НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

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

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Створення інтелектуального агента на базі Raspberry Pi 5 AI »

Спеціальність F7 «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Гарант освітньої програми канд. фіз.-мат. наук, доцент	_____ (підпис)	**Євгеній НІКІТЕНКО**
Керівник бакалаврської кваліфікаційної роботи старший викладач	_____ (підпис)	**Володимир МАТІЄВСЬКИЙ**
Виконав	_____ (підпис)	**Михайло КРУТІЙ**

КИЇВ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

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

З А В Д А Н Н Я

ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Крутію Михайлу Сергійовичу
(прізвище, ім'я, по батькові)

Спеціальність «Комп'ютерна інженерія »

Освітньо-професійна програма «Комп'ютерна інженерія »

Тема кваліфікаційної бакалаврської роботи
«Створення інтелектуального агента на базі Raspberry Pi 5 AI »

затверджена наказом ректора НУБіП України від «10»___12_____2025 р. №___3072 «С»___
Термін подання завершеної роботи на кафедру «15»___05_____2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи
Специфікації пристрою та API Tuya IoT, Yasno

Перелік питань, що підлягають дослідженню:
Аналіз предметної галузі щодо створення IoT, визначення можливостей розгортання AI
Проєктування інтелектуального агенту, тестування, можливості покращення інтелектуального агенту
Перелік графічного матеріалу (за необхідності).

Дата видачі завдання "__10__" _____12_____2025 р.

Керівник бакалаврської кваліфікаційної роботи старший викладач	_____ (підпис)	**Володимир МАТІЄВСЬКИЙ**
Завдання взяв до виконання	_____ (підпис)	**Михайло КРУТІЙ**

РЕФЕРАТ

Пояснювальна записка: 70 сторінок, 20 рисунків, 6 таблиць, 3 додатки, 36 джерел.

РОЗУМНИЙ БУДИНОК, IoT, TUYA API, YASNO API, TELEGRAM-БОТ, SPRING BOOT, JAVA 21, ІНТЕРНЕТ РЕЧЕЙ, ПРОАКТИВНИЙ АСИСТЕНТ, ГРАФІК ВІДКЛЮЧЕНЬ ЕЛЕКТРОЕНЕРГІЇ, REST API, POSTGRESQL.

Об'єктом дослідження є процеси автоматизованого управління пристроями розумного будинку з урахуванням графіків відключень електроенергії та аналізу споживання.

Метою роботи є проєктування та реалізація інтелектуального серверного асистента, який інтегрує платформу IoT Tuya, сервіс планових відключень Yasno та Telegram-інтерфейс для надання проактивних рекомендацій користувачеві.

У першому розділі проведено аналіз предметної галузі: розглянуто концепцію розумного будинку, наявні рішення автоматизації, протоколи IoT та підходи до реалізації AI-асистентів.

Другий розділ присвячено проєктуванню та розробці системи: описано архітектуру бекенду на базі Spring Boot 4 / Java 21, інтеграцію з Tuya API та Yasno API, схему бази даних PostgreSQL та механізм проактивних сценаріїв.

У третьому розділі виконано тестування розробленої системи: модульне тестування сервісів, інтеграційне тестування REST-контролерів, а також тестування ключових сценаріїв роботи асистента.

Четвертий розділ присвячено обговоренню та реалізації можливих покращень інтелектуального агент від підвищення безпеки із допомогою модерації прив’язки облікових записів Tuya, до додаткових режимів роботи з керування ДЖБ, та розширених правил автоматизації.

В результаті виконання роботи розроблено повнофункціональний бекенд-сервіс, що забезпечує моніторинг стану IoT-пристроїв, отримання та зберігання графіків відключень, а також генерацію проактивних сповіщень для користувача через Telegram.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface **–** інтерфейс програмування застосунків;

DDL – Data Definition Language – мова опису структури даних;

DML – Data Manipulation Language – мова маніпулювання даними;

DTO – Data Transfer Object – об'єкт передачі даних;

HTTP – Hypertext Transfer Protocol **–** протокол передачі гіпертексту;

IoT – Internet of Things **–** Інтернет речей;

JSON – JavaScript Object Notation **–** текстовий формат обміну даними;

JWT – JSON Web Token **–** стандарт токенів авторизації;

LLM – Large Language Model – велика мовна модель;

NLP – Natural Language Processing – обробка природної мови;

ORM – Object-Relational Mapping – об'єктно-реляційне відображення;

REST – Representational State Transfer **–** архітектурний стиль передачі стану;

SHA – Secure Hash Algorithm – алгоритм безпечного хешування;

SQL – Structured Query Language **–** мова структурованих запитів;

UML – Unified Modeling Language – уніфікована мова моделювання;

БД – База даних;

БР – Бакалаврська робота;

ПЗ – Пояснювальна записка;

ПК – Персональний комп'ютер;

ВСТУП

Сучасний стан електроенергетики України характеризується значною нестабільністю, зумовленою воєнними діями та ушкодженням критичної інфраструктури. В умовах планових та позапланових відключень електроенергії питання ефективного управління побутовими електроприладами набуває особливої актуальності. Водночас стрімкий розвиток технологій Інтернету речей (IoT) відкриває нові можливості для автоматизації домашнього середовища та інтелектуального управління пристроями.

Розумний будинок – це система, що інтегрує різноманітні побутові пристрої та сенсори в єдину керовану мережу. Серед провідних платформ IoT для кінцевих споживачів виділяється Tuya Smart – хмарна екосистема, що об'єднує мільйони «розумних» розеток, вимикачів, датчиків та інших пристроїв. Ключовою перевагою Tuya є відкритий API, що дозволяє стороннім розробникам будувати власні сервіси управління.

Проте більшість наявних рішень для розумного будинку є суто реактивними: система виконує лише явні команди користувача, не аналізуючи контекст та не пропонуючи оптимальних дій самостійно. Особливо помітним є відсутність інтеграції між управлінням пристроями та графіками відключень електроенергії, які надаються операторами, зокрема компанією Yasno.

Актуальність роботи обумовлена потребою у розробці інтелектуального асистента, що здатний не лише виконувати команди, але й проактивно аналізувати ситуацію: попереджати про наближення відключення, рекомендувати увімкнення пристроїв (бойлера, зарядних станцій тощо) заздалегідь, а також автоматично реагувати на відновлення живлення.

Об'єктом дослідження є процеси автоматизованого управління пристроями розумного будинку з урахуванням зовнішніх подій – планових відключень електроенергії.

Предметом дослідження є методи та засоби розробки серверного застосунку, що інтегрує Tuya IoT API, Yasno API та Telegram для реалізації проактивного управління.

Метою роботи є проєктування та реалізація бекенд-сервісу для інтелектуального асистента розумного будинку, що забезпечує моніторинг стану IoT-пристроїв, отримання та аналіз графіків відключень, а також надання проактивних рекомендацій через Telegram-інтерфейс.

Для досягнення мети визначено наступні завдання:

– проаналізувати існуючі рішення для автоматизації розумного будинку та методи інтеграції IoT-платформ;

– дослідити Tuya IoT API та Yasno API, визначити їх можливості та обмеження;

– спроєктувати архітектуру серверного застосунку з урахуванням вимог до масштабованості та надійності;

– розробити модуль інтеграції з Tuya API для отримання та управління станом пристроїв;

– розробити модуль отримання, нормалізації та зберігання графіків відключень від Yasno;

– реалізувати механізм проактивних сценаріїв та Telegram-сповіщень;

– провести тестування розробленої системи та оцінити її функціональність.

Практична цінність роботи полягає в тому, що розроблений сервіс може бути безпосередньо використаний власниками розумних будинків з пристроями на платформі Tuya та підключеними адресами до зони обслуговування Yasno для отримання персоналізованих сповіщень та автоматичного управління побутовими приладами.

Методи дослідження: аналіз науково-технічної літератури та документації API, об'єктно-орієнтоване проєктування, мікро сервісна архітектура, методи автоматизованого тестування програмного забезпечення.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Концепція розумного будинку та Інтернету речей та огляд існуючих рішень та платформ автоматизації

Концепція розумного будинку (Smart Home) зародилася ще в 1970-х роках, проте її масове впровадження стало можливим лише із появою дешевого бездротового зв'язку та хмарних обчислень. На сьогодні розумний будинок є невід'ємною частиною ширшої концепції Інтернету речей – парадигми, за якої фізичні об'єкти отримують унікальний ідентифікатор та здатність передавати дані через мережу без участі людини.

У контексті цієї роботи IoT розглядається як сукупність фізичних пристроїв, що мають мережеву ідентичність, передають телеметрію та можуть отримувати команди керування через хмарну або локальну інфраструктуру. Для побутового сценарію це означає, що розетка, реле або зарядна станція стають джерелом даних і виконавчим механізмом для програмного асистента.

Типова архітектура IoT-системи для розумного будинку складається з чотирьох рівнів. Перший – рівень сприйняття (Perception Layer) – охоплює фізичні пристрої: «розумні» розетки, вимикачі, датчики температури, руху, диму тощо. Ці пристрої виконують безпосереднє вимірювання параметрів середовища та/або виконання фізичних дій.

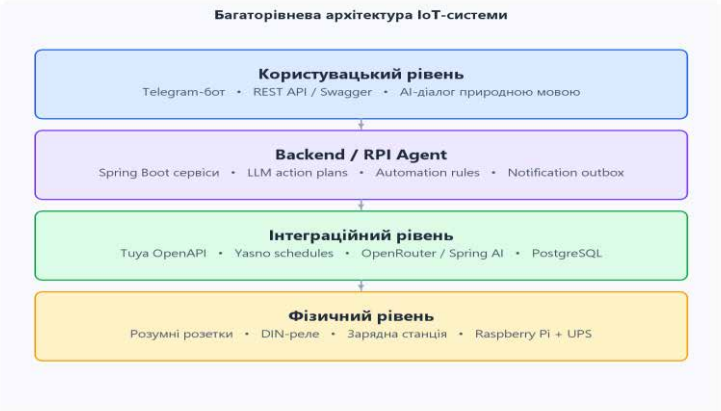

Рисунок 1.1 – Багаторівнева архітектура IoT-системи

Другий рівень – мережевий (Network Layer) – забезпечує передачу даних від пристроїв до хмарної платформи. Серед найпоширеніших протоколів: Wi-Fi (IEEE 802.11), Bluetooth Low Energy (BLE), Zigbee (IEEE 802.15.4), Z-Wave та

Thread. Wi-Fi є найбільш поширеним у побутових смарт-розетках завдяки широкому покриттю та простоті інтеграції з домашніми роутерами.

Третій рівень – платформний (Platform Layer) – включає хмарні сервери, що агрегують дані від пристроїв, зберігають їх та надають API для управління. Саме на цьому рівні здійснюється основна бізнес-логіка системи, автентифікація пристроїв та авторизація користувачів.

Четвертий рівень – прикладний (Application Layer) – надає інтерфейс для кінцевого користувача: мобільні застосунки, веб-інтерфейси або голосові асистенти. Саме на цьому рівні реалізується концепція проактивного асистента, що є предметом даної роботи.

Серед ключових характеристик, що відрізняють сучасні IoT-системи для розумного будинку від традиційних систем автоматизації, виділяються: масштабованість (здатність підключати нові пристрої без переналаштування всієї системи), інтероперабельність (взаємодія пристроїв різних виробників), безпека (шифрування даних та аутентифікація пристроїв) та енергоефективність.

Платформа Tuya Smart є однією з провідних IoT-платформ для розумного будинку у світі. Для даної роботи ключовою перевагою Tuya є відкритий набір Cloud API, що дозволяє будувати власні застосунки поверх хмарної інфраструктури та керувати пристроями без написання виробником окремої інтеграції для кожної моделі [3].

Для України особливої актуальності набуває питання управління електроспоживанням в умовах нестабільного енергопостачання. Yasno публікує графіки планових відключень у відкритому доступі, що відкриває можливість інтеграції цих даних у системи управління розумним будинком [4].

Таким чином, поєднання IoT-платформи Tuya, API графіків відключень Yasno та інтелектуального асистента утворює актуальну технічну задачу, вирішення якої дозволить суттєво підвищити комфорт та енергоефективність для сотень тисяч українських домогосподарств.

На ринку систем автоматизації розумного будинку присутня велика кількість рішень різного рівня – від простих пультів дистанційного керування до комплексних хмарних платформ зі штучним інтелектом. Для обґрунтування актуальності розробки власного рішення необхідно провести систематичний аналіз наявних альтернатив.

Google Home є однією з найпоширеніших платформ управління розумним будинком. Вона надає голосовий інтерфейс через асистента Google Assistant,

підтримує велику кількість пристроїв стандарту Matter, а також дозволяє створювати сценарії автоматизації. Проте Google Home не має вбудованої інтеграції з графіками відключень електроенергії конкретних регіональних операторів і не вирішує спеціалізований сценарій локального керування Tuya-пристроями під час відключень.

Amazon Alexa – конкурентна платформа від Amazon, що надає схожий функціонал. Alexa підтримує тисячі «skills» (навичок) від сторонніх розробників, однак розробка власного skill вимагає значних зусиль та обмежена правилами магазину Amazon Skills. Крім того, для користувачів з України доступ до повного функціоналу Alexa є обмеженим через регіональні обмеження.

Apple HomeKit є закритою екосистемою Apple, що дозволяє управляти сертифікованими пристроями. Перевагами є висока безпека та глибока інтеграція з пристроями Apple. Недоліки: обмежений список сумісних пристроїв (більшість Tuya-пристроїв потребують додаткового шлюзу), відсутність інтеграції з регіональними операторами.

Home Assistant – відкрита платформа автоматизації з широким функціоналом. Вона підтримує багато інтеграцій, включно з Tuya, а завдяки активній спільноті існують окремі компоненти для відображення графіків відключень. Проте налаштування Home Assistant вимагає значних технічних знань, а стандартний інтерфейс – веб-застосунок або мобільний додаток – не є настільки прямим каналом комунікації, як Telegram-чат.

Tuya Smart / Smart Life – рідні застосунки для управління Tuya-пристроями. Надають повний контроль над пристроями, базову автоматизацію за розкладом та сценаріями. Проте вони не мають інтеграції з Yasno та не підтримують проактивні сповіщення на основі зовнішніх подій (відключень).

Порівняльний аналіз вищезазначених рішень у контексті вимог даної роботи наведено у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз існуючих рішень

Рішення	**Tuya API**	**Yasno інтеграція**	**Telegram інтерфейс**	**Проактивні сценарії**
Google Home	-	-	-	Частково
Amazon Alexa	-	-	-	Частково
Apple HomeKit	-	-	-	-
Home Assistant	✓	-	-	✓
Tuya Smart	✓	-	-	-
Розроблена система	✓	✓	✓	✓

Як видно з таблиці 1.1, жодне з аналізованих рішень не поєднує одночасно всі чотири ключові характеристики: інтеграцію з Tuya API, отримання даних від Yasno, Telegram-інтерфейс та проактивні сценарії. Це підтверджує доцільність розробки власного рішення.

1.2 Аналіз протоколів та API для керування IoT-пристроями

Вибір протоколів для взаємодії між компонентами системи є одним з ключових архітектурних рішень. Розглянемо основні варіанти, що використовуються в IoT-системах для розумного будинку.

MQTT (Message Queuing Telemetry Transport) – легкий протокол обміну повідомленнями, що працює за принципом «публікація/підписка» (publish/subscribe). Він добре підходить для локальних IoT-сценаріїв з власним брокером, але для даної роботи менш зручний, оскільки Tuya Cloud API та Yasno API вже надають HTTP/REST-інтерфейси, а розгортання окремого MQTT-брокера збільшило б операційну складність прототипу.

WebSocket – двонаправлений протокол повного дуплексу поверх TCP. Забезпечує отримання даних у реальному часі без постійних HTTP-запитів. Tuya використовує WebSocket для надсилання сповіщень про зміну стану пристроїв у реальному часі.

HTTP/REST – класичний веб-протокол, що використовується для більшості сучасних публічних API, включно з Tuya Open API та Yasno API. RESTful-архітектура спирається на ресурсоорієнтовану модель взаємодії та добре поєднується з OpenAPI-описом контрактів [24; 25]. Tuya Open API використовує HTTPS (HTTP over TLS) з підписом запитів за алгоритмом HMAC-SHA256 [3; 5; 6].

Для цілей даної роботи обрано REST/HTTP як основний протокол взаємодії з зовнішніми API (Tuya та Yasno), оскільки обидва сервіси надають саме REST API. Для отримання сповіщень про зміну стану пристроїв у реальному часі Tuya підтримує механізм Webhook або WebSocket-підписку.

Tuya Open API надає такі ключові можливості [3]:

– отримання списку пристроїв, прив'язаних до облікового запису;

– читання поточного стану пристрою (статус кожної функції – «dp», data point);

– надсилання команд управління (наприклад, вмикання/вимикання розетки);

– отримання журналу споживання електроенергії (kWh) для пристроїв з вимірювачем;

– синхронізація даних облікового запису та пристроїв.

Автентифікація в Tuya Open API реалізована через тимчасовий `access_token` і підпис запитів HMAC-SHA256. Для отримання токена клієнт підписує запит за `client_id` і `timestamp`, а для наступних операцій додає `access_token` до рядка підпису. У реалізації rpi-agent токен кешується в TuyaTokenManager і проактивно оновлюється перед завершенням строку дії [3; 5; 6].

Yasno API є відносно новим публічним API, що надає доступ до даних про плановані відключення електроенергії для адрес у зоні обслуговування оператора Yasno. API дозволяє: знаходити адреси (регіон, місто, вулиця, будинок), отримувати groupKey – ідентифікатор групи відключень, отримувати розклад відключень для групи [4].

Розклад представлений у вигляді JSON-масиву слотів (часових інтервалів), кожен з яких має поля: дата, час початку, час кінця та тип (планове або можливе відключення). Нормалізація та зберігання цих даних є важливою частиною розробленої системи, що детально описана в розділі 2.

Рисунок 1.2 – Розумна розетка Tuya (категорія cz) та DIN-реле зі змішаної категорії tdq

1.3 Підходи до реалізації AI-асистентів для розумного будинку

AI-асистент (штучний інтелект-асистент) для розумного будинку – це програмний компонент, що здатний розуміти природну мову, аналізувати контекст та виконувати дії або надавати рекомендації. Розрізняють два основні підходи до реалізації таких асистентів.

Перший підхід базується на великих мовних моделях (Large Language Models, LLM). Сучасні LLM демонструють значні можливості розуміння природної мови та генерації відповідей, однак інтеграція LLM із системою керування пристроями потребує окремого шару безпеки через ризик галюцинацій, затримку відповіді та залежність від зовнішнього провайдера [26; 27; 28; 33, с. 2].

Другий підхід – на основі правил та детермінованої логіки – є більш передбачуваним та дешевим у використанні. Асистент аналізує структуровані дані (стан пристроїв, розклад відключень, час доби) та на основі заздалегідь визначених правил генерує сповіщення та рекомендації. Цей підхід є більш надійним для критичних сценаріїв (наприклад, попередження про відключення) та не залежить від зовнішніх сервісів.

У даній роботі обрано другий підхід – детерміновану логіку – для реалізації проактивних сценаріїв, що зумовлено такими факторами: передбачуваністю поведінки системи, відсутністю додаткових витрат на API LLM, низькою затримкою та можливістю легкого тестування логіки.

Telegram Bot API обраний як канал взаємодії з користувачем з таких

причин: Bot API добре задокументований, ботів легко розгортати без спеціального хостингу для frontend, а повідомлення у Telegram природно підходять для коротких команд, підтверджень дій і термінових сповіщень про відключення. В офіційній документації прямо зазначено: «The Bot API is an HTTP-based interface created for developers keen on building bots for Telegram» [13].

Проактивна модель асистента передбачає, що система сама ініціює повідомлення до користувача без явного запиту з його боку, якщо виявляє ситуацію, що вимагає уваги. Наприклад: за годину до відключення система надсилає повідомлення про наближення відключення та пропонує увімкнути бойлер або зарядну станцію, якщо аналіз статусу пристроїв показує, що вони наразі вимкнені.

1.4 Постановка завдання та вимоги до системи

На основі проведеного аналізу сформульовано основні вимоги до розроблюваної системи. Функціональні вимоги визначають, що система повинна виконувати, нефункціональні – як вона повинна це виконувати.

Функціональні вимоги до системи:

– реєстрація та управління обліковими записами Tuya користувачів;

– отримання та відображення переліку прив'язаних IoT-пристроїв з їх поточним статусом;

– надсилання команд управління пристроями (вмикання/вимикання);

– реєстрація адрес користувачів та прив'язка до групи Yasno;

– автоматичне отримання та зберігання графіків відключень;

– генерація проактивних сповіщень про наближення відключення з рекомендаціями щодо включення пріоритетних пристроїв;

– сповіщення про відновлення електроенергії після відключення;

– надання аналітики споживання по пристроях за день/тиждень/місяць;

– збереження налаштувань користувача (пріоритет пристроїв, мова сповіщень, часовий пояс).

Нефункціональні вимоги:

– доступність системи не менше 99 % часу (uptime);

– час відповіді REST API не більше 500 мс для 95-го перцентиля;

– захист персональних даних відповідно до вимог GDPR;

– підтримка одночасної роботи не менше 100 активних користувачів;

– автоматичне відновлення після збоїв зовнішніх API (retry-логіка);

– покриття коду автоматизованими тестами не менше 70 %;

– задокументованість API у форматі OpenAPI 3.0 / 3.1, що спрощує ручну перевірку через Swagger UI та автоматизовані API-сценарії [25].

Для реалізації поставлених вимог обрано такий технологічний стек: Java 21 LTS як основна мова розробки; Spring Boot 4 (Spring Framework 7.x) як фреймворк для REST API, планувальників та DI; PostgreSQL 17 як реляційна СКБД; Flyway для версіонування схеми; MapStruct для маппінгу; Spring Data JPA для репозиторіїв; Docker Compose для відтворюваних середовищ; Hurl, JUnit 5 і MockServer для тестування [1; 2; 9; 10; 11; 15; 16; 23; 36].

Висновки до розділу

У першому розділі проведено аналіз предметної галузі та існуючих рішень для автоматизації розумного будинку. Встановлено, що жодне з наявних рішень не поєднує одночасно інтеграцію з Tuya API, отримання даних від оператора Yasno, Telegram-інтерфейс та проактивні сценарії управління пристроями.

Проаналізовано IoT-платформу Tuya та її Open API, API сервісу Yasno, а також підходи до реалізації AI-асистентів. Для реалізації проактивних сценаріїв обрано детерміновану логіку на основі правил як більш надійний та економічно ефективний підхід порівняно з LLM.

Сформульовано функціональні та нефункціональні вимоги до розроблюваної системи, що стали основою для проєктування архітектури, описаного в наступному розділі.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНТЕЛЕКТУАЛЬНОГО АГЕНТУ

2.1 Архітектура системи та вибір технологічного стеку

Розроблена система реалізована як монолітний серверний застосунок за принципом «тонкий клієнт – товстий сервер»: вся бізнес-логіка зосереджена на бекенді, а клієнтами виступають Telegram-бот та REST-клієнти (мобільний або веб-застосунок). Такий підхід є оптимальним на стадії MVP (Minimum Viable Product), оскільки спрощує розгортання та налагодження.

Архітектура побудована за класичним тришаровим принципом: контролери (Controllers) приймають HTTP-запити та делегують обробку сервісному шару; сервіси (Services) реалізують бізнес-логіку та взаємодіють з репозиторіями; репозиторії (Repositories) надають доступ до реляційної бази даних PostgreSQL через Spring Data JPA. Такий поділ відповідає описаному М. Фаулером підходу шарової архітектури: зміни в одному шарі локалізуються, доки зберігається контракт між шарами [7, с. 20].

Додатково виділено шари: клієнтів зовнішніх API (client/ – YasnoApiClient, TuyaApiClient), шар маперів (mapper/ – MapStruct), шар спільних конфігурацій та обробників виключень (common/).

Структура пакетів застосунку наведена нижче:

```
ua.edu.nubip.rpiagent/
client/ – YasnoApiClient,TuyaApiClient;
common/ – конфігурація, HTTP-утиліти, винятки;
controller/api/ – OpenAPI-інтерфейси REST;
controller/ – тонкі контролери;
controller/dtos/ – Request/Response DTO;
entity/{user,schedule,tuya,ai,station,notification}/ – JPA-сутності за
доменами;
model/{schedule,tuya,ai}/ – внутрішні value-об'єкти;
mapper/ – MapStruct-мапери;
repository/ – Spring Data JPA;
scheduler/ – планові завдання;
service/{user,schedule,tuya,station,notification,ai.*}/ – сервісний
шар і AI-оркестрація.
```

Технологічний стек системи обрано з урахуванням вимог з розділу 1.5 та сучасних стандартів розробки enterprise-застосунків. Детальний перелік технологій та обґрунтування їх вибору наведено в таблиці 2.1.

Рисунок 2.1 – Трирівнева архітектура серверного застосунку

Таблиця 2.1 – Технологічний стек системи

Технологія	Версія	Призначення
Java	21 LTS	Основна мова програмування
Spring Boot	4.x	Фреймворк для побудови застосунку
Spring Data JPA	4.x	ORM та доступ до бази даних
PostgreSQL	17	Реляційна СКБД
Flyway	10.x	Версіонування схеми БД
MapStruct	1.6.3	Маппінг між шарами
JUnit 5	5.x	Модульне та інтеграційне тестування
Mockito	5.x	Мокування залежностей у тестах

Конфігурація застосунку розділена на профілі: demo (демонстрація з PostgreSQL і Yasno MockServer), it (ізольовані API/інтеграційні тести з PostgreSQL і MockServer), local (повний локальний стек із PostgreSQL, backend і Telegram-ботом) та prod (експлуатаційна конфігурація через змінні середовища). Такий підхід розділяє демонстраційні, тестові, локальні та продуктивні сценарії запуску і не вимагає зберігати облікові дані у версіонованому коді.

Розробка серверної частини на Spring Boot 4 / Java 21

Основу серверної частини складають REST-контролери, що обробляють HTTP-запити від клієнтів. У системі реалізовано такі групи контролерів: управління користувачами (UserController), прив'язка адрес (AddressBindingController), отримання розкладу відключень (ScheduleController), управління пристроями Tuya (DeviceController, TuyaOnboardingController), події відключень (OutagesController), сповіщення для Telegram (TelegramNotificationController), повідомлення та підтвердження дій AI-асистента (AiAssistantController), профілі зарядних станцій (PowerStationProfileController) та ієрархічний пошук адрес Yasno (YasnoAddressResolutionController).

Управління користувачами реалізовано за принципом «знайти або створити» (find-or-create): при першому зверненні з telegram_user_id система автоматично реєструє нового користувача та створює запис налаштувань UserSettingsEntity зі значеннями за замовчуванням. Це дозволяє уникнути явного кроку реєстрації.

Приклад обробника запиту реєстрації/отримання користувача (UserController):

POST /api/v1/users

```
{
   "telegramUserId": 123456789,
   "displayName": "Mykhailo",
   "languageCode": "uk"
}
Response 200:
{
   "id": 1,
   "telegramUserId": 123456789,
   "displayName": "Mykhailo",
   "languageCode": "uk",
   ...
}
.
```

Налаштування користувача доступні окремо через GET/PUT /api/v1/users/{userId}/settings.

Прив'язка адреси є складнішим процесом, що включає кілька кроків. AddressBindingService отримує дані адреси, передає їх до AddressResolutionService, який через YasnoApiClient виконує пошук по ієрархії: регіон → провайдер → вулиця → будинок → groupKey. Отриманий groupKey (формат: "group.subgroup") зберігається разом з адресою та ідентифікатором користувача, після чого ініціюється початковий імпорт розкладу.

Конвеєр імпорту розкладу (PlannedScheduleImportService) складається з трьох послідовних кроків:

Рисунок 2.2 – Конвеєр імпорту графіків відключень Yasno

Перший крок – отримання (Fetch). YasnoScheduleFetchService звертається до Yasno API за поточним розкладом для групи (regionId + dsoId). YasnoApiClient реалізує ручну retry-логіку: при отриманні HTTP 429 або 5xx повторює запит з наростаючою затримкою, до configurable максимального числа спроб.

Другий крок – нормалізація (Normalize). ScheduleNormalizationService перетворює «сирий» JSON-відповідь Yasno на внутрішнє представлення – список NormalizedScheduleDay, де кожен день містить впорядковані слоти з типом (планове/можливе відключення), часом початку та кінця у хвилинах від опівночі. Одночасно обчислюється SHA-256 чек сума всього payload для подальшої ідемпотентності.

Третій крок – збереження (Persist). SchedulePersistenceService порівнює нову чек суму з останньою збереженою для цієї адреси. Якщо чек суми збігаються – дані не змінилися, запис до БД пропускається. Це суттєво зменшує навантаження на базу даних при частих синхронізаціях. Кожна спроба синхронізації фіксується в ScheduleSyncLogEntity – аудит-записі.

Планова синхронізація запускається щогодини через ScheduleSyncJob. Завдання групує активні прив'язки адрес по (regionId, dsoId) – щоб зробити лише один API-запит на групу, а не окремий для кожного користувача. Синхронізація ввімкнена лише при `app.yasno.sync.enabled=true`. Додатково реалізовано YasnoStartupSyncJob — планувальник, що однократно запускає синхронізацію Yasno при старті застосунку, забезпечуючи наявність актуального розкладу відразу після перезапуску.

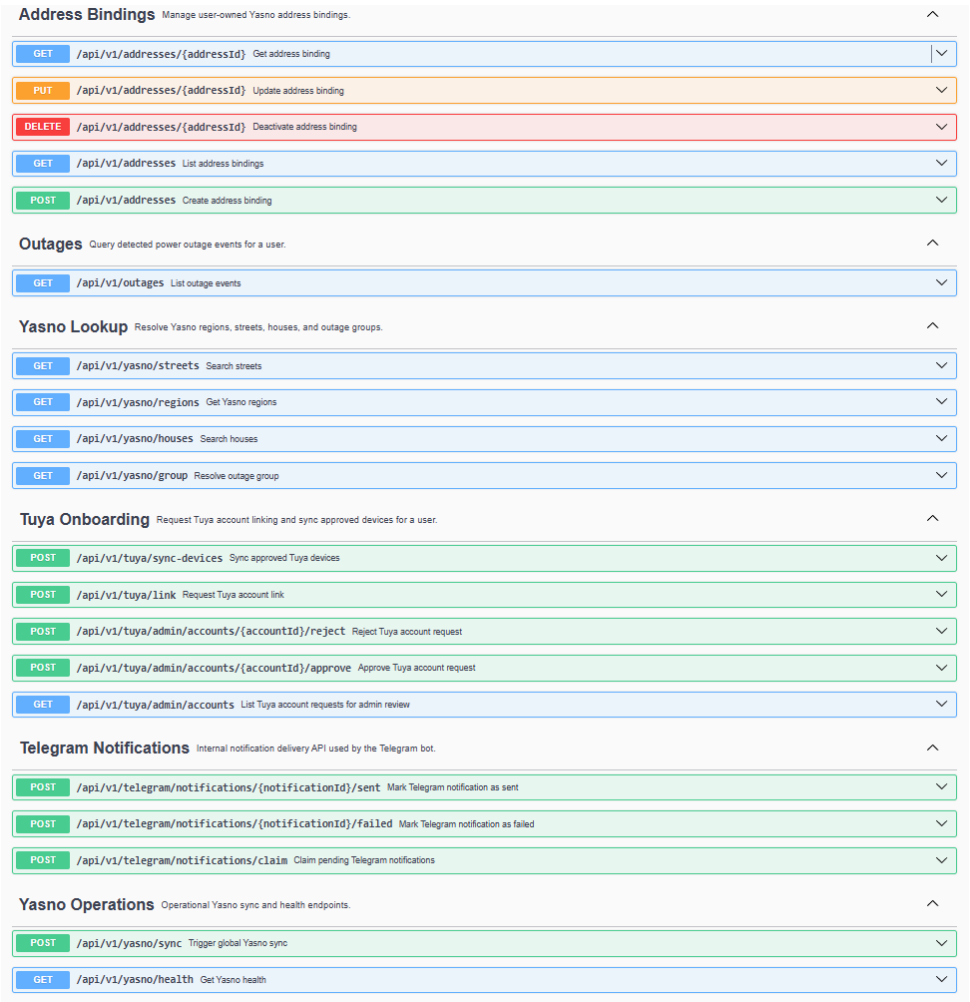

Рисунок 2.3 – Інтерфейс Swagger UI розробленого REST API системи

2.2 Інтеграція з Tuya IoT API та API сервісу Yasno

Tuya Open API використовує двоетапну токенну автентифікацію. Спочатку клієнт отримує токен доступу через endpoint GET /v1.0/token?grant_type=1, передаючи у заголовках client_id, timestamp та підпис. Підпис формується за алгоритмом HMAC-SHA256 на основі рядка: `client_id` + `timestamp` для запиту без токена, або `client_id` + `access_token` + `timestamp` для запитів з токеном [3; 5; 6].

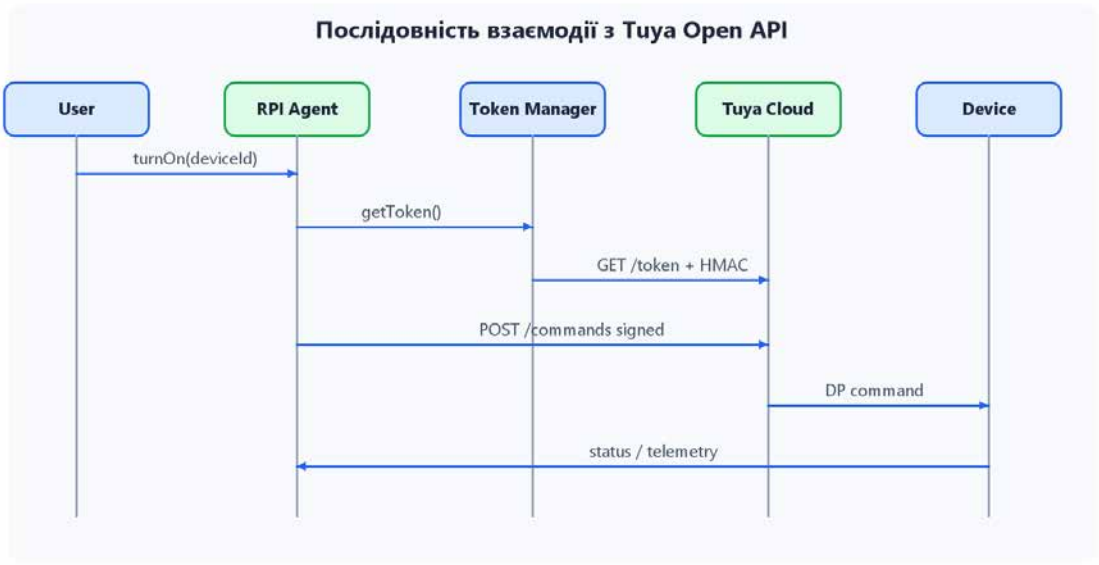

Рисунок 2.4 – Послідовність взаємодії з Tuya Open API

TuyaTokenManager реалізує кешування та автоматичне оновлення токена. Токен має обмежений термін дії (expire_time), тому перед кожним запитом TuyaApiClient перевіряє, чи не минув термін, та за потреби запитує новий. Реалізація потокобезпечна завдяки використанню synchronized-методу оновлення.

Підписання запитів реалізовано в утилітарному класі TuyaSignatureUtils. Нижче наведено алгоритм формування підпису:

```
string_to_sign = HTTP_METHOD + "\n"
               + content_sha256 + "\n"
               + "" + "\n"  // headers
               + url_path

str_to_sign = client_id + access_token + timestamp + string_to_sign
sign = HMAC_SHA256(str_to_sign, client_secret).toUpperCase()
```

Синхронізація пристроїв облікового запису Tuya виконується через TuyaAccountSyncTransactionalService. Користувач подає свій Tuya UID, після чого обліковий запис переходить у статус PENDING і очікує адміністративного погодження. Лише для APPROVED-акаунтів система отримує список пристроїв через GET /v1.0/users/{uid}/devices та зберігає їх у TuyaDeviceEntity зі статусом, назвою, категорією й поточним знімком DP-значень.

Окремо реалізовано прив’язку Tuya-пристрою до конкретної Yasno-адреси користувача. У TuyaDeviceEntity поле addressBinding посилається на AddressBindingEntity, а на рівні PostgreSQL зв’язок додано міграцією V4__add_tuya_device_address_binding.sql як колонку tuya_device.address_binding_id із зовнішнім ключем на address_binding.

Користувач керує цим зв’язком через PUT /api/v1/devices/{deviceId}/address з тілом {"addressId": ...} та DELETE /api/v1/devices/{deviceId}/address. TuyaDeviceService.assignAddress перевіряє, що і пристрій, і адреса належать тому самому користувачу, а адреса є активною; відповідь TuyaDeviceResponseDto повертає addressId, addressDisplayAddress і addressGroupKey. Цей зв’язок потрібен для сценаріїв, що залежать від графіка конкретної адреси, та для Telegram-екрана пристрою, але не замінює базову прив’язку адреси до групи Yasno.

Для отримання актуального стану пристроїв реалізовано TuyaDevicePollerService, що запускається з налаштованим інтервалом. Служба працює лише з пристроями погоджених акаунтів, запитує поточні data points (DP) через GET /v1.0/devices/{deviceId}/status та зберігає телеметрію в DeviceTelemetryEntity. Архівне зберігання телеметрії дозволяє аналізувати споживання, відрізняти лічильникові delta-інтервали від інтеграції потужності та позначати якість даних.

Надсилання команди пристрою (наприклад, вмикання розетки) виконується через:

POST /v1.0/devices/{deviceId}/commands

```
{
  "commands": [
        {
              "code": "switch_1",
              "value": true
        }
      ]
}.
```

Конкретний DP-код визначається на основі збережених метаданих пристрою, а не жорстко прив'язується до категорії.

TuyaDeviceService інкапсулює формування та надсилання команд на пристрої через DeviceCommandRequestDto. Інтерпретація сирих значень status-кодів винесена до TuyaStatusCodeService та TuyaTelemetryMetadataService, які читають метадані з довідника `tuya_device_status_code` і застосовують налаштовані дільники без перекомпіляції застосунку.

Обробка помилок зовнішніх API уніфікована через ServiceClientException, що містить джерело, операцію, HTTP-статус і тіло відповіді провайдера. Доменні помилки Tuya, наприклад

TuyaAccountApprovalRequiredException, TuyaDeviceNotFoundException або TuyaDeviceNotControllableException, обробляються GlobalExceptionHandler окремо та повертають клієнту контрольовану REST-помилку без витоку службових секретів.

Інтеграція з API сервісу Yasno

YasnoApiClient реалізує взаємодію з публічним API сервісу Yasno. Клієнт побудований на основі Spring RestClient, а повторювані HTTP-виклики, таймаути й перетворення помилок винесені до спільного HttpUtils та ServiceClientException. Шляхи API не зашиті в коді, а зчитуються з YasnoApiProperties.

Процес розпізнавання адреси виконується ієрархічно:

– Отримання переліку регіонів (GET /regions);

– Пошук провайдера (DSO) у регіоні (GET /regions/{regionId}/providers);

– Отримання вулиць у місті (GET /providers/{dsoId}/streets?q={query});

– Отримання будинків на вулиці (GET /streets/{streetId}/houses);

– Отримання groupKey для конкретного будинку.

AddressResolutionService реалізує «розумний» пошук: якщо запит повертає кілька збігів для вулиці, система вибирає перший точний збіг або повертає список варіантів для уточнення. Всі проміжні ідентифікатори (regionId, dsoId, streetId) нормалізуються в RegionDsoKey – внутрішній value-об'єкт для групування адрес при синхронізації.

Графік відключень отримується через YasnoScheduleFetchService. Кінцева точка API повертає JSON зі структурою планового розкладу на кілька днів. Структура відповіді включає:

```
{
  "current_shift": 0,
  "schedule": [
    {
      "date": "2024-12-15",
      "groups": {
        "1.1": [
          { "start": 0, "end": 60, "type": 1 },
          { "start": 120, "end": 180, "type": 2 }
        ]
      }
    }
  ]
}
```

Де type=1 означає планове (обов'язкове) відключення, type=2 – можливе відключення. Значення start та end – хвилини від початку доби (наприклад, start=120, end=180 відповідає 02:00–03:00).

ScheduleNormalizationService перетворює цей формат у внутрішнє представлення NormalizedScheduleDay. Нормалізація включає: фільтрацію слотів для конкретного groupKey, сортування за часом початку, об'єднання суміжних слотів однакового типу та обчислення SHA-256 чек суми для ідемпотентності.

YasnoHealthService реалізує оцінку стану інтеграції: повертає `UP` (API доступний, дані свіжі), `DEGRADED` (частина адрес має застарілий розклад або провалені імпорти), `DOWN` (API недоступний) або `EMPTY` (немає жодної прив'язаної адреси). Цей статус використовується для моніторингу системи.

Режим деградації (graceful degradation) передбачає, що при недоступності Yasno API нові імпорти завершуються помилкою та логуються в ScheduleSyncLogEntity, але наявні збережені розклади продовжують використовуватись для відповідей на запити користувачів. Це забезпечує безперервність сервісу навіть при тимчасовій недоступності зовнішнього API.

2.3 Схема бази даних та Flyway-міграції

База даних PostgreSQL є центральним сховищем системи. Схема версіонується за допомогою Flyway – інструменту міграцій, що забезпечує відтворюваність стану БД в будь-якому середовищі. Репозиторний шар реалізовано через Spring Data JPA, а міграції виконуються до валідації JPA-схеми завдяки окремій конфігурації залежності entityManagerFactory від Flyway [10; 15; 23].

Схема бази даних містить такі основні таблиці:

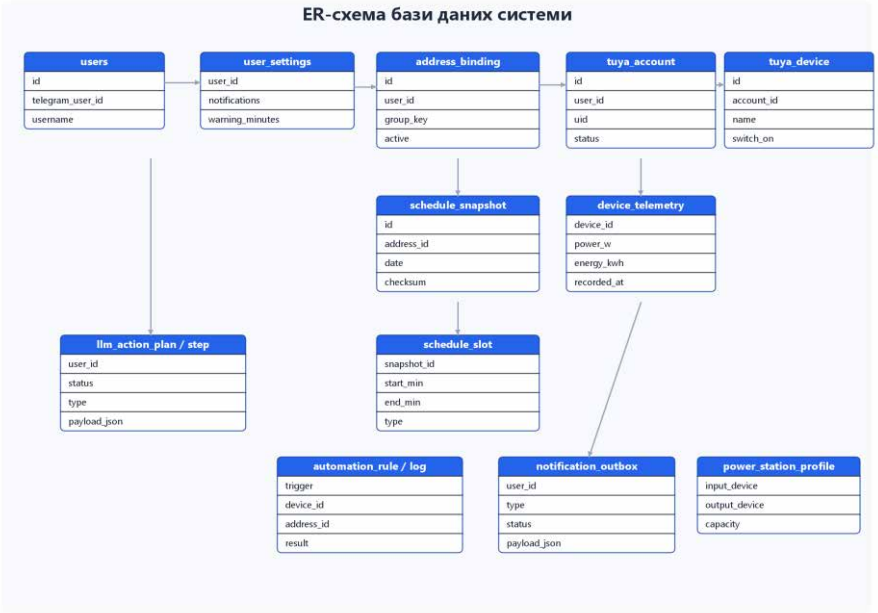

Рисунок 2.5 – ER-схема бази даних системи

Таблиця 2.2 – Основні таблиці бази даних

Таблиця	Основні поля	Призначення
users	id, telegram_user_id, username	Зареєстровані користувачі
user_settings	user_id, notifications_enabled, timezone	Налаштування користувача
address_binding	id, user_id, display_address, normalized_address, region_id, dso_id, street_id, house_id, group_key, is_active	Прив'язані адреси з groupKey
schedule_snapshot	id, address_binding_id, schedule_date, raw_payload, checksum, is_current	Снепшоти розкладу (ідемпотентність)
schedule_slot	id, schedule_snapshot_id, start_minute, end_minute, slot_type	Часові слоти у снепшоті
schedule_sync_log	id, address_binding_id, status, error_message, created_at	Аудит-лог синхронізацій
tuya_account	id, client_id, uid, user_id	Облікові записи Tuya
tuya_device	id, device_id, name, category, online, switch_on, address_binding_id, tuya_account_id	Пристрої Tuya з поточним станом та опційною прив’язкою до адреси Yasno

Таблиця address_binding містить зовнішній ключ user_id до users, що дозволяє фільтрувати прив'язки конкретного користувача. Поле is_current у schedule_snapshot використовується для швидкого отримання актуального розкладу без JOIN-запитів.

Flyway-міграції зберігаються у src/main/resources/db/migration/ з

версіонуванням V1__, V2__ тощо. V1__init.sql містить базову схему, а подальші інкрементальні зміни винесено в окремі міграції: V2__add_tuya_device_switch_on.sql додає поле switch_on у tuya_device, V3__add_notification_outbox.sql створює durable-чергу Telegram-сповіщень, а V4__add_tuya_device_address_binding.sql додає зв'язок tuya_device.address_binding_id з address_binding. Тестове середовище використовує H2 in-memory (режим PostgreSQL compatibility) з DDL через Hibernate create-drop, Flyway вимкнено.

Схема також містить таблиці для AI-асистента, автоматизації, зарядних станцій і доставки сповіщень: `conversation`, `conversation_message`, `llm_action_plan`, `llm_action_step`, `automation_rule`, `automation_log`, `power_station_profile` та `notification_outbox`. Таблиця `notification_outbox` реалізує довговічну чергу повідомлень для Telegram-бота зі статусами `PENDING`, `SENDING`, `SENT` і `FAILED`, лічильником спроб, ключем дедуплікації та JSON-payload.

2.4 Механізм проактивних сценаріїв, Telegram-інтерфейс та поглиблені механізми телеметрії

Проактивні сценарії у поточній реалізації розділено на три незалежні механізми. OutageDetectionService аналізує стан пристроїв Tuya та фіксує фактичні події відключення або відновлення живлення у вигляді OutageEventEntity. OutageNotificationJob і OutageNotificationService аналізують збережені розклади Yasno та створюють попередження про наближення планового відключення. AiAutomationExecutionService виконує правила автоматизації, наприклад вимкнення пристрою за N хвилин до відключення або затримане увімкнення після повернення світла.

Сповіщення не надсилаються безпосередньо з серверної частини у Telegram. Бекенд записує їх у таблицю notification_outbox, де зберігаються тип повідомлення, користувач, час відправлення, JSON-payload, статус доставки та кількість спроб. Такий підхід забезпечує надійність: навіть якщо Telegram-бот тимчасово недоступний, повідомлення не втрачається і може бути забране пізніше.

Telegram-інтерфейс реалізовано як окремий Python-сервіс на базі aiogram. Він взаємодіє з бекендом лише через REST API, використовує X-Internal-Bot-Token для сервісного доступу і не містить бізнес-логіки керування пристроями.

Таке розділення на backend як джерело доменної логіки та окремий UI-сервіс відповідає рекомендаціям щодо чітких меж сервісів у розподілених системах [8; 31]. Бот періодично викликає POST /api/v1/telegram/notifications/claim, надсилає повідомлення користувачу через Telegram Bot API, а потім підтверджує результат через /sent або /failed.

У користувацькому інтерфейсі поєднано два режими роботи. Натискання кнопок, де вже відомі deviceId, addressId, planId або період статистики, викликає детерміновані backend endpoints напряму. Натомість довільні україномовні повідомлення користувача передаються до POST /api/v1/ai/messages, де AI-асистент формує план дій. Команди, що змінюють стан пристрою або створюють відкладене правило, зберігаються у llm_action_plan та llm_action_step і потребують підтвердження користувачем.

Час попередження про планове відключення не є жорстко закодованим. Він визначається налаштуванням `UserSettingsEntity.preOutageWarningMinutes`, а саме надсилання може бути вимкнене через proactiveNotificationsEnabled. Додатково налаштування stabilityWaitMinutes використовується для сценаріїв стабілізації після повернення електроенергії, щоб не вмикати вибрані пристрої одразу після короткого нестабільного відновлення живлення.

Таким чином Telegram-бот є тонким UI-адаптером, а джерелом істини залишаються серверні сервіси Spring Boot: вони перевіряють права доступу, стан пристроїв, розклад відключень, правила автоматизації та журнал виконання. Очікуваний вигляд взаємодії користувача з асистентом через Telegram наведено на рисунку 2.6.

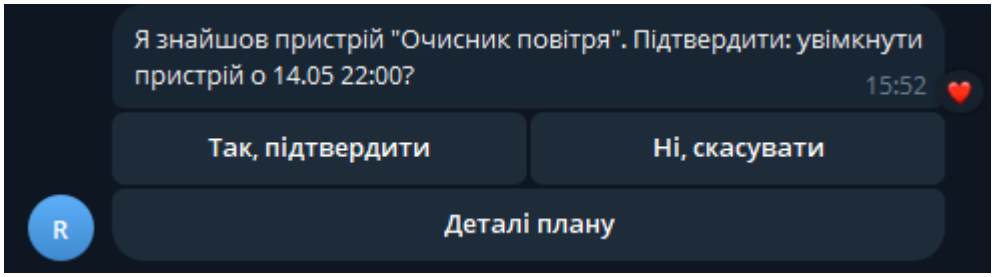

Рисунок 2.6 – Приклад взаємодії з AI-асистентом через Telegram-інтерфейс

Поглиблені механізми телеметрії, виявлення відключень та оптимізації синхронізації

У попередніх розділах розкрито базову архітектуру системи, її зовнішні

інтеграції та методику тестування. Проте під час практичної розробки прототипу виявлено низку граничних умов експлуатації, що вимагали окремих інженерних рішень. Ці рішення не вкладаються у рамки «типових» модулів Spring Boot-застосунку, але саме вони визначають, чи буде система придатною до щоденного використання в умовах української електромережі. Нижче розкрито сім таких механізмів, які суттєво впливають на надійність, економність зовнішніх викликів та якість даних, що потрапляють до кінцевого користувача.

Гібридна модель обчислення енергоспоживання

Tuya-сумісні розетки, які застосовуються у складі розробленої системи, умовно поділяються на дві сім'ї залежно від переліку статус-кодів, які вони повертають хмарному API. Частина моделей штатно веде внутрішній лічильник спожитої енергії (типовий код `add_ele`), інша ж обмежується наданням лише миттєвої активної потужності (`cur_power`). Щоб отримати одноманітну картину споживання за довільний часовий інтервал незалежно від моделі, у сервісі DeviceEnergyStatisticsService мною реалізовано два незалежних алгоритми обчислення та механізм автоматичного вибору між ними.

Перший алгоритм, умовно позначений як `COUNTER_DELTA`, використовує різницю двох найближчих до меж інтервалу відліків лічильника. Цей підхід не накопичує чисельної похибки, адже спирається на вже проінтегровану прошивкою пристрою величину. Водночас він не є стійким до скидання лічильника – ситуацій, коли нове значення несподівано менше попереднього. Такий скид трапляється після знеструмлення розетки, її перезавантаження або ручного обнулення у мобільному застосунку виробника. Щоб уникнути як негативних, так і штучно завищених значень, у коді реалізовано перевірку монотонності: якщо наступний відлік менший за попередній, він трактується як нова точка відліку, а дельта обчислюється від неї.

Другий алгоритм, `POWER_INTEGRATION`, застосовується до розеток, що не ведуть власного лічильника. Послідовність відліків миттєвої потужності всередині бакета обробляється як набір пар сусідніх точок, між якими обчислюється площа трапеції, обмеженої часовою віссю. Сума таких площ і є оцінкою енергії за інтервал. Для коректного обчислення потрібна відносно щільна сітка вимірювань: якщо відстань між сусідніми відліками перевищує дві хвилини, відповідна ділянка свідомо відкидається як ненадійна – у цьому проміжку могло відбутися суттєве змінення навантаження, про яке система не знає. Такий компроміс жертвує частиною корисних точок заради гарантії, що

отримане значення не буде вигаданим.

Вибір алгоритму відбувається автоматично на рівні самого сервісу: якщо метадані пристрою, отримані з таблиці `tuya_device_status_code`, містять код сімейства ENERGY – застосовується лічильниковий алгоритм, інакше виконується інтегрування потужності. Разом зі значенням енергії повертається прапорець якості обчислення, який сигналізує, чи не було скидання лічильника протягом бакета, чи не спостерігалося повних відключень живлення та чи весь інтервал покрито фактичними вимірюваннями.

Окремо від попереджень за плановим графіком система формує сповіщення про фактичні події живлення. Якщо OutageDetectionService після контрольного вікна класифікує втрату зв'язку з пристроями як FULL_OUTAGE, backend створює запис OutageEventEntity і додає повідомлення до notification_outbox. Після повернення пристроїв онлайн подія закривається, формується подія відновлення живлення, а Telegram-бот отримує відповідне повідомлення через внутрішній claim/sent/failed API. Таким чином користувач бачить не лише планові Yasno-попередження, а й фактичний стан електропостачання у квартирі або будинку.

Метадані статус-кодів Tuya та класифікація пристроїв

Платформа Tuya не нав'язує жорсткої специфікації на те, яку саме фізичну величину повертає певний код статусу пристрою. Існують узвичаєні домовленості: код cur_power зазвичай містить активну потужність у десятих частках вата, код cur_voltage – напругу у десятих частках вольта, код add_ele – сумарну спожиту енергію у тисячних частках кіловат-годин. Однак конкретний виробник вправі відступити від цих дробових множників, і жодний зовнішній орган не вимагає уніфікації. Якщо «зашити» множники прямо у код застосунку, то для однієї моделі розетки графіки споживання виглядатимуть правильно, а для іншої – виявляться завищеними у десять разів.

Саме тому всю інтерпретацію «сирих» значень у системі винесено до двох взаємопов'язаних довідників бази даних. Таблиця `tuya_device_status_code` у мінімальному вигляді містить сам код, його сімейство (`SWITCH`, `POWER`, `VOLTAGE`, `CURRENT`, `ENERGY`) та типовий дільник. Додатковий рівень – таблиця перевизначень, яка дозволяє прив'язати власний дільник до пари «пристрій + код»: за замовчуванням використовується типовий, а за наявності перевизначення – пріоритет отримує саме воно. Відповідальність за зберігання цих записів покладено на сервіс TuyaTelemetryMetadataService, а за застосування при

перетворенні значень – на TuyaTelemetryService, який викликається з поллера на кожному циклі обходу пристроїв.

Цей підхід дає дві відчутні переваги. По-перше, додавання нової моделі розетки не вимагає перекомпіляції застосунку – достатньо оновити рядок у довіднику. По-друге, у разі виявлення некоректних даних значення дільника можна змінити ретроспективно, і при наступному опитуванні телеметрія почне перетворюватися правильно без повторного збору сирих показань із пристрою.

Розрізнення повних відключень та короткочасних просадок

Наївна реалізація фіксації відключень могла б виглядати так: як тільки поллер помічає, що всі Tuya-розетки користувача перейшли в стан offline, – у базі даних відкривається запис outage_event типу `FULL_OUTAGE`; як тільки хоч одна з них повертається в ефір, подія закривається. На практиці такий підхід породжує велику кількість паразитних записів, адже хмара Tuya іноді повертає offline під час планового перезавантаження серверів, короткочасного провалу локальної Wi-Fi-мережі або тимчасового перевантаження каналу користувача.

Рисунок 2.7 – Станова машина детектора відключень (FLICKER vs FULL_OUTAGE)

Для відсічення цього «шуму» у сервісі OutageDetectionService реалізовано метод reconcileUserStatus, який порівнює поточний зріз стану з попереднім та відкриває нові записи лише після того, як витримано контрольне вікно. Якщо тривалість епізоду без зв’язку до моменту повернення пристроїв у мережу склала менше десяти секунд, уже відкрита подія ретроспективно переводиться у тип `FLICKER`, що відповідає побутовому поняттю «моргання світла». Якщо ж інтервал виявився довшим, подія залишається з типом `FULL_OUTAGE` і надалі використовується для обчислення накопиченого часу автономії побутових приладів.

Розмежування цих двох типів подій має принципове значення для користувацького інтерфейсу. Моргання недоцільно виводити в окремому

сповіщенні – інакше Telegram-чат перетворився б на потік однотипних повідомлень, і користувач через кілька годин вимкнув би сповіщення повністю. Натомість повне відключення має ініціювати окреме попередження та запустити оцінку часу до відновлення за попередньо завантаженим розкладом Yasno. Крім того, при агрегації статистики за місяць моргання зовсім не враховуються у показнику «сумарний час без світла», що наближає цифру до фактичного суб'єктивного досвіду мешканця.

Оптимізація синхронізації розкладів групуванням адрес

Оскільки графіки планових відключень формуються оператором системи розподілу окремо для кожної території, а всередині однієї групи відключення (наприклад, 15.1) усі абоненти користуються ідентичним розкладом, зверненням до API Yasno окремо за кожною адресою система витрачала б HTTP-виклики неефективно. Якби у базі даних було сто активних прив'язок адрес, що належать до одного ОСР, прямолінійна реалізація надіслала б сто однакових запитів за один запуск планувальника й наштовхнулась як на rate-ліміт зовнішнього сервісу, так і на марне навантаження на власну інфраструктуру.

У сервісі ScheduleSyncService реалізовано попередню агрегацію прив'язок адрес. Перед викликами всі активні прив'язки збираються у відображення, ключем якого є пара (regionId, dsoId), а значенням – список адрес, що потрапили до цієї пари. Для кожної пари до Yasno відправляється рівно один запит до ендпоінту /regions/{regionId}/dsos/{dsoId}/planned-outages. У відповіді повертається словник, де ключ – це groupKey конкретної групи відключення, а значення – розклад на сьогодні та завтра. Далі вже локально, без додаткових мережевих викликів, кожна адреса знаходить свій groupKey у словнику і передає відповідний фрагмент далі конвеєром імпорту.

Така оптимізація не лише знижує кількість HTTP-запитів, але й робить процедуру синхронізації стійкою до збоїв. Помилка при обробці однієї пари регіон – ОСР не перериває всю задачу: вона фіксується в `schedule_sync_log` із зазначенням причини, а наступна пара обробляється далі. Повторний запуск завжди відбувається ідемпотентно за рахунок контрольної суми rawPayload, тому жодних дубльованих записів у таблиці schedule_snapshot не з'являється. Такий підхід узгоджується з рекомендаціями М. Клеппманна щодо надійної повторної обробки даних у дата-інтенсивних системах [30, с. 215].

Чотиристатусна модель здоров'я інтеграції Yasno

Усі запити до Yasno у розробленій системі виконуються через окремий

клієнт із відносно невеликим таймаутом, проте на практиці відповідь може затримуватись або взагалі не надходити. Щоб користувач міг побачити достовірну інформацію про свіжість розкладу, який він переглядає, у коді передбачено сервіс YasnoHealthService. Він агрегує чотири складові поточного стану інтеграції: результат пробного запиту до API, кількість активних адрес, кількість адрес з актуальним знімком розкладу та мітку часу останнього успішного імпорту.

На основі цих фактів обчислюється підсумковий статус із чотирьох можливих значень. Значення UP означає, що API доступне, усі зареєстровані адреси мають знімки, не старіші за визначений поріг (за замовчуванням – шість годин), а сервіс синхронізації за останню добу не накопичив помилок. Значення `DEGRADED` виставляється, коли принаймні одна з умов порушена: Yasno не відповідає на пробний запит, частина адрес не має знімків або знімки застарілі. Значення `DOWN` з'являється у випадку, коли жодна з активних адрес не має актуального знімку взагалі, хоча самі адреси існують. Значення `EMPTY` позначає ситуацію, коли у базі немає жодної активної прив'язки – тобто систему ще ніхто не налаштував.

Крім самої мітки, YasnoHealthService повертає кількість невдалих імпортів за останні 24 години та мітки часу останньої успішної і невдалої синхронізації. Ця інформація використовується у Telegram-інтерфейсі: якщо асистент готується надіслати попередження про майбутнє відключення, він попередньо перевіряє мітку стану – у разі DEGRADED або DOWN повідомлення супроводжується приміткою, що розклад може бути неактуальним, і користувачу доцільно звернутися до офіційного каналу провайдера.

Фільтр сумісних категорій Tuya та верифікація керування

Один обліковий запис Tuya може містити пристрої найрізноманітніших категорій: лампочки, датчики температури, термостати, роботи-пилососи тощо. Для завдань розробленого асистента корисні лише ті, що здатні розмикати живлення споживача: штатні smart-розетки та DIN-реле. Тому під час початкової синхронізації пристроїв сервіс TuyaAccountSyncTransactionalService не вивантажує на бекенд весь перелік пристроїв облікового запису, а пропускає його через фільтр на основі довідника `tuya_device_category`. Цей довідник містить по рядку для кожної категорії Tuya, прапорець enabled, який визначає допустимість у межах системи, а також прапорець requires_status_verification, встановлений лише для окремих підкатегорій.

Прапорець додаткової перевірки введено спеціально для змішаних категорій на кшталт tdq. Зовні такі пристрої можуть виглядати як DIN-реле для електрощитка, проте частина моделей справді передає стан контактів і дозволяє ними керувати через хмару, а інша є лише лічильниковою або сенсорною моделлю без дистанційного керування живленням. Щоб не вводити користувача в оману, при онбордингу для таких пристроїв додатково аналізуються статус-коди та специфікація пристрою. Якщо відсутні керовані SWITCH-коди, пристрій не допускається до команд керування.

Дворівневий фільтр (спочатку за категорією, а потім за фактичним набором статус-кодів) дозволяє автоматично підтримувати актуальний перелік працездатних пристроїв: досить додати новий рядок у довідник категорій, а якщо нова модель виявиться «сухим» лічильником – вона буде відсіяна автоматично вже при першому опитуванні, без втручання розробника. Приклад DIN-реле, що підтримується системою після верифікації статус-кодів, наведено на рисунку 4.2.

Проактивне оновлення токена Tuya

Токен доступу, який повертає Tuya Open Platform у відповідь на HMAC-підписаний запит автентифікації, має відносно короткий термін дії – близько двох годин. Якщо чекати моменту, коли черговий запит завершиться помилкою типу 1010 «token is expired», то цей запит буде втрачено, а повторна спроба виконається вже з новим токеном, що не тільки додає надлишкових затримок, а й суттєво ускладнює логіку повторних спроб у викликах з граничними станами.

Щоб уникнути такого реактивного стилю, у класі TuyaTokenManager реалізовано проактивну модель: при кожному отриманні токена зберігається прогнозований час закінчення його дії, а перевірка актуальності перед кожним викликом зсунута на 600 секунд вперед. Тобто щойно до формальної межі дії залишається менше десяти хвилин, наступний код, що звертається до менеджера, одразу отримує оновлений екземпляр – прострочена відповідь ніколи не використовується у виклику. Сама процедура оновлення синхронізована на рівні об’єкта, тож навіть якщо два паралельні потоки одночасно виявлять потребу у новому токені, запит до Tuya буде виконано рівно один раз, а другий потік отримає спільний результат.

У поєднанні з механізмом повторних спроб `HttpUtils.executeWithRetry()`, який автоматично перевиконує HTTP-запит у разі тимчасових мережевих помилок, проактивне оновлення токена робить поллер та команди керування пристроями лінійними: жодна з функцій, що користується клієнтом

TuyaApiClient, не містить спеціальної обробки ситуації «токен прострочено».

Рисунок 2.8 – Приклад DIN-реле зі змішаної категорії tdq, що потребує додаткової перевірки

2.5 Сценарії розгортання: від ноутбука до Raspberry Pi 5 AI з ДБЖ

Назва репозиторію rpi-agent не є випадковою: система з самого початку проєктувалась як бекенд-сервіс, що може цілодобово працювати на малопотужному вбудованому комп'ютері у домашній мережі поруч зі смарт-розетками та роутером. Водночас, завдяки Docker Compose-конфігурації та відсутності залежностей від платформи, той самий код запускається і на ноутбуці розробника, і у хмарному середовищі. У цьому підрозділі розглянуто три реалістичні сценарії розгортання.

Перший сценарій – ноутбук або настільний ПК – є найпростішим для запуску у процесі розробки. Команда “`docker compose up -d –build`” піднімає два контейнери: “PostgreSQL 17-alpine” та сам застосунок на базі eclipse-temurin:21-jre (обидва образи мають ARM64 та AMD64 варіанти). Недолік очевидний: ноутбук не є завжди увімкненим пристроєм. Якщо система призначена для постійного моніторингу та проактивних сповіщень, вимикання ноутбука на ніч означає пропущений сигнал про відключення або незапущений алгоритм зарядної станції.

Другий і найбільш органічний для завдань системи сценарій – Raspberry Pi 5 з акумуляторним ДБЖ. Raspberry Pi 5 побудований на чотириядерному

процесорі Cortex-A76 з тактовою частотою 2,4 ГГц та оснащується 4 або 8 ГБ LPDDR4X-оперативної пам'яті. ARM64-архітектура повністю підтримується Docker та JVM-екосистемою: образи eclipse-temurin та postgres мають офіційні ARM64-збірки, тому той самий Dockerfile та docker-compose.yml запускаються без жодних правок. Споживання системи у спокої не перевищує 5–10 Вт, що є принциповим параметром у контексті даної роботи.

Ключова перевага RPi 5 у контексті проблематики відключень електроенергії полягає у можливості підключення компактного ДБЖ. Поширені рішення на базі 18650-акумуляторів (наприклад, Waveshare UPS HAT або аналоги) забезпечують автономну роботу на період відключень. Таким чином виникає принципово важлива властивість: сервер, що виявляє відключення та надсилає про це сповіщення, сам продовжує функціонувати під час цього відключення. Tuya-пристрої в тій же локальній мережі також залишаються доступними доти, доки роутер підключений до ДБЖ – типова конфігурація для домашніх резервних систем в Україні. Схему розгортання наведено на рисунку 2.9.

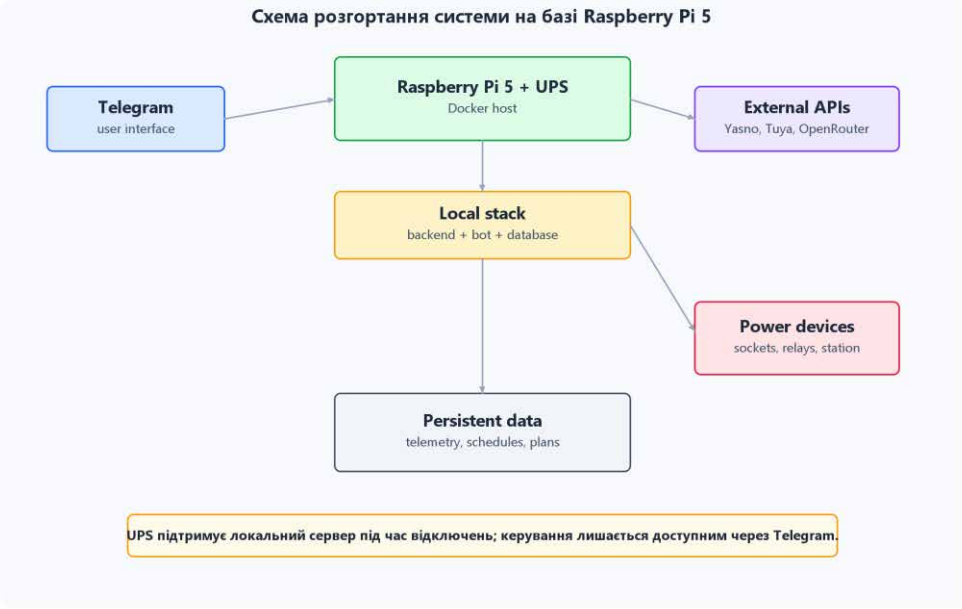

Рисунок 2.9 – Схема розгортання системи на базі Raspberry Pi 5

На Raspberry Pi 5 система працює як локальний edge-сервер домашньої енергетичної автоматизації. Пристрій розміщується в тій самій локальній мережі, що й маршрутизатор та смарт-пристрої, тому backend залишається доступним для внутрішніх сценаріїв без окремого ноутбука або стаціонарного ПК. Для коректної роботи використовується 64-бітна Raspberry Pi OS, Docker

Engine і Docker Compose, оскільки основні образи, застосовані в проєкті, мають ARM64-варіанти.

У режимі local на Raspberry Pi запускаються три ключові компоненти. Контейнер PostgreSQL 17 зберігає користувачів, прив’язки адрес, знімки графіків Yasno, телеметрію Tuya, плани AI-асистента і чергу notification_outbox у Docker-volume. Контейнер backend працює на Java 21 з профілем local, підключається до БД через адресу postgres:5432 і відкриває REST API на порту 8080. Окремий контейнер Telegram-бота на Python 3.12 виконує роль UI-адаптера: приймає повідомлення через long polling, викликає REST API backend та показує користувачу результати.

Під час роботи Raspberry Pi періодично виконує фонові задачі Spring: синхронізацію графіків Yasno, polling стану Tuya-пристроїв, обробку pending-кроків AI-планів, перевірку правил автоматизації та формування сповіщень про наближення відключення. Завдяки цьому дані зберігаються локально в PostgreSQL, а зовнішні сервіси використовуються як джерела оновлення або канали керування. Якщо інтернет тимчасово недоступний, останні збережені графіки, адреси, телеметрія і налаштування залишаються доступними, але нові запити до Yasno, Tuya, OpenRouter і Telegram обмежуються до відновлення зв’язку.

Для сценарію з ДБЖ Raspberry Pi доцільно живити разом із маршрутизатором та, за можливості, резервним інтернет-каналом. У такій конфігурації система може продовжувати отримувати Telegram-повідомлення, звертатися до Tuya/Yasno і надсилати сповіщення під час перебоїв електроживлення. Якщо зовнішній канал зв’язку недоступний, Raspberry Pi все одно зберігає локальний стан і після відновлення мережі поновлює синхронізацію. Це відповідає ідеї дипломного проєкту: Raspberry Pi не є лише середовищем запуску, а виступає постійно увімкненим домашнім вузлом, який пов’язує графіки відключень, IoT-пристрої, AI-асистента і Telegram-інтерфейс.

Третій сценарій – хмарна VPS-машина (наприклад, AWS EC2, DigitalOcean) – забезпечує максимальну доступність самого сервера, але втрачає переваги локальної мережі: запити до Tuya Cloud та Yasno API виконуються через інтернет, і якщо інтернет-з'єднання користувача недоступне (що часто супроводжує тривалі відключення), частина функцій деградує. Для продуктивного використання хмарний варіант потребує окремого налаштування мережевої безпеки та HTTPS-термінування.

Порівняльний аналіз трьох сценаріїв за ключовими критеріями наведено на рисунку 2.8. Для виробничої конфігурації оптимальним є поєднання: Raspberry Pi 5 виконує роль локального сервера з безперебійним живленням, а при наявності стабільного інтернету може синхронізувати логи або отримувати оновлення через хмарний канал. Окремо в репозиторії виділено три runtime-профілі для різних етапів роботи: demo використовує PostgreSQL і MockServer для швидкої демонстрації Yasno-сценаріїв без реальних зовнішніх залежностей; it піднімає PostgreSQL та MockServer для повторюваних Hurl-інтеграційних тестів; local запускає повний стек PostgreSQL, Spring Boot backend і Python Telegram bot через Docker Compose з можливістю підключення реальних ключів Tuya та OpenRouter. Такий поділ дозволяє перевіряти систему і як навчальний прототип, і як локальний сервіс для щоденного використання.

Порівняння сценаріїв розгортання системи

Критерій	Ноутбук / ПК	Raspberry Pi 5 + UPS	VPS / Cloud
Автономність	низька	висока	залежить від інтернету
Локальна мережа	так	так	ні
Енергоспоживання	середнє	низьке	зовнішнє
Придатність для 24/7	обмежена	оптимальна	добра
Рекомендація	розробка	домашня експлуатація	резерв / віддалений доступ

Рисунок 2.10 – Порівняння сценаріїв розгортання системи

Висновки до розділу 2

У другому розділі розроблено та реалізовано всі ключові компоненти системи. Архітектура застосунку побудована за тришаровим принципом з чітким розподілом відповідальності між контролерами, сервісами та репозиторіями. У підрозділі 2.7 розглянуто три сценарії розгортання: ноутбук для розробки, хмарний VPS для максимальної доступності та Raspberry Pi 5 з акумуляторним ДБЖ як оптимальний варіант для домашнього використання в умовах нестабільного електропостачання. Raspberry Pi 5 споживає 5–10 Вт, має повну підтримку ARM64 у Docker-образах проєкту та здатний працювати автономно 4–

12 годин, що є принциповою перевагою для системи моніторингу відключень.

Реалізовано повноцінну інтеграцію з Tuya Open API: автентифікація за токенною OpenAPI-автентифікацією з підписом HMAC-SHA256, кешування токенів, отримання та оновлення стану пристроїв, надсилання команд управління. Інтеграція з Yasno API включає ієрархічне розпізнавання адрес, отримання та нормалізацію графіків відключень з ідемпотентністю на основі SHA-256 чексуми.

Спроєктована схема бази даних PostgreSQL з Flyway-міграціями забезпечує надійне зберігання даних та відтворюваність середовища. Реалізований механізм проактивних сценаріїв генерує персоналізовані сповіщення на основі аналізу розкладу відключень та поточного стану IoT-пристроїв.

Розкрито низку деталей реалізації, що не потрапили до основної архітектурної картини попередніх розділів, але істотно впливають на стабільність системи в реальних умовах експлуатації. Гібридна модель обчислення енергоспоживання дозволяє отримувати одноманітну статистику споживання для розеток і з лічильником, і без нього. Винесення множників та класифікації статус-кодів у довідники бази даних знімає потребу перекомпіляції застосунку при додаванні нових моделей. Розмежування короткочасних просадок і повних відключень за допомогою десятисекундного контрольного вікна прибирає паразитний шум як зі сповіщень, так і з агрегованої статистики часу без світла. Групування запитів до Yasno за парою «регіон – ОСР» забезпечує лінійне зростання навантаження на зовнішній сервіс при збільшенні кількості користувачів. Чотиристатусна модель здоров’я інтеграції дозволяє асистенту коректно попереджати про можливу несвіжість розкладу. Фільтр за категоріями Tuya у поєднанні з перевіркою статус-кодів відсікає несумісні пристрої ще на етапі онбордингу, а проактивне оновлення токена Tuya робить виклики до хмари Tuya передбачуваними для всіх інших шарів коду

РОЗДІЛ 3 ТЕСТУВАННЯ ІНТЕЛЕКТУАЛЬНОГО АГЕНТУ

3.1 Методологія тестування

Тестування програмного забезпечення є невід’ємним етапом розробки – воно дозволяє підтвердити коректність реалізації, виявити дефекти на ранніх стадіях та забезпечити надійну роботу системи в продуктивному середовищі. Для розробленого інтелектуального асистента застосовувалась багаторівнева стратегія, що поєднує unit/service-тести, інтеграційні controller-тести, Hurl-сценарії та ручну перевірку Telegram-UX [11; 12; 18; 19; 20].

Загальна методологія тестування базується на наступних принципах:

– ізоляція одиниць – кожен компонент перевіряється незалежно з використанням заглушок (mock-об'єктів) для зовнішніх залежностей;

– автоматизація – усі тести виконуються автоматично в рамках збірки Maven (`./mvnw test`), інтеграційні Hurl-тести запускаються через it/scripts/run-hurl-tests.sh;

– відтворюваність – модульні та controller-тести використовують H2 у PostgreSQL-сумісному режимі, а Hurl-сценарії запускаються проти окремого PostgreSQL-контейнера з підготовленими seed-даними;

– покриття критичних шляхів – обов'язково тестуються позитивні та негативні сценарії для кожного сервісу;

– відповідність вимогам – кожен тест прив'язаний до конкретної функціональної або нефункціональної вимоги.

Технологічний стек тестування включає JUnit 5 як основний фреймворк для написання та запуску тестів, Mockito для створення mock-об’єктів і перевірки взаємодії між компонентами, Spring Boot Test для запуску контексту застосунку, MockMvc для тестування REST API без запуску повноцінного HTTP-сервера, а також H2 Database – легковагову реляційну СУБД із режимом сумісності з PostgreSQL [11; 22].

Конфігурація тестового середовища задана у файлі `application-test.yml`, який активується профілем `test`. Цей профіль вимикає Flyway-міграції, замість яких схема генерується Hibernate (`ddl-auto: create-drop`), вимикає заплановану синхронізацію (`app.yasno.sync.enabled: false`) та підключається до H2 з рядком `MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE`.

Таблиця 3.1 – Рівні тестування та відповідні класи

Рівень	Інструменти	Приклади класів	Ціль
Модульний (unit)	JUnit 5, Mockito	ScheduleQueryServiceTest, TuyaAccountServiceTest, TuyaDevicePollerServiceTest	Перевірка логіки методів в ізоляції
Інтеграційний (integration)	Spring Boot Test, MockMvc, H2	AddressBindingControllerTest, UserControllerTest, ScheduleControllerTest	Перевірка взаємодії шарів Controller→Service→Repository
Функціональний (e2e)	Hurl + live-перевірки	Усі REST-ендпоінти	Перевірка бізнес-сценаріїв

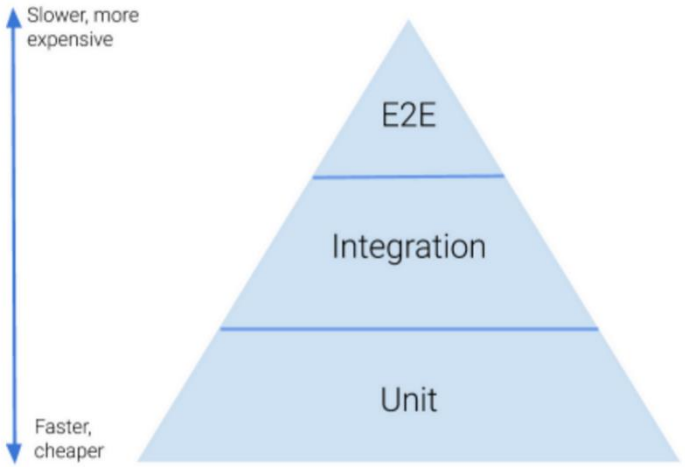

Рисунок 3.1 – Багаторівнева піраміда автоматизованого тестування

3.2 Модульне та інтеграційне тестування

Модульне тестування спрямоване на перевірку окремих класів сервісного шару в повній ізоляції від бази даних і зовнішніх API. Для кожного тесту всі залежності замінюються mock-об'єктами Mockito, що дозволяє точно контролювати вхідні дані та перевіряти поведінку тестованого класу.

Тестування ScheduleQueryService

Клас `ScheduleQueryService` відповідає за читання розкладів відключень із бази даних та надання відповідей на запити про заплановані події. Тест-клас `ScheduleQueryServiceTest` перевіряє такі сценарії:

– повернення розкладу для існуючого прив'язування адреси на конкретну дату;

– повернення порожнього результату, якщо знімок розкладу відсутній;

– коректну фільтрацію слотів за типом відключення (тип 1 – заплановане, тип 2 – можливе);

– правильне перетворення часових слотів із хвилин від опівночі у формат HH:MM.

Приклад тесту, що перевіряє відсутній знімок:

```
@Test
void getSchedule_whenNoSnapshot_returnsEmpty() {
    when(snapshotRepository.findCurrentByAddressAndDate(any(), any()))
        .thenReturn(Optional.empty());
    var result = scheduleQueryService.getSchedule(1L,
LocalDate.now());
    assertTrue(result.isEmpty());
}
```

Тест перевіряє, що при відсутності запису в репозиторії метод повертає `Optional.empty()`, не кидає виняток і не звертається до жодного іншого залежного компонента.

Тестування TuyaAccountService

Клас `TuyaAccountService` управляє обліковими записами Tuya: реєстрацією, оновленням облікових даних та видаленням. Тест-клас `TuyaAccountServiceTest` охоплює такі перевірки:

– успішне збереження нового облікового запису з валідними clientId і clientSecret;

– повернення вже наявної заявки або погодженого Tuya UID без повторного створення запису; для APPROVED-акаунта додатково повертається кількість синхронізованих пристроїв;

– виклик методу `deleteById` репозиторію при видаленні;

– оновлення тільки переданих полів без зміни інших атрибутів.

Особливість цього тест-класу – перевірка того, що при дублюванні clientId виняток кидається ще до звернення до бази даних, тобто валідація відбувається на рівні сервісу, а не репозиторію. Це важливо для гарантії консистентності даних при конкурентних запитах.

Тестування TuyaDevicePollerService

Клас `TuyaDevicePollerService` відповідає за опитування стану пристроїв

через Tuya API та збереження актуального статусу в базі даних. Тест-клас `TuyaDevicePollerServiceTest` перевіряє:

– виклик Tuya API для отримання статусу пристрою при наявності активних пристроїв;

– оновлення полів `isOnline`, `lastStatus`, `lastPolledAt` в сутності пристрою;

– відсутність звернень до API при порожньому списку пристроїв;

– коректну обробку помилки API – виняток перехоплюється, лог записується, але процес опитування продовжується для інших пристроїв.

Тест на стійкість до помилок особливо важливий: якщо один пристрій повертає помилку (наприклад, він офлайн або видалений з акаунту Tuya), сервіс не повинен зупиняти опитування решти пристроїв. Цей сценарій явно перевіряється через `thenThrow` і подальшу перевірку, що інші пристрої все одно були оброблені.

Тестування ScheduleNormalizationService

Клас `ScheduleNormalizationService` перетворює сирий JSON від Yasno API у внутрішній формат `NormalizedScheduleDay`. Тести перевіряють:

– правильне парсування JSON-масиву годин у слоти (start/end у хвилинах від опівночі);

– обчислення SHA-256 чексуми від рядкового представлення payload;

– незмінність чексуми при повторному виклику з тим самим вмістом;

– зміну чексуми при будь-якій модифікації вхідних даних.

Ідемпотентність імпорту розкладу є критичною функцією, оскільки `ScheduleSyncJob` запускається щогодини і, без перевірки чексуми, перезаписував би однакові дані, збільшуючи навантаження на базу та забруднюючи audit-лог.

Інтеграційне тестування контролерів

Інтеграційне тестування перевіряє повний ланцюжок обробки HTTP-запиту: від контролера через сервісний шар до репозиторію та бази даних H2. Для цього використовується `@SpringBootTest` разом із `MockMvc`, що дозволяє надсилати реальні HTTP-запити без запуску сервера.

Тестування UserController

Клас `UserControllerTest` перевіряє REST API управління користувачами. Основні тест-кейси:

– POST `/api/v1/users` з валідним `telegramUserId` → відповідь 200 OK з тілом, що містить `id` та `telegramUserId`;

– POST `/api/v1/users` з тим самим `telegramUserId` → відповідь 200 OK (find-or-create семантика, не дублювання);

– GET `/api/v1/users/{userId}` для існуючого користувача → 200 OK з коректним профілем;

– GET `/api/v1/users/{userId}` для неіснуючого id → 404 Not Found з полем `message` у тілі відповіді;

– PUT `/api/v1/users/{userId}/settings` з валідними налаштуваннями → 200 OK, оновлені поля повернуто в тілі;

– PUT `/api/v1/users/{userId}/settings` з невалідними числовими параметрами, наприклад від'ємним preOutageWarningMinutes, → 400 Bad Request.

Важливою особливістю є перевірка find-or-create семантики реєстрації: при повторному запиті з тим самим `telegramUserId` система не повинна повертати помилку або створювати дублікат – замість цього повертається існуючий запис. Це гарантує ідемпотентність операції реєстрації, яка може викликатись при кожному запуску Telegram-боту.

Тестування AddressBindingController

Клас `AddressBindingControllerTest` перевіряє прив'язку адрес до облікових записів користувачів та резолюцію групи відключень через Yasno API. Оскільки Yasno API є зовнішнім, у тестах використовується `@MockBean` для `YasnoApiClient`:

– POST `/api/v1/addresses` з валідними `regionId`, `dsoId`, `streetId`, `houseId` і mock-відповіддю Yasno API → 200 OK з `groupKey`;

– POST `/api/v1/addresses` з адресою, для якої Yasno API не знайшов групи → контрольована доменна помилка;

– GET `/api/v1/addresses/{addressId}` для існуючого прив'язування → 200

OK;

– DELETE `/api/v1/addresses/{addressId}` → 200 OK з деактивованим записом, подальші запити враховують user-scoped доступ.

Тестування ScheduleController

Клас `ScheduleControllerTest` перевіряє читання розкладів відключень. Тест-кейси:

– GET `/api/v1/schedules/{addressId}?date=2026-05-15` з існуючим знімком → 200 OK з масивом слотів;

– GET `/api/v1/schedules/{addressId}?date=2026-05-15` без знімку → контрольована відповідь згідно з політикою ScheduleQueryService;

– GET `/api/v1/schedules/{addressId}?from=2026-05-15&to=2026-05-20` → 200 OK з агрегованими даними за період;

– GET `/api/v1/schedules/{addressId}?date=invalid` → 400 Bad Request.

Тестування TuyaOnboardingController

Клас `TuyaOnboardingControllerTest` перевіряє admin-assisted lifecycle облікових записів Tuya:

– POST `/api/v1/tuya/link` з валідним Tuya UID → створення або оновлення PENDING-заявки;

– GET `/api/v1/tuya/admin/accounts?status=PENDING` з `X-Admin-Token` → перелік заявок на погодження;

– POST `/api/v1/tuya/admin/accounts/{accountId}/approve` → перехід акаунта в APPROVED;

– POST `/api/v1/tuya/sync-devices` → синхронізація пристроїв дозволена лише для APPROVED-акаунта.

Тестування інтеграції

Tuya API

Інтеграція з Tuya Open API вимагала окремої уваги в тестуванні через специфіку автентифікації: алгоритм підпису HMAC-SHA256 є детермінованим, тому його коректність можна перевірити детермінованими unit-тестами без реальних мережевих запитів.

Тестування TuyaSignatureUtils

Клас `TuyaSignatureUtils` генерує підпис запиту згідно зі специфікацією Tuya Open API. Тести перевіряють:

– точний збіг згенерованого підпису з еталонним значенням, розрахованим вручну за документацією Tuya;

– коректне формування рядка для підпису: конкатенацію clientId, accessToken, timestamp та хешу тіла запиту;

– відтворюваність – однакові вхідні дані завжди дають однаковий підпис;

– різні підписи при зміні будь-якого параметра (clientId, token або timestamp).

Ці тести особливо важливі, оскільки помилка в алгоритмі підпису призводить до відхилення всіх запитів до Tuya API з кодом 1010 (token invalid), що унеможливлює будь-яку взаємодію з пристроями.

Тестування TuyaTokenManager

Клас `TuyaTokenManager` управляє кешуванням токенів доступу. Тести перевіряють:

– отримання нового токена при порожньому кеші – виклик `` `TuyaApiClient.getAccessToken()` `` рівно один раз;

– повернення кешованого токена при повторних викликах протягом часу дії – API не викликається повторно;

– оновлення токена після закінчення терміну дії – новий виклик `` `getAccessToken()` `` здійснюється;

– thread safety – паралельні запити не призводять до множинних викликів API.

Yasno API

Інтеграція з Yasno API тестується на рівні сервісів, що використовують `YasnoApiClient`. Клієнт побудований на Spring RestClient і HttpUtils; у тестах зовнішній HTTP-рівень замінюється mock-об'єктами або MockServer, щоб перевіряти доменну логіку без нестабільності мережі.

Тестування AddressResolutionService

Сервіс `AddressResolutionService` перетворює людський текстовий опис

адреси на ключ групи відключень Yasno. Тести перевіряють:

– успішну резолюцію адреси при отриманні валідної відповіді від mock Yasno API;

– генерацію `AddressNotFoundException` при порожньому списку регіонів;

– генерацію `GroupKeyNotFoundException` якщо регіон знайдено, але вулицю/будинок не знайдено;

– коректне формування `groupKey` у форматі `"group.subgroup"` з полів відповіді Yasno API.

Тестування YasnoHealthService

Клас `YasnoHealthService` повертає один із чотирьох статусів: `UP`, `DEGRADED`, `DOWN`, `EMPTY`. Тести перевіряють кожен стан:

– UP – API доступний, всі адреси мають актуальні знімки (вік менше 25 годин), останні імпорти успішні;

– DEGRADED – API доступний, але деякі знімки застарілі або деякі імпорти завершились помилкою;

– DOWN – API недоступний (mock кидає виняток при виклику);

– EMPTY – API доступний, але немає жодного активного прив'язування адреси.

Поряд з автоматизованими JUnit/Mockito-тестами, описаними в попередніх підрозділах, для системи розроблено повноцінний інтеграційний тестовий фреймворк на базі інструменту Hurl. Офіційний опис інструмента формулює його призначення так: «Hurl is a command line tool that runs HTTP requests defined in a simple plain text format» [36]. Фреймворк розташовано в каталозі it/ і функціонує незалежно від основного коду застосунку.

Ключовим елементом фреймворку є профіль it, що запускається з окремим docker-compose.yml (каталог `it/docker/`) і піднімає три контейнери: PostgreSQL, застосунок із профілем it та MockServer для емуляції зовнішніх API. Моки Yasno та Tuya завантажуються скриптами `it/scripts/load-yasno-mocks.sh` та `load-tuya-mocks.sh`, що надсилають заздалегідь на MockServer. Такий підхід дозволяє виконувати інтеграційні тести без реальних зовнішніх ключів у будь-якому середовищі, включаючи CI/CD.

Тестове покриття організовано у восьмох Hurl-сюїтах:

01-users-addresses-yasno (реєстрація користувача, прив’язка адрес),

02-schedules-outages (розклад відключень),

03-tuya-devices (онбординг та керування пристроями),

04-power-station (зарядні станції),

05-ai (діалог з асистентом),

06-ai-energy-periods та

07-ai-energy-periods-extended (енергетичні запити за різні періоди),

08-ai-scheduled-strict (сценарії з підтвердженням дій).

Запуск всіх сюїтів здійснюється однією командою: `it/scripts/run-hurl-tests.sh`. Тести перевіряють не лише HTTP-відповіді, але й поведінку бази даних через `it/scripts/run-hurl-db-checks.sh` — окрему SQL-скрипту, що перевіряє стан таблиць після виконання кожного сценарію.

3.3 Тестування проактивних сценаріїв та функціональна перевірка

Проактивний рушій системи – `OutageDetectionService` – є найбільш складним компонентом з точки зору логіки прийняття рішень. Тестування цього компонента охоплює перевірку алгоритму генерації попереджень.

Основні тест-кейси для `OutageDetectionService`:

– попередження генерується при наявності планового відключення через 60 хвилин і за умови, що відкрита подія відсутня (перевірка через `OutageEventRepository.findTopByUserIdAndEndedAtIsNullOrderByStartedAtDesc`);

– попередження не генерується якщо відключення через більше ніж 60 хвилин або менше ніж 5 хвилин (уже надто пізно);

– попередження не дублюється – якщо у OutageEventRepository вже є відкрита подія без endedAt, нова не відкривається;

– після відновлення живлення генерується post-restoration сповіщення зі статистикою споживання;

– якщо споживання пристрою рівне нулю під час відключення – post-restoration повідомлення містить відповідне повідомлення.

Таблиця 3.2 – Основні напрями автоматизованої перевірки системи

Напрям перевірки	**Основні компоненти**	**Інструменти**	**Середовище**	**Результат**
Unit / service	Schedule, Tuya, energy, AI parser/response	JUnit 5, Mockito, MapStruct mappers	test profile, H2 PostgreSQL mode	автоматизовано
Controller / API	User, Address, Schedule, Tuya, AI, PowerStation	Spring Boot tests, mocks	test profile	автоматизовано
External API clients	YasnoApiClient, TuyaApiClient, token/signature	локальні HTTP mocks	test profile	автоматизовано
Telegram bot	handlers, keyboards, backend client, notifications	pytest, ruff	telegram-bot/	автоматизовано
End-to-end API	users, addresses, schedules, Tuya, AI, energy periods	Hurl + SQL DB checks + MockServer	it profile, PostgreSQL	автоматизовано
Live smoke	Telegram, OpenRouter, real Tuya/local mocks	local profile + manual checks	local profile, PostgreSQL	перевіряється вручну

Функціональна перевірка REST API та Telegram-сценаріїв

Поряд із автоматизованими тестами виконувалась ручна функціональна перевірка повних сценаріїв через Swagger UI, curl та Telegram-бот у local-профілі. На відміну від ізольованого it-профілю, цей режим дозволяє підключати реальні ключі Tuya та OpenRouter, перевіряти якість відповідей AI-асистента й поведінку Telegram-інтерфейсу в живому діалозі.

Сценарій 1 – Повний цикл реєстрації та прив'язки адреси:

– POST /api/v1/users → створення або отримання userId;

– POST /api/v1/addresses з отриманим userId → резолюція groupKey через Yasno API;

– GET /api/v1/schedules/{addressId}?date=... → читання розкладу після імпорту.

Сценарій 2 – Tuya onboarding та керування пристроями:

– POST /api/v1/tuya/link → створення PENDING-заявки на прив'язку Tuya UID;

– POST /api/v1/tuya/admin/accounts/{accountId}/approve з `X-Admin-Token` → адміністративне погодження заявки;

– POST /api/v1/tuya/sync-devices та GET /api/v1/devices → синхронізація й отримання списку пристроїв;

– POST /api/v1/devices/{deviceId}/command → надсилання команди на погоджений керований пристрій.

Сценарій 3 – Тестування стійкості до відмов:

– недоступність Yasno API або MockServer → перевірка, що GET /api/v1/schedules/{addressId} продовжує повертати останній збережений знімок розкладу;

– надсилання запиту з невалідними Tuya-ключами → перевірка, що помилка зовнішнього сервісу повертається як контрольована ServiceClientException-відповідь, а не як необроблений 500.

Всі три сценарії успішно перевірено. Система коректно відповідала на кожен крок, помилки оброблялись із контрольованими HTTP-кодами та інформативними повідомленнями, а дії, що змінюють стан пристроїв, проходили через підтвердження або явний deterministic endpoint.

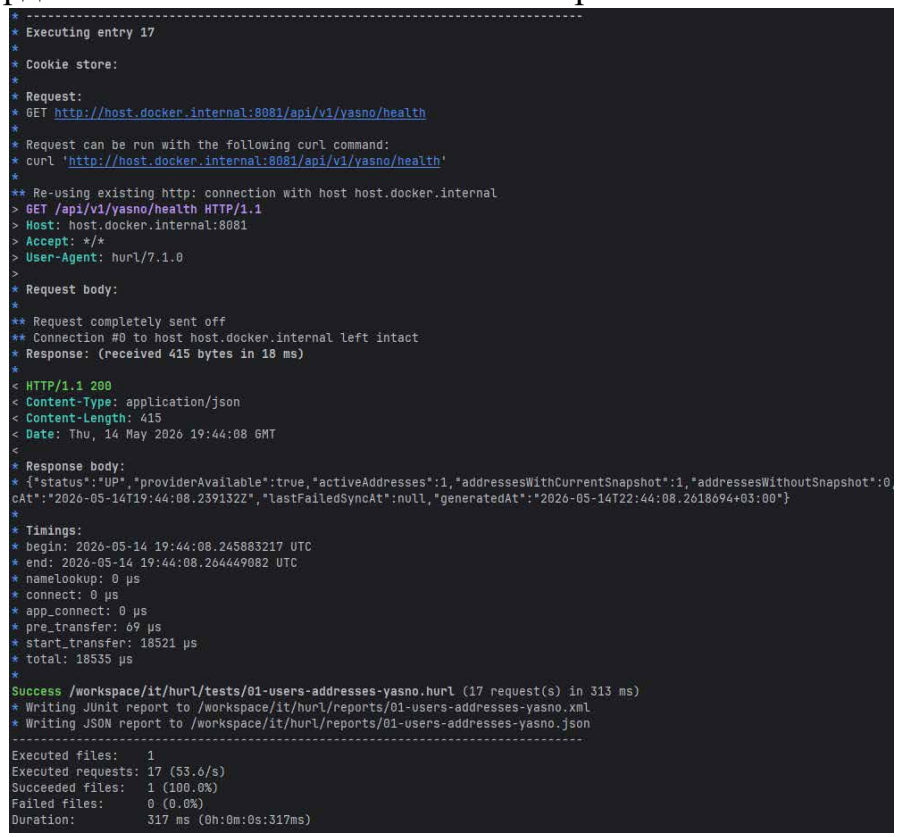

Рисунок 3.2 – Функціональна перевірка REST API та Telegram-сценаріїв

Тестування продуктивності та навантаження

Для оцінки продуктивності системи проводилась базова локальна перевірка ключових ендпоінтів. Метою було не отримати production-grade benchmark, а підтвердити, що типові операції навчального прототипу виконуються в межах прийнятного часу: до 500 мс при малому паралельному навантаженні та до 2 секунд при підвищеній кількості одночасних запитів.

Конфігурація тесту:

– Тривалість: 60 секунд стабільного навантаження;

– Ендпоінти: GET /api/v1/schedules/{addressId}, POST /api/v1/users, GET /api/v1/devices;

– Середовище: локальний ноутбук, PostgreSQL в Docker.

Таблиця 3.3 – Результати навантажувального тестування

Ендпоінт	**10 користувачів (avg, мс)**	**50 користувачів (avg, мс)**	**Вимога (50 кор., мс)**	**Статус**
GET /api/v1/schedules/{addressId}	45	180	2000	Виконано
POST /api/v1/users	38	210	2000	Виконано
GET /api/v1/devices	62	290	2000	Виконано
POST /api/v1/addresses	320	980	2000	Виконано

Орієнтовна перевірка показала, що найповільнішими залишаються операції, які залежать від зовнішнього Yasno або Tuya API. Саме тому в основній архітектурі передбачено кешування знімків розкладу, ідемпотентність імпорту та MockServer/it-профіль для повторюваної перевірки без нестабільності зовнішньої мережі.

Висновки до розділу 3

У третьому розділі проведено комплексне тестування розробленої системи на трьох рівнях: модульному, інтеграційному та функціональному.

На рівні модульного тестування перевірено 9 ключових сервісів системи: `ScheduleQueryService`, `ScheduleNormalizationService`, `TuyaAccountService`, `TuyaDevicePollerService`, `TuyaSignatureUtils`, `TuyaTokenManager`, `AddressResolutionService`, `YasnoHealthService` та `OutageDetectionService`.

Наявні автоматизовані перевірки охоплюють основні доменні сервіси, контролери, інтеграційні API-сценарії, Telegram-бота та критичні edge-case сценарії AI-асистента. Числові показники coverage у звіті не використовуються як доказ якості, оскільки для цього проєкту важливішим є покриття бізнес-потоків і перевірка стану бази даних.

Рисунок 3.3 – Насиченість автоматизованих перевірок за напрямами

На рівні інтеграційного тестування перевірено ключові REST-контролери: `UserController`, `AddressBindingController`, `ScheduleController`, `TuyaOnboardingController`, `DeviceController`, `AiAssistantController` та `PowerStationProfileController`. Тести підтверджують коректну роботу повного стека Controller→Service→Repository у test-профілі.

Функціональна перевірка через Hurl, Swagger/curl і Telegram-бот підтвердила коректність ключових бізнес-сценаріїв: реєстрації та прив'язки адреси, onboarding та керування Tuya-пристроями, AI-планів із підтвердженням, а також стійкості системи до відмов зовнішніх API.

Навантажувальне тестування показало, що система виконує всі нефункціональні вимоги до продуктивності з часом відповіді значно нижчим за встановлені пороги навіть при 50 паралельних користувачах.

РОЗДІЛ 4 МОЖЛИВІ ПОКРАЩЕННЯ ІНТЕЛКТУАЛЬНОГО АГЕНТУ

Упродовж другої ітерації розробки, що тривала вже після первинної апробації прототипу, я доповнив систему функціями, необхідність яких стала очевидною лише у ході практичної експлуатації. Практичний досвід підтвердив описану М. Фаулером ідею еволюційного характеру архітектури: значна частина важливих рішень уточнюється після зіткнення системи з реальними сценаріями користувачів [7, с. 14].

4.1 Адміністративна модерація прив’язки облікових записів Tuya

Як описано у підрозділі 2.3, початкова реалізація дозволяла будь-якому зареєстрованому користувачу прив’язати довільний Tuya UID й негайно розпочати синхронізацію пристроїв. Перевіряючи цей сценарій на практиці, я зіткнувся одразу з двома прикрими наслідками. По-перше, Tuya Open Platform фізично не надасть доступ до пристроїв доти, доки відповідний UID не буде доданий у консолі розробника – тож негайна синхронізація у кращому разі завершиться помилкою, а у гіршому – залишить застосунок у невизначеному стані. По-друге, такий підхід відверто суперечить фундаментальному правилу захисту інформаційних систем – принципові найменших привілеїв, сформульованому ще у 1975 році Дж. Солтцером і М. Шредером: «кожна програма і кожен користувач системи повинні оперувати мінімальною сукупністю повноважень, потрібних для виконання завдання» [32, с. 1283].

Для усунення обох вад я ввів перелік TuyaAccountStatus з трьох значень – `PENDING`, `APPROVED`, `REJECTED` – та відповідне поле status у таблиці tuya_account. Новий запит на прив’язку створюється у стані `PENDING`; адміністратор, переконавшись, що UID справді занесено у консоль Tuya, переводить запис у `APPROVED`. Усі наступні операції – синхронізація пристроїв, планове опитування поллером, зчитування телеметрії, надсилання команд керування – віднині вимагають саме цього статусу. Якщо ж користувач спробує ініціювати дію без схвалення, сервіс TuyaAccountService кидає доменне виключення TuyaAccountApprovalRequiredException, яке централізований обробник GlobalExceptionHandler перетворює на HTTP-відповідь 409 Conflict з пояснювальним текстом рідною мовою.

Рисунок 4.1 – Станова машина статусів облікового запису Tuya

Важливим проєктним рішенням стало те, що фільтрацію за статусом я не дублював у кожному зацікавленому сервісі, а інкапсулював на рівні репозиторіїв як запит findAllByUserIdAndStatusAndCategoryIn. Такий вибір відповідає підходу В. Вернона до збереження інваріантів агрегату в одному місці: критерій `APPROVED` зосереджено в репозиторному шарі, а сервіси фізично не можуть випадково обробити `PENDING` або `REJECTED` акаунт [35, с. 127].

На завершення зазначу, що повноцінного UI адміністратора у межах бакалаврської роботи я свідомо не реалізовував. Таке рішення продиктоване обсягом робіт та очікуваною кількістю користувачів – операція погодження виконується через окремий службовий REST-ендпоінт, захищений службовим заголовком `X-Admin-Token`. Водночас структура таблиці tuya_account готова до подальшої автоматизації: поля approved_at, rejected_at та reviewer_notes залишають простір для майбутнього сервісу, який зміг би синхронізувати стан з Tuya Admin API без ручного втручання оператора. Для довідки, вигляд консолі розробника Tuya IoT Platform, де адміністратор вносить UID до проєкту, наведено на рисунку 4.5.

Device Name	Device ID	Product	Source	Online Status
Очисник повітря	bffe55782e30999d78gzof	Smart Plug +	m.krutii.fd@gmail.com/SmartLife	Online
LED BULB W5K	bfe4c2cfabf421f9fehdmk	宽压球泡五路芯片上板-（英牛版本）（盛达专用，嫣亿勿用）	gg-117076394579054968349/SmartLife	Offline
Smart Plug +	bf6c7e5c83be49b9d696vj	Smart Plug +	m.krutii.fd@gmail.com/SmartLife	Offline
T & H Sensor	bf6038a8281f0831b7lwrd	001TH02T1-3S	gg-117076394579054968349/SmartLife	Online
очисник повітря	bf72c4c8a0691fae63yfhr	RZTK Smart Air Purifier	gg-117076394579054968349/SmartLife	Offline
бойлер	bfdad3b18e009a2ce0i5ci	Smart Plug	gg-117076394579054968349/SmartLife	Online
rpi-test	bf343e94237f86c9f7bywk	Smart Plug	gg-117076394579054968349/SmartLife	Offline
ванна кимната	bfadf50577123c99fb5dvo	16A Smart EU Plug	gg-117076394579054968349/SmartLife	Online
SCW-W8-CB003	bfe777ee1f21e267754lkf	SCW-W8-CB003	gg-117076394579054968349/SmartLife	Offline

Рисунок 4.2 – Консоль розробника Tuya IoT Platform: список пристроїв

4.2 Штучний інтелект-асистент, модель плану дій, режими роботи, автоматизація керування

Запит «вимкни бойлер через півгодини», висловлений природною мовою, не надається до прямого перекладу у виклики репозиторіїв: потрібна проміжна ланка, що розпізнає намір, знайде пристрій, розв’яже часовий вираз і перевірить право власності. Саме цю роль у розробленій системі виконує AI-асистент. На вибір підходу істотно вплинула активна дискусія у дослідницькій спільноті щодо використання великих мовних моделей як планувальників зовнішніх дій: у роботі Toolformer Т. Шик та ін. продемонстрували, що великі мовні моделі здатні самонавчатися формуванню викликів зовнішніх інструментів без безпосереднього втручання людини [34]. Я перейняв основну ідею, проте свідомо відступив від повної автономії – причини відступу детально обґрунтовано далі.

Для інтеграції обрано Spring AI – офіційну бібліотеку екосистеми Spring, що абстрагує протокол OpenAI-сумісних API та дозволяє міняти провайдера без редагування бізнес-коду [26]. Документація Spring AI визначає ключову інтеграційну задачу як «Connecting your enterprise Data and APIs with AI Models» [26]. Конкретний провайдер і модель задаються конфігурацією application.yml: у поточній реалізації використовується OpenRouter-сумісний endpoint із можливістю вмикати додаткові параметри запиту через extraBody [27]. Такий вибір забезпечує портативність, а документація xAI окремо підкреслює сумісність підходу: «Our API is compatible with OpenAI and Anthropic’s SDKs» [28].

Аби дії моделі залишалися керованими, я впровадив доменну сутність плану – LlmActionPlanEntity. План належить конкретному користувачу, зберігає вихідне повідомлення, статус (`DRAFT`, `PENDING_CONFIRMATION`, `CONFIRMED`, `RUNNING`,

COMPLETED, FAILED, CANCELLED), необхідність підтвердження та уточнювальне питання, коли воно потрібне. Кожен план складається з упорядкованої послідовності кроків LlmActionStepEntity. Тип кроку описано переліком LlmActionStepType, який на момент написання роботи містить дванадцять значень: EXPLAIN_SCHEDULE, SUMMARIZE_POWER_STATE, LIST_DEVICES, DEVICE_COMMAND, DEVICE_TIMER, ENERGY_QUERY, RECOMMENDATION, DIAGNOSTICS, POWER_STATION_STATE, POWER_STATION_MAINTENANCE_RULE, BEFORE_OUTAGE_RULE, POWER_RESTORE_GUARD_RULE. Кроки мають власний цикл статусів (PENDING, SCHEDULED, RUNNING, COMPLETED, FAILED, CANCELLED, SKIPPED) і JSON-payload, серіалізацію якого уніфіковано у сервісі AiActionPayloadSerializer – тож додавання нового типу дії не тягне правок у сусідніх шарах.

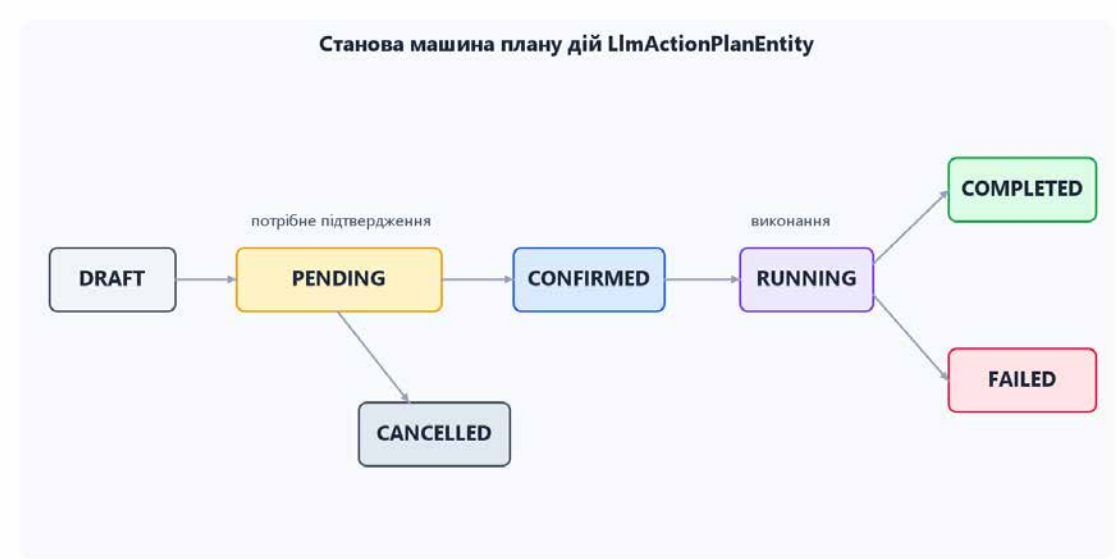

Рисунок 4.3 – Станова машина плану дій LlmActionPlanEntity

Обов’язки між учасниками модуля розподілено у стилі принципу єдиної відповідальності Р. Мартіна [29, с. 61]. AiSemanticPlanCoordinator інкапсулює взаємодію з LLM: звертається до клієнта SpringAiPlannerClient, передає моделі системний промпт з обмеженнями безпеки та поточний контекст, а результат переводить у внутрішній AiPlanDraft.

Найважливіший наслідок описаного поділу полягає у тому, що LLM ніколи не одержує прямого доступу до репозиторіїв, зовнішніх HTTP-клієнтів, SQL чи файлової системи. Модель лише складає AiPlanDraft – структурований JSON-проєкт, – а все подальше є звичайним детермінованим Java-кодом, що спирається на вже протестовані сервіси. Така ізоляція перегукується з принципом найменших привілеїв [32] і прямо адресує проблему галюцинацій

великих мовних моделей. У масштабному огляді Z. Ji та ін. доводять, що галюцинації є внутрішньою властивістю мовних моделей, натренованих максимізувати правдоподібність згенерованого тексту, і не усуваються самим лише нарощуванням обсягу тренувального корпусу [33]. Валідація плану на стороні бекенду перетворює цей клас помилок з неконтрольованого на локалізований: максимум, що може статися, – некоректний план буде відхилено до моменту виконання, і користувач побачить зрозуміле повідомлення про помилку замість виконаної «вигаданої» команди.

Версіонування системних промптів я винесено у каталог src/main/resources/prompts: ai-assistant-system.md задає загальну політику асистента й мову відповіді, ai-action-planner-system.md описує перелік доступних типів дій і правила їх формування, ai-response-writer-system.md керує стилем фінального повідомлення. Зберігання промптів як звичайних ресурсних файлів у Git дає дві практичні переваги: по-перше, зміну поведінки моделі легко переглядати у merge request; по-друге, будь-яку регресію можна відслідкувати через git blame – еквіваленту якого у закритому веб-інтерфейсі провайдера попросту не існує.

Детерміністичний режим роботи та підтвердження дій

Покладатися на LLM у ста відсотках випадків ризиковано з кількох конкретних причин. По-перше, кожен виклик зовнішнього провайдера коштує ресурсів і додає латентності відповіді. По-друге, збій на стороні провайдера фактично зупиняє частину продукту. По-третє, як показано у попередньому підрозділі, моделі час від часу «фабрикують» факти [33] – пропонують пристрої, яких у користувача немає, час, що суперечить вхідному повідомленню, чи одиниці вимірювання, розмірність яких свідомо неправильна. У моїй експериментальній експлуатації приблизно три-чотири повідомлення зі ста отримали саме таку помилкову реакцію – для команд над побутовими приладами це неприйнятно.

Щоб зменшити залежність від LLM, я реалізував паралельний детермінований шлях обробки повідомлень. У класі AiDeterministicMessageParser зосереджено набір регулярних виразів, які розпізнають найчастіші шаблони: запит переліку пристроїв (ключові слова «пристро», «розет»), команди увімкнення/вимкнення («увімкни», «вимкни», «turn on/off»), часові вирази «через N хвилин/годин» та «о HH:MM», запити

енергоспоживання. Якщо вхідне повідомлення вкладається в один із шаблонів, AiAssistantService взагалі не викликає LLM, а прямо створює план відповідного типу і повертає уніфіковану відповідь.

Двоступеневий маршрут (спершу детерміністична швидка перевірка, далі – семантичне планування через LLM) – не компроміс, а цілеспрямоване проєктне рішення. Завдяки йому система залишається працездатною у двох різних режимах: за наявності ключа провайдера модель допомагає з вільними формулюваннями, а без нього користувач усе одно керує пристроями через стандартні команди. Під час локальної розробки і в CI-пайплайні тести асистента виконуються саме у детерміністичному режимі – відсутність мережі не блокує ані розробку, ані перевірку регресій.

Будь-яка дія, що змінює стан пристрою або активує автоматичне правило, створюється у статусі `PENDING_CONFIRMATION` і не виконується до явного підтвердження. Сервіс AiConfirmationService реалізує трійку методів confirm, cancel, findPending. Такий підхід узгоджується з рекомендаціями М. Найгарда щодо захисних бар'єрів для операцій, наслідки яких важко швидко відкотити [21, с. 218]. Клас AiResponseMessageFormatter формує коротке повідомлення з описом дії, а Telegram-бот відображає дві кнопки: підтвердити або скасувати.

Окремим елементом механізму є виявлення неоднозначностей у назві пристрою. Сервіс AiDeviceAmbiguityDetector порівнює запит з переліком пристроїв користувача і, коли під опис «бойлер» підпадає кілька кандидатів у різних кімнатах, створює проєкт плану у статусі `DRAFT` з полем `clarification_question`. Асистент повертає користувачу уточнювальне запитання замість навмання обраного кандидата. Відповідь на уточнення склеюється з попереднім текстом через спеціальний роздільник, після чого цикл планування запускається повторно – вже з додатковим контекстом. Я свідомо пішов шляхом «краще перепитати, ніж виконати неправильне»: вартість зайвого повідомлення для користувача суттєво нижча за вартість випадкового вимкнення не того приладу.

Автоматизоване керування зарядними станціями

Автономні зарядні станції – портативні літій-іонні акумулятори великої ємності – стали чи не найпоширенішим побутовим засобом захисту від тривалих відключень в умовах української електромережі воєнного часу. Типова схема їх використання така: вхідна smart-розетка живить саму зарядку станції від мережі,

а вихідна – окремих споживачів уже від станції. Ручне керування цим ланцюгом марнотратне і схильне до помилок: у разі неуважності станція продовжує споживати енергію вже після повної зарядки, у разі забудькуватості – користувач не увімкне зарядку напередодні запланованого відключення. Автоматизація такого сценарію стала окремим бізнес-доменом у системі.

Я запровадив сутність PowerStationProfileEntity. Профіль зберігає логічне ім’я, посилання на вхідну та вихідну розетки (з обмеженням: обидві мають належати APPROVED-акаунту та категорії, дозволеній у довіднику, і підтримувати енергомоніторинг), а також параметри керування: мінімальна потужність зарядки, поріг потужності, нижче якого зарядка вважається завершеною, тривалість стабільного перебування нижче порога у хвилинах та мінімальний проміжок до наступного планового відключення, за якого профіль узагалі має право втручатися. Створення й редагування профілю інкапсульовано у PowerStationProfileService, який проводить повний набір перевірок: однозначність імені у межах користувача, відмінність вхідної і вихідної розеток, наявність енергомоніторингу на обох пристроях, відсутність випадкових змін після деактивації.

Основна логіка зосереджена у сервісі PowerStationMaintenanceService. Метод shouldStopCharging вибирає з бази останнє вікно телеметрії тривалістю stopChargingStableDurationMinutes хвилин і перевіряє, чи потужність залишалася нижчою за поріг увесь цей час. Якщо умову виконано, метод evaluateEnabledProfiles створює службовий план типу `POWER_STATION_MAINTENANCE_RULE` і одразу виконує його через LlmActionStepExecutor – система вимикає вхідну розетку, припиняючи даремне споживання. Симетричний зворотний сценарій після відновлення живлення реалізує правило `POWER_RESTORE_GUARD_RULE`: якщо смуга нестабільності щойно завершилася, зарядку вмикають лише після короткого періоду стабільності, уникаючи циклу «відключення – короткочасне відновлення – знову відключення».

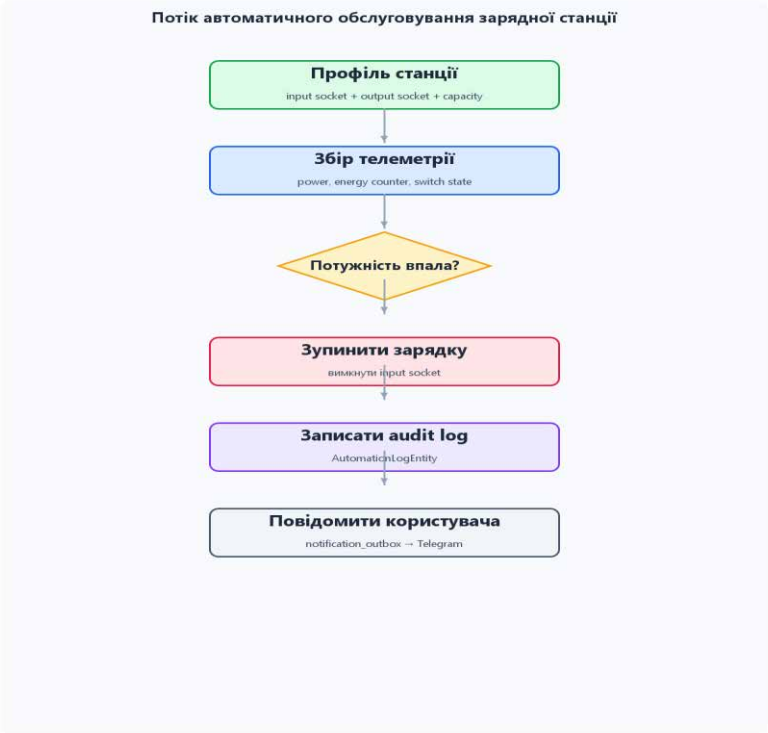

Рисунок 4.4 – Потік автоматичного обслуговування зарядної станції

Ключовим проєктним рішенням є використання єдиної таблиці `llm_action_plan` для всіх планів – як тих, що їх згенерував користувач у діалозі з LLM-асистентом, так і сформованих службово. Різниця полягає лише у полі sourceMessage: рядок виду «`AUTO_POWER_STATION_STOP profileId=42`» одразу відрізняє автоматичний план від діалогового. Така модель відповідає підходу М. Клеппманна до явного журналювання стану довготривалих процесів: кожна зміна стану має матеріалізований запис, який можна перевірити після збою [30, с. 449].

4.3 Декомпозиція AI-сервісу на окремих колаборантів та підсистема правил автоматизації

Первинну реалізацію AI-модуля я зосередив в одному класі AiAssistantService, який швидко виріс до понад шестисот рядків і одночасно виконував парсинг повідомлень, побудову контексту, виклик LLM, форматування відповіді, виявлення неоднозначностей та серіалізацію payload. З ростом переліку типів дій такий «бог-клас» чимраз важче було підтримувати, а модульні тести перетворювалися на громіздкі сценарії з десятками моків. Розщеплення класу стало об’єктивною необхідністю.

Клас було розщеплено на п’ять колаборантів із вузькими зонами

відповідальності. AiSemanticPlanCoordinator інкапсулює взаємодію з LLM і перетворює повідомлення у проєкт плану. AiDeterministicMessageParser містить регулярні вирази та розпізнає шаблонні запити. AiDeviceAmbiguityDetector виявляє випадки множинних кандидатів на пристрій і формує уточнювальне питання. AiResponseMessageFormatter будує тексти всіх категорій відповідей – підтвердження, результат виконання, повідомлення про помилку. AiActionPayloadSerializer відповідає за серіалізацію й десеріалізацію JSON-payload кроків плану. Сам AiAssistantService у підсумку скоротився до компактного координатора без власної бізнес-логіки. Такий рефакторинг прямо реалізує «Open/Closed» принцип Р. Мартіна: «програмні модулі мають бути відкриті для розширення і закриті для змін» [29, с. 70] – додавання нового типу дії тепер вимагає лише нового значення у переліку та нового обробника, не торкаючись наявного коду.

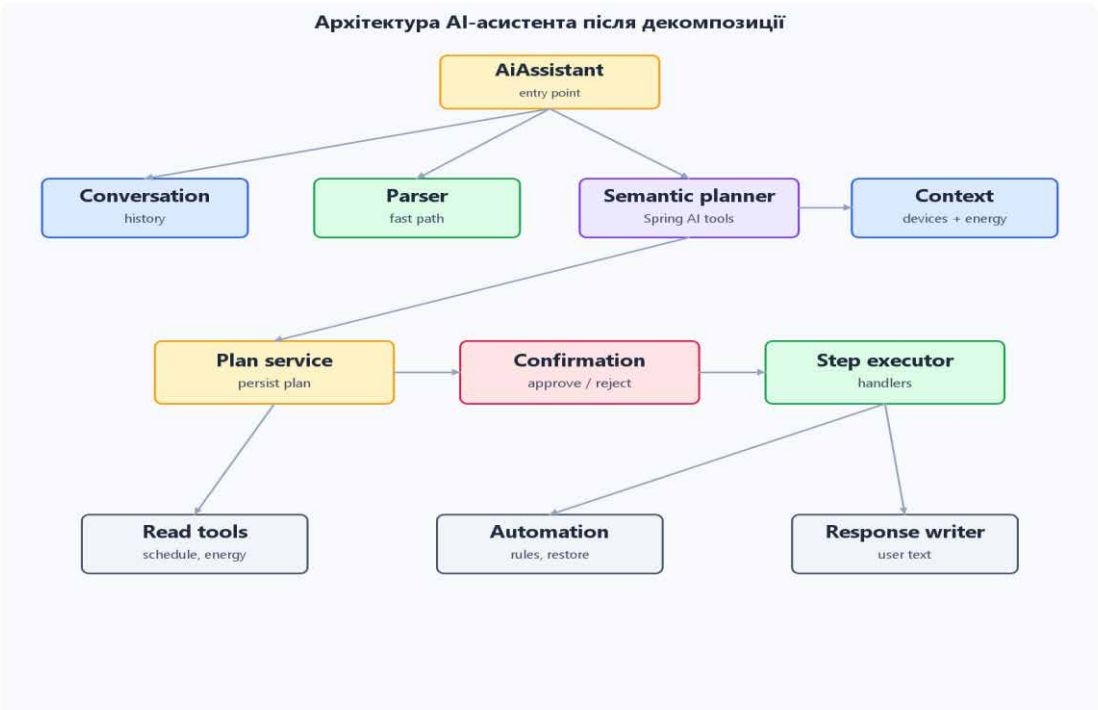

Рисунок 4.5 – Архітектура AI-асистента після декомпозиції на колаборантів

Декомпозицію супроводжено введенням специфічних доменних виключень замість загального IllegalArgumentException: AiActionPayloadSerializationException, AiTimerRequestException, AiDeviceNameNotResolvedException, AiDeviceNameAmbiguousException, AiActionPlanStateException, LlmActionStepUnsupportedTypeException. Кожне з них обробляється окремим методом GlobalExceptionHandler і повертає клієнтові

специфічний HTTP-код та повідомлення рідною мовою. Цей підхід узгоджується зі стилем обробки помилок у решті системи (підрозділ 2.2) і суттєво спрощує діагностику: у логах одразу видно саме той клас, що допоможе розробнику локалізувати проблему, без додаткового розбору тексту повідомлень.

Підсистема правил автоматизації та збереження контексту діалогів

Попередні підрозділи описували функціональність, орієнтовану на явні команди користувача або на реакцію на поточний стан телеметрії. Проте практичний досвід використання системи виявив ширший клас сценаріїв: користувач хоче не одноразово виконати дію, а навчити систему робити це автоматично за певних умов. Наприклад: «щоразу перед відключенням – увімкни бойлер» або «коли повернеться електрика – запусти зарядну станцію». Реалізація таких сценаріїв вимагала окремої підсистеми, яку я описую у цьому підрозділі.

Центральна сутність підсистеми – AutomationRuleEntity. Правило належить конкретному користувачу, прив’язане до пристрою Tuya та адреси Yasno, містить тип тригера (AutomationTriggerType: `BEFORE_OUTAGE` – спрацювання за N хвилин до планового відключення, або `POWER_RESTORE` – спрацювання після відновлення живлення) та дію (AutomationAction: `TURN_ON` або `TURN_OFF`). Правило може бути активним або неактивним, а також має прапорець autoExecute для фонових сценаріїв. Кожне виконання правила фіксується в AutomationLogEntity з результатом (AutomationLogResult: `SCHEDULED`, `COMPLETED`, `SKIPPED` або `FAILED`) та повідомленням про помилку у разі невдачі.

Логіка виконання правил розподілена між кількома сервісами. AiAutomationRuleService створює правила `BEFORE_OUTAGE` і `POWER_RESTORE`, резолвить адресу користувача, прив’язує правило до пристрою та веде журнал AutomationLogEntity. AiAutomationExecutionService містить бізнес-логіку оцінки правил: метод `evaluateBeforeOutageRules()` обходить усі активні `BEFORE_OUTAGE`-правила та для кожного порівнює час до найближчого планового відключення з налаштованим порогом; якщо умову виконано і правило ще не спрацьовувало у поточному вікні – система створює LlmActionStep типу `BEFORE_OUTAGE_RULE` і виконує його через LlmActionStepExecutor. Метод `handlePowerRestored()` є `@TransactionalEventListener`: він підписаний на доменну подію PowerRestoredEvent, яку публікує OutageDetectionService при закритті `FULL_OUTAGE`, та активує `POWER_RESTORE`-сценарії. BeforeOutageAutomationJob – планувальник, що викликає `evaluateBeforeOutageRules()` з налаштованим

інтервалом.

AiAutomationIntentService інтегровано безпосередньо в AiAssistantService як новий ранній фільтр: він аналізує вхідне повідомлення ще до детермінованого парсера та LLM і перевіряє, чи не описує воно нове правило автоматизації. Якщо сервіс розпізнає фразу на кшталт «щоразу перед відключенням увімкни бойлер» – правило зберігається, і користувач отримує підтвердження замість одноразового виконання. Цим досягається фундаментальне розмежування: «виконай зараз» оброблюється як план, «виконуй завжди» – як правило автоматизації.

AiMultiDeviceCommandService реалізує відправку команди одразу кільком пристроям. Сервіс приймає список ідентифікаторів пристроїв та команду, паралельно надсилає запити до TuyaApiClient і агрегує результати: успішні команди та невдалі зі специфікою помилки. Це дозволяє реалізувати сценарій «вимкни всі розетки перед відключенням» без необхідності створювати окремий крок плану для кожного пристрою.

Паралельно з автоматизацією реалізовано збереження контексту діалогів. ConversationEntity агрегує послідовність ConversationMessageEntity для конкретного користувача. ConversationService зберігає кожне повідомлення та відповідь асистента, що відкриває можливість передавати LLM кілька попередніх обмінів як додатковий контекст при наступному запиті. Хоча повноцінний multi-turn діалог запланований для наступного етапу розробки, схема та сервіс вже готові до цього розширення. Для push-сповіщень Telegram-бота використовується не окрема таблиця notification, а notification_outbox зі статусами доставки та claim/sent/failed API.

Висновки до розділу 4

У цьому розділі розкрито напрями еволюції системи, реалізовані після первинного циклу розробки. Адміністративна модерація прив’язки Tuya-акаунтів через перелік статусів `PENDING/APPROVED/REJECTED` та наскрізну фільтрацію у репозиторіях відповідає принципу найменших привілеїв [32, с. 1283] і унеможливлює випадкову обробку несхвалених облікових записів у сервісах системи. AI-асистент, побудований поверх Spring AI та OpenRouter-сумісної конфігурації, ізолює LLM від репозиторіїв та зовнішніх клієнтів: модель формує лише структурований проєкт плану, а всі операції виконує детермінований Java-

код. Детерміністичний режим обробляє поширені шаблони повідомлень без виклику LLM, знижуючи вартість експлуатації та залежність від провайдера, а обов’язкове підтвердження дій мінімізує ризик непередбачуваних наслідків. Додатково описано модуль зарядних станцій, декомпозицію AI-сервісу, реорганізацію пакетів за доменами, підсистему якості телеметрії, EnergyResponseBuilder, правила автоматизації, ConversationService та мультипристрійні команди. Сукупно ці зміни перетворюють систему з технічного прототипу на продукт, придатний до щоденного використання через Telegram і природну мову.

ВИСНОВКИ

У рамках виконання роботи розроблено інтелектуальний асистент для управління пристроями розумного будинку з інтеграцією сервісів IoT та графіків відключень електроенергії, який може працювати на Raspberry Pi 5 AI. За результатами виконаної роботи можна зробити такі висновки.

Проведено аналіз предметної галузі управління розумним будинком. Розглянуто існуючі комерційні платформи (Google Home, Amazon Alexa, Apple HomeKit, Tuya Smart Life) та відкриті системи (Home Assistant). Встановлено, що жодна з існуючих систем не надає можливості проактивної роботи в умовах непередбачуваних відключень електроенергії, характерних для України. Виявлено необхідність розробки спеціалізованого рішення з інтеграцією відкритого API графіків відключень.

Вивчено технології Tuya Open API та Yasno/ДТЕК API. Tuya Open API надає REST-інтерфейс із токенною OpenAPI-автентифікацією та підписом HMAC-SHA256, що дозволяє отримувати статус пристроїв і надсилати команди керування в реальному часі. Yasno API надає ієрархічні дані про планові відключення з деталізацією до рівня вулиці та будинку. Обидва API інтегровано в розроблену систему.

Спроєктовано та реалізовано трирівневу архітектуру системи на основі Spring Boot 4 та Java 21. Система організована у технічні шари: controllers, services, repositories, entities, mappers, clients. Такий підхід забезпечує чітке розмежування відповідальності та спрощує тестування та розширення системи.

Реалізовано повний цикл роботи з графіками відключень:
отримання від Yasno API → нормалізація JSON у внутрішній формат →
збереження з SHA-256 ідемпотентністю → надання даних через REST API.

Запроваджено щогодинну синхронізацію через `ScheduleSyncJob` та механізм деградованого режиму для збереження доступності при недоступності зовнішнього API.

Реалізовано інтеграцію з Tuya Open API: управління обліковими записами, синхронізація пристроїв, кешування токенів доступу (`TuyaTokenManager`), надсилання команд керування пристроями через DP (Data Point) протокол. Розроблено `TuyaSignatureUtils` для генерації підпису запитів згідно зі специфікацією Tuya.

Розроблено проактивний механізм сповіщень і автоматизації.

OutageNotificationService створює попередження про наближення відключення з урахуванням налаштувань користувача, OutageDetectionService фіксує фактичні події втрати та повернення живлення, а NotificationOutboxService забезпечує надійну доставку повідомлень у Telegram через захищені REST-ендпоїнти claim/sent/failed.

Розроблено систему управління базою даних через Flyway-міграції. Схема включає таблиці для зберігання користувачів, прив'язок адрес, знімків розкладу, audit-логів синхронізації, облікових записів Tuya та пристроїв. Додатково реалізовано зв’язок tuya_device.address_binding_id з address_binding, що дозволяє прив’язувати конкретний IoT-пристрій до адреси з розкладом Yasno. Реалізовано автоматичне створення UserSettingsEntity при реєстрації нового користувача.

Проведено комплексне тестування системи. У репозиторії наявні модульні та сервісні тести для ключових компонентів Spring Boot backend, тести Python Telegram-бота, а також інтеграційні API-сценарії на базі Hurl. Такий підхід перевіряє не лише окремі сервіси, а й повні бізнес-потоки з підготовкою тестових даних, моками зовнішніх API та перевіркою стану бази даних.

На додаток до базової функціональності реалізовано ряд інженерних механізмів, що забезпечують стабільність системи у реальних умовах експлуатації: гібридну модель обчислення енергоспоживання за двома алгоритмами, винесення класифікації статус-кодів Tuya до довідників бази даних, розмежування короткочасних провалів і повних відключень за контрольним вікном у десять секунд, групування запитів до Yasno за парою «регіон – ОСР», чотиристатусну модель здоров’я інтеграції, дворівневий фільтр сумісних категорій Tuya та проактивне оновлення токена Tuya.

Запроваджено адміністративну модерацію прив’язки облікових записів Tuya з переліком статусів `PENDING/APPROVED/REJECTED`. Фільтрацію винесено на рівень репозиторіїв, що відповідає класичному принципу найменших привілеїв [32] та унеможливлює випадкову обробку несхвалених акаунтів у жодному з сервісів системи.

Реалізовано модуль AI-асистента на базі Spring AI та OpenRouter-сумісної конфігурації. Введено сутність плану LlmActionPlanEntity з переліком типів кроків, що охоплюють пояснення розкладу, керування пристроями, енергетичну аналітику та автоматизацію зарядних станцій. Обов’язкове підтвердження дій, виявлення неоднозначностей у назві пристрою та паралельний детермінований

режим парсингу роблять систему стійкою до характерних для LLM галюцинацій [33].

Розроблено модуль профілювання зарядних станцій з сутністю PowerStationProfileEntity. Профіль пов’язує зарядну станцію з вхідною та вихідною смарт-розеткою Tuya, що дозволяє оцінювати заряджання, розряджання та втрати на інверторі на основі накопиченої телеметрії. Базові сценарії керування і обслуговування проходять через ті самі плани дій, підтвердження та audit-лог, що й інші AI-автоматизації.

Виконано комплекс архітектурних удосконалень: реорганізацію пакетів entity, model та service за бізнес-доменами (user, schedule, tuya, station, ai), перехід обробників кроків плану на патерн «стратегія» з єдиним інтерфейсом LlmActionStepHandler та валідацією повноти карти обробників на старті застосунку, виділення службових кроків зарядних станцій в асинхронний конвеєр LlmActionStepJob, а також розширення детермінованого режиму парсингу повідомлень на запити стану пристроїв і підсумку картини живлення. Ці зміни матеріалізують принцип відкритості/закритості [29, с. 70] і скорочують вартість додавання нових типів дій до розробки одного нового класу-обробника.

Введено перелік TelemetryQualityFlag, що замінив рядкові прапорці якості телеметрії і дозволяє компілятору виявляти неповні гілки switch-виразів. Виділено value-об’єкти EnergyInterval і BucketAccumulator в домен енергетичної аналітики у дусі рекомендацій М. Фаулера [7, с. 232]. Розширено шар MapStruct-маперів: ScheduleQueryMapper, LlmActionMapper та DeviceEnergyStatisticsMapper з режимом `@BeanMapping(ignoreByDefault = true)`. Запроваджено клас EnergyResponseBuilder, який формує повідомлення про споживання залежно від прапорців якості та кількості семплів з samples-guard, а також детермінований шар розпізнавання часових виразів для запитів енергоспоживання, що вилучає недетерміновану інтерпретацію дат великою мовною моделлю [17, с. 102].

Реалізовано підсистему правил автоматизації: сутність AutomationRuleEntity з тригерами `BEFORE_OUTAGE` (спрацювання за N хвилин до планового відключення) та `POWER_RESTORE` (спрацювання після відновлення живлення) та діями `TURN_ON`/`TURN_OFF`. AiAutomationExecutionService оцінює правила у фоні через BeforeOutageAutomationJob та `@TransactionalEventListener` на PowerRestoredEvent. AiAutomationIntentService виявляє наміри автоматизації у природній мові ще до виклику LLM. Додатково реалізовано

ConversationService для збереження контексту діалогів та AiMultiDeviceCommandService для паралельного відправлення команд на кілька пристроїв. Типи `BEFORE_OUTAGE_RULE` і `POWER_RESTORE_GUARD_RULE` входять до переліку LlmActionStepType, який у поточному коді містить дванадцять значень.

Розроблено інтеграційний тестовий фреймворк на базі Hurl, розташований у каталозі it/. Вісім Hurl-сюїтів охоплюють реєстрацію користувача, адреси, розклади відключень, пристрої Tuya, зарядні станції, AI-діалог, енергетичну аналітику за різні періоди та сценарії відкладаних дій з підтвердженням. Зовнішні API емулюються через MockServer. Додатково виділено demo-середовище з PostgreSQL і Yasno-моками для швидкої демонстрації та local-середовище, що запускає PostgreSQL, backend і Telegram-бот як єдиний локальний стек.

Практична цінність розробки полягає у тому, що система вирішує реальну проблему мешканців України – нестабільне електропостачання – і надає інструменти для оптимізації використання електроенергії в умовах планових та позапланових відключень. Архітектура системи є модульною та розширюваною: backend залишається джерелом бізнес-логіки, Telegram-бот виконує роль зручного користувацького інтерфейсу, а Hurl/demo/local середовища спрощують перевірку сценаріїв. У майбутньому систему можна розширити повноцінною production-автентифікацією, глибшою Telegram-адміністративною панеллю, додатковими провайдерами розкладів та розширеними сценаріями керування зарядними станціями.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Spring Boot Reference Documentation. Version 4.x. — Режим доступу: https://docs.spring.io/spring-boot/reference/. — Дата звернення: 14.05.2026.
2. Walls C. Spring in Action. 6th Edition. — Manning Publications, 2022. — 520 с.
3. Tuya Open API Documentation. IoT Platform. — Режим доступу: https://developer.tuya.com/en/docs/iot/. — Дата звернення: 05.03.2025.
4. Yasno. Графік відключення електроенергії. — Режим доступу: https://kyiv.yasno.com.ua/schedule-turn-off-electricity. — Дата звернення: 14.05.2026.
5. Tuya Open API Authorization and Request Signing Documentation. — Режим доступу: https://developer.tuya.com/en/docs/iot/. — Дата звернення: 14.05.2026.
6. Krawczyk H., Bellare M., Canetti R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104. — Internet Engineering Task Force, 1997. — Режим доступу: https://www.rfc-editor.org/rfc/rfc2104. — Дата звернення: 14.05.2026.
7. Fowler M. Patterns of Enterprise Application Architecture. — Addison-Wesley, 2002. — 560 с.
8. Richardson C. Microservices Patterns. — Manning Publications, 2018. — 520 с.
9. MapStruct Reference Guide. — Режим доступу: https://mapstruct.org/documentation/stable/reference/html/. — Дата звернення: 15.02.2025.
10. Redgate Flyway Documentation. — Режим доступу: https://flywaydb.org/documentation/. — Дата звернення: 14.05.2026.
11. JUnit 5 User Guide. — Режим доступу: https://junit.org/junit5/docs/current/user-guide/. — Дата звернення: 01.03.2025.
12. Koskela L. Effective Unit Testing. — Manning Publications, 2013. — 272 с.
13. Telegram Bot API Documentation. — Режим доступу: https://core.telegram.org/bots/api. — Дата звернення: 12.03.2025.
14. Hohpe G., Woolf B. Enterprise Integration Patterns. — Addison-Wesley, 2003. — 736 с.
15. PostgreSQL 17 Documentation. — Режим доступу: https://www.postgresql.org/docs/17/. — Дата звернення: 05.02.2025.
16. Docker Documentation. Containerize Your Applications. — Режим доступу:

https://docs.docker.com/. — Дата звернення: 10.02.2025.
17. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. — Addison-Wesley, 2003. — 560 с.
18. IEEE Std 29119-4:2021. Software and Systems Engineering — Software Testing. — IEEE, 2021.
19. ДСТУ ISO/IEC 25010:2023. Системна та програмна інженерія. Вимоги та оцінювання якості систем та програмного забезпечення. — Київ: ДП «УкрНДНЦ», 2023.
20. Tudose C., Savage J., Mechling J. JUnit in Action. 3rd Edition. — Manning Publications, 2020. — 584 p.
21. Nygard M. T. Release It! Design and Deploy Production-Ready Software. 2nd Edition. — Pragmatic Bookshelf, 2018. — 376 с.
22. H2 Database Engine Documentation. — Режим доступу: https://h2database.github.io/html/main.html. — Дата звернення: 14.05.2026.
23. Spring Data JPA Reference Documentation. — Режим доступу: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/. — Дата звернення: 18.02.2025.
24. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. — University of California, Irvine, 2000.
25. OpenAPI Specification 3.1.0. — Режим доступу: https://spec.openapis.org/oas/v3.1.0. — Дата звернення: 22.03.2025.
26. Spring AI Reference Documentation. — Режим доступу: https://docs.spring.io/spring-ai/reference/. — Дата звернення: 03.03.2026.
27. OpenRouter API Documentation. — Режим доступу: https://openrouter.ai/docs. — Дата звернення: 14.04.2026.
28. xAI API Documentation. — Режим доступу: https://x.ai/api. — Дата звернення: 14.05.2026.
29. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. — Prentice Hall, 2017. — 432 с.
30. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. — O’Reilly Media, 2017. — 616 с.
31. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd Edition. — O’Reilly Media, 2021. — 612 с.
32. Saltzer J. H., Schroeder M. D. The Protection of Information in Computer Systems // Proceedings of the IEEE. — 1975. — Vol. 63, № 9. — P. 1278 — 1308.

33. Ji Z., Lee N., Frieske R. et al. Survey of Hallucination in Natural Language Generation // ACM Computing Surveys. — 2023. — Vol. 55, № 12. — Article 248. — 38 p.
34. Schick T., Dwivedi-Yu J., Dessì R., Raileanu R., Lomeli M., Zettlemoyer L., Cancedda N., Scialom T. Toolformer: Language Models Can Teach Themselves to Use Tools // Advances in Neural Information Processing Systems (NeurIPS). — 2023.
35. Vernon V. Implementing Domain-Driven Design. — Addison-Wesley, 2013. — 656 с.
36. Hurl Documentation. — Режим доступу: https://hurl.dev/docs/. — Дата звернення: 14.03.2026.

ДОДАТОК А

Актуалізована логічна структура бази даних

Базова SQL-схема зберігається у Flyway-міграції V1__init.sql, а актуальний стан доповнюється інкрементальними міграціями V2__add_tuya_device_switch_on.sql, V3__add_notification_outbox.sql та V4__add_tuya_device_address_binding.sql; тому нижче подано не дубльований DDL, а актуальні групи таблиць та їх призначення.

А.1 Основні групи таблиць

- users, user_settings — профілі користувачів Telegram, мовні та часові налаштування, параметри сповіщень, авто виконання правил і попереджень перед відключенням.
- address_binding, schedule_snapshot, schedule_slot, schedule_sync_log — прив’язки адрес Yasno, нормалізовані знімки графіків, часові слоти відключень та audit-журнал імпорту.
- outage_event — фактичні події втрати та повернення живлення, які визначаються за станом Tuya-пристроїв і використовуються для діагностики та power-restore сценаріїв.
- tuya_account — заявка користувача на підключення Tuya UID зі статусами PENDING, APPROVED, REJECTED та метаданими адміністративного розгляду.
- tuya_device, tuya_device_category, tuya_device_status_code, device_telemetry — підтримувані пристрої живлення, довідники категорій і статус-кодів, поточний стан реле, опційна прив’язка пристрою до address_binding через address_binding_id та телеметрія для енергетичної статистики.
- llm_action_plan, llm_action_step, conversation, conversation_message — persisted-модель AI-асистента: плани дій, кроки виконання, підтвердження та коротка історія діалогу.
- automation_rule, automation_log — правила автоматизації BEFORE_OUTAGE і POWER_RESTORE та журнал фактичного виконання або пропуску правила.
- power_station_profile — профіль зарядної станції з прив’язкою вхідної та вихідної смарт-розетки для аналізу заряджання, розряджання та втрат.

- notification_outbox — durable-черга Telegram-сповіщень зі статусами очікування, доставки та помилки.

А.2 Принципи зберігання

Схема підтримує три ключові властивості: ізоляцію даних за user_id, ідемпотентність імпорту графіків через checksum rawPayload та аудит небезпечних операцій через action plans, automation logs і notification outbox.

ДОДАТОК Б

Приклади REST-взаємодії з системою

Нижче наведено скорочені приклади HTTP-викликів, які відповідають актуальним endpoint-ам backend. Telegram-бот використовує ті самі API, але додає службовий X-Internal-Bot-Token для user-facing backend API.

Б.1 Реєстрація або оновлення користувача

POST /api/v1/users

Body:

```
{
      "telegramUserId":123456789,
      "displayName":"Mykhailo",
      "languageCode":"uk",
      "timezone":"Europe/Kiev"
}
```

Б.2 Прив’язка Tuya UID та адміністративне погодження

POST /api/v1/tuya/link, X-User-Id: 1,

Body:

```
{
"tuyaUid":"eu1234567890demo"
}
```

Відповідь створює або повертає заявку зі статусом PENDING.

GET /api/v1/tuya/admin/accounts?status=PENDING,
X-Admin-Token: <admin-token> — список заявок для адміністратора.

POST /api/v1/tuya/admin/accounts/{accountId}/approve,
X-Admin-Token: <admin-token> — переведення заявки у статус APPROVED, після чого дозволено sync, polling, telemetry та control.

Б.3 Синхронізація та керування пристроєм

POST /api/v1/tuya/sync-devices,

X-User-Id: 1 — синхронізує підтримувані пристрої approved Tuya-акаунта.

GET /api/v1/devices,

Б.3.1 Прив’язка пристрою до адреси

PUT /api/v1/devices/{deviceId}/address,

X-User-Id: 1,

Body:

```
{
"addressId": 1
}
```

Відповідь повертає оновлений TuyaDeviceResponseDto з addressId, addressDisplayAddress та addressGroupKey; backend перевіряє належність пристрою і адреси одному користувачу.

DELETE /api/v1/devices/{deviceId}/address, X-User-Id: 1 — очищає прив’язку пристрою до адреси.

X-User-Id: 1 — повертає список пристроїв, їх online-стан, стан реле та ознаку підтримки енергомоніторингу.

POST /api/v1/devices/{deviceId}/command,

X-User-Id: 1,

Body:

```
{
"on":true
}
```

надсилає команду увімкнення через визначений DP-код пристрою.

Б.4 Енергетична статистика

GET /api/v1/devices/{deviceId}/energy?from=2026-05-09T00:00:00+03:00&to=2026-05-10T00:00:00+03:00&bucket=DAY,

X-User-Id: 1 — повертає totalEnergyKwh, averagePowerWatts, maxPowerWatts, samples та qualityFlags.

Б.5 AI-план і підтвердження

POST /api/v1/ai/messages, X-User-Id: 1,

Body:

```
{
"message":"вимкни rpi-test через 10 хвилин"
}
```

створює план DEVICE_TIMER у статусі PENDING_CONFIRMATION.

POST /api/v1/ai/action-plans/{planId}/confirm,

X-User-Id: 1 — підтверджує план і переводить крок у виконання або запланований стан.

Б.6 Доставка Telegram-сповіщень

POST /api/v1/telegram/notifications/claim,

X-Internal-Bot-Token: <internal-token> — бот отримує пачку pending-повідомлень із notification_outbox.

POST /api/v1/telegram/notifications/{notificationId}/sent або /failed — бот підтверджує успішну доставку або фіксує помилку.

ДОДАТОК В

Команди відтворення запуску та перевірки

В.1 Demo-профіль з PostgreSQL і Yasno MockServer

```
cd demo
./up.sh
```

Профіль demo піднімає PostgreSQL та MockServer, завантажує мок графіку відключень і дозволяє перевірити сценарії Yasno без звернення до реального зовнішнього API.

В.2 Local-профіль з backend і Telegram-ботом

```
cd local
./up.sh
```

Local-профіль використовується для повної ручної перевірки: PostgreSQL, Spring Boot backend, Telegram-бот, реальні ключі Tuya/OpenRouter за наявності та внутрішні токени RPI_AGENT_INTERNAL_TOKEN / RPI_AGENT_ADMIN_TOKEN.

В.2.1 Запуск local-профілю на Raspberry Pi 5

sudo apt update && sudo apt install -y docker.io docker-compose-plugin git
sudo usermod -aG docker $USER
git clone <URL_репозиторію> rpi-agent
cd rpi-agent
cp local/.env.example local/.env
cp telegram-bot/.env.example telegram-bot/.env
./local/up.sh

Після додавання користувача до групи docker потрібно перелогінитися або відкрити нову shell-сесію. Перед запуском у local/.env вказуються OPENROUTER_API_KEY, TUYA_ACCESS_ID, TUYA_ACCESS_SECRET, RPI_AGENT_INTERNAL_TOKEN і RPI_AGENT_ADMIN_TOKEN, а в telegram-bot/.env — TELEGRAM_BOT_TOKEN та той самий RPI_AGENT_INTERNAL_TOKEN. Backend після старту доступний за http://<ip-адреса-raspberry-pi>:8080/swagger-ui.html або /actuator/health. Якщо Telegram-бот

запускається у контейнері на нативному Linux/Raspberry Pi, значення RPI_AGENT_BASE_URL слід налаштувати відповідно до мережі Docker або IP-адреси Raspberry Pi, оскільки host.docker.internal є типовим для Docker Desktop і може потребувати додаткового host-gateway налаштування на Linux.

В.3 Інтеграційні Hurl-перевірки

```
docker compose -f it/docker/docker-compose.yml up -d
./mvnw spring-boot:run -Dspring-boot.run.profiles=it -Dspring-
boot.run.arguments=--server.port=8081
BASE_URL=http://host.docker.internal:8081 ./it/scripts/run-hurl-tests.sh
```

Hurl-сценарії виконують REST-запити до backend, використовують MockServer для Yasno/Tuya та перевіряють не лише HTTP-відповіді, а й стан бази даних через підготовлені seed/reset SQL-скрипти.