МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

**РЯБОШАПКА ОЛЕКСАНДР ВАДИМОВИЧ**

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____О. В. Зелінська
«____»_____2025р.

# СИСТЕМА МОНІТОРИНГУ ТЕЛЕФОННИХ ДЗВІНКІВ У БІЗНЕС-СЕРЕДОВИЩІ

Спеціальність 122 Комп’ютерні науки
**Кваліфікаційна (бакалаврська) робота**

Керівник:
Р. М. Бабаков, професор кафедри
інформаційних технологій,
д. т. н., доцент

_____

Оцінка:___/ ___/ _____
(бали за шкалою ЄКТS/за національно шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

## АНОТАЦІЯ

**Рябошапка О.В. Система моніторингу телефонних дзвінків у бізнес-середовищі.** Спеціальність 122 «Комп'ютерні науки» Донецький національний університет імені Василя Стуса, Вінниця, 2025, 48 с.

У кваліфікаційній (бакалаврській) роботі досліджено процес створення мобільної системи моніторингу телефонних дзвінків, орієнтованої на використання у малих та середніх підприємствах. Розроблено Android-додаток, який фіксує голосову активність, зберігає історію дзвінків локально та надає доступ до неї через вбудований HTTP-сервер без використання хмарної інфраструктури. У роботі проаналізовано особливості телефонної комунікації в бізнесі, обґрунтовано вибір архітектурних рішень, описано ключові компоненти системи, результати тестування та можливі сценарії практичного застосування.

**Ключові слова:** моніторинг дзвінків, Android-додаток, локальний сервер, бізнес-комунікація, мобільне програмування, Ktor, Jetpack Compose.

Табл. 3. Рис. 9. Бібліограф.: 35 найм.

## ABSTRACT

**Riaboshapka O.V. A System for Monitoring Phone Calls in a Business Environment.** Specialty 122 "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia, 2025, 48 p.

This bachelor's thesis presents the development of a mobile system for monitoring phone calls, aimed at small and medium-sized businesses. The implemented Android application captures call activity, stores call history locally, and provides access via an embedded HTTP server without relying on cloud infrastructure. The research explores the role of voice communication in business, justifies the chosen architectural solutions, describes the system's core components, presents testing results, and outlines practical use cases.

Keywords: call monitoring, Android app, local server, business communication, mobile development, Ktor, Jetpack Compose.

Table 3. Fig. 9. Bibliography: 35

# ЗМІСТ

# ВСТУП

**Актуальність теми дослідження.** У сучасному бізнес-середовищі ефективна комунікація з клієнтами та партнерами є критично важливим чинником конкурентоспроможності підприємств. Телефонні дзвінки залишаються одним із ключових каналів взаємодії, особливо в сферах продажу, підтримки, логістики та сервісного обслуговування. За відсутності належного контролю така комунікація може виходити з-під уваги керівництва, що негативно впливає на якість обслуговування, прозорість роботи персоналу та фінансові результати.

З розвитком мобільних технологій з’явилася можливість впровадження локальних систем моніторингу дзвінків, які не потребують хмарної інфраструктури чи складного налаштування. Це відкриває перспективи для створення автономних мобільних рішень, здатних забезпечити збір, обробку та передачу інформації про телефонні дзвінки безпосередньо з пристрою.

У даній роботі розглянуто створення Android-додатку, який фіксує виклики в реальному часі, зберігає інформацію локально, а також запускає вбудований HTTP-сервер, що дозволяє переглядати інформацію про дзвінки в локальній мережі через звичайний браузер. Рішення орієнтоване на використання у відділах обслуговування клієнтів, невеликих кол-центрах, сервісних службах та компаніях, яким потрібен простий та ефективний інструмент контролю.

**Мета дослідження** – дослідити теоретичні засади моніторингу телефонних дзвінків у бізнес-середовищі та розробити Android-додаток для їх збору, локального збереження і доступу через внутрішню мережу підприємства.

**Завдання дослідження:**

1. Проаналізувати сутність та роль телефонної комунікації у сучасному бізнесі.
2. Розглянути існуючі системи моніторингу викликів та підходи до їх реалізації.

3. Проаналізувати інструменти та технології розробки мобільних додатків для Android.

4. Визначити вимоги до системи моніторингу дзвінків у бізнес-середовищі.

5. Розробити архітектуру та дизайн Android-додатку відповідно до заданих вимог.

6. Реалізувати функціональність моніторингу, зберігання та обробки даних про дзвінки.

7. Реалізувати локальний сервер на пристрої для надання доступу до зібраної інформації.

8. Провести тестування додатку та оцінити його ефективність у практичному застосуванні.

9. Визначити перспективи розвитку системи та можливості її масштабування.

**Об’єкт дослідження** – мобільна система збору, обробки та відображення даних про телефонні виклики у бізнес-середовищі.

**Предмет дослідження** – процес моніторингу телефонних дзвінків за допомогою мобільного додатку з локальним сервером у контексті оптимізації бізнес-процесів.

**Методи дослідження.** Аналіз наукових джерел та існуючих технічних рішень у сфері мобільного моніторингу; системний аналіз вимог до Android-додатку; розробка програмного забезпечення на базі Android SDK з використанням Jetpack Compose, Ktor, DataStore, Hilt; тестування на реальному пристрої.

**Теоретичне та практичне значення одержаних результатів.** Результати дослідження мають як теоретичну, так і прикладну цінність для побудови ефективних систем мобільного моніторингу комунікацій у межах підприємства. Запропонований додаток може бути використаний малими та середніми підприємствами для забезпечення прозорості комунікацій, підвищення якості клієнтського обслуговування та контролю телефонної активності персоналу.

Рішення має потенціал для подальшого розвитку, зокрема шляхом додавання вебінтерфейсу, аналітичного модуля або інтеграції з внутрішніми CRM-системами.

**Структура (кваліфікаційної) бакалаврської роботи.** Бакалаврська робота складається зі вступу, трьох основних розділів та одинадцятьох підрозділів, висновку, списку використаних посилань та додатків (35 найменувань). Загальний обсяг роботи – 48 сторінок.

# РОЗДІЛ 1
# ТЕОРЕТИЧНІ ОСНОВИ СТВОРЕННЯ СИСТЕМ МОБІЛЬНОГО МОНІТОРИНГУ ДЗВІНКІВ У БІЗНЕС-СЕРЕДОВИЩІ

## 1.1. Поняття та роль моніторингу комунікацій у бізнесі

У добу цифрової трансформації, коли компанії дедалі активніше застосовують автоматизовані інструменти управління, електронну пошту, чат-боти та месенджери, саме телефонний дзвінок залишається одним з найефективніших способів комунікації. Його ключова перевага полягає у можливості встановлення швидкого та емоційно-забарвленого контакту, що особливо важливо у сферах, де персоналізоване обслуговування клієнта є критичним чинником успіху. Телефонна розмова дозволяє не лише оперативно вирішити нагальні питання, а й точніше з’ясувати деталі, які важко передати через текстові повідомлення, а також краще зрозуміти емоційний стан співрозмовника.

У порівнянні з електронною поштою або чатами, телефонна розмова значно знижує ризик непорозумінь, пов’язаних із неоднозначним тлумаченням тексту. Голосове спілкування дозволяє одразу уточнити неясності, уникнути затримок у відповідях і краще контролювати тональність розмови. Зокрема, це актуально у критичних ситуаціях, коли навіть незначна емоційна похибка може вплинути на рішення клієнта або внутрішній клімат у команді.

Телефонна розмова – це специфічна форма усного мовлення, яка має низку особливостей, зумовлених як технічними, так і комунікативними факторами. На відміну від безпосереднього спілкування у візуальній присутності співрозмовників, телефонна комунікація позбавлена можливості використовувати невербальні засоби вираження – жести, міміку, погляд, зоровий контакт чи зміну положення тіла. Ці візуальні сигнали відіграють важливу роль у традиційній міжособистісній взаємодії, адже сприяють кращому розумінню емоційного стану мовця, його намірів і реакцій. У телефонній розмові ж усе

навантаження лягає виключно на голосові параметри – інтонацію, темп, гучність, паузи, а також на змістовне наповнення мовлення.

Крім того, телефонна бесіда зазвичай має обмеження в часі. Її тривалість часто диктується не лише службовими чи побутовими обставинами, а й внутрішніми психологічними й енергетичними витратами учасників діалогу. Телефонне спілкування вимагає більшої концентрації, оскільки співрозмовники не мають змоги візуально коригувати один одного або підтримувати розмову через невербальні сигнали, що ускладнює сприйняття та інтерпретацію реплік.

Такі особливості мають особливе значення у кризових ситуаціях, де важлива оперативна координація (наприклад, під час логістичних збоїв, технічних інцидентів або медичних викликів). У таких випадках телефонна розмова дозволяє не лише швидко передати інформацію, а й гнучко адаптувати її під змінні обставини, реагуючи на тон, емоційний стан та уточнення співрозмовника в реальному часі.

Ще одним важливим чинником є наявність технічних перешкод, що властиві телефонним комунікаціям. Йдеться, зокрема, про можливі порушення якості зв’язку, затримки сигналу, фонові шуми, погану чутність або втручання сторонніх абонентів у разі нестабільного з’єднання. Усе це створює додаткове навантаження на учасників комунікації, вимагаючи від них чіткішого мовлення, повторення важливих фраз та уточнення окремих моментів, що в іншому форматі спілкування могли б передаватися невербально.

Значну роль голосова комунікація відіграє в таких галузях, як: продажі, клієнтський сервіс, технічна підтримка, медицина, логістика та державне управління. Саме через телефонний канал підприємства формують перше враження, вибудовують довіру та утримують лояльність клієнтів. Усередині організації телефонна комунікація виконує функцію координуючого механізму, що забезпечує злагоджену роботу між департаментами, дозволяє в реальному часі вирішувати завдання.

У контексті цифрової трансформації бізнес-процесів, коли дедалі більше компаній впроваджують автоматизовані платформи взаємодії з клієнтами,

телефонна комунікація зберігає своє ключове значення як канал оперативного, особистісного та емоційно забарвленого спілкування. Вона виступає незамінним інструментом у сферах продажів, клієнтського сервісу, логістики, охорони здоров'я, державного управління та консалтингу.

Усе вищезазначене зумовлює підвищені вимоги до якості телефонного спілкування, що, у свою чергу, ставить на перший план необхідність його систематичного контролю та аналізу. Усе частіше компанії прагнуть не лише відповідати на дзвінки, а й управляти всією системою телефонної взаємодії як стратегічним інструментом впливу на клієнтський досвід.

Телефонні дзвінки забезпечують:

1. Оперативність реагування на звернення клієнтів;
2. Точність уточнення деталей, що складно реалізувати в текстовій формі;
3. Емоційний контакт між компанією та клієнтом;
4. Підвищення довіри через особисте спілкування.

Внутрішня телефонна комунікація також сприяє координації дій між підрозділами, дозволяючи оперативно вирішувати виробничі питання, особливо у великих організаціях з динамічними процесами.

У міру масштабування компанії, зростання кількості клієнтських звернень та збільшення навантаження на контактні центри, виникає об'єктивна необхідність у впровадженні засобів контролю за якістю та структурою телефонних викликів. У разі відсутності належного моніторингу телефонна взаємодія втрачає прозорість, що може спричинити зниження якості обслуговування, втрату важливої інформації, неефективне використання людських ресурсів та зниження загальної продуктивності команди [1].

Сучасні системи моніторингу телефонних дзвінків (Call Monitoring Systems) вирішують зазначені проблеми, забезпечуючи фіксацію ключових параметрів викликів. До таких параметрів належать дата, час, тривалість дзвінка, напрям (вхідний або вихідний), номер абонента, статус відповіді, а в деяких випадках — і аудіозапис розмови, якщо це дозволено чинним законодавством. У

розробленому нами додатку, крім цих параметрів, також зберігається інформація про пристрій, з якого був здійснений виклик (callerName, callerNumber), що дозволяє фіксувати персоналізовану активність оператора [Додаток А].

Упровадження таких систем також вимагає дотримання чинного законодавства у сфері захисту персональних даних, а також інформування співрозмовників про можливість запису розмов. Це особливо актуально в міжнародних компаніях, де необхідно враховувати як локальні норми (наприклад, український Закон «Про захист персональних даних»), так і загальні регламенти, зокрема GDPR.

Аналітична обробка отриманої інформації дозволяє виявляти пікові періоди навантаження, визначати ефективність працівників, виявляти системні помилки у спілкуванні з клієнтами [2], а також будувати прогнозовану модель навантаження на персонал. На основі зібраних даних формується стратегічна звітність для керівництва, що, у свою чергу, створює підґрунтя для прийняття обґрунтованих управлінських рішень [3].

Наприклад, на основі аналізу тривалості дзвінків у пікові години можна скоригувати графік змін працівників, а статистика пропущених викликів допомагає виявити критичні точки навантаження. Також автоматичний підрахунок середнього часу відповіді, рівня дотримання скриптів або ефективності окремих операторів дозволяє об’єктивно оцінити якість роботи та вчасно вносити корективи у процеси.

Запровадження систем моніторингу телефонної комунікації має безпосередній вплив на покращення клієнтського досвіду та ефективність внутрішніх процесів. Регулярний аналіз записів розмов дає змогу виявляти характерні помилки у взаємодії з клієнтами, удосконалювати комунікаційні скрипти.

З боку працівників присутність системи моніторингу стимулює підвищення рівня відповідальності, дотримання корпоративних стандартів, увагу до деталей у розмові та належну мовленнєву культуру. Знаючи, що дзвінки можуть бути

проаналізовані, оператори з більшою серйозністю ставляться до своєї роботи, що безпосередньо відображається на загальному рівні обслуговування [4].

Тому, доречним є виділити типові показники ефективності до та після впровадження системи моніторингу (табл. 1.1).

Таблиця 1.1 – Типові показники ефективності до та після впровадження системи моніторингу

| **Індикатор ефективності** | **До впровадження** | **Після впровадження** |
|---|---|---|
| Середній час обробки одного звернення | 8 хв | 5 хв |
| Рівень задоволеності клієнтів (CSAT) | 72% | 87% |
| Кількість пропущених викликів | 15% | 5% |
| Дотримання стандартів спілкування | 65% | 90% |

Також моніторингові дані широко застосовуються у HR-практиках: під час адаптації нових працівників, в оцінюванні ефективності персоналу за KPI, а також при плануванні навчання. Таким чином, телефонний моніторинг виконує не лише контрольну функцію, а стає інструментом глибокого аналізу бізнес-процесів, підвищення рівня сервісу, оптимізації витрат та формування конкурентної переваги.

Отже, телефонна комунікація залишається однією з найважливіших складових корпоративної взаємодії, а її моніторинг — ефективним інструментом підвищення якості обслуговування. Завдяки сучасним аналітичним можливостям компанії отримують доступ до якісної інформації, яка дозволяє не лише вирішувати поточні завдання, а й формувати стратегічну політику у сфері комунікації та клієнтського досвіду.

Таким чином, моніторинг телефонних комунікацій виходить за межі простої технічної функції. Він стає важливою частиною цифрової трансформації, де кожен контакт із клієнтом — це потенційне джерело цінної інформації для вдосконалення продукту, сервісу й організаційної культури загалом.

## 1.2. Інструменти та технології розробки мобільних застосунків

Під час реалізації мобільної системи моніторингу телефонних дзвінків ключовим завданням стало обрання технологій, які дозволили б створити стабільний, адаптивний і розширюваний програмний продукт із мінімальними вимогами до інфраструктури та з урахуванням реального бізнес-контексту. Пріоритетним чинником при цьому була можливість реалізувати усю логіку моніторингу, обробки даних та надання доступу до них локально — без використання зовнішніх серверів або підключення до хмари. З цією метою було обрано платформу Android, яка забезпечує найширший доступ до системних ресурсів мобільного пристрою [5].

Крім доступу до системних API, Android вирізняється високим рівнем кастомізації, широкою базою пристроїв на ринку та активною спільнотою розробників, що забезпечує наявність тисяч відкритих бібліотек, шаблонів і рішень. Це робить платформу ідеальною для побудови бізнес-додатків з нестандартною логікою або з підвищеними вимогами до безпеки й автономності. Також велика кількість документації та підтримка з боку Google полегшує інтеграцію складних технологічних рішень, включаючи роботу з апаратними модулями та фоновими службами.

Android надає розробникам можливість взаємодіяти з такими компонентами, як журнали дзвінків, стан мобільної мережі, API телефонії, дозволи на читання системних подій тощо. Крім того, ця платформа дозволяє вбудовувати локальні сервери безпосередньо в додаток, що критично важливо для проєктів, які мають працювати в автономному середовищі — наприклад, локальних офісних мережах [12]. Такий підхід дозволяє зберігати та обробляти дані без передачі їх у зовнішні сервіси, що знижує ризики витоку інформації та підвищує конфіденційність.

Розробка інтерфейсу користувача здійснювалася за допомогою фреймворку Jetpack Compose — сучасного інструменту для побудови UI в Android, який став офіційно рекомендованим Google з 2021 року [7]. Compose базується на

декларативному підході: інтерфейс описується у вигляді функцій Kotlin, що реагують на зміну стану. Це дозволяє створювати адаптивні екрани з мінімальними зусиллями, легко підтримувати повторне використання компонентів, а також ефективно працювати з потоками даних у реальному часі. Для даного застосунку такий підхід був особливо корисним, оскільки інтерфейс потребував постійного оновлення у відповідь на зміни стану викликів [Додаток В].

Архітектура додатку базується на принципах **MVVM (Model–View–ViewModel)**, що дозволяє відокремити логіку представлення даних від бізнес-логіки. Це дає змогу ефективно тестувати компоненти, уникати дублювання коду та забезпечити масштабованість. Взаємодія між шарами реалізована через ViewModel, який містить стан додатку, а також репозиторій, що керує доступом до джерел даних (локального сховища або мережі). Таким чином, UI-частина залишається реактивною, а бізнес-логіка — чистою та незалежною від фреймворків відображення.

Особливу увагу було приділено взаємодії з системними API Android, зокрема PhoneStateListener та TelephonyManager. Перший дозволяє в режимі реального часу отримувати інформацію про зміну станів телефонного з’єднання: перехід у стан вхідного або вихідного дзвінка, його початок або завершення [8]. Ці дані є основою для фіксації викликів у системі. TelephonyManager, у свою чергу, надає доступ до вхідного номера, стану мережі, а також дозволяє ідентифікувати подію як вхідну чи вихідну [9].

Для забезпечення асинхронної обробки подій використовувалися Kotlin Coroutines — сучасний механізм керування багатопоточністю в Android. Завдяки ним стало можливим запускати паралельні задачі, такі як запис виклику в сховище або оновлення інтерфейсу, без блокування основного потоку. У поєднанні з StateFlow, який виступає реактивним джерелом стану, було реалізовано оновлення UI в режимі реального часу [3].

Окремим викликом стало забезпечення зберігання даних користувача — зокрема, імені оператора та номера пристрою. Для цього було використано

Android DataStore — рекомендовану Google технологію для збереження ключ-значення налаштувань або профільної інформації [10]. На відміну від застарілого SharedPreferences, DataStore працює на Kotlin Flow, є безпечним для багатопоточної роботи, і чудово інтегрується з іншими компонентами Jetpack [Додаток Г].

Серед альтернативних рішень розглядалися також бази даних SQLite через Room або використання EncryptedSharedPreferences для чутливих даних. Проте саме DataStore було обрано як більш гнучке, сучасне рішення з підтримкою асинхронної роботи, що важливо в контексті паралельної обробки викликів. Крім того, можливість створення власних Serializer-ів дозволила налаштувати збереження складніших об’єктів, зокрема профільної інформації оператора.

Таким чином, обрана сукупність інструментів дозволила реалізувати стабільну систему з високим рівнем автономності, що не потребує зовнішніх серверів, працює локально й дозволяє розширення — зокрема, шляхом підключення вебінтерфейсу до локального сервера [6].

У процесі реалізації було використано такі ключові технології:

- Android SDK — як основна платформа;
- Jetpack Compose — для розробки інтерфейсу;
- Kotlin Coroutines + StateFlow — для асинхронної обробки та реактивності [3];
- PhoneStateListener і TelephonyManager -для моніторингу стану дзвінків [8], [9];
- DataStore Preferences — для локального зберігання даних користувача [10];
- Ktor embedded server (CIO) — для розгортання локального HTTP-сервера прямо на пристрої [6];
- Hilt — для впровадження залежностей і забезпечення модульності архітектури.

Усі ці компоненти працюють узгоджено, забезпечуючи ефективне функціонування мобільної системи моніторингу викликів у бізнес-середовищі.

Для покращення обробки HTTP-запитів на стороні сервера використовувалася бібліотека Ktor Serialization, яка дозволила зручно працювати з форматами JSON та передавати структуровані дані через API. Також було реалізовано логування мережевих запитів за допомогою Ktor CallLogging, що допомогло в налагодженні протоколу взаємодії з клієнтською частиною. На клієнтському рівні, для керування правами доступу, застосовувався ActivityResultContracts.RequestPermission, що дозволяє динамічно обробляти дозволи Android 11+ у рамках нового API.

Для розробки програмного забезпечення використовувалося офіційне середовище Android Studio, що надає повний набір інструментів для створення, налагодження та тестування Android-додатків. Android Studio базується на IntelliJ IDEA, має потужний редактор коду, візуальний редактор інтерфейсу, вбудовану підтримку систем контролю версій, симулятори пристроїв та інструменти аналізу продуктивності (рис. 1.1). Особливе значення мали можливості Live Preview для Jetpack Compose, що дозволяли миттєво бачити зміни в інтерфейсі під час розробки [14].

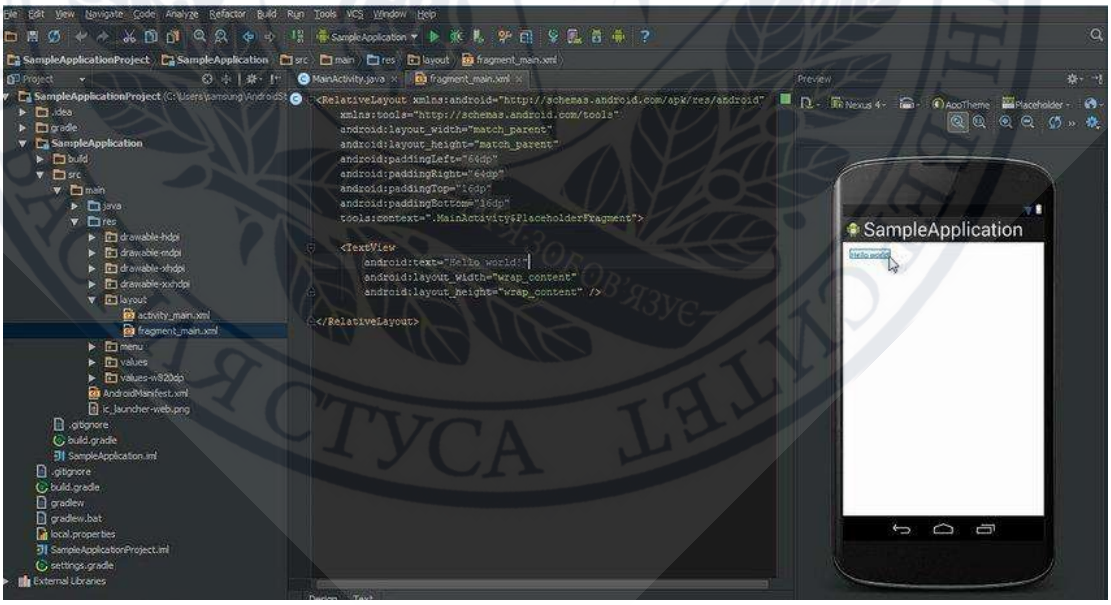


Рисунок 1.1 – Інтерфейс Android Studio

Основною мовою програмування проєкту є Kotlin — сучасна, лаконічна, безпечна за замовчуванням мова, яка повністю сумісна з Java та рекомендована Google для Android-розробки [15]. Kotlin забезпечує зменшення кількості

шаблонного коду, підтримує розширення функціональності через extension-функції, має потужну підтримку для функціонального програмування та роботи з потоками, включно з корутинами. Його синтаксис добре поєднується з декларативною природою Jetpack Compose, що значно покращує загальний досвід розробки.

Для підвищення надійності проєкту було впроваджено базові модулі **автоматизованого тестування**: Unit-тестування логіки репозиторію, а також Instrumented-тести для перевірки роботи ViewModel та UI в реальних сценаріях за допомогою Jetpack Test та Espresso. Це дозволило виявити помилки ще на етапі розробки, забезпечивши стабільність при зміні версій Android SDK.

Для тестування REST API, який реалізований через Ktor-сервер на мобільному пристрої, використовувався Postman — популярний інструмент для перевірки HTTP-запитів. Він дозволяв емулювати запити до локального сервера, перевіряти відповідність форматів відповіді, відслідковувати параметри дзвінків, що надходили до системи, а також виконувати симуляцію клієнтської сторони без розгортання окремого інтерфейсу [11].

Завдяки комплексному підходу до вибору інструментів, який включає офіційні розробницькі засоби, сучасну мову програмування та підтримку розширеного тестування, вдалося створити повноцінну автономну систему, що поєднує мобільний клієнт та сервер у межах одного пристрою.

## 1.3. Локальні сервери у мобільних додатках: огляд архітектурних рішень

У класичних програмних системах клієнт-серверна модель передбачає розподіл ролей між пристроєм користувача (клієнтом) та окремим сервером, що розміщується в інтернеті чи в локальній мережі. Проте з розвитком мобільних платформ та збільшенням їхніх обчислювальних можливостей з’явилась нова концепція — локального вбудованого сервера, який працює безпосередньо на

мобільному пристрої. Такий підхід дозволяє реалізувати повноцінну серверну логіку без потреби у зовнішній інфраструктурі [12].

Вбудований сервер на мобільному пристрої — це процес, який запускається всередині додатку й обробляє HTTP-запити локально. Він може бути доступним іншим пристроям у локальній мережі (через Wi-Fi) або використовуватись лише самим застосунком. Цей підхід особливо актуальний у випадках, коли важлива автономність, офлайн-доступ, низька затримка, а також захищеність даних — наприклад, у корпоративних середовищах або при використанні на закритих об’єктах.

Подібні архітектурні рішення дедалі частіше застосовуються у таких галузях, як логістика, охорона здоров’я, автоматизація складських процесів, польові дослідження, інспекції або виробничий контроль, де передавання конфіденційних даних до хмари є небажаним або технічно складним. Наприклад, у медичних закладах, що працюють з персональними даними пацієнтів, локальний сервер дозволяє обробляти запити лікарів або медсестер без виходу за межі внутрішньої мережі, дотримуючись вимог безпеки та нормативних обмежень (зокрема GDPR чи HIPAA). У польових умовах — наприклад, під час будівельного нагляду або екологічного моніторингу — мобільні пристрої з вбудованими серверами дозволяють збирати й передавати дані іншим пристроям у режимі офлайн, без постійного доступу до Інтернету.

У рамках розробленої системи моніторингу було впроваджено вбудований сервер на базі фреймворку Ktor, який зазвичай застосовується у веброзробці на мові Kotlin, але також підтримує запуск як embedded-сервер на Android [13]. Це рішення дозволило обробляти HTTP-запити без залучення зовнішніх серверів: усі дані про виклики залишаються на пристрої, і лише за запитом надаються у вигляді JSON через відкритий порт. Це забезпечує прозорість комунікацій і простоту доступу до них з будь-якого пристрою в локальній мережі — без складної інфраструктури або облікових записів [Додаток Ґ].

Архітектурно сервер реалізований у вигляді окремого фонової компоненти — ServerProviderImpl.kt, яка виконує роль HTTP-сервера на базі Ktor. Він не

припиняє свою роботу навіть при згортанні застосунку. У реалізації передбачені чітко визначені REST-маршрути:

- /api/v1/log — отримання історії викликів;
- /api/v1/status — поточний стан дзвінка;
- /api/v1 — базовий стан сервера.

Уся логіка маршрутизації реалізована через окремі routing-модулі Ktor, а дані про виклики зберігаються у вигляді реактивного потоку (StateFlow) [3], що дозволяє швидко реагувати на запити та уникати застарілої інформації.

Уся структура взаємодії між сервером, ViewModel та репозиторієм побудована на принципах розділення відповідальностей. Комунікація між сервером і логікою моніторингу здійснюється через абстрактні інтерфейси, що дозволяє легко змінювати спосіб зберігання або формат відповіді без зміни решти коду. Наприклад, у разі розширення функціоналу (додавання нових маршрутів або інтеграції з веб-клієнтом), достатньо реалізувати нові endpoint-и у відповідному routing-модулі, не змінюючи логіку взаємодії з бізнес-рівнем. Також було передбачено механізм обробки винятків (exception handling), що дозволяє уникнути аварійних зупинок сервера при некоректних запитах або збої в обробці даних.

Окрім REST-інтерфейсу, у сервері реалізовано механізм потокобезпеки за допомогою Mutex — примітиву синхронізації з kotlinx.coroutines, що дозволяє уникати race conditions при старті або зупинці сервера. Це особливо важливо при роботі з одночасними запитами у фоновому режимі. Уся ця логіка реалізована у [Додаток Ґ].

Для моніторингу стабільності роботи сервера реалізовано логування ключових подій (start, stop, errors) із можливістю перегляду системних логів безпосередньо з додатку. Це дозволяє адміністратору або технічному спеціалісту отримувати діагностичну інформацію без підключення до зовнішніх систем моніторингу. Також було передбачено механізм автоматичного перезапуску сервера у разі непередбаченої зупинки — наприклад, через збої в Android-ресурсах або фонове вивантаження процесу.

У порівнянні з хмарними рішеннями, локальний сервер має низку переваг:

- **Автономність** — не потребує підключення до інтернету;
- **Безпека** — дані не виходять за межі локальної мережі;
- **Контроль** — сервер повністю підконтрольний власнику пристрою;
- **Простота** — не потрібно налаштовувати сторонні сервіси, домени чи хостинг.

Проте такий підхід має і свої обмеження: обробка запитів можлива лише в межах локальної мережі, немає автоматичного масштабування, а також існують обмеження на продуктивність, пов’язані з ресурсами мобільного пристрою. У нашому випадку, де система призначена для використання у межах одного офісу або організації, всі ці обмеження є прийнятними.

Отже, використання локального сервера в Android-додатку на основі Ktor дозволяє реалізувати повноцінну серверну логіку прямо на пристрої, що ідеально підходить для задач моніторингу в локальному бізнес-середовищі. Це рішення поєднує в собі мобільність, простоту та повний контроль над даними, що робить його актуальним для широкого кола практичних задач.

У перспективі зростання інтересу до edge computing (обчислень "на краю") відкриває нові можливості для розвитку вбудованих серверів на мобільних пристроях. Очікується, що кількість додатків, які поєднують локальну обробку з періодичною синхронізацією в хмару, буде зростати — особливо у сферах з підвищеними вимогами до затримки, захищеності або безперервності роботи. У таких умовах рішення на основі Ktor embedded server можуть стати технологічним стандартом для мобільних систем збору, обробки та обміну даними.

# РОЗДІЛ 2
# РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ ДЛЯ МОНІТОРИНГУ ТЕЛЕФОННИХ ДЗВІНКІВ У БІЗНЕС-СЕРЕДОВИЩІ

## 2.1. Аналіз бізнес-вимог до системи моніторингу

У сучасному конкурентному бізнес-середовищі організації, що працюють із великою кількістю клієнтських звернень, особливо через голосовий канал, стикаються з проблемою контролю ефективності телефонної комунікації. Це особливо актуально для кол-центрів, сервісних відділів, логістичних служб, консультаційних бюро, медичних закладів, техпідтримки тощо — тобто структур, де якість і своєчасність відповіді на дзвінок безпосередньо впливає на прибуток, лояльність клієнтів та імідж компанії [2].

У таких сферах часто виникає необхідність аналізувати не лише факт комунікації, а й її якість: наскільки швидко оператор відповідає на дзвінки, скільки часу витрачає на вирішення проблеми, як часто клієнти звертаються повторно. Наприклад, у медичних установах важливо зафіксувати всі звернення пацієнтів із точністю до хвилини, а в службах доставки — контролювати реакцію диспетчерів у години пікового навантаження. Відсутність таких даних або неможливість отримати їх вчасно часто призводить до втрат — як матеріальних, так і репутаційних.

Найбільш поширеною проблемою в таких організаціях є відсутність прозорості у процесах телефонної комунікації. Керівник не має змоги швидко оцінити, скільки дзвінків здійснив або прийняв оператор, яка була тривалість розмов, хто з клієнтів звертався повторно, які дзвінки залишились без відповіді, і як загалом виглядає навантаження на команду. У більшості випадків такі дані доступні лише за умови використання складної телефонної інфраструктури — наприклад, IP-телефонії, VoIP-CRM-інтеграцій або платних хмарних сервісів. Однак для малого та середнього бізнесу подібні системи є надмірно дорогими, технічно складними або просто непотрібними за масштабом [4].

Використання смартфона як автономного моніторингового пристрою дозволяє поєднати гнучкість мобільного середовища з можливістю локального зберігання даних. Мобільний пристрій уже має вбудовані модулі комунікації, енергозабезпечення, обробки подій, доступ до API телефонії — тобто виконує функції, які у класичних системах вимагали б окремих пристроїв або серверів. Завдяки цьому загальна вартість впровадження системи значно знижується без втрати функціональності.

Саме тому постає потреба в автономному, недорогому рішенні, яке можна розгорнути безпосередньо на мобільному пристрої працівника.

В основі проєкту, що описується у даній роботі, лежить модель використання мобільного Android-пристрою як універсального інструменту прийому та моніторингу дзвінків. Оператор (співробітник кол-центру, диспетчер тощо) працює зі смартфоном, який виконує роль як клієнта (приймає виклики), так і сервера (надає дані про активність). Власне, це і є ключовою інновацією запропонованої системи — використання мобільного телефону як повноцінного локального вузла збору і обробки телефонної статистики. Такий підхід зменшує вартість входу, не потребує централізованої інфраструктури й забезпечує повну автономність.

Основними функціональними вимогами до системи, сформульованими на підставі вивчення типових бізнес-кейсів, є:

- автоматичне виявлення всіх вхідних і вихідних викликів [Додаток Б];
- реєстрація часу початку виклику, його тривалості, номера абонента і його імені (за наявності);
- визначення статусу виклику: триває / завершено / пропущено;
- зберігання історії дзвінків локально з можливістю їх перегляду [Додаток В];
- надання доступу до історії дзвінків у локальній мережі через браузер без встановлення додаткових програм;
- можливість швидкого налаштування користувача — збереження його імені та номера пристрою [Додаток Г].

Крім базового збору статистики, передбачена можливість розширення системи шляхом інтеграції з зовнішніми інструментами аналітики. Наприклад, у перспективі до системи може бути підключено вебінтерфейс, що дозволить візуалізувати динаміку навантаження, формувати зведені звіти або проводити аналіз повторних звернень. Для цього система має зберігати дані у форматі, придатному для обробки — наприклад, у вигляді JSON-структур або таблиць, сумісних з Excel / Google Sheets.

Окрім функціональних, надзвичайно важливими є нефункціональні вимоги. По-перше, система повинна бути повністю автономною: працювати без доступу до інтернету, хмари чи сторонніх серверів. Уся логіка — як клієнтська, так і серверна — повинна бути вбудована в мобільний застосунок. По-друге, система має бути безпечною: інформація про виклики не повинна передаватись за межі пристрою, а локальний сервер має бути доступним лише у межах локальної мережі [Додаток Ґ]. По-третє, важливою є продуктивність — додаток повинен працювати у фоновому режимі, не впливаючи на роботу пристрою, не перевантажуючи оперативну пам’ять і не впливаючи на якість виклику.

Також у фокусі розробки — простота використання. Для запуску системи користувачеві достатньо один раз надати дозволи на доступ до дзвінків і ввести свої дані. Додаток автоматично почне роботу у фоновому режимі та відкриє локальний сервер, доступ до якого можна отримати з будь-якого пристрою в мережі за IP-адресою [Додаток Б, Ґ].

Такий підхід дозволяє масштабувати систему горизонтально — встановлюючи застосунок на декількох пристроях, які працюють незалежно, але водночас забезпечують менеджмент актуальними даними про ситуацію в кожному підрозділі. Менеджер може отримувати статистику з кількох операторів у межах однієї мережі, не потребуючи складного серверного середовища. Це особливо корисно для організацій із розгалуженою структурою — наприклад, мереж медичних пунктів, філіалів логістичних компаній або мобільних сервісних бригад.

## 2.2. Архітектура рішення: клієнт-серверна модель на Android

Запропонована система моніторингу дзвінків побудована за принципами клієнт-серверної архітектури, в якій обидві частини — і клієнт, і сервер — реалізовані всередині одного Android-додатку. Такий підхід дозволяє уникнути залежності від зовнішньої інфраструктури й досягти максимальної автономності, зберігаючи водночас гнучкість у доступі до даних через звичайні HTTP-запити [6].

Загалом систему можна поділити на три основні рівні: системний рівень взаємодії з Android API, логічний рівень обробки подій та рівень доступу до даних через сервер. На кожному з цих рівнів реалізовано окремі компоненти, які взаємодіють між собою за допомогою реактивних потоків даних і чітко розмежованих інтерфейсів [3].

На системному рівні працює компонент **CallObserver**, що використовує PhoneStateListener та TelephonyManager для виявлення змін у стані телефонного виклику [8][9]. Він реагує на події на кшталт початку або завершення дзвінка, а також ідентифікує вхідний номер. У момент фіксації виклику CallObserver викликає методи збереження даних у репозиторії дзвінків [Додаток Б].

На логічному рівні діє CallRepository — клас, відповідальний за збереження історії викликів. Він підтримує список дзвінків у вигляді MutableStateFlow, що дозволяє в реальному часі реагувати на зміну стану. Інші частини системи — наприклад, сервер або інтерфейс — підписуються на ці потоки та автоматично отримують оновлення [Додаток А, В].

Особисті дані оператора — його ім’я та номер телефону — зберігаються в окремому репозиторії UserInfoRepository, який взаємодіє з локальним сховищем Android DataStore [10]. Це забезпечує збереження навіть після перезапуску додатку, а також дозволяє отримувати ці дані в будь-якому компоненті системи [Додаток Г].

Серверна частина реалізована через компонент ServerProviderImpl, який за допомогою фреймворку **Ktor** створює локальний HTTP-сервер [6][13]. Він

запускається на мобільному пристрої у фоновому режимі та обслуговує запити до маршруту /api/v1/.... Сервер має кілька окремих модулів:

- **RootRouting** — маршрут для перевірки статусу сервера;
- **StatusRouting** — маршрут для отримання поточного активного виклику;
- **LogRouting** — маршрут для отримання повної історії викликів у форматі JSON.

Кожен маршрут взаємодіє з відповідним репозиторієм — переважно з CallRepository — і формує відповідь на основі його поточного стану. Всі маршрути захищені в межах локальної мережі: запити з інших IP-адрес або ззовні не обробляються [Додаток Ґ].

Важливо, що всі компоненти взаємодіють між собою через інжекцію залежностей за допомогою Hilt, що забезпечує тестованість, масштабованість і розмежування логіки [5]. Наприклад, CallObserver не створює CallRepository напряму — він отримує його як залежність, що робить можливим заміну реалізації без зміни архітектури [Додаток Б].

Окрему увагу приділено реактивному підходу. За допомогою StateFlow усі компоненти — від інтерфейсу до API — отримують актуальні дані без необхідності ручного оновлення. Це дозволяє підтримувати інтерфейс в актуальному стані в реальному часі, навіть коли сервер надає доступ до тих самих даних зовнішнім клієнтам у мережі [3].

Серверна логіка запускається через спеціальний сервіс, який тримає HTTP-сервер активним незалежно від стану UI. Це дозволяє навіть при згортанні додатку продовжити моніторинг викликів і надавати доступ до них з інших пристроїв (наприклад, з ноутбука керівника) [Додаток Ґ].

Нижче представлено архітектурну схему взаємодії компонентів системи (рис. 2.1):

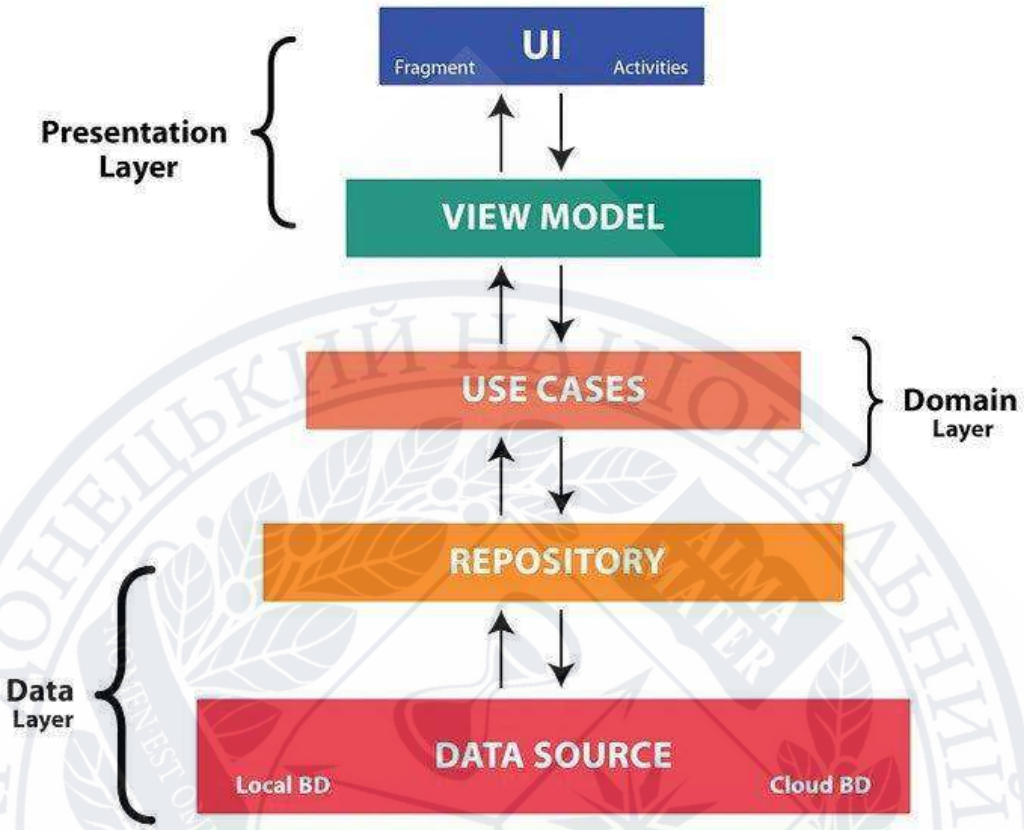


Рисунок 2.1 – Архітектура системи моніторингу

Важливим архітектурним рішенням у розробці стало винесення всієї серверної логіки в окремий модуль проєкту. Завдяки цьому вдалось чітко розмежувати відповідальність клієнтської частини (UI, обробка викликів, локальне збереження даних) та серверної (обробка HTTP-запитів, маршрутизація, серіалізація). Модуль серверу не має жодних залежностей від Android SDK, що дозволяє перевикористовувати його компоненти, тестувати окремо або за потреби розгорнути в іншому середовищі.

Така структура забезпечує кращу підтримуваність проєкту, пришвидшує компіляцію та відкриває можливості до розширення — наприклад, створення окремого десктопного або серверного застосунку на базі тих самих REST-ендпоінтів. Модулі взаємодіють один з одним через чітко визначені інтерфейси

та контракти, що знижує зв’язаність системи і відповідає принципам Clean Architecture [5].

Такий підхід дозволяє будувати модульну, надійну й розширювану систему, яку в майбутньому можна масштабувати — зокрема, додавши вебінтерфейс, централізований збір статистики або інтеграцію з корпоративними системами.

### 2.3. Ключові компоненти додатку: обробка викликів, передача даних, інтерфейс

Архітектура розробленої мобільної системи моніторингу телефонних дзвінків передбачає чітке розділення відповідальностей між окремими компонентами. Це дозволяє досягнути високої гнучкості, повторного використання коду, масштабованості та спрощує підтримку проєкту. Кожен компонент виконує окрему функцію: обробляє події телефонії, зберігає дані, передає їх через HTTP-протокол або відображає в інтерфейсі. Нижче докладніше описано ключові компоненти системи та принципи їх взаємодії.

Центральною точкою обробки телефонної активності є компонент CallObserver, який реалізує моніторинг системних подій за допомогою API PhoneStateListener і TelephonyManager [8, 9]. Завдяки цим API, CallObserver здатен виявляти зміну стану телефонного з’єднання: перехід у режим виклику, завершення розмови або пропущений дзвінок. На основі отриманих даних він ініціює запис у сховище викликів через CallRepository [Додаток Б].

CallRepository виконує роль сховища телефонної історії. Він підтримує список дзвінків у вигляді реактивного потоку StateFlow<List> [3]. Кожен запис представлений структурою CallInfo, що включає номер абонента, ім’я (якщо відоме), час початку, тривалість, статус (наприклад, ongoing або completed), а також унікальні ідентифікатори пристрою, з якого здійснено виклик [Додаток А]. Завдяки використанню StateFlow усі зміни в колекції дзвінків автоматично відображаються у підписаних на неї компонентах — наприклад, в інтерфейсі або HTTP-сервері.

Персональні дані користувача — ім’я оператора та номер телефону — зберігаються в окремому сховищі UserInfoRepository, який взаємодіє з DataStore Preferences API. Ці дані не змінюються часто, але необхідні для фіксації походження кожного виклику. Унікальність таких даних дозволяє формувати статистику по окремих пристроях або операторах, а також використовувати їх для фільтрації або аналітики [10, Додаток Г].

Компонент ServerProviderImpl відповідає за запуск вбудованого HTTP-сервера, реалізованого за допомогою Ktor Embedded Server [6]. Сервер працює у фоновому режимі, обробляє запити до локальних REST-ендпоінтів і повертає інформацію у форматі JSON. Наприклад, маршрут /api/v1/log обробляється модулем LogRouting і повертає повну історію дзвінків, отриману з CallRepository, а маршрут /api/v1/status — актуальний активний виклик (якщо він є) [Додаток Ґ]. Важливо, що сервер працює без необхідності доступу до Інтернету й обмежується лише локальною мережею, підвищуючи безпеку та контрольованість.

Інтерфейс користувача реалізовано за допомогою Jetpack Compose — сучасного декларативного фреймворку для побудови UI, який дозволяє напряму працювати з реактивними потоками даних [7]. Він складається з кількох ключових екранів: початкового (надання дозволів), екрану введення користувача та головного екрану, де виводиться історія дзвінків і поточна IP-адреса сервера. Головні UI-компоненти, такі як CallHistoryComponent і ServerInfoComponent, підписані на відповідні StateFlow і автоматично оновлюються при зміні стану [Додаток В].

Для реалізації архітектури було використано шаблон інжекції залежностей через Hilt, що дозволило ізолювати компоненти, забезпечити їхню тестованість і гнучкість. Наприклад, CallObserver отримує залежності CallRepository та UserInfoRepository через конструктор, що дозволяє легко замінити їх на мок-реалізації при тестуванні. Аналогічно, ServerProviderImpl може бути переініціалізований із тестовим репозиторієм або налаштуванням порту.

Додатково варто зазначити, що вся логіка серверної частини винесена в окремий модуль проєкту, який не має залежностей від Android SDK. Завдяки цьому сервер може бути легко адаптований для використання на десктопі або в контейнерному середовищі, без змін основної логіки. Це дозволяє створити повноцінну серверну інфраструктуру на основі наявного мобільного коду, що відкриває перспективи масштабування та централізованого збору даних [6].

Система побудована так, щоб обробка подій (викликів), передача інформації через HTTP та відображення в UI відбувалися в єдиному реактивному потоці — від джерела до споживача. Завдяки цьому забезпечується повна синхронізація станів, стабільна продуктивність і відсутність дублювання логіки в різних частинах системи. Такий підхід відповідає принципам сучасного реактивного програмування та дозволяє легко масштабувати систему в майбутньому, додаючи нові можливості без змін базової структури.

### 2.4. Реалізація збереження користувацьких даних та серверної логіки

Одним із критично важливих аспектів у розробці системи моніторингу викликів є збереження та актуалізація персоніфікованих даних користувача — зокрема, імені оператора та номера пристрою, з якого здійснюються або приймаються дзвінки. Ці дані дозволяють точно ідентифікувати джерело кожного виклику, що особливо важливо у багатокористувацьких середовищах (наприклад, диспетчерських або кол-центрах), де різні працівники можуть працювати із різними пристроями.

Для реалізації цього механізму було обрано сучасний підхід до локального зберігання — Android DataStore, що є реактивною, безпечною та потокобезпечною альтернативою застарілим SharedPreferences [10]. В рамках проєкту використано API DataStore Preferences, який зберігає дані у форматі ключ–значення та тісно інтегрується з Kotlin Coroutines і Flow. Це дозволяє будувати реактивні потоки даних, до яких можна підписуватись з будь-якого компонента програми.

Компонент UserLocalDataSource реалізує взаємодію з DataStore — він надає два основні потоки: fromNameFlow та fromNumberFlow, які несуть значення відповідно імені та номера користувача. Ці значення не лише зберігаються між сесіями, а й автоматично транслюються у всі частини системи, де вони потрібні — наприклад, для ініціалізації запису виклику або відображення в інтерфейсі [Додаток Г].

Роботу із UserLocalDataSource координує UserInfoRepository, який виступає посередником між джерелом даних і споживачами, спрощуючи доступ до даних та забезпечуючи централізовану логіку обробки. Компонент CallObserver, що відповідає за моніторинг змін у телефонії, використовує саме цей репозиторій для зчитування актуального імені та номера оператора у момент обробки дзвінка. Усі ці дані об’єднуються в модель CallInfo — унікальний запис виклику, який включає всю необхідну інформацію: номер, ім’я абонента, час, тривалість, статус виклику, а також джерело дзвінка (callerName, callerNumber) [Додаток А, Додаток Б].

Паралельно з локальним збором даних функціонує вбудований HTTP-сервер, реалізований на базі Ktor Embedded Server (CIO) [6, 13]. Запуск сервера відбувається через компонент ServerProviderImpl, який належить до окремого серверного модуля і реалізує усю необхідну логіку: старт/зупинка сервера, конфігурація портів, маршрутизація запитів, серіалізація відповідей тощо [Додаток Ґ]. У процесі запуску сервер ініціалізує фоновий процес, що продовжує працювати незалежно від активності користувача чи стану UI. В результаті, IP-адреса та порт доступу до API відображаються безпосередньо в інтерфейсі мобільного додатку, а доступ до даних стає можливим з будь-якого пристрою в локальній мережі.

Сервер побудований за модульною архітектурою, де кожен маршрут реалізується окремим модулем. Основні маршрути включають:

- /api/v1 — кореневий маршрут, що підтверджує активність сервера та повертає загальний статус системи.

• /api/v1/log — обробляється модулем LogRouting, повертає список усіх збережених викликів у форматі JSON, використовуючи CallRepository як джерело даних.

• /api/v1/status — маршрут, що надає інформацію про поточний активний виклик у реальному часі (якщо він триває).

Кожен з маршрутів має ізольовану реалізацію логіки, що дозволяє легко розширювати сервер новими функціями, не змінюючи основну структуру. Зокрема, планується додати маршрути для фільтрації викликів за датами, експорт у CSV, статистику за оператором тощо.

Особливу увагу приділено потокобезпеці. Для захисту від race conditions під час обробки одночасних запитів сервер використовує Mutex з kotlinx.coroutines [16]. Це об’єкт синхронізації, який гарантує, що лише один потік може виконати критичну секцію коду в певний момент. У нашому випадку це стосується доступу до CallRepository — читання списку викликів або його модифікація (наприклад, додавання нових записів під час активного дзвінка). Таким чином, навіть при одночасному надходженні HTTP-запитів і паралельному оновленні стану, серверна частина гарантує узгодженість даних.

Варто підкреслити, що серверний модуль цілком ізольований від Android-залежностей. Це дозволяє використовувати його не лише у мобільному середовищі, але й повторно — наприклад, для створення десктопного застосунку, або для розгортання на локальному сервері підприємства. Такий підхід відповідає принципам багатоплатформної розробки та забезпечує майбутню масштабованість рішення [6].

Таким чином, реалізоване рішення поєднує:

• реактивне локальне збереження персональних даних користувача (через DataStore);

• уніфіковану модель дзвінка з усією необхідною інформацією (CallInfo);

• безпечний вбудований сервер із підтримкою REST API та контролем доступу;

• потокобезпечну обробку запитів завдяки Mutex;

• можливість масштабування та повторного використання серверної частини поза межами Android-середовища.

Усе це дозволяє створити повноцінну, автономну систему, яка може ефективно використовуватись у корпоративних середовищах, де важлива ізоляція даних, локальність, простота налаштування та контроль за комунікаційними процесами.

## 2.5. Тестування системи та перевірка стабільності в реальних умовах

Для оцінки надійності, продуктивності та коректності роботи системи моніторингу телефонних дзвінків було проведено всебічне ручне тестування в умовах, наближених до реального використання. Попри відсутність автоматизованих модульних або інтеграційних тестів, особливу увагу приділено перевірці роботи всіх ключових компонентів: CallObserver, DataStore, ServerProviderImpl, а також UI-компонентів, реалізованих у Jetpack Compose.

Одразу після запуску додаток виводить екран із запитом необхідних дозволів (рис. 2.1), без яких неможливий моніторинг дзвінків. Було протестовано сценарії як повного, так і часткового надання дозволів. У випадку відсутності доступу до телефонії система коректно повідомляє про неможливість роботи, що відповідає рекомендаціям Android щодо permission-handling [5].

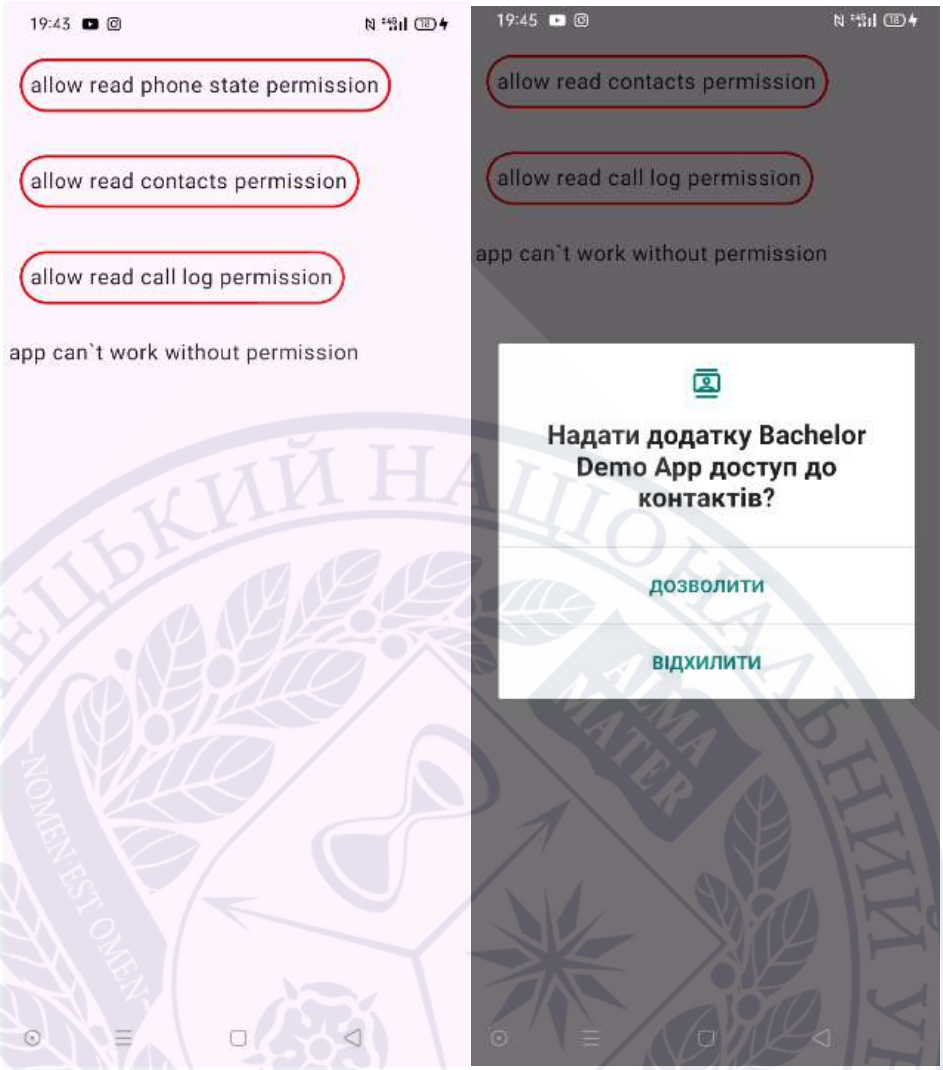


Рисунок 2.2 – Екрани із запитом дозволів

Після отримання дозволів відображається екран введення користувацьких даних — імені та номера телефону (рис. 2.3). Це забезпечує унікальну ідентифікацію кожного пристрою в межах мережі, що критично для багатокористувацького середовища.

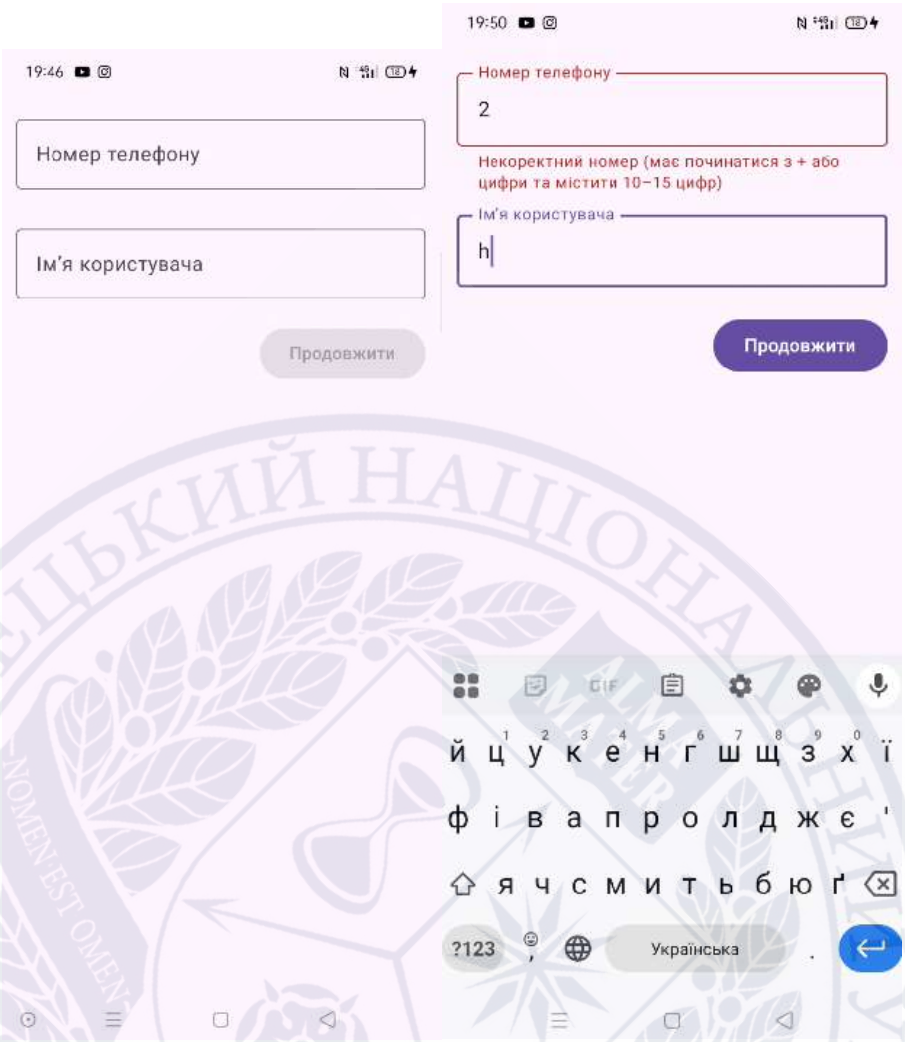


Рисунок 2.3 – Екрани введення даних користувача

Далі система переходить до головного екрана, який відображає статус сервера та історію дзвінків. Якщо сервер не запущено — інтерфейс містить відповідне повідомлення, а дані недоступні зовнішнім клієнтам (рис. 2.4).

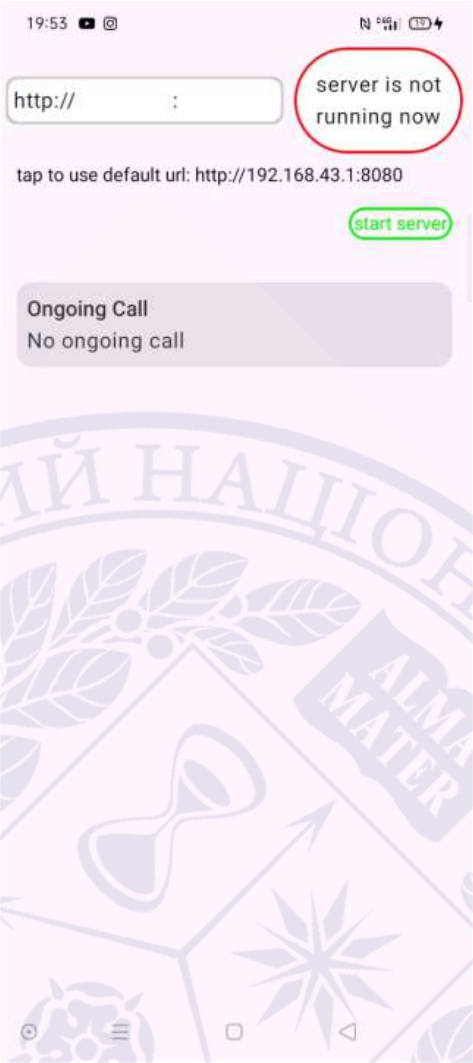


Рисунок 2.4 – Головний екран із неактивним сервером

При запуску локального HTTP-сервера (через ServerProviderImpl [Додаток Ґ]) інтерфейс оновлюється та відображає IP-адресу, за якою сервер доступний іншим пристроям у локальній мережі (рисунок 2.4). Це дозволяє адміністратору або керівнику отримати доступ до статистики безпосередньо з комп’ютера або планшета.


Рисунок 2.5 – Головний екран із активним сервером

Особливу увагу приділено перевірці REST API за допомогою Postman [11]. На рисунку 2.6 представлено приклад відповіді сервера на запит /api/v1/status у форматі JSON. Усі запити /log, /status та / повертали коректну, актуальну інформацію, що підтверджує цілісність інтеграції між CallRepository, StateFlow та маршрутизацією серверної частини.

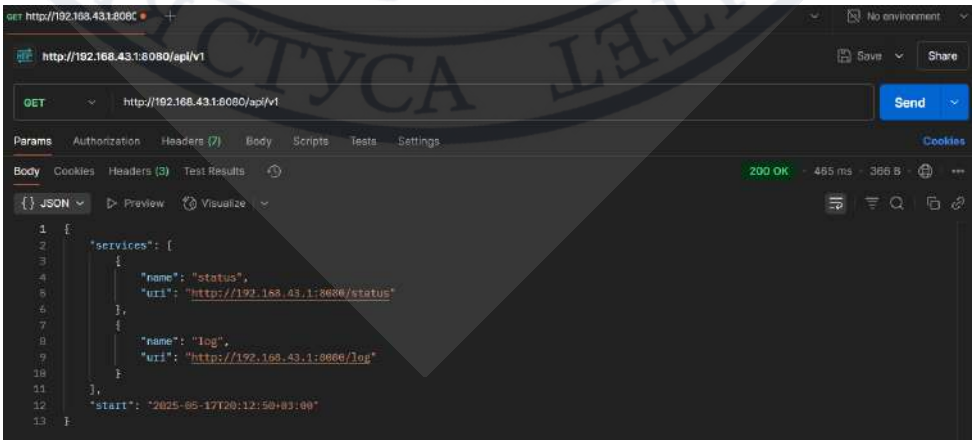


Рисунок 2.6 – Приклад відповіді сервера в Postman “/api/v1”

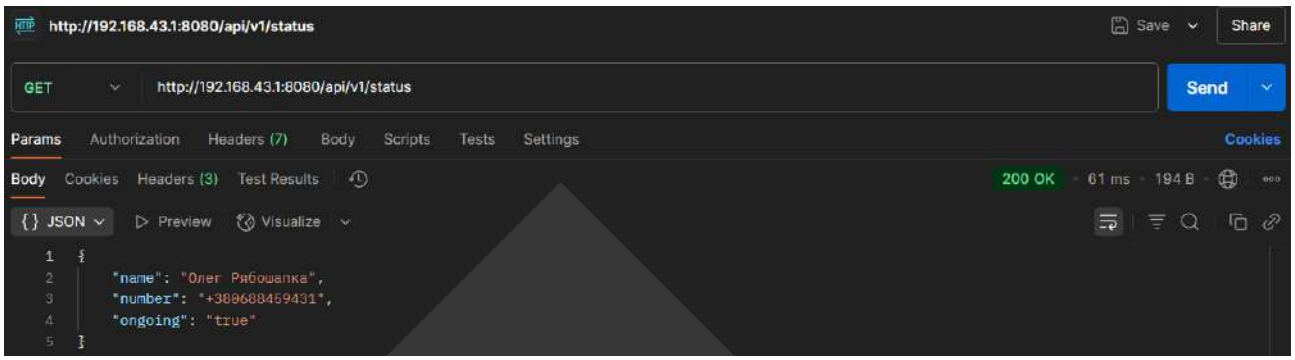


Рисунок 2.7 – Приклад відповіді сервера в Postman “/api/v1/status”

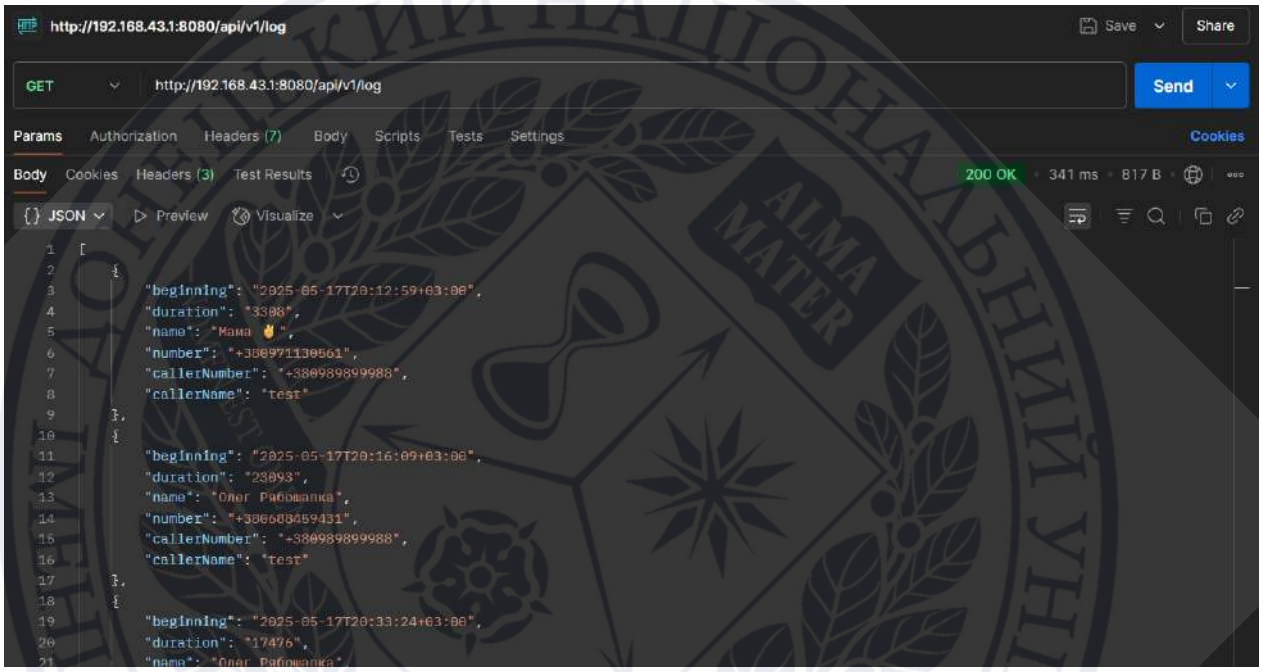


Рисунок 2.8 – Приклад відповіді сервера в Postman “/api/v1/log”

Було змодельовано різні типи викликів: короткі та довгі, пропущені, вхідні та вихідні. Усі вони коректно реєструвались компонентом CallObserver [Додаток Б] незалежно від того, чи перебував додаток у фоновому режимі або пристрій був заблокованим. Це досягнуто завдяки глибокій інтеграції з системними API телефонії [8, 9].

Дані користувача, збережені через DataStore API [10, Додаток Г], зберігалися між перезапусками додатку й автоматично підставлялися до нових записів. Це продемонструвало стабільність корутинаційного підходу до роботи з налаштуваннями, згідно з практиками, рекомендованими Google [5].

Окремо тестувалася продуктивність серверної частини. Додаток залишався активним у фоновому режимі понад 6 годин, обробляючи періодичні дзвінки, при цьому сервер не припиняв роботу. Механізм Mutex [16], використаний у ServerProviderImpl, забезпечував синхронізований доступ до CallRepository, виключаючи race conditions. Цей підхід відповідає найкращим практикам безпечного багатопотокового програмування [17].

Також змодельовано граничні сценарії:

- вимкнення сервера вручну — додаток коректно припиняв обробку запитів;
- зміна IP-адреси при переході між Wi-Fi-мережами — IP оновлювався автоматично;
- часткове відкликання дозволів — інтерфейс відображав помилку та підказку дій.

Загалом, результати тестування продемонстрували:

- стабільність роботи без падінь або втрати даних;
- коректну взаємодію між клієнтською, серверною та інтерфейсною частиною;
- готовність системи до використання в реальному середовищі з низьким рівнем технічної підтримки.

Окрему увагу у подальшому можна приділити автоматизованому тестуванню – зокрема, використанню інструментів Espresso для UI-тестів [18], JUnit для модульних перевірок логіки [19], а також інтеграції CI/CD-рішень на базі GitHub Actions або GitLab Pipelines (табл. 2.1) [20].

Таблиця 2.1 – Основні сценарії тестування мобільної системи моніторингу дзвінків

| **№** | **Сценарій тестування** | **Вхідні умови** | **Очікуваний результат** |
|---|---|---|---|
| 1 | Вхідний дзвінок | Додаток активний, дозволи надані | Дзвінок зафіксовано, відображено в історії |
| 2 | Вихідний дзвінок | Користувач ініціює дзвінок | Дзвінок зафіксовано з тривалістю, статус – завершено |

| № | Сценарій тестування | Вхідні умови | Очікуваний результат |
|---|---|---|---|
| 3 | Пропущений дзвінок | Дзвінок не було прийнято | Запис з нульовою тривалістю, статус – завершено |
| 4 | Запит до /api/v1/log | Сервер запущений | Повертається список дзвінків у форматі JSON |
| 5 | Запит до /api/v1/status під час активного виклику | Дзвінок триває | Повертається CallInfo з ongoing = true |
| 6 | Повторний запуск додатку | Дані користувача вже введені | Ім’я та номер автоматично підставляються у нові записи |
| 7 | Вимкнення/перезапуск пристрою | Після перезапуску | Сервер та моніторинг відновлюються після запуску вручну |
| 8 | Відсутність дозволу READ_CALL_LOG | Користувач не надав дозвіл | Додаток не працює, з’являється відповідне повідомлення |
| 9 | Тривала робота у фоновому режимі | Пристрій не використовується вручну | Сервер залишається активним, нові виклики фіксуються |
| 10 | Доступ до сервера з іншого пристрою в мережі | Відомий IP-адрес, активне підключення | Дані доступні через браузер або Postman |

Отже, тестування системи в умовах, наближених до реального використання, підтвердило її стабільну роботу, коректне функціонування основних компонентів і готовність до практичного впровадження. Система адекватно реагує на типові та граничні ситуації, забезпечуючи надійність і безперервність роботи. У майбутньому доцільно впровадити автоматизоване тестування для підвищення ефективності перевірки та підтримки проєкту.

# РОЗДІЛ 3
# ВПРОВАДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МОБІЛЬНОЇ СИСТЕМИ МОНІТОРИНГУ ТЕЛЕФОННИХ ДЗВІНКІВ У БІЗНЕС-СЕРЕДОВИЩІ

## 3.1. Сценарії використання в малому та середньому бізнесі

Розроблена система моніторингу голосової активності орієнтована на застосування у малих та середніх підприємствах (МСП), що працюють із клієнтами в телефонному режимі. Її основна мета – забезпечити простий і доступний інструмент для реєстрації вхідних та вихідних викликів, зберігання історії взаємодій і надання до них доступу без необхідності використання складної інфраструктури або хмарних сервісів [2, 4, 19].

Однією з ключових переваг є автономність системи: вона працює локально на Android-пристрої та не потребує постійного підключення до Інтернету, VPN або зовнішніх серверів [5, 12, 20]. Це робить рішення придатним для бізнесів, які прагнуть мінімізувати витрати на IT-інфраструктуру та одночасно контролювати якість обслуговування клієнтів у голосовому каналі. Саме тому, доречним буде розглянути можливі сценарії застосування розробленої системи.

**Сценарій 1. Диспетчерські служби**

У галузі перевезень (таксі, логістика, кур’єрські послуги), диспетчер працює з великим обсягом дзвінків, часто маючи лише один мобільний пристрій. Запропонована система дозволяє автоматично реєструвати всі виклики: фіксувати час, номер, тривалість, ім’я абонента (за наявності). Керівник служби має змогу у будь-який момент перевірити журнал дзвінків, зайшовши на локальний сервер за IP-адресою пристрою [6, Додаток Ґ].

**Сценарій 2. Невеликі відділи продажів або підтримки**

У малих офісах, де працює один-два менеджери із клієнтами, система може використовуватись як локальний монітор комунікацій. Наприклад, у відділі продажів, який не має доступу до централізованої CRM, запис телефонної

активності може забезпечити зворотний зв’язок для керівника. Через локальний сервер історія дзвінків доступна в браузері без інсталяції додаткових програм [11, 18].

**Сценарій 3. Сфера обслуговування (майстерні, клініки, сервісні центри)**

Часто у таких закладах співробітники використовують особисті або корпоративні телефони для зв’язку з клієнтами. Система дозволяє бачити, чи були пропущені дзвінки, скільки часу витрачено на розмови, і чи всі клієнти отримали відповідь. Це не лише полегшує адміністрування, а й допомагає підтримувати якість сервісу без потреби в IP-телефонії чи підписках на хмарні CRM-системи [1, 19].

Відповідно, подана система може мати низку переваг для малого та середнього бізнесу. До таких можна віднести:

1. **Мінімальні вимоги до обладнання:** необхідно лише мати Android-смартфон та Wi-Fi-з’єднання.
2. **Відсутність сторонніх серверів:** локальний HTTP-сервер реалізується через ServerProviderImpl (див. Додаток Ґ) на основі Ktor Embedded Server [6, 13].
3. **Автоматична реєстрація дзвінків:** відбувається за допомогою API TelephonyManager та PhoneStateListener [8, 9, Додаток Б].
4. **Негайна доступність історії:** зберігання дзвінків у StateFlow забезпечує актуальність даних у реальному часі [3, Додаток А].
5. **Безпека даних:** усі дані залишаються на пристрої та недоступні стороннім сервісам, що відповідає стандартам локальної безпеки [21].
6. **Швидке впровадження:** достатньо кількох хвилин для налаштування системи після встановлення.
7. **Відповідність Android for Work:** система може бути інтегрована в корпоративне середовище з використанням Android Enterprise [17].

Хоча система створювалась із розрахунку на мікробізнес, її архітектура дозволяє адаптацію для більших команд або декількох пристроїв. Дані з кожного пристрою можуть бути об’єднані через централізовану точку збору на базі REST

API. Це відкриває шлях до створення десктопного моніторингового інструмента або вебінтерфейсу для віддаленого контролю [13, 15].

Оскільки, подана система може стати важливим елементом у різноманітних підприємствах важливо підтримувати її розвиток та актуальність. Тому, у майбутньому можливе додавання таких функцій:

- експорт дзвінків у форматі CSV або Excel;
- підключення до зовнішніх календарів;
- інтеграція з e-mail сервісами для звітності;
- підтримка push-сповіщень.

Усі ці функції можуть бути додані без зміни базової логіки, завдяки модульній архітектурі системи на основі Hilt [7].

Отже, розроблена система демонструє високу практичну цінність для малого та середнього бізнесу, що працює з клієнтами телефоном. Вона є простою у впровадженні, не потребує додаткової інфраструктури й забезпечує ефективний моніторинг дзвінків у локальному середовищі. Завдяки автономності, безпеці даних і гнучкій архітектурі, система може бути адаптована до різних бізнес-сценаріїв та розширена відповідно до майбутніх потреб підприємств.

### 3.2. Аналіз переваг для керування телефонною активністю персоналу

У малому та середньому бізнесі ефективне управління людськими ресурсами є визначальним чинником стабільного обслуговування клієнтів. Особливо це стосується телефонної комунікації, яка часто залишається єдиним або основним каналом взаємодії з клієнтами [4, 19]. В умовах обмежених ресурсів і відсутності централізованих CRM-систем, керівникам складно об’єктивно оцінити якість роботи співробітників, своєчасність зворотного зв’язку або рівень завантаженості працівника. Саме тому впровадження локальної системи моніторингу викликів може стати ключовим інструментом для оптимізації комунікаційних процесів.

Розроблений у межах цього проєкту додаток дозволяє в реальному часі фіксувати всі дзвінки, що здійснює чи приймає працівник, а також надавати до цієї інформації доступ безпосередньо у локальній мережі — без зовнішніх серверів або підключення до хмари [5, 6]. Усі події телефонії обробляються через CallObserver (див. Додаток Б), записуються у CallRepository та зберігаються у вигляді структур CallInfo (див. Додаток А), які автоматично передаються як у UI, так і в API-сервер [3, 13].

На цьому етапі виникає потреба детального аналізу переваг поданого сервісу задля наочного розуміння потреби та його важливості у використанні.

**1. Прозорість і контроль без інтеграції з CRM**

Система дозволяє менеджеру бачити:

- кількість дзвінків, здійснених кожним працівником за обраний період;
- час початку та тривалість розмови;
- частоту повторних звернень;
- пропущені або короткі виклики.

Ці метрики дозволяють отримати уявлення про динаміку завантаженості, виявити критичні періоди навантаження та за необхідності відкоригувати графіки або процеси комунікації. Наприклад, якщо аналіз дзвінків показує, що понад 30% викликів залишаються без відповіді у певний часовий проміжок, це є сигналом для оперативних змін [2, 21].

**2. Підвищення ефективності та дисципліни**

Факт фіксації телефонної активності сам по собі стимулює працівників до підвищення самодисципліни. Як свідчать дослідження у сфері поведінкової економіки, навіть проста наявність нагляду змінює поведінку персоналу у бік більшої відповідальності [22]. Це особливо актуально для невеликих команд, де один пропущений дзвінок може призвести до втрати клієнта.

Також система дозволяє формувати історію звернень, яка може бути використана:

- як доказ у разі конфліктної ситуації з клієнтом;
- як база для навчання та адаптації нових співробітників;

• як матеріал для внутрішнього аудиту або щотижневих звітів.

**3. Гнучкий доступ до даних**

Однією з особливостей системи є можливість переглядати дані не лише на смартфоні оператора, але й з будь-якого пристрою в мережі — наприклад, ноутбука або планшета керівника. Серверна логіка, реалізована через ServerProviderImpl [Додаток Ґ], надає REST-доступ до маршрутів /api/v1/log і /api/v1/status, що дозволяє побудувати власні засоби візуалізації або інтеграції з іншими локальними сервісами [6, 11, 20].

Доречним є здійснити аналіз порівняльних переваг системи (табл. 3.1).

Таблиця 3.1 – Порівняльні переваги системи:

| **Параметр** | **Без моніторингу** | **Із системою моніторингу** |
|---|---|---|
| Облік кількості дзвінків | Неможливий | Автоматичний через CallRepository |
| Виявлення простоїв або перевантаження | Інтуїтивне, суб’єктивне | На основі фактичних даних [2, 3] |
| Доступ до статистики | Відсутній | Через локальний HTTP-сервер |
| Вплив на дисципліну | Мінімальний | Працівник знає, що його дії фіксуються |
| Аналіз повторних звернень | Неможливий | Ідентифікація за номером абонента |
| Інструменти аналізу | Відсутні | JSON-дані через API [11, 13] |

Система забезпечує не лише базовий моніторинг викликів, але й слугує інструментом аналітики для керівника будь-якого рівня. Її використання дозволяє реалізувати підхід «data-driven management» навіть у мікрокомандах, які раніше покладались на суб’єктивну оцінку. З урахуванням простоти впровадження, низького порогу входу та повної автономності, така система є оптимальним рішенням для SMB-сегменту.

### 3.3. Перспективи масштабування: вебінтерфейс, аналітика, інтеграції з CRM

Хоча поточна реалізація системи моніторингу дзвінків повністю виконує своє завдання в умовах малого бізнесу, її архітектура була спроєктована з урахуванням подальшого масштабування як з функціональної, так і з технічної точки зору. Всі компоненти — від CallRepository до ServerProviderImpl (див. Додатки А, Ґ) — взаємодіють через чітко визначені контракти, що дозволяє розширювати функціонал без зміни вже реалізованих базових механізмів [6, 13].

Така модульна побудова дозволяє легко додавати нові компоненти — наприклад, модуль обробки подій, голосових тегів або навіть механізм класифікації дзвінків за типом (скарга, консультація, замовлення). Завдяки використанню чітких контрактів між шарами (інтерфейси, sealed-класи), система не вимагає зміни існуючої логіки — нові модулі можуть підключатися окремими сервісами через dependency injection. Це особливо важливо при розвитку з MVP до повноцінної бізнес-платформи.

Оскільки серверна частина надає повноцінні REST-ендпоінти у форматі JSON (наприклад, /api/v1/log, /api/v1/status), цілком логічним є наступний крок — розробка вебінтерфейсу для зручнішої візуалізації даних. Це може бути як базова HTML-сторінка, так і повноцінний односторінковий застосунок на React, Vue або Angular. Такий інтерфейс дозволить:

- швидко переглядати історію дзвінків із фільтрами;
- бачити статус активних викликів у реальному часі;
- виводити графіки та діаграми з даними про навантаження.

Враховуючи, що сервер працює локально, вебінтерфейс може бути відкритий з будь-якого пристрою, підключеного до тієї ж мережі. Це зберігає автономність системи та не потребує сторонніх сервісів чи реєстрацій [6, 25].

У випадку розгортання вебінтерфейсу, доцільним є використання таких технологій, як React (з бібліотекою Recharts або Chart.js для візуалізації), а також Axios для з’єднання з REST API. Інтерфейс може бути реалізований як

Progressive Web App (PWA), що дозволить зберігати частину даних офлайн, кешувати запити та працювати з мобільних пристроїв без встановлення додатків. Таким чином, до даних системи зможуть звертатися як керівники, так і аналітики без втручання в мобільну частину.

Підрахунок ключових метрик — середньої тривалості розмов, кількості дзвінків на день, частки пропущених викликів — дозволить перетворити систему з простого реєстратора подій у повноцінний аналітичний інструмент. Ці дані можна:

- збирати в окремому /api/v1/stats;
- періодично експортувати у CSV/Excel;
- автоматично виводити у звіти з допомогою зовнішніх бібліотек, таких як Apache POI або Ktor Excel Writer [32].

За оцінками експертів, надання керівникам візуалізованої статистики може підвищити ефективність управлінських рішень до 28% у сфері SMB (джерело: [33]).

Ще одним напрямом розвитку є інтеграція з існуючими CRM-системами. Завдяки тому, що дані про дзвінки представлені у вигляді серіалізованих DTO (CallInfo, LogResponseItem), вони легко піддаються обробці сторонніми сервісами. Інтеграція можлива за допомогою:

- вебхуків, які надсилають події на CRM після кожного дзвінка;
- регулярного експорту історії викликів у зовнішню базу;
- прямого REST-запиту від CRM до локального сервера за параметром номера телефону [34].

Інтеграція не лише забезпечить автоматичне ведення клієнтських карток, але й дозволить поєднати дзвінки з історією покупок, замовлень або звернень — що значно розширює аналітичний потенціал системи.

На технічному рівні реалізація вже підтримує багатоджерельність: у CallInfo присутні поля callerNumber та callerName, що однозначно ідентифікують пристрій-джерело (див. Додаток А). Це означає, що декілька Android-пристроїв можуть одночасно передавати свої дані на спільний сервер — локальний або

зовнішній. Таким чином, можливо побудувати централізовану систему обліку дзвінків усієї команди без використання складної IP-телефонії.

Цей підхід відкриває перспективи масштабування до формату внутрішнього контакт-центру або фронт-офісу з багатьма операторами, що працюють у спільному середовищі [4].

Зрештою, запропоноване рішення можна трансформувати у повноцінний локальний інформаційний хаб, який включатиме:

- моніторинг комунікацій;
- модульну аналітику;
- персоніфіковані дашборди;
- інтеграцію з внутрішніми ERP/CRM.

При цьому воно зберігає простоту впровадження: не потребує підключення до хмари, не має щомісячної підписки, працює на будь-якому сучасному Android-пристрої. Подібна архітектура відповідає принципам **offline-first development** і дозволяє розгортати цифрові рішення навіть у тих компаніях, де є обмеження на використання зовнішніх сервісів [35].

У подальшому система може еволюціонувати в платформу з розширюваним API, яка дозволить зовнішнім розробникам або підрозділам компанії створювати власні модулі, звіти чи сценарії інтеграції. Такий підхід відповідає концепції low-code / no-code середовищ, де бізнес-користувачі зможуть створювати автоматизовані процеси без глибоких знань програмування, зберігаючи при цьому контроль над локальними даними.

# ВИСНОВКИ

У ході виконання роботи було розроблено автономну мобільну систему для моніторингу телефонних дзвінків, орієнтовану на потреби малого та середнього бізнесу. Система дозволяє автоматично фіксувати вхідні та вихідні виклики, зберігати інформацію про них локально, а також надавати доступ до історії дзвінків у зручному форматі через вбудований HTTP-сервер, який працює в межах локальної мережі.

На основі аналізу сучасних підходів до моніторингу комунікацій було сформульовано вимоги до функціональності системи. В результаті було створено мобільний додаток на платформі Android, реалізований з використанням Jetpack Compose, Kotlin Coroutines, DataStore, Ktor та Hilt. Архітектура проєкту побудована на основі реактивного підходу, багатомодульності та чіткого розмежування відповідальностей.

Практична реалізація продемонструвала стабільність, простоту використання та готовність до роботи в реальних умовах. Система протестована вручну у низці типових сценаріїв, включаючи активні дзвінки, роботу у фоновому режимі, запуск без інтернету, запити до API з інших пристроїв.

Розроблене рішення може бути застосоване в диспетчерських службах, кол-центрах, сервісних компаніях, де потрібен простий, прозорий та недорогий інструмент контролю телефонної активності персоналу. Його основними перевагами є автономність, відсутність потреби в зовнішніх серверах, підтримка локального доступу та можливість розширення.

У роботі також окреслено перспективи розвитку системи: створення вебінтерфейсу, додавання аналітичного модуля, інтеграція з CRM та підтримка кількох пристроїв одночасно. Завдяки обраній архітектурі, такі доповнення можуть бути реалізовані без суттєвої перебудови наявної логіки.

Таким чином, результатом проєкту стала ефективна, практично орієнтована система, яка має потенціал до подальшого розвитку як повноцінний інструмент для покращення бізнес-комунікацій.

# СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ


1. Google Android Developers – Call Log and Telephony APIs URL: https://developer.android.com/reference/android/provider/CallLog
2. Analytics Vidhya – How to Analyze Call Center Data URL: https://www.analyticsvidhya.com/blog/2021/06/how-to-analyze-call-center-data/
3. KotlinLang – StateFlow Documentation URL: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/
4. TMCNet – Voice Communication in Business URL: https://www.tmcnet.com/voip/0504/featurearticle.htm
5. Android Developers – Platform Overview URL: https://developer.android.com/guide
6. Ktor Embedded Server (CIO) — Documentation URL: https://ktor.io/docs/embedded-server.html
7. Jetpack Compose – Developer Documentation URL: https://developer.android.com/jetpack/compose/documentation
8. PhoneStateListener – Android Developer Docs URL: https://developer.android.com/reference/android/telephony/PhoneStateListener
9. TelephonyManager – Android API Reference URL: https://developer.android.com/reference/android/telephony/TelephonyManager
10. Android DataStore – Jetpack Guide URL: https://developer.android.com/topic/libraries/architecture/datastore
11. Postman – API Testing Tool Documentation URL: https://learning.postman.com/docs/getting-started/introduction/
12. Android Developers – Embedded Servers URL: https://developer.android.com/topic/libraries/architecture/guide#local
13. Ktor Embedded Server – Documentation URL: https://ktor.io/docs/embedded-server.html

14. Android Studio – Official Developer Site URL: https://developer.android.com/studio

15. KotlinLang – About the Kotlin Programming Language URL: https://kotlinlang.org/docs/whatsnew.html

16. Kotlin Coroutines – kotlinx.coroutines.sync.Mutex URL: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.sync/-mutex/

17. Baeldung – Kotlin Coroutine Mutex Explained URL: https://www.baeldung.com/kotlin/coroutine-mutex

18. Android Developers – Testing UI with Espresso URL: https://developer.android.com/training/testing/espresso

19. Android Developers – Testing Fundamentals (JUnit) URL: https://developer.android.com/training/testing/fundamentals

20. CircleCI – Android Testing Best Practices URL: https://circleci.com/blog/android-testing-best-practices/

21. HeadSpin – Manual vs Automated Testing in Mobile Apps URL: https://www.headspin.io/blog/manual-vs-automated-testing

22. Google Developers – Guide to Background Tasks on Android URL: https://developer.android.com/guide/background

23. Android Enterprise – Build for Work URL: https://developer.android.com/work/overview

24. Atlassian – Why small businesses should embrace simple tooling URL: https://www.atlassian.com/blog/teamwork/small-business-tools

25. Small Business Trends – Top CRM Alternatives for Small Businesses URL: https://smallbiztrends.com/2020/10/crm-alternatives-small-business.html

26. Google Developers – Connectivity Best Practices for Android Apps URL: https://developer.android.com/training/basics/network-ops

27. OWASP – Mobile Application Security Verification Standard (MASVS) URL: https://owasp.org/www-project-mobile-app-security/

28. Behavioural Insights Team – Improving Productivity with Transparent Metrics URL: https://www.bi.team/publications/productivity-nudge/

29. TechTarget – Call monitoring systems: benefits and use cases URL: https://www.techtarget.com/searchcustomerexperience/definition/call-monitoring

30. Harvard Business Review – Manage Your Team's Phone Time URL: https://hbr.org/2018/06/manage-your-teams-phone-time-effectively

31. Android Developers – Networking Best Practices URL: https://developer.android.com/training/basics/network-ops/connecting

32. Apache POI – Excel for Java URL: https://poi.apache.org/

33. McKinsey Digital – Using analytics to drive SME growth URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/the-value-of-analytics-in-sme-operations

34. HubSpot – Webhooks API Documentation URL: https://developers.hubspot.com/docs/api/webhooks

35. Google Developers – Offline-first apps URL: https://developer.android.com/topic/performance/vitals/offline

# ДОДАТКИ

# ДОДАТОК А

## Модель CallInfo

```
@Serializable
data class CallInfo(
    val beginning: Long,
    val number: String,
    val name: String,
    val callerNumber: String,
    val callerName: String,
    var duration: Long = 0,
    var timesQueried: Int = 0,
    var ongoing: Boolean = true
) {
    override fun toString(): String {
        return "CallInfo"
    }
}
```

# ДОДАТОК Б

## Обробка дзвінків (CallObserver.kt)

```
class CallObserver @Inject constructor(
    @ApplicationContext private val context: Context,
    private val callRepository: CallRepository,
    private val userInfoRepository: UserInfoRepository
) {
    private val callerName = MutableStateFlow("")
    private val callerNumber = MutableStateFlow("")

    private var telephonyManager: TelephonyManager =
        getSystemService(context, TelephonyManager::class.java) as TelephonyManager
    private val phoneStateListener = object : PhoneStateListener() {
        private var isCallStarted = false

        @Deprecated("Deprecated in Java")
        override fun onCallStateChanged(state: Int, incomingNumber: String?) {
            super.onCallStateChanged(state, incomingNumber)

            when (state) {
                TelephonyManager.CALL_STATE_OFFHOOK -> {
                    isCallStarted = true
                    incomingNumber?.let {
                        val contactName = getContactName(it)

                        callRepository.addOngoingCall(
                            it,
                            contactName,
                            callerNumber.value,
                            callerName.value
                        )
                    }
                }

                TelephonyManager.CALL_STATE_IDLE -> {
                    if (!isCallStarted) {
                        return
                    }
                    incomingNumber?.let {
                        callRepository.endCall(it)
                    }
                    isCallStarted = false
                }

                else -> Unit
            }
        }
    }

    fun start() {
        CoroutineScope(Dispatchers.Default).launch {
            launch {
                userInfoRepository.fromName.collect { callerName.value = it }
            }
            launch {
                userInfoRepository.fromNumber.collect { callerNumber.value = it }
            }
        }
        telephonyManager.listen(phoneStateListener, PhoneStateListener.LISTEN_CALL_STATE)
    }

    fun end() {
        telephonyManager.listen(phoneStateListener, PhoneStateListener.LISTEN_NONE)
    }
}
```

# ДОДАТОК В

## Компонент CallComponent (UI на Jetpack Compose)

```
@Composable
fun CallComponent(call: CallHistory) {
    val dateFormatter = remember {
        DateTimeFormatter.ofPattern("dd MMM yyyy, HH:mm")
    }

    val formattedDate = remember(call.beginning) {
        val instant = Instant.ofEpochMilli(call.beginning)
        val zonedDateTime = instant.atZone(ZoneId.systemDefault())
        dateFormatter.format(zonedDateTime)
    }

    val formattedDuration = remember(call.duration) {
        val hours = call.duration / 3600
        val minutes = (call.duration % 3600) / 60
        val seconds = call.duration % 60

        buildString {
            if (hours > 0) append("$hours год ")
            if (minutes > 0) append("$minutes хв ")
            append("$seconds с")
        }.trim()
    }

    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 4.dp)
    ) {
        Column(modifier = Modifier.padding(8.dp)) {
            Text(text = stringResource(R.string.number_format, call.number))
            Text(text = stringResource(R.string.name_format, call.name))
            Text(text = stringResource(R.string.duration_format, formattedDuration))
            Text(text = stringResource(R.string.ongoing_format, call.ongoing))
            Text(text = stringResource(R.string.beginning_format, formattedDate))
            Text(text = stringResource(R.string.from_number_format, call.callerNumber))
            Text(text = stringResource(R.string.from_name_format, call.callerName))
        }
    }
}
```

# ДОДАТОК Г

## Реалізація локального зберігання користувацьких даних через DataStore (UserLocalDataSource)

```
class UserLocalDataSource(
    private val dataStore: DataStore<Preferences>
) {

    val fromNameFlow: Flow<String> = dataStore.data.map { prefs ->
        prefs[FROM_NAME] ?: ""
    }

    val fromNumberFlow: Flow<String> = dataStore.data.map { prefs ->
        prefs[FROM_NUMBER] ?: ""
    }

    suspend fun saveUserData(name: String, number: String) {
        dataStore.edit { prefs ->
            prefs[FROM_NAME] = name
            prefs[FROM_NUMBER] = number
        }
    }

    companion object{
        private val FROM_NAME = stringPreferencesKey("from_name")
        private val FROM_NUMBER = stringPreferencesKey("from_number")
    }
}
```

# ДОДАТОК Ґ

## Серверна логіка (ServerProviderImpl.kt)

```
internal class ServerProviderImpl(
    private val serverAddressProvider: ServerAddressProvider,
    private val callRepository: CallRepository
) : ServerProvider {
    private var server: EmbeddedServer<*, *>? = null
    private val mutex = Mutex()

    override suspend fun startServer(host: String, port: Int) {
        serverAddressProvider.post(host, port)
        if (!UrlValidator.isValidPort(port)) {
            val serverUrl = serverAddressProvider.getUrl()
            serverAddressProvider.clear()
            throw ServerNotStartedException(serverUrl)
        }
        try {
            mutex.withLock {
                val startTime = System.currentTimeMillis()

                server = embeddedServer(
                    CIO,
                    host = host,
                    port = port,
                ) {
                    install(ContentNegotiation) {
                        json(
                            Json {
                                prettyPrint = true
                                isLenient = true
                            }
                        )
                    }
                    rootRouting(startTime, serverAddressProvider.getUrl()!!.toString())
                    statusRouting(callRepository)
                    logRouting(callRepository)
                }.start(wait = true)
            }
        } catch (e: BindException) {
            e.printStackTrace()
            val serverUrl = serverAddressProvider.getUrl()
            serverAddressProvider.clear()
            throw ServerNotStartedException(serverUrl)
        } catch (e: Exception) {
            e.printStackTrace()
            val serverUrl = serverAddressProvider.getUrl()
            serverAddressProvider.clear()
            throw ServerNotStartedException(serverUrl)
        }
    }

    override suspend fun stopServer() {
        mutex.withLock {
            server?.stop()
            server = null
        }
    }
}
```