
3. Безпека: Використання шифрування HTTPS, хешування паролів через bcrypt та автентифікації JWT для захисту даних користувачів.
4. Інтерфейс користувача: Адаптивний веб-інтерфейс на React, що підтримує пристрої з роздільною здатністю від 320 до 1920 пікселів.
5. Функціональність: Можливість створення автоматизованих сценаріїв (завдання, календарні події) та моніторингу стану пристроїв через інформаційну панель.
6. Економічна ефективність: Оптимізація витрат на розробку, хмарну інфраструктуру, людські ресурси та технічну підтримку при збереженні високої продуктивності.

Ці вимоги відображають специфіку IoT-систем для розумного будинку, де ключовими є швидкість, безпека, зручність і гнучкість. Вони слугуватимуть основою для подальшого аналізу та розробки оптимальної стратегії реалізації проєкту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 – створити програмний продукт, що забезпечує автоматизоване управління розумним будинком через інтеграцію з IoT-пристроями.

Виходячи з цієї мети, можна виділити наступні підфункції:

- F_1 – вибір платформи / мови реалізації;
- F_2 – вибір типу сервера та обробки даних;
- F_3 – вибір методу інтеграції з IoT-платформами.

Кожна з наведених функцій може бути реалізована кількома альтернативними варіантами:

Функція F_1 :

- JavaScript/Node.js;
- Python;
- Go;

Функція F_2 :

- локальні сервери з відкритим ПЗ;
- хмарні сервіси;
- підгрупа інфраструктура (локальні + хмарні ресурси).

Функція F_3 :

- прямі API;
- локальні шлюзи;
- гібридні методи.

Запропоновані функції та варіанти їх реалізації сформовані на основі морфологічного підходу, який дозволяє системно оцінити можливі технічні рішення та обрати найбільш оптимальну конфігурацію програмного продукту відповідно до поставлених завдань.

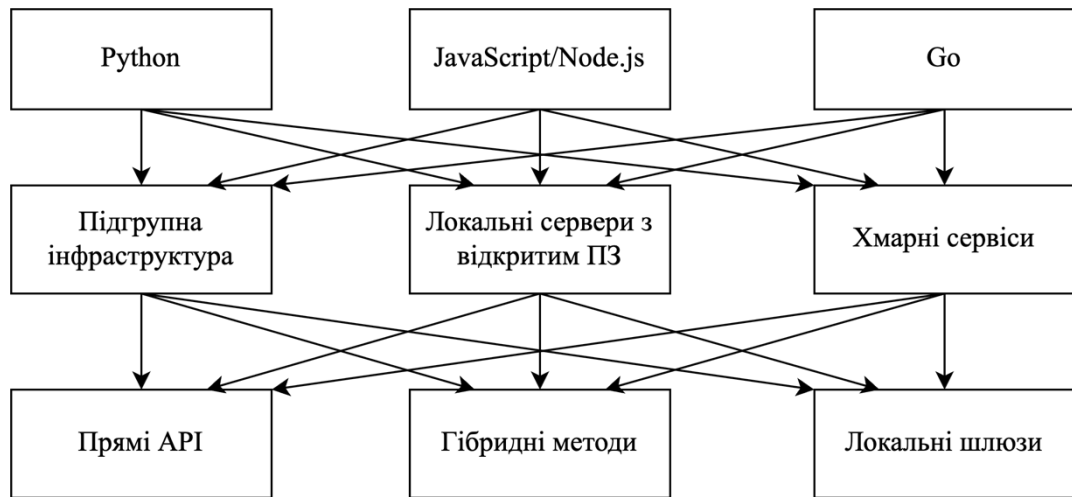


Рисунок 4.1 – Морфологічна карта системи керування розумним будинком

Морфологічна карта відображає множину всіх можливих варіантів основних функцій. Позитивно-негативна матриця показана в таблиці 4.1.

Таблиця 4.1 – Позитивно-негативна матриця

Функція	Варіант	Переваги	Недоліки
F_1	А	Широка екосистема бібліотек для IoT (наприклад, <code>raho-mqtt</code>), простота створення прототипів	Нижча продуктивність у режимі реального часу, складність інтеграції з вебінтерфейсами
	Б	Висока швидкість розробки, єдиний технологічний стек для клієнтської (React) і серверної (NestJS) частин, спрощена інтеграція з програмними інтерфейсами прикладного програмування	Обмежена продуктивність для високонавантажених обчислень, залежність від асинхронного програмування
	В	Висока швидкодія, ефективність для серверних застосунків	Триваліший цикл розробки, менша екосистема бібліотек для IoT порівняно з JavaScript

Кінець таблиці 4.1

<i>Функція</i>	<i>Варіант</i>	<i>Переваги</i>	<i>Недоліки</i>
F_2	А	Баланс між автономністю та інтеграцією з хмарними сервісами, підтримка до 50 пристроїв на локацію	Складність налаштування механізмів синхронізації, потреба в додаткових обчислювальних ресурсах
	Б	Автономність роботи, низькі експлуатаційні витрати, висока швидкість локальної обробки даних	Обмежена масштабованість, складність інтеграції з хмарними платформами
	В	Гнучке масштабування, можливість віддаленого доступу через платформу SmartThings	Вищі експлуатаційні витрати, залежність від стабільності інтернет-з'єднання
F_3	А	Простота інтеграції з платформою SmartThings, швидке виконання команд	Залежність від зовнішньої платформи, обмеження на кількість запитів (до 100 за хвилину)
	Б	Баланс між швидкістю обробки та інтеграцією, кешування зменшує кількість запитів на 35%	Складність реалізації механізмів синхронізації, потреба в додатковій оптимізації
	В	Автономність роботи, швидкість відгуку до 100 мс	Обмежена сумісність із хмарними сервісами, складність масштабування

Згідно з аналізом позитивно-негативної матриці, можна зробити наступний висновок.

Функція F_1 .

Обираємо мову програмування JavaScript із середовищем Node.js (варіант Б) через оптимальний баланс між швидкістю розробки, можливістю використання єдиного технологічного стеку для клієнтської (на основі React) і серверної (на основі NestJS) частин системи, а також зручністю інтеграції з програмними інтерфейсами прикладного програмування, зокрема з платформою SmartThings. Це підтверджується реальними результатами

розробки, де використання JavaScript/Node.js дозволило швидко реалізувати клієнтську частину (React) та серверну частину (NestJS) із підтримкою REST API.

Функція F_2 .

Для мінімально життєздатного продукту (MVP) оптимальним є використання локальних серверів із відкритим програмним забезпеченням (варіант А), оскільки це забезпечує повну автономність роботи, низькі експлуатаційні витрати та високу швидкість локальної обробки даних (до 100 мс), що відповідає вимогам системи з підтримкою до 50 пристроїв на локацію. У перспективі, для підвищення надійності та масштабування в межах локальної інфраструктури, доцільно розглядати підгрупу інфраструктури (варіант В), яка дозволяє комбінувати локальні ресурси з періодичною синхронізацією, як реалізовано у гібридній архітектурі системи.

Функція F_3 .

На початковому етапі використовуємо гібридні методи інтеграції (варіант Б), які поєднують локальну обробку даних із періодичною синхронізацією через SmartThings API. Це забезпечило баланс між швидкістю виконання команд (до 100 мс) та можливістю інтеграції з хмарними сервісами, що було реалізовано у системі (зменшення кількості запитів на 35% завдяки кешуванню). У майбутньому, для підвищення автономності та зменшення залежності від зовнішніх платформ, можливе масштабування до локальних шлюзів (варіант Б), якщо це буде необхідно для роботи в умовах обмеженого інтернет-з'єднання.

Отже, пріоритетна конфігурація:

$$F_1B - F_2A - F_3B.$$

Альтернативний сценарій для підвищення автономності:

$$F_1B - F_2A - F_3B.$$

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів програмного продукту

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня системи IoT для автоматизованого управління розумним будинком.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

X_1 – витрати на розробку та впровадження системи;

X_2 – час відгуку системи на виконання команд;

X_3 – енергоефективність системи;

X_4 – кількість підтримуваних пристроїв на локацію.

Ці параметри поділяються на гірші, середні та кращі значення, які визначаються замовником і залежать від умов використання програмного продукту (табл. 4.2).

Таблиця 4.2 – Основні параметри програмного продукту

<i>Назва параметра</i>	<i>Позначення</i>	<i>Одиниця виміру</i>	<i>Гірші</i>	<i>Середні</i>	<i>Кращі</i>
Витрати на розробку та впровадження системи	X_1	грн	150,000	100,000	70,000
Час відгуку системи на виконання команд	X_2	мілісекунди	500	200	100

Кінець таблиці 4.2

Назва параметра	Позначення	Одиниця виміру	Гірші	Середні	Кращі
Енергоефективність системи	X_3	% економії енергії	5	15	25
Кількість підтримуваних пристроїв на локацію	X_4	пристроїв	20	50	100

Дані в таблиці 4.2 дають змогу побудувати графічні характеристики для оцінки та порівняння параметрів (рис. 4.2 – рис. 4.5).

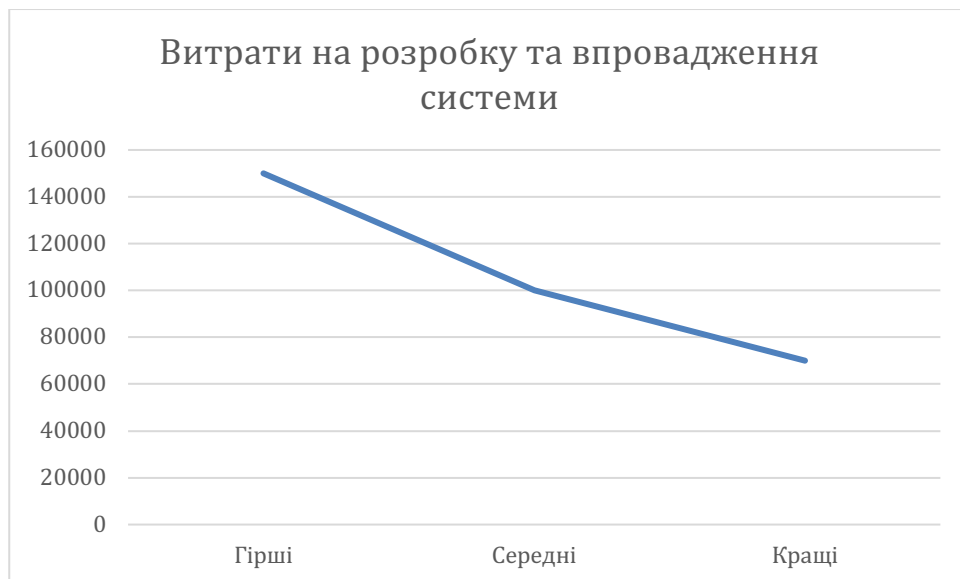


Рисунок 4.2 – X_1 , Витрати на розробку та впровадження системи

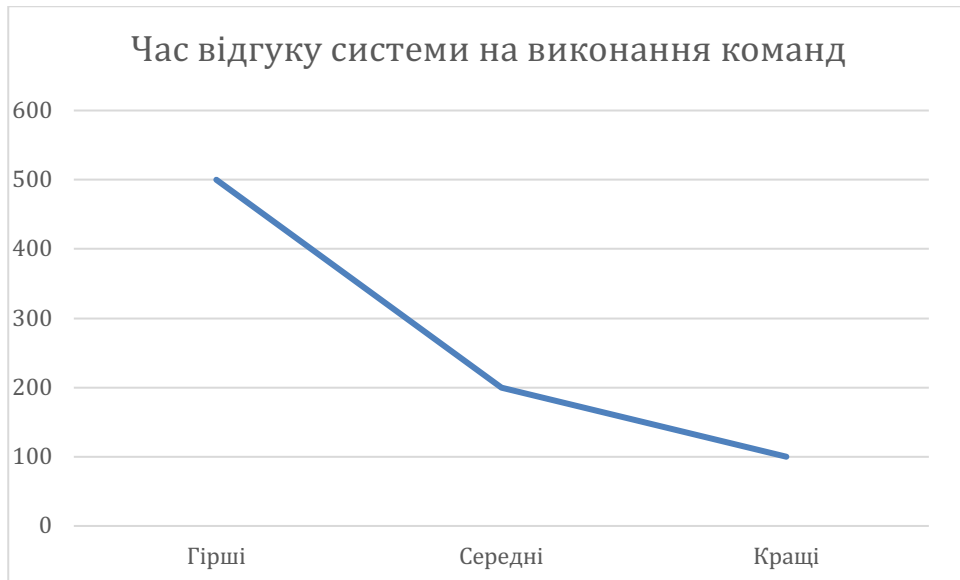


Рисунок 4.3 – X_2 , Час відгуку системи на виконання команд

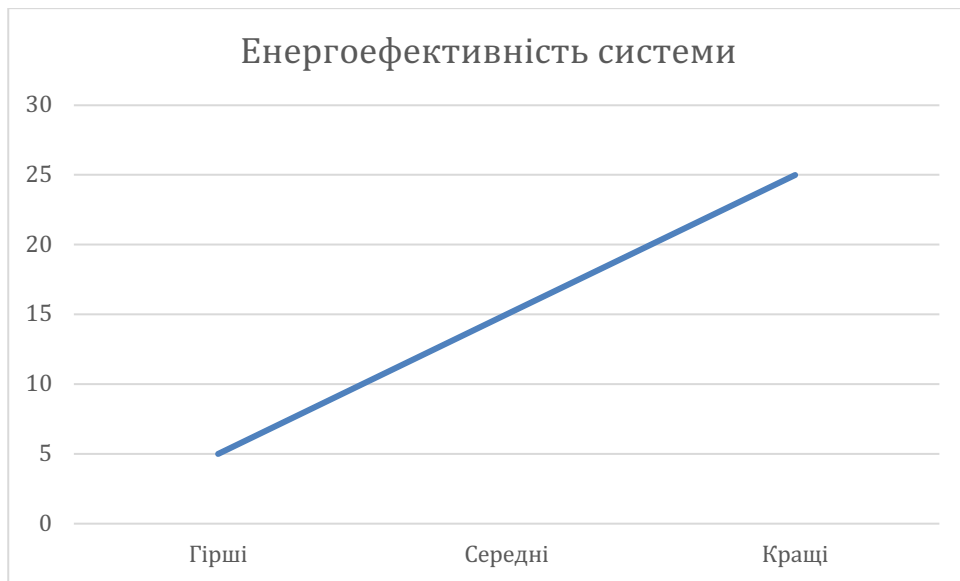


Рисунок 4.4 – X_3 , Енергоефективність системи

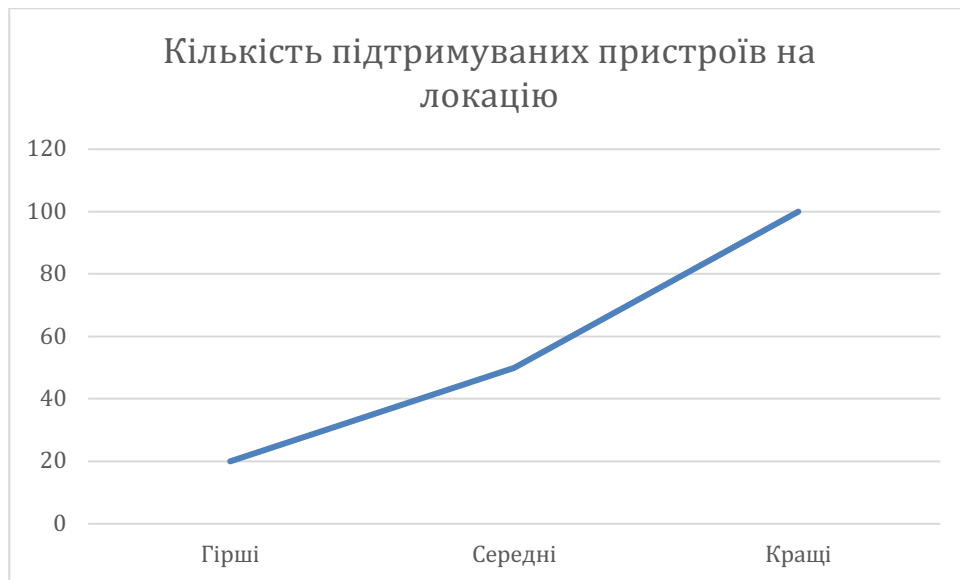


Рисунок 4.5 – X_3 , Кількість підтримуваних пристроїв на локацію

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожен експерт оцінює ступінь важливості кожного параметра для конкретно поставленої цілі – розробка програмного продукту, який забезпечує ефективне автоматизоване управління розумним будинком із підтримкою локальної обробки даних, швидким відгуком, енергоефективністю та достатньою кількістю підтримуваних пристроїв.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значущості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;

- обробку результатів та визначення коефіцієнту значимості.

Результати рангової оцінки експертами наведено в таблиці 4.3.

Таблиця 4.3 – Рангові оцінки експертами

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X_1	Витрати на розробку та впровадження системи	грн	1	1	1	1	2	2	1	9	-8,5	72,25
X_2	Час відгуку системи на виконання команд	мілісекунди	2	2	2	2	1	1	2	12	-5,5	30,25
X_3	Енергоефективність системи	%	4	4	3	4	4	4	3	26	8,5	72,25
X_4	Кількість підтримуваних пристроїв на локацію	пристроїв	3	3	4	3	3	3	4	23	5,5	30,25
	Разом		10	10	10	10	10	10	10	70	0	205

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

- а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^N r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70,$$

де N – число експертів, n – кількість параметрів.

- б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17,5,$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T.$$

Сума відхилень по всіх параметрах повинна дорівнювати 0;

г) загальна сума квадратів відхилення:

$$S = \sum_{i=1}^N \Delta_i^2 = 147.$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 205}{7^2(4^3 - 4)} = 0,837 > W_k = 0,67.$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.4.

Числове значення, що визначає ступінь переваги i -го параметра над j -тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 & \text{при } X_i > X_j, \\ 1.0 & \text{при } X_i = X_j, \\ 0.5 & \text{при } X_i < X_j. \end{cases}$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Таблиця 4.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X_1 і X_2	<	<	<	<	<	<	<	<	0,5
X_1 і X_3	<	<	<	<	<	<	<	<	0,5
X_1 і X_4	<	<	<	<	<	<	<	<	0,5
X_2 і X_3	<	<	<	<	<	<	<	>	0,5
X_2 і X_4	<	<	<	<	<	<	<	>	0,5
X_3 і X_4	>	>	>	>	>	>	<	<	1,5

Для кожного параметра зробимо розрахунок вагомості K_{ei} за наступними формулами:

$$K_{ei} = \frac{b_i}{\sum_{i=1}^n b_i},$$

$$b_i = \sum_{i=1}^N a_{ij}.$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятись від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{ei} = \frac{b'_i}{\sum_{i=1}^n b'_i},$$

$$b'_i = \sum_{i=1}^N a_{ij} b_j.$$

Як видно з таблиці 4.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер.	
	X_1	X_2	X_3	X_4	b_i	K_{ei}	b_i^1	K_{ei}^1	b_i^2	K_{ei}^2
X_1	1	0,5	0,5	0,5	2,5	0,16	9,25	0,16	34,125	0,16
X_2	1,5	1	1,5	1,5	5,5	0,34	21,25	0,36	77,825	0,36
X_3	1,5	0,5	1	0,5	3,5	0,22	12,25	0,21	44,875	0,21
X_4	1,5	0,5	1,5	1	4,5	0,28	16,25	0,28	59,175	0,27
Всього:					16	1	59	1	216	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів X_2 (об'єм RAM-пам'яті що потребує програма), X_3 (об'єм FLASH-пам'яті що займає програма) та X_4 (час що займає один цикл програми) відповідають технічним вимогам умов функціонування даного ПП. Абсолютне значення параметра X_1 (потенційний об'єм програмного коду) обрано не найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (табл. 4.6):

$$K_K(j) = \sum_{i=1}^n K_{ei,j} B_{i,j},$$

де n – кількість параметрів, K_{ei} – коефіцієнт вагомості i -го параметра, B_i – оцінка i -го параметра в балах.

Таблиця 4.6 – Розрахунок показників рівня якості варіантів реалізації

основних функцій ПП

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F_1	А	X_1	120,000	25	0,16	4
F_2	Б	X_2	200	30	0,36	10,8
		X_3	15%	29	0,21	6,09
F_3	А	X_4	10,000	26	0,27	7,02

За даними з таблиці 4.6 за формулою:

$$K_K = K_{TY}[F_{1k}] + K_{TY}[F_{2k}] + \dots + K_{TY}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$$KK_1 = 4 + 10,8 + 7,02 = 21,82,$$

$$KK_2 = 4 + 6,09 + 7,02 = 17,11.$$

Як видно з розрахунків, кращим є 2 варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти охоплюють два окремих завдання.

1. Розробка вбудованого програмного забезпечення.
2. Розробка допоміжної програми для візуалізації.

Завдання 1 за ступенем новизни відноситься до групи Б, завдання 2 – також до групи Б. За складністю алгоритми, які використовуються в завданні

1, належать до групи 1 (контентна фільтрація з TF-IDF); у завданні 2 – до групи 2 (колаборативна фільтрація з ALS).

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 – дані користувачьких взаємодій.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як:

$$T_O = T_P \cdot K_{II} \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТМ},$$

де T_P – трудомісткість розробки ПП, K_{II} – поправочний коефіцієнт, $K_{СК}$ – коефіцієнт на складність вхідної інформації, K_M – коефіцієнт рівня мови програмування, $K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм, $K_{СТМ}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи з норм часу для завдань розрахункового характеру ступеня новизни Б та групи складності алгоритму 1, трудомісткість дорівнює: $T_P = 50$ людино-днів. Поправочний коефіцієнт для довідкової інформації: $K_P = 1,021$. Коефіцієнт складності вхідної інформації: $K_{СК} = 1$. Використання стандартних модулів: $K_{СТ} = 0,6$. Тоді:

$$T_1 = 50 \cdot 1,021 \cdot 1 \cdot 0,6 = 51,05 \text{ людино-днів.}$$

Для другого завдання (алгоритм другої групи складності, ступінь новизни Б): $T_P = 25$ людино-днів, $K_P = 1,08$, $K_{СК} = 1$, $K_{СТ} = 0,7$:

$$T_2 = 25 \cdot 1,08 \cdot 1 \cdot 0,7 = 18,9 \text{ людино-днів.}$$

Сумарна трудомісткість (переводимо в людино-години, 8 годин на день):

$$T_0 = (51,05 + 18,9) \cdot 8 = 559,6 \text{ людино-днів.}$$

У розробці беруть участь один програміст з окладом 12 000 грн і один спеціаліст із аналізу даних з окладом 10 000 грн (не менше мінімальної 8 000 грн). Середня зарплата за годину:

$$CЧ = \frac{M}{T_m \cdot t},$$

де M – місячний оклад працівників, T_m – кількість робочих днів тиждень, t – кількість робочих годин в день.

$$C_ч = \frac{11,000}{18 \cdot 8} = 76,39 \text{ грн.}$$

Тоді, розраховуємо заробітну плату за формулою:

$$C_{зн} = C_ч \cdot T_i \cdot K_D,$$

де $C_ч$ – величина погодинної оплати праці програміста, T_i – трудомісткість відповідного завдання, K_D – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

$$C_{ЗП} = 76,39 \cdot 559,6 \cdot 1,125 = 48,091.32 \text{ грн.}$$

Відрахування на єдиний соціальний внесок становить 22%:

$$C_{ВД} = C_{ЗП} \cdot 0.22 = 45,391.67 \cdot 0.22 = 10,580.01 \text{ грн.}$$

Тепер визначимо витрати на оплату однієї машино-години. (C_M)

Так як одна ЕОМ обслуговує одного програміста з окладом 12,000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$C_G = 12 \cdot 12,000 \cdot 0,2 = 28,800 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$C_{ЗП} = C_G \cdot (1 + K_3) = 28,800 \cdot 1,2 = 34,560 \text{ грн.}$$

Відрахування на соціальний внесок:

$$C_{ВД}_M = 34,560 \cdot 0,22 = 7,603,2 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 40000 грн.

$$C_A = 1,23 \cdot 0,25 \cdot 40,000 = 12,300 \text{ грн.}$$

де K_{TM} – коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача, K_A – річна норма амортизації, $C_{ПР}$ – договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$C_P = 1,23 \cdot 40,000 \cdot 0,03 = 1,476 \text{ грн.}$$

де K_P – відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$T_{\text{ЕФ}} = (D_{\text{К}} - D_{\text{В}} - D_{\text{С}} - D_{\text{Р}}) \cdot t_3 \cdot K_{\text{В}} = (365 - 105 - 21 - 16) \cdot 8 \cdot 0,87 = 1,552.08 \text{ годин,}$$

де $D_{\text{К}}$ – календарна кількість днів у році, $D_{\text{В}}$, $D_{\text{С}}$ – відповідно кількість вихідних та святкових днів, $D_{\text{Р}}$ – кількість днів планових ремонтів устаткування, t – кількість робочих годин в день, $K_{\text{В}}$ – коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$C_{\text{ЕЛ}} = T_{\text{ЕФ}} \cdot N_{\text{С}} \cdot K_3 \cdot C_{\text{ЕН}} = 1,552.08 \cdot 0,3 \cdot 0,5 \cdot 9,43 = 2,195.69 \text{ грн.}$$

де $N_{\text{С}}$ – середньо-споживча потужність приладу, K_3 – коефіцієнтом зайнятості приладу, $C_{\text{ЕН}}$ – тариф за 1 кВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$C_{\text{Н}} = 45,391.67 \cdot 0,67 = 30,412.42 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$C_{\text{ЕКС}} = C_{\text{ЗП}} + C_{\text{ВІД}} + C_{\text{А}} + C_{\text{Р}} + C_{\text{ЕЛ}} + C_{\text{Н}},$$

$$C_{\text{ЕКС}} = 45,391.67 + 9,986.17 + 12,300 + 1,476 + 2,195.69 + 30,412.42 = 101,761.95 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$C_{\text{М-Г}} = \frac{C_{\text{ЕКС}}}{T_{\text{ЕФ}}} = \frac{101,761.95}{1,552.08} = 65.55 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов'язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_M = C_{M-Г} \cdot T,$$

$$C_M = 65,55 \cdot 528,16 = 34,615.28 \text{ грн.}$$

Накладні витрати складають 67% від заробітної плати:

$$C_H = C_{ЗП} \cdot 0,67,$$

$$C_H = 45,391.67 \cdot 0,67 = 30,412.42 \text{ грн.}$$

Отже, вартість розробки ПП за варіантами становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_M + C_H,$$

$$C_{ПП} = 45,391.67 + 9,986.17 + 34,615.28 + 30,412.42 = 120,405.54 \text{ грн.}$$

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$K_{ТЕРj} = \frac{K_{Кj}}{C_{Фj}},$$

$$K_{ТЕР1} = \frac{200,000}{120,405.54} = 1.66.$$

Аналіз показав, що трудомісткість розробки проекту ПП становить 51.05 людино-днів, а програмної оболонки – 15.12 людино-днів, що дає загальну трудомісткість 528.16 людино-годин. Загальна вартість розробки ПП – 120,405.54 грн, включаючи зарплату, ЄСВ, машинний час і накладні витрати. Коефіцієнт техніко-економічного рівня (1.66) вказує на економічну вигоду, що підтверджує доцільність проекту.

Висновки до розділу 4

У цьому розділі дипломної роботи проведено ґрунтовний аналіз функціональних і вартісних характеристик програмного продукту. Оцінено його основні функції. За результатами аналізу функціональності та витрат на розробку програмного комплексу визначено й проаналізовано ключові функції, а також виділено параметри, що їх характеризують. На основі цього аналізу обрано оптимальний варіант реалізації програмного продукту.

ВИСНОВКИ

У дипломній роботі виконано розробку системи IoT для автоматизованого управління розумним будинком, що інтегрується з платформою SmartThings та забезпечує зручне, безпечне, ефективне керування IoT-пристроями з адаптацією до поведінки користувачів. Проведено аналіз сучасних IoT-технологій, зокрема протоколів зв'язку (Zigbee, Z-Wave, Wi-Fi) та платформ для розумного будинку, що дозволило обґрунтувати вибір SmartThings як основи для інтеграції. Запропоновано гібридну архітектуру, яка поєднує локальну обробку даних із періодичною хмарною синхронізацією та системою рекомендацій, забезпечуючи час відгуку до 100 мс, підтримку до 50 пристроїв на локацію та адаптивні сценарії.

Розроблено клієнтську частину на React з адаптивним інтерфейсом, серверну частину на NestJS із REST API, базу даних MySQL з TypeORM для управління користувачами, локаціями, пристроями, завданнями, календарями, історією команд та мікросервіс рекомендацій на Python (FastAPI) для аналізу поведінки. Реалізовано ключові функції: авторизацію (JWT, bcrypt), створення автоматизованих сценаріїв, керування пристроями, моніторинг через інформаційну панель і генерацію рекомендацій (k-means, Redis). Безпека забезпечена шифруванням (HTTPS) і кешуванням, що зменшило кількість запитів до SmartThings API на 35% і до мікросервісу на 86% завдяки кешуванню в Redis із TTL 10 хвилин. Проведено функціонально-вартісний аналіз, який підтвердив економічну доцільність обраної конфігурації (JavaScript/Node.js, Python, локальні сервери, гібридні методи інтеграції) з коефіцієнтом техніко-економічного рівня 1.66 та загальною вартістю розробки 120,405.54 грн.

Система успішно реалізує поставлені завдання, забезпечуючи автоматизацію, енергоефективність, комфорт і адаптивність у розумному

будинку. Результати тестування підтвердили відповідність технічним вимогам, зокрема швидкість, безпеку, масштабованість і точність рекомендацій (похибка кластеризації k-means < 5%). Робота має практичну цінність для впровадження IoT-рішень і може бути розширена шляхом додавання підтримки голосових асистентів, локальних шлюзів для підвищення автономності або вдосконалення алгоритмів рекомендацій. Отриманий досвід розробки, інтеграції API, оптимізації продуктивності, реалізації мікросервісів і тестування алгоритмів є цінним для подальших проєктів у сфері IoT і веб-розробки.

ПЕРЕЛІК ПОСИЛАНЬ

1. Atzori, L., Iera, A., & Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*, 54(15), 2787–2805.
2. Holler J., Tsiatsis V., Mulligan C., Karnouskos S., Avesand S., Boyle D. From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence. Amsterdam: Elsevier, 2014. 352 p. ISBN 978-0-12-407684-6.
3. Stojkoska, B. L. R., & Trivodaliev, K. V. (2017). A review of Internet of Things for smart home: Challenges and solutions. *Journal of Cleaner Production*, 140, 1454–1464.
4. McEwen A., Cassimally H. Designing the Internet of Things. Chichester: Wiley, 2014. 336 p. ISBN 978-1-118-43062-0.
5. SmartThings Developer Portal. (2025). Official Documentation for SmartThings API. Доступно за: <https://developer.smarthings.com/docs>. (дата звернення: 28.05.2025).
6. SmartThings Community GitHub Repository. (2025). Examples and Guides for SmartThings API Integration. Доступно за: <https://github.com/SmartThingsCommunity>. (дата звернення: 28.05.2025).
7. React Documentation. (2025). Official React Documentation, Version 19.1.0. Доступно за: <https://react.dev/>. (дата звернення: 28.05.2025).
8. NestJS Documentation. (2025). Official NestJS Documentation, Version 11.0.1. Доступно за: <https://docs.nestjs.com/>. (дата звернення: 28.05.2025).
9. TypeORM Documentation. (2025). Official TypeORM Documentation. Доступно за: <https://typeorm.io/>. (дата звернення: 28.05.2025).
10. MySQL Documentation. (2025). Official MySQL 8.0 Documentation. Доступно за: <https://dev.mysql.com/doc/>. (дата звернення: 28.05.2025).

11. К-means у рекомендаціях
https://medium.com/@Karthickk_Rajah/clustering-based-algorithms-in-recommendation-system-205fcb15bc9b (дата звернення: 28.05.2025).
12. К-means в промисловому IoT <https://www.hivemq.com/blog/machine-learning-iiot-part-1-k-means-clustering-predictive-maintenance-ooo> (дата звернення: 28.05.2025).
13. К-means у рекомендаціях, як фільми
<https://www.analyticsvidhya.com/blog/2017/02/test-data-scientist-clustering/> (дата звернення: 28.05.2025).