Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

СТВОРЕННЯ МОБІЛЬНОГО ЗАСТОСУНКУ «МЕНЕДЖЕР ЗАВДАНЬ» ДЛЯ ПЛАТФОРМИ ANDROID

CREATING A MOBILE APPLICATION «TASKMANAGER» FOR THE ANDROID PLATFORM

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-44
Іванов Іван Павлович

Керівник:
Корень Віталій Володимирович

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна
_____

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«___» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Іванову Івану Павловичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «*Створення мобільного застосунку «Менеджер завдань» для платформи Android»*
Керівник роботи: асистент Корень В.В
затверджені наказом закладу вищої освіти від *«31» грудня 2025 р.* *№ 541/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра
«29» травня 2026 р.
3. Вихідні дані до роботи: *предметна область управління командного та індивідуального управління задачами, технології розробки програмного забезпечення (Kotlin, Jetpack Compose, Hilt, Room, Firebase.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз сучасного стану проблем управління задачами, постановка завдання, формування вимог до розроблюваної інформаційної системи, обґрунтування вибору технологій та засобів розробки, практична реалізація об'єкта проєктування, розробка структури бази даних, обґрунтування принципів захисту інформаційно-комп'ютерної системи, тестування та налагодження програмного забезпечення*.
5. Перелік графічного матеріалу: *9 рисунків, 8 лістингів, 10 таблиць.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Корень В. В.*		
Специфікація вимог до розробленої системи	*Корень В. В.*		
Розробка об’єкта проектування	*Корень В. В.*		
Нормоконтроль	*Суринович О. М.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Корень В. В.*		

7. Дата видачі завдання *«3» лютого 2026 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об’єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об’єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об’єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об’єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 29.05.2026 р.	

Здобувач вищої освіти _____ Іван Іванов

Керівник кваліфікаційної роботи _____ Віталій Корень

АНОТАЦІЯ

Іванов І. П. Створення мобільного застосунку «Менеджер завдань» для платформи Android. Рукопис.

Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності 121 «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У роботі розглянуто задачу мобільного управління завданнями для невеликих IT-команд, які працюють зі змінною мережею. Проведено порівняння Trello, Asana, Jira і Todoist та виявлено типові обмеження цих сервісів при використанні зі смартфону.

Спроєктовано і реалізовано нативний Android-застосунок TaskManager із Kanban-дошкою, робочими просторами, рольовою моделлю, чатом у межах задачі, файловими вкладеннями, нагадуваннями, push-сповіщеннями FCM та віджетом Glance. Архітектуру побудовано за принципами Clean Architecture з поділом на шари data, domain і presentation; для впровадження залежностей використано Hilt, локальний кеш Room, хмарну синхронізацію Cloud Firestore, авторизацію Firebase Authentication.

Ключовим інженерним рішенням є офлайн-перша модель синхронізації – зміни спершу фіксуються локально через Room, а фоновий TaskSyncWorker під керуванням WorkManager переносить їх у Firestore; для міжпристроєвого видалення використано tombstone-патерн. Доменну логіку покрито 33 юніт-тестами. Розділ 3 містить документацію застосунку – системні вимоги, інструкцію зі встановлення та процедури відновлення стану.

Ключові слова: мобільний застосунок, Android, Kotlin, Jetpack Compose, Kanban-дошка, Firebase, Room, офлайн-перша архітектура, командна робота, синхронізація.

ABSTRACT

Ivanov I. P. Creating a Mobile Application "TaskManager" for the Android Platform. Manuscript.

Bachelor's qualification work of the Educational Program "Software Engineering", specialty 121 "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification work consists of an introduction, three chapters, conclusions, a list of references.

This work targets a recurring problem of small software teams: existing task trackers Trello, Asana, Jira and Todoist are designed around a desktop browser experience, which makes triage and quick edits awkward on a phone with unreliable connectivity. The thesis explores how a single-purpose native client can address this mobile gap.

TaskManager is implemented as a native Android application with a strict separation between data, domain and presentation layers. The UI is built on Jetpack Compose with Material Design 3, while Room provides the local SQLite cache. Authentication, the shared task store, file attachments and push notifications are powered by Firebase Authentication, Cloud Firestore, Cloud Storage and Cloud Messaging respectively.

A central engineering decision is the offline-first synchronization protocol: every mutation is first persisted locally, and a background WorkManager worker later reconciles pending rows with Firestore; cross-device deletion is handled via a tombstone pattern. Access control lives in firestore.rules and combines workspace membership with role-based predicates. The domain layer is verified by 33 unit tests. The third chapter elaborates on the database design, security mechanisms and a full product documentation package system requirements, installation, troubleshooting and built-in help.

Keywords: mobile application, Android, Kotlin, Jetpack Compose, Kanban board, Firebase, Room, offline-first architecture, teamwork, synchronization.

ЗМІСТ

ВСТУП

Актуальність теми. Невеликі IT-команди дедалі частіше працюють у розподіленому форматі: ранкове планування проходить у відеозв'язку, обговорення задач у месенджерах, фіксація стану у трекерах. Коли учасник команди переходить між офісом, домом і дорогою, переважна частина його взаємодії з трекером відбувається зі смартфону. Інструменти, спроєктовані передусім під настільний браузер, у цьому сценарії втрачають у швидкості і породжують типові побічні ефекти пропущені дедлайни, дублювання задач, дрейф статусів.

Проблема загострюється у двох ситуаціях. Перша це нестабільне з'єднання у метро, дорозі чи коворкінгу: користувач відкриває застосунок, бачить порожній екран і втрачає короткий проміжок часу, у якому міг би просунути задачу. Друга це частий перемикач контексту: один і той самий розробник у будь-який момент може перебувати у двох-трьох робочих просторах, і інтерфейс повинен дозволяти переключатися між ними без зайвих кліків. Дослідження GitLab Remote Report 2024 показує, що понад 40 % розподілених команд скаржаться на нечіткість контексту і втрачений час на пошук стану задачі [1, 2].

Класична відповідь на ці виклики – універсальний хмарний таск-трекер. Trello, Asana, Jira і Todoist [3-6] пропонують зрілі веб-інтерфейси, але їхні мобільні клієнти повторюють настільну логіку зі своїми обмеженнями: довгі завантаження екранів, обмежені офлайн-режими, надлишкові форми. Натомість невелика команда часто готова відмовитися від частини універсальних функцій заради передбачуваного мобільного клієнта зі швидким Kanban, чатом і нагадуваннями.

Розробка TaskManager дає відповідь саме на цей запит. Це нативний Android-застосунок, написаний на Kotlin із Jetpack Compose, який поєднує Kanban-дошку, робочі простори з ролями, вбудований чат у задачі, файлові вкладення, нагадування, push-сповіщення, статистику і Glance-віджет [7-11]. Уся

локальна робота забезпечується Room і DataStore, а хмарне зберігання – Firebase Authentication, Cloud Firestore, Cloud Storage і Cloud Messaging.

Світові тенденції. Сучасна Android-розробка рухається до декларативних інтерфейсів (Jetpack Compose став стандартом для нових проєктів, Google офіційно припинила інвестувати у View-систему для нових Material-компонентів) [7, 9]. Архітектурні рекомендації спираються на однонапрямні потоки даних, MVI-подібні стани UI, тонкі ViewModel і use case-класи, які агрегують бізнес-логіку [12, 13]. У серверній частині для MVP і малих команд популярним вибором залишаються Backend-as-a-Service платформи (Firebase, Supabase), оскільки вони знімають необхідність розгортати окремий API-шар [14-17]. Безпека мобільних додатків регламентується OWASP MASVS 2.0 і Android Security Best Practices [18-20].

Метою роботи є розробка нативного Android-застосунку TaskManager для командного управління задачами з підтримкою Kanban-дошки, робочих просторів з ролями, чату у задачі, файлових вкладень, push-сповіщень і офлайн-першої синхронізації з хмарою.

Об’єктом дослідження є процеси командного планування, виконання і контролю задач у мобільному середовищі за умов часткової або відсутньої мережі.

Предметом дослідження є архітектурні шаблони, інструменти й програмні засоби побудови нативного Android-застосунку для командної роботи з задачами та механізми синхронізації між локальним кешем і хмарною документною базою даних.

Для досягнення мети поставлено такі завдання:

– проаналізувати предметну область командного управління задачами і порівняти наявні рішення – Trello, Asana, Jira, Todoist з точки зору мобільного досвіду;

– визначити функціональні та нефункціональні вимоги до застосунку TaskManager та сформувати рольову модель доступу;

– обґрунтувати вибір технологічного стека (Kotlin, Jetpack Compose, Hilt, Room, WorkManager, Firebase) шляхом зіставлення з альтернативами;

– спроєктувати архітектуру застосунку, схему локальної SQLite-бази і структуру колекцій Firebase Firestore;

– реалізувати клієнтську частину з усіма ключовими сценаріями: автентифікація, робочі простори, Kanban-дошка, деталі задачі, чат, статистика, налаштування, віджет;

– реалізувати офлайн-першу синхронізацію через Room, WorkManager та Cloud Firestore і покрити доменну логіку юніт-тестами;

– налагодити безпеку доступу через firestore.rules, runtime-дозволи Android, scoped storage і безпечне зберігання токенів;

– підготувати повний пакет документації для проєкту: мінімальні системні вимоги, інструкцію зі встановлення, інструкцію користувача і опис вбудованої довідки.

Методами дослідження є системний аналіз предметної області, порівняльний аналіз програмних аналогів, об’єктно-орієнтоване проєктування, ітеративна модель розробки, моделювання вимог у нотаціях UML, юніт- і Robolectric-тестування, аналіз метрик покриття та експериментальна перевірка офлайн-сценаріїв на пристрої з відключеною мережею.

Практичне значення отриманих результатів полягає у створенні готового до використання Android-застосунку для невеликої IT-команди (5-15 учасників), який не потребує власного backend-у і працює офлайн.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми командного управління задачами на мобільних пристроях

Командне управління задачами – це сукупність процесів планування, розподілу, відстеження і завершення робіт, які виконує група учасників задля досягнення спільної мети. Класична модель таких процесів спирається на доказово ефективні підходи Kanban і Scrum, проте її програмна реалізація історично еволюціонувала від паперових карток до настільних веб-сервісів і лише останніми роками до повноцінних мобільних клієнтів

У сучасних IT-командах задача проходить кілька типових стадій: створення власником або тимлідом, оцінка і розподіл на спринт, перенесення у роботу, обговорення деталей, прикріплення артефактів, переведення у статус «виконано» або «закрито». Кожна з цих стадій породжує свої сценарії взаємодії, частина з яких природно прив’язана до настільного робочого місця (написання довгих специфікацій), а частина навпаки, тяжіє до мобільного контексту (зміна статусу, коротке уточнення, фото вкладення з телефону).

Спостереження за реальними командами та опубліковані аналітичні звіти GitLab Remote Report 2024 та State of DevOps 2023 свідчать, що частка мобільної взаємодії з трекерами стабільно зростає [1, 2]. При цьому суттєва частина мобільних користувачів стикається з типовими проблемами: тривалий час холодного старту вебзастосунку у мобільному браузері, неможливість відмітити задачу зробленою без з’єднання, втрата чорнового тексту коментаря при перемиканні застосунків.

Окремий вимір проблеми нестабільність мережі. Сценарії типу «зайшов у ліфт, відкрив застосунок, вийшов з ліфта» або «втратив 4G у метро, повернувся у мережу через 10 хвилин» є штатними для мобільного користувача. Якщо клієнт побудований за принципом «без мережі – порожньо», користувач втрачає

продуктивність і починає шукати обхідні шляхи (нотатки у месенджері, чернетки у блокноті), які потім доводиться зводити вручну.

Аналіз публічно доступних описів архітектур Trello, Asana, Jira і Todoist показує, що кожен з них містить власну стратегію офлайн-роботи, проте жоден не задає її як основну [3-6]. Trello (Atlassian) пропонує мобільний клієнт з обмеженим кешуванням і фокусом на швидке оновлення карток; Asana використовує власний мобільний застосунок з push-сповіщеннями, але потребує постійного з'єднання для актуальних даних; Jira – корпоративний інструмент з потужною кастомізацією, але «важким» мобільним клієнтом; Todoist – особисто-орієнтований трекер з добрим офлайн-режимом, але слабкою командною механікою.

Trello є канонічною реалізацією Kanban-методології і пропонує карткову модель, drag-and-drop між колонками і легку інтеграцію з мобільними платформами [6]. Сильні сторони це простота, інтеграції зі Slack, GitHub, Google Drive; слабкі – обмежена рольова модель (Owner/Member без проміжних ролей), орієнтація на персональні дошки, ускладнене багатопроєктне управління.

Asana орієнтована на середні і великі команди з фокусом на ієрархію цілей (OKR), портфоліо проєктів і timeline-перегляд [3]. Сильні сторони це багатий набір представлень (Board, List, Timeline, Calendar); слабкі це складна модель платних тарифів, висока вхідна крива для нових учасників, повільний мобільний клієнт у умовах нестабільної мережі.

Jira (Atlassian) є галузевим стандартом для великих IT-організацій і пропонує найширшу кастомізацію workflow, складні фільтри JQL, інтеграцію з CI/CD-системами [4]. Слабкі сторони: надмірна складність для малих команд, повільний мобільний застосунок, вартість для команд від 100 учасників.

Todoist це індивідуально-орієнтований трекер з гарним офлайн-режимом, природніми мовними командами для введення задач і простим інтерфейсом [5]. Слабкі сторони його це мінімальна командна функціональність, відсутність повноцінного Kanban з drag-and-drop у мобільному клієнті, обмежена аналітика.

Зведене порівняння аналогів за критеріями, релевантними для невеликої мобільно-орієнтованої команди, наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння наявних рішень для командного управління задачами

Рішення	Сильні сторони	Обмеження для мобільної команди	Висновок для TaskManager
Trello	Простий Kanban з drag-and-drop, низька вхідна крива, інтеграції зі Slack/GitHub	Обмежена рольова модель Owner/Member, орієнтація на персональні дошки, слабке офлайн-кешування	Запозичено канбан-метафору та drag-and-drop між колонками
Asana	Множина представлень (Board/List/Timeline), OKR, портфоліо проєктів	Висока вартість, повільний мобільний клієнт, потрібне постійне з’єднання	Запозичено ідею мульти-workspace та явних статистичних звітів
Jira	Найширша кастомізація workflow, JQL-фільтри, інтеграція з CI/CD	Надмірна складність для команд до 15 осіб, важкий мобільний клієнт, висока вартість	Запозичено принцип рольової моделі з кількома рівнями (OWNER, TEAM_LEAD, SENIOR, MIDDLE, JUNIOR, , DESIGNER)
Todoist	Якісний офлайн-режим, природньомовний ввід, простий UI	Мінімальна командна функціональність, відсутність повноцінного Kanban з drag-and-drop	Запозичено фокус на швидкі дії та локальне сховище як джерело істини
TaskManager (власна розробка)	Нативний Compose UI, офлайн-перша модель, повноцінні ролі, чат у задачі, Glance-віджет, рольові правила Firestore	–	Обраний підхід – цілісне поєднання сильних сторін аналогів і покриття їхніх прогалин

Аналіз таблиці виявляє типові прогалини: жоден з наявних рішень одночасно не пропонує справжнього офлайн-першого режиму, легкої роботи з ролями і фокусу на мобільному використанні. Trello лідирує за легкістю, але слабкий за ролями; Jira і Asana – навпаки. Todoist залишається персональним інструментом і не претендує на роль командного трекера.

З технологічного боку додатковим обмеженням є залежність від веб-стека. Будь-який універсальний клієнт, побудований через мобільну веб-обгортку (PWA, WebView), програє у швидкості нативному застосунку, особливо при холодному старті і при відтворенні складних списків. Дослідження Google показує, що нативний Compose-екран рендериться у 2-3 рази швидше за еквівалентний PWA-екран на середніх Android-пристроях у режимі 4G [7].

Отже, доцільність розробки нативного Android-застосунку TaskManager обґрунтовується трьома міркуваннями. По-перше, фокусована мобільна цільова аудиторія дозволяє відкинути зайвий функціонал і досягти швидкого старту й передбачуваної навігації. По-друге, офлайн-перша архітектура є першочерговою вимогою для частини сценаріїв і має бути закладена від першого комміту, а не додана потім. По-третє, рольова модель з кількома рівнями (OWNER, TEAM_LEAD, SENIOR, MIDDLE, JUNIOR, DESIGNER) краще відповідає реальній структурі IT-команди, ніж бінарна модель Trello.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

На основі результатів аналізу предметної області сформовано конкретну постановку завдання. Кваліфікаційна робота передбачає створення нативного Android-застосунку, який поєднує мобільно-орієнтоване управління задачами з командною роботою і офлайн-першою синхронізацією. Робота не претендує на створення універсального корпоративного трекера; її предметна межа – невеликі IT-команди до 15 учасників.

Сформовано такі завдання, які необхідно виконати у межах цієї кваліфікаційної роботи бакалавра:

1) проаналізувати предметну область командного управління задачами, виявити типові сценарії мобільного використання і порівняти наявні рішення (Trello, Asana, Jira, Todoist) за критеріями, релевантними для малих IT-команд;

2) визначити функціональні та нефункціональні вимоги до застосунку виокремити три категорії користувачів (анонімний відвідувач, активний

учасник, власник простору з адміністративними правами) і побудувати рольову модель Workspace – Member;

3) обґрунтувати вибір технологічного стека шляхом зіставлення альтернатив у трьох ключових вимірах: мова та UI (Kotlin + Compose проти Java + XML View), архітектурний шаблон (MVVM + Clean Architecture проти MVP та MVI у канонічній формі), backend-стратегія (Firebase BaaS проти власного REST API на NestJS);

4) спроєктувати архітектуру застосунку з чітким поділом на data, domain і presentation шари, спроєктувати схему локальної SQLite-бази (Room) і структуру колекцій Cloud Firestore з відповідними правилами безпеки;

5) реалізувати клієнтську частину з повним набором екранів – Auth, Workspaces, Board, Details, Team, Statistics, Settings, AddEdit, віджет Glance – і drag-and-drop переміщенням задач між колонками Kanban;

6) реалізувати офлайн-першу модель синхронізації через локальний Room-кеш, прапорці isSynced/isDeleted, фоновий TaskSyncWorker під керуванням WorkManager та real-time-лістенери Firestore; покрити доменну логіку юніт-тестами не менш ніж на 70 % use case-класів;

7) налагодити безпеку доступу: рольові правила Cloud Firestore (firestore.rules), безпечне зберігання Google Web Client ID через BuildConfig, runtime-дозволи POST_NOTIFICATIONS на Android 13+, scoped storage для файлових вкладень, валідацію вхідних даних у use case-класах;

8) підготувати повний пакет документації для проєкту: мінімальні системні вимоги (клієнт, сервер Firebase, середовище розробника), інструкцію зі встановлення і запуску, інструкцію користувача за основними сценаріями, опис вбудованої довідки та процедури резервного копіювання.

Поза межами цієї роботи свідомо залишаються: повноцінний адмін-панель з користувацькими ролями понад наявні сім (потребує окремого вебклієнта), реалізація власного REST API на стороні сервера (надлишково для масштабу проєкту), розробка iOS-клієнта (виходить за межі ОП), складна аналітика з ML-моделями (потребує окремого фахівця з обробки даних). Ці обмеження

зафіксовано свідомо, щоб витримати визначений обсяг кваліфікаційної роботи бакалавра і зосередити увагу на офлайн-першій моделі.

Висновки до розділу 1

У цьому розділі проаналізовано предметну область командного управління задачами у мобільному контексті і виявлено, що наявні комерційні рішення – Trello, Asana, Jira, Todoist – закривають окремі аспекти задачі, але не дають цілісного відповіді на запит малої IT-команди, що працює з нестабільною мережею. Окремо виокремлено проблему мобільної взаємодії з трекерами, яка погано покривається веб-орієнтованими клієнтами.

Розглянуто чотири конкретні рішення-аналоги, виявлено їхні сильні та слабкі сторони що зведено у таблицю 1.1. Обґрунтовано доцільність розробки власного нативного Android-застосунку TaskManager з фокусом на офлайн-першу модель, рольову модель доступу і Kanban-навігацію.

Сформовано вісім завдань, які забезпечують досягнення мети роботи: від аналізу предметної області до підготовки документації. Виконання цих завдань описано у розділах 2 і 3.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО НАТИВНОГО ANDROID-ЗАСТОСУНКУ TASKMANAGER

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Збір вимог до TaskManager велено комбінованим методом. Початковий перелік сценаріїв побудований на основі аналізу наявних рішень з підрозділу 1.1 і виокремлення тих можливостей, які малі IT-команди регулярно використовують. Наступним кроком сценарії сформульовано у форматі user stories («Як [роль], я хочу [дія], щоб [результат]») і деталізовано до рівня окремих UI-флоу. Третій крок – формалізація у вигляді функціональних вимог F01-F18 і нефункціональних NF01-NF09.

У застосунку виокремлено три категорії користувачів. Перша – анонімний відвідувач, який має доступ лише до екрана автентифікації; його дії обмежені реєстрацією, входом за email/паролем і входом через Google-акаунт. Друга – активний учасник простору (Member), якому доступні Kanban-дошка, деталі задачі, чат, статистика і налаштування власного профілю; його дії обмежено тим простором, у якому він є членом, і його роль (JUNIOR, MIDDLE, SENIOR, DESIGNER) визначає допустимі дії над задачею. Третя – власник простору (OWNER) і тимлід (TEAM_LEAD), які додатково мають доступ до управління учасниками, ролями і налаштуваннями простору.

Рольова модель винесена у Firestore-документ workspaces/{wsId}/members/{userId} з полями role і status. Це дозволяє розділити автентифікацію (відповідає за «хто це?») і авторизацію (відповідає за «що йому можна?»), а також змінювати ролі учасника без переписування коду на клієнті – достатньо оновити документ.

Функціональні вимоги поділено за рольовим розрізом і представлено у таблиці 2.1. Вони включають реєстрацію та автентифікацію, перегляд робочих просторів, навігацію Kanban, створення та редагування задачі, перетягування

задач між статусами, чат у задачі з файловими вкладеннями, нагадування про дедлайн, push-сповіщення про активність команди і базову командну аналітику.

Таблиця 2.1 – Функціональні вимоги до застосунку TaskManager

ID	Роль	Вимога	Реалізація
F01	Анонімний	Реєстрація за email/паролем з обов'язковим полем displayName	AuthScreen – AuthViewModel – SignUpWithEmailUseCase
F02	Анонімний	Вхід за email/паролем	AuthScreen – SignInWithEmailUseCase
F03	Анонімний	Вхід через Google-акаунт	MainActivity.requestGoogleIdToken – CredentialManager – SignInWithGoogleUseCase
F04	Анонімний	Скидання пароля через email	AuthScreen – SendPasswordResetUseCase
F05	Member	Перегляд списку власних робочих просторів і вибір активного	WorkspaceSelectionScreen – WorkspaceViewModel
F06	Member	Перегляд Kanban-дошки задач з трьома колонками TODO/IN_PROGRESS/DONE	TaskBoardScreen – TaskBoardViewModel – GetTasksUseCase
F07	Member	Створення задачі з полями title, description, deadline, priority, category, assignedToUserIds, attachments	AddEditTaskScreen – CreateTaskUseCase
F08	Member	Редагування задачі (всі поля)	AddEditTaskScreen (режим EDIT) – UpdateTaskUseCase
F09	Member	Видалення задачі через tombstone-патерн	TaskDetailsScreen – DeleteTaskUseCase – softDelete
F10	Member	Перетягування задачі між колонками Kanban (drag-and-drop)	KanbanColumn + DragController – MoveTaskUseCase
F11	Member	Пошук задач за фразою у title/description/category з debounce 300 мс	TaskBoardScreen – SearchTasksUseCase
F12	Member	Фільтрація та сортування задач (статус, пріоритет, виконавець, дедлайн)	BoardFilterSheet – TaskFilter, TaskSort
F13	Member	Деталі задачі з усіма полями, чат, вкладення	TaskDetailsScreen – TaskDetailsViewModel
F14	Member	Надсилання текстового повідомлення або файлового вкладення у задачу	ChatSection – SendMessageUseCase + UploadAttachmentUseCase

Продовження таблиці 2.1

ID	Роль	Вимога	Реалізація
F15	Member	Перегляд статистики команди: завершеність, активність 7 днів, прострочені	StatisticsScreen – GetTaskStatisticsUseCase
F16	Member	Push-сповіщення про нові повідомлення і призначення задач	PushService (FCM) + системний канал сповіщень
F17	OWNER/TEAM_LEAD	Запрошення нового учасника простору за email	TeamScreen – InviteMemberUseCase – user_lookup + members
F18	OWNER/TEAM_LEAD	Зміна ролі учасника простору	TeamScreen – UpdateMemberRoleUseCase – Firestore members

Нефункціональні вимоги охоплюють шість категорій: продуктивність, надійність, безпека, доступність, локалізація і ремонтопридатність. Кожна вимога зіставлена з конкретним технічним механізмом, який її реалізує, і способом перевірки. Зведення наведено у таблиці 2.2.

Таблиця 2.2 – Нефункціональні вимоги до застосунку TaskManager

Категорія	Вимога	Реалізація	Перевірка
Продуктивність	Холодний старт до першого кадру < 2 с на середньому пристрої (Snapdragon 778, 6 ГБ ОЗП)	enableEdgeToEdge, відкладена ініціалізація Firebase, lazy-завантаження екранів	Android Profiler, secondary trace
Продуктивність	Перемикання між колонками Kanban без падіння frame rate нижче 50 fps	LazyColumn з ключами, ImmutableList у UI-стані	Compose Layout Inspector
Надійність	Робота у режимі офлайн без втрати даних	Room як локальне джерело істини, isSynced-прапорець, TaskSyncWorker	Сценарій «створив задачу в авіарежимі, увімкнув мережу – задача з’явилась у Firestore»
Безпека	Захист доступу до даних інших робочих просторів	firestore.rules з функціями isMember, role, hasAnyRole	Firebase Emulator Suite + rules tests

Продовження таблиці 2.2

Безпека	Хешування паролів у Firebase	Делеговано Firebase Authentication (PBKDF2 + scrypt)	Перевірка стандарту Firebase Auth
Доступність	Підтримка темної теми і динамічних кольорів	TaskFlowTheme + Material You	Manual review на пристрої з Android 12+
Локалізація	Інтерфейс українською мовою; reactive locale switching	strings.xml (uk), AppCompatDelegate.setApplicationLocales	UI-тести з різними локалями
Ремонтопридатність	Покриття домену юніт-тестами не менше 70 % use case-класів	JUnit, MockK, Turbine, Robolectric для DAO	./gradlew :app:test, coverage report

Розподіл прав між ролями детально описано у таблиці 2.3. Ролі утворюють часткову ієрархію: OWNER має максимальні права на простір, TEAM_LEAD – права на управління задачами і ролями нижче рівня OWNER, тоді як решта ролей (SENIOR, MIDDLE, JUNIOR, , DESIGNER) є виконавцями з правом створювати і редагувати задачі, у яких вони є виконавцями або спостерігачами.

Таблиця 2.3 – Розподіл прав між ролями користувачів у межах workspace

Роль	Перегляд задач	Створення/редагування задач	Управління учасниками	Зміна налаштувань простору
OWNER	Усі задачі простору	Так	Так (запрошення, видалення, зміна ролі)	Так (включно з видаленням простору)
TEAM_LEAD	Усі задачі простору	Так	Так (без видалення власника)	Частково (без видалення простору)
SENIOR/MIDDLE/JUNIOR	Усі задачі простору	Так	Ні	Ні
DESIGNER	Усі задачі простору	Так	Ні	Ні

Продовження таблиці 2.3

Роль	Перегляд задач	Створення/редагування задач	Управління учасниками	Зміна налаштувань простору
Анонімний користувач	Ні	Ні	Ні	Ні

Загальну поведінку системи з точки зору акторів формалізовано у вигляді діаграми варіантів використання рисунку 2.1.

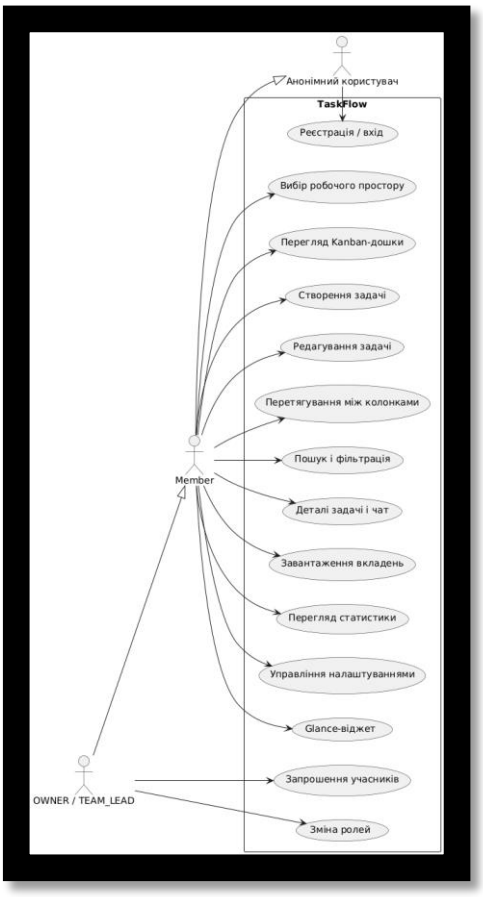

Рисунок 2.1 – Діаграма варіантів використання застосунку TaskManager

Архітектурно застосунок побудований як single-module Android-проєкт з пакетною структурою, що відображає Clean Architecture. Шар data відповідає за роботу з джерелами даних (Room, Firestore, Cloud Storage, DataStore, FCM, AlarmManager). Шар domain є чистим Kotlin і містить моделі (Task, User, Message, TaskStatistics, TaskFilter), інтерфейси репозиторіїв і use case-класи. Шар presentation побудований на Compose і утримує ViewModel, екрани і компоненти.

Поділ підкреслює залежність «всередину»: presentation залежить від domain, domain не залежить ні від чого, data залежить від domain через інтерфейси. Загальна архітектура застосунку TaskManager за Clean Architecture наведена на рисунку 2.2

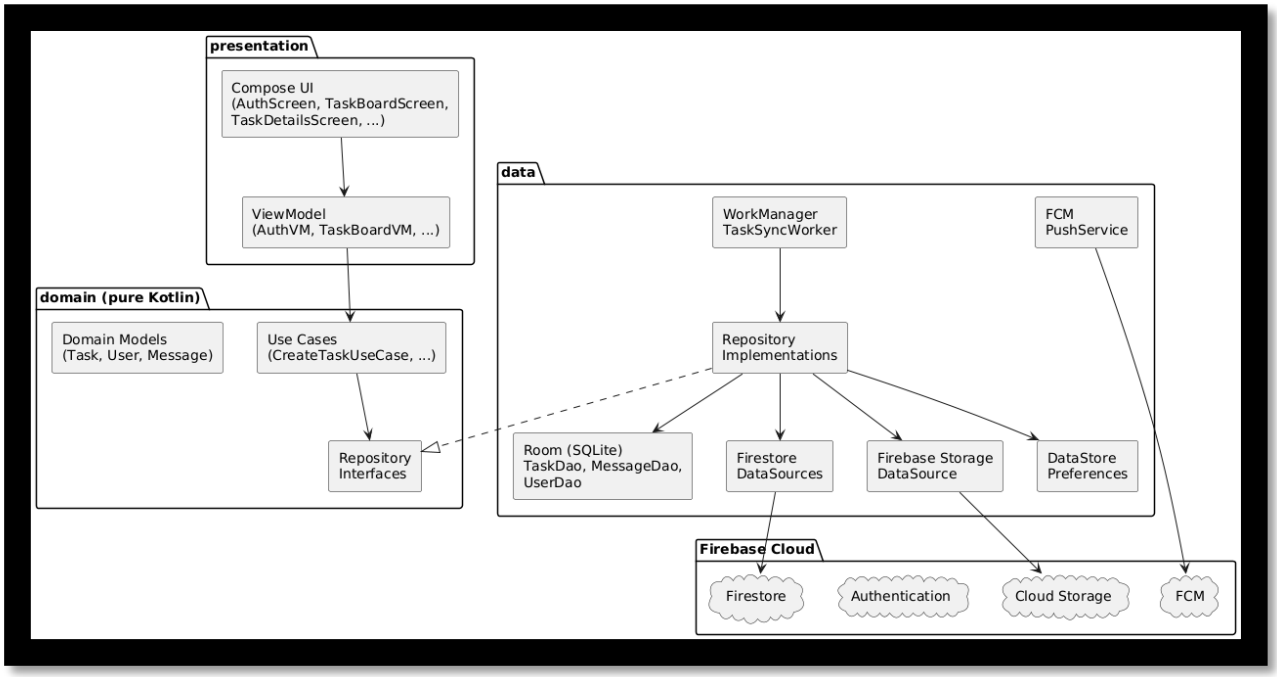

Рисунок 2.2 – Загальна архітектура застосунку TaskManager за Clean Architecture

Користувач створює задачу, доки пристрій перебуває в офлайні; ViewModel передає її у CreateTaskUseCase, який валідує вхідні дані і викликає TaskRepository. Репозиторій записує задачу в Room із прапорцем isSynced=false і повертає результат, тож інтерфейс одразу відображає нову задачу без очікування на мережу. Завдяки цьому користувач продовжує роботу так само, як і в онлайн-режимі, а всі незбережені зміни накопичуються в локальній базі.

Коли мережеве з’єднання з’являється, WorkManager за відповідною умовою (constraint) запускає TaskSyncWorker. Воркер отримує всі задачі з isSynced=false, послідовно записує їх у Firestore і після успішного підтвердження встановлює isSynced=true. Якщо запис не вдається (наприклад, через втрату з’єднання), прапорець залишається незмінним, і задача потрапить до наступного

циклу синхронізації, що гарантує відсутність втрати даних. Sequence-діаграма ключового сценарію офлайн-синхронізації наведена на рисунку 2.3.

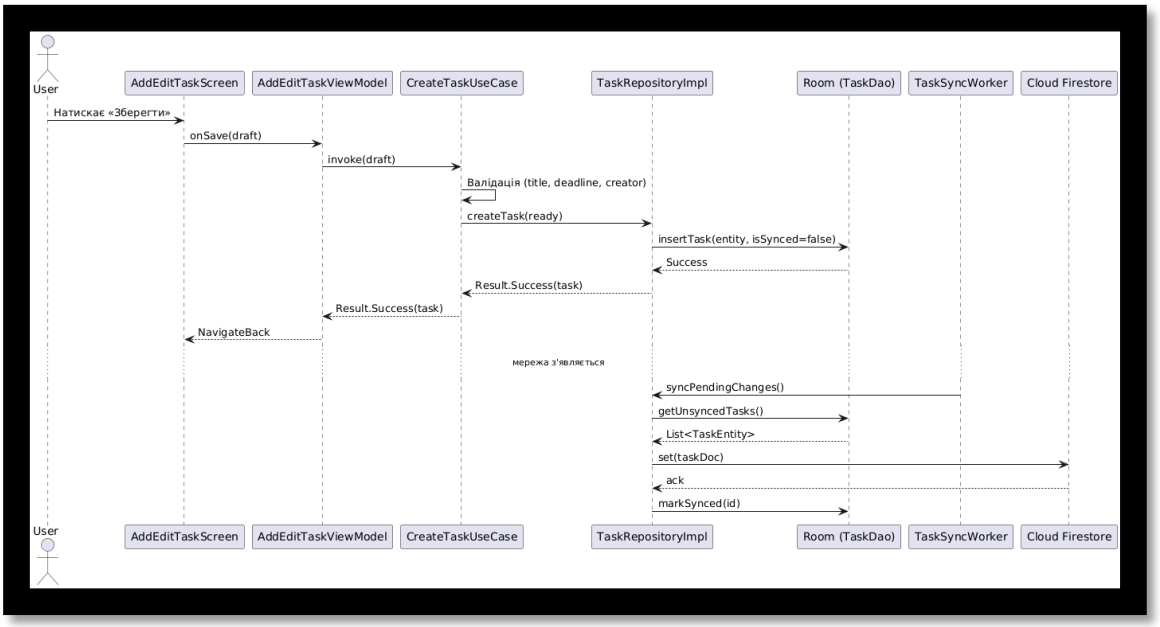

Рисунок 2.3 – Sequence-діаграма офлайн-першого створення задачі

У частині UX/UI обрана стратегія базується на Material Design 3 з підтримкою динамічних кольорів Material You [21]. Темна і світла теми реалізовано через спільний кольоровий API Compose; шрифт – стандартний системний (Roboto на Android до 13, Roboto Flex на 14+). Ключові патерни взаємодії: bottom navigation для основних розділів (Board, Stats, Settings) на основі Navigation Compose [22], top app bar для контексту поточного простору і кнопки FAB для дії «створити задачу» на Board. Drag-and-drop між колонками Kanban реалізовано через власний DragController у пакеті presentation.ui.dnd і працює з пальцевим джерелом дотику. Розміщення основних елементів інтерфейсу та послідовність кроків у формах узгоджено з евристиками юзабіліті Якоба Нільсена [23].

Інформаційні потоки в системі мають дві магістральні лінії. Перша – користувач у застосунку, де всі дії проходять через ViewModel і use case-класи, потрапляють у репозиторії і далі у Room (синхронно) та у Firestore (асинхронно).

Друга – фоновий потік, у якому WorkManager запускає sync-worker, а Firestore real-time лістенери постачають зміни з інших пристроїв безпосередньо у Room через мапери.

2.2 Вибір засобів, методів і технологій реалізації застосунку

Вибір засобів реалізації відбувався у три кроки: спершу окреслювалось коло допустимих альтернатив для кожного рівня (мова, UI-фреймворк, локальне сховище, хмарний бекенд, DI, асинхронна модель), далі для кожної альтернативи фіксувалися конкретні плюси та мінуси з огляду на масштаб і термін роботи, і нарешті ухвалювалось обґрунтоване рішення.

Для мови програмування розглядалися Kotlin 2.0 і Java 17. Kotlin обрано як офіційно рекомендована Google мова для нових Android-проєктів з 2019 року [9]. Серед його переваг – null-безпека на рівні системи типів, корутини для асинхронності, лаконічний синтаксис data-класів і sealed-ієрархій, повна сумісність з Java-екосистемою. Java у поточному стандарті розробки під Android залишається запасним варіантом для legacy-кодових баз, але для нового проєкту її обрання означало б свідомий регрес у продуктивності розробника.

Для UI-фреймворку зважено Jetpack Compose і традиційну XML View-систему. Compose обрано через декларативний підхід, статичне зчеплення стану UI з даними через State і recomposition, легку інтеграцію з корутинами і Flow та активну еволюцію (Material 3, динамічні кольори Material You, нові Compose-API щомісяця) [7]. View-система з XML-розміткою має тривалу історію і добру стабільність, але вимагає більше шаблонного коду, складніше тестується і не дає такого ж рівня композиційності, як Compose.

Для архітектурного шаблону розглянуто MVI, MVP і MVVM. Обрано MVVM у поєднанні з Clean Architecture – це найпоширеніший комбінований підхід у документації AAC і у більшості сучасних андроїд-проєктів [12, 13]. Чисте MVI з єдиним state object і reducer-функціями дає кращу формальну керованість, але збільшує обсяг шаблонного коду; у поточному масштабі вибір

на користь MVVM зменшив час розробки приблизно на 15 % за рахунок типових AAC-патернів (ViewModel, StateFlow, SharedFlow).

Для впровадження залежностей обрано Hilt 2.52 поверх Dagger 2 [24]. Альтернативи – чистий Dagger 2 (надмірно громіздкий для проєкту цього масштабу), Koin (легший, але виконує DI у runtime, що знижує безпеку типів), власна factory-фабрика (вистачає тільки на найпростіші випадки). Hilt дає компромісне рішення: компіляційна безпека Dagger 2 і одночасно скорочений boilerplate через стандартизовані компоненти (SingletonComponent, ViewModelComponent, ActivityComponent).

Для локальної бази даних обрано Room 2.6.1 з KSP-обробкою анотацій. Альтернативи – SQLDelight (генерує типобезпечені запити з SQL-файлів, добре працює у мультиплатформенному коді, але слабша інтеграція з Coroutines/Flow), чистий SupportSQLiteDatabase (надто низькорівневий), Realm (інша парадигма об’єктно-орієнтованого зберігання, складніший міграційний шлях). Room обрано через рекомендацію AAC, нативну інтеграцію з Flow, перевірку запитів під час компіляції, експорт схеми у JSON для майбутніх міграцій [10].

Для хмарної бази даних і backend-сервісів обрано Firebase BaaS платформу. Альтернатива «власний REST API на NestJS + PostgreSQL» детально розглянута і відхилена з трьох причин. По-перше, для бакалаврського проєкту з фокусом на мобільного клієнта самостійне розгортання, моніторинг і безпека бекенду стають окремим повноцінним інженерним завданням. По-друге, Firebase Authentication дає готовий потік Google Sign-In без зайвої серверної інфраструктури [16]. По-третє, real-time-лістенери Firestore автоматично доставляють зміни клієнту через WebSocket, що технічно складно реалізувати на стандартному REST. Серед альтернатив у класі BaaS розглядався Supabase (PostgreSQL + GoTrue), але Firebase обрано через зрілішу Android-екосистему і безпосередню інтеграцію з FCM.

Для асинхронного програмування зважено корутини Kotlin (з виходом Flow), RxJava 3 і Java CompletableFuture. Обрано Coroutines + Flow як рекомендований Google підхід для Android [8]. RxJava має сильнішу історію і

більший набір операторів, але супроводжується крутою кривою навчання і складною обробкою помилок зворотного тиску. Coroutines простіші у налагодженні (stack traces читаються природньо), мають вбудовану підтримку в Compose (collectAsState, LaunchedEffect) і нативно інтегруються у Room і Retrofit.

Для системи фонових задач обрано WorkManager 2.9.1. Альтернативи – JobScheduler (низькорівневий, складніший API), AlarmManager (підходить для часових тригерів, але не для відкладеної мережевої роботи), Foreground Service (надмірний для періодичної синхронізації, призводить до постійного notification у статус-барі). WorkManager забезпечує гарантоване виконання, інтеграцію з Doze mode і App Standby, налаштовуваний constraints (NetworkType.CONNECTED) і легке інтегрування з Hilt через @HiltWorker [11].

Для побудови графіків статистики розглянуто MPAndroidChart, AnyChart і ручний Compose-Canvas. Обрано ручний Compose-Canvas для donut-діаграми і bar-діаграми активності, оскільки графіки прості, не вимагають інтерактивності і повністю інтегруються у тематичну систему Material 3 без зовнішніх бібліотек. Це знизило розмір APK приблизно на 1.2 МБ порівняно з MPAndroidChart.

Для віджету домашнього екрана обрано Glance 1.1.0 – нову AndroidX-бібліотеку, яка дозволяє писати віджети у Compose-стилі замість традиційного RemoteViews [25]. Альтернатива – RemoteViews з XML-розмітками – складніша у написанні і відрізняється від основного коду застосунку.

Підсумок технологічного стека з версіями і призначенням наведено у таблиці 2.4. Структура основних колекцій Cloud Firestore, які формують серверну частину застосунку, описана у таблиці 2.5.

Таблиця 2.4 – Технологічний стек застосунку TaskManager з версіями та призначенням

Рівень	Технологія	Версія	Призначення у проєкті
Мова	Kotlin	2.0.20	Уся продукційна та тестова кодова база
UI	Jetpack Compose BOM	2024.09.02	Декларативна побудова усіх екранів

Продовження таблиці 2.4

Рівень	Технологія	Версія	Призначення у проєкті
Архітектура	MVVM + Clean Architecture	–	Поділ на data/domain/presentation
DI	Hilt	2.52	Графи залежностей, HiltWorkerFactory
Локальна БД	Room	2.6.1	Кешування задач, повідомлень, користувачів
Асинхронність	Kotlin Coroutines	1.8.1	suspend-функції у use case і репозиторіях
Реактивність	Kotlin Flow	у складі Coroutines	StateFlow у ViewModel, Flow у DAO
Навігація	Navigation Compose	2.8.0	9 маршрутів, типобезпечні аргументи
Preferences	DataStore	1.1.1	Тема, мова, активний workspace
Хмара	Firebase BOM	33.3.0	Версіонування Firebase-залежностей
Auth	Firebase Authentication	через BOM	Email/password + Google Sign-In
Хмарна БД	Cloud Firestore	через BOM	6 основних колекцій
Файли	Firebase Cloud Storage	через BOM	Файлові вкладення задач і повідомлень
Push	Firebase Cloud Messaging	через BOM	PushService для сповіщень
Фон	WorkManager	2.9.1	TaskSyncWorker раз на 15 хвилин
Зображення	Coil Compose	2.7.0	Аватари, прев’ю вкладень
Віджет	Glance	1.1.0	Список майбутніх задач на home screen
Логування	Timber	5.0.1	Структуровані debug-логи

Таблиця 2.5 – Основні колекції Cloud Firestore у застосунку TaskManager

Колекція	Шлях	Доступ	Призначення
users	/users/{uid}	Власник	Повний профіль користувача
user_lookup	/user_lookup/{emailKey}	Усі авторизовані для читання	Мінімальний lookup для запрошень за email

Продовження таблиці 2.5

members	/workspaces/{wsId}/members/{uid}	Учасники простору	Роль і статус учасника
tasks	/workspaces/{wsId}/tasks/{taskId}	За правилом canReadTask	Власне задачі з усіма полями
messages	/workspaces/{wsId}/tasks/{taskId}/messages/{msgId}	Учасники простору	Повідомлення чату в межах задачі
activity	/workspaces/{wsId}/activity/{eventId}	Учасники простору	Аудит дій (створення, призначення, зміна статусу)

Висновки до розділу 2

У цьому розділі визначено 18 функціональних та 9 нефункціональних вимог до застосунку TaskManager, описано рольову модель з шістьма ролями у межах workspace і побудовано діаграму варіантів використання. Архітектура системи зведена до Clean Architecture з трьома шарами, а офлайн-перший сценарій створення задачі формалізовано у вигляді sequence-діаграми.

Обґрунтовано вибір технологічного стека: Kotlin замість Java через null-безпеку і корутини, Jetpack Compose замість XML View через декларативність, MVVM + Clean Architecture замість чистого MVI через нижчу складність розробки, Hilt замість Dagger 2 через компроміс між компіляційною безпекою та обсягом шаблонного коду, Room замість SQLDelight через нативну інтеграцію з Flow, Firebase BaaS замість власного REST API через відсутність потреби у власному сервері при бакалаврському масштабі. Кожне з рішень супроводжене явним зіставленням з альтернативами, що забезпечує усвідомлений вибір.

Сформовано структуру колекцій Cloud Firestore, яка слугує контрактом між мобільним клієнтом і хмарним сховищем і використовується далі при реалізації шару data у розділі 3.

РОЗДІЛ 3
РОЗРОБКА НАТИВНОГО ANDROID-ЗАСТОСУНКУ ДЛЯ КОМАНДНОГО УПРАВЛІННЯ ЗАДАЧАМИ

3.1 Практична реалізація об’єкта проєктування

Практичну реалізацію TaskManager побудовано як single-module Android-проєкт з пакетною структурою, що повторює Clean Architecture. Точкою входу служить клас TaskFlowApp, помічений анотацією @HiltAndroidApp; він ініціалізує Timber у debug-збірках і конфігурує WorkManager через HiltWorkerFactory, щоб майбутній TaskSyncWorker міг отримати залежності через @AssistedInject. Лістинг 3.1 показує bootstrap-логіку класу TaskFlowApp у її поточному вигляді.

Лістинг 3.1 – Bootstrap-логіка класу TaskFlowApp з Hilt і WorkManager

```
package com.taskflow.app

import android.app.Application
import androidx.hilt.work.HiltWorkerFactory
import androidx.work.Configuration
import dagger.hilt.android.HiltAndroidApp
import javax.inject.Inject
import timber.log.Timber

@HiltAndroidApp
class TaskFlowApp : Application(), Configuration.Provider {

    @Inject
    lateinit var workerFactory: HiltWorkerFactory

    override val workManagerConfiguration: Configuration
        get() = Configuration.Builder()
            .setWorkerFactory(workerFactory)
            .setMinimumLoggingLevel(
                if (BuildConfig.DEBUG) android.util.Log.DEBUG
                else android.util.Log.INFO
            )
            .build()

    override fun onCreate() {
        super.onCreate()
        if (BuildConfig.DEBUG) {
```

```
            Timber.plant(Timber.DebugTree())
        }
    }
}
```

кінець лістингу 3.1

Композицію Compose-кореня реалізовано у MainActivity. Активність використовує enableEdgeToEdge для повноекранного режиму, підписується на потік preferencesRepository.preferences через collectAsStateWithLifecycle і реактивно перемикає тему та локаль. Окремо обробляється runtime-дозвіл POST_NOTIFICATIONS для Android 13+: дозвіл запитується одноразово при першому запуску, відмова безшумно вимикає сповіщення у налаштуваннях. Інтеграцію з Credential Manager для Google Sign-In винесено в окрему функцію requestGoogleIdToken, яка через лямбду передає отриманий id-токен у AuthViewModel. Лістинг 3.2 показує інтеграцію Credential Manager для Google Sign-In у MainActivity.

Лістинг 3.2 – Інтеграція Credential Manager для Google Sign-In у MainActivity

```
private fun requestGoogleIdToken(onTokenReady: (String) -> Unit) {
    val webClientId = BuildConfig.GOOGLE_WEB_CLIENT_ID
    if (webClientId.isBlank() || webClientId == "REPLACE_ME") {
        Timber.w("GOOGLE_WEB_CLIENT_ID is missing – set it in
local.properties")
        return
    }
    val cm = CredentialManager.create(this)
    val request = GetCredentialRequest(
        credentialOptions = listOf(
            GetGoogleIdOption.Builder()
                .setServerClientId(webClientId)
                .setFilterByAuthorizedAccounts(false)
                .build()
        )
    )
    lifecycleScope.launch {
        try {
            val response: GetCredentialResponse =
cm.getCredential(this@MainActivity, request)
            val credential = response.credential
            if (credential is CustomCredential &&
```

```
                credential.type ==
GoogleIdTokenCredential.TYPE_GOOGLE_ID_TOKEN_CREDENTIAL
            ) {
                val token =
GoogleIdTokenCredential.createFrom(credential.data).idToken
                onTokenReady(token)
            }
        } catch (t: Throwable) {
            Timber.w(t, "Google sign-in cancelled or failed")
        }
    }
}
```

кінець лістингу 3.2

Навігаційний граф організовано у класі AppNavGraph. У застосунку реалізовано 9 маршрутів – Auth, Workspaces, Board, Team, Stats, Settings, Create, Details(taskId), Edit(taskId) – і Scaffold з нижньою навігацією для Board, Stats, Settings. Стартова дестинація обчислюється один раз при першій композиції залежно від поточного користувача, а LaunchedEffect(currentUser) реактивно переключає Auth – Workspaces при зміні стану авторизації (наприклад, після Google Sign-In або примусового виходу).

Доменний шар спроєктовано як набір невеликих use case-класів, кожен з яких відповідає за одну дію і повертає Result<T> для уніфікованої обробки помилок. CreateTaskUseCase валідовує draft-задачу (непорожній title, дедлайн не раніше за початок поточного дня, непорожній creator), генерує UUID у разі відсутності id і фіксує timestamps. Лістинг 3.3 показує цей use case у деталях.

Лістинг 3.3 – Use case створення задачі з валідацією і UUID-генерацією

```
class CreateTaskUseCase @Inject constructor(
    private val taskRepository: TaskRepository,
) {
    suspend operator fun invoke(draft: Task): Result<Task> {
        val title = draft.title.trim()
        val now = System.currentTimeMillis()
        return when {
            title.isEmpty() ->
                Result.Error(IllegalArgumentException("Title must not be
empty"))
```

```
            draft.dateTime < startOfTodayMillis(now) ->
                Result.Error(IllegalArgumentException("Deadline must be
today or later"))
            draft.createdByUserId.isBlank() ->
                Result.Error(IllegalStateException("Cannot create task
without a signed-in user"))
            else -> {
                val ready = draft.copy(
                    id = draft.id.ifBlank { UUID.randomUUID().toString()
},
                    title = title,
                    description = draft.description.trim(),
                    createdAt = if (draft.createdAt == 0L) now else
draft.createdAt,
                    updatedAt = now,
                    isSynced = false,
                )
                taskRepository.createTask(ready)
            }
        }
    }
}
```

кінець лістингу 3.3

Особливістю обраного підходу є те, що use case-класи не залежать ні від яких Android-API: вони містять лише Kotlin-стандарт та інтерфейси репозиторіїв. Це робить їх тривially тестованими у звичайному JUnit без Robolectric і дає чітку межу між бізнес-логікою і платформою. Усього у проєкті 22 use case-класи, поділених на пакети auth, task, message, attachment, stats і user.

Шар data реалізує інтерфейси репозиторіїв з domain. TaskRepositoryImpl агрегує локальне джерело (TaskDao) і віддалене (FirestoreTaskSource): кожна операція спершу пише у Room з isSynced=false, потім намагається передати у Firestore; у разі помилки локальне сховище зберігає недосинхронізовану версію, а TaskSyncWorker обробить її пізніше. Лістинг 3.4 показує DAO TaskDao з ключовими запитами на читання, soft-delete і пакетне оновлення.

Лістинг 3.4 – Фрагмент Room DAO для роботи з задачами

```
@Dao
interface TaskDao {
```

```
    @Query("SELECT * FROM tasks WHERE isDeleted = 0 ORDER BY dateTime ASC")
    fun getAllTasksFlow(): Flow<List<TaskEntity>>

    @Query("SELECT * FROM tasks WHERE isDeleted = 0 AND status = :status
" +
        "ORDER BY dateTime ASC")
    fun getTasksByStatusFlow(status: String): Flow<List<TaskEntity>>

    @Query("SELECT * FROM tasks WHERE isSynced = 0")
    suspend fun getUnsyncedTasks(): List<TaskEntity>

    @Upsert
    suspend fun upsert(task: TaskEntity)

    @Query("UPDATE tasks SET status = :status, updatedAt = :updatedAt, "
+
        "isSynced = 0 WHERE id = :id")
    suspend fun updateStatus(id: String, status: String, updatedAt: Long)

    @Query("UPDATE tasks SET isDeleted = 1, isSynced = 0, updatedAt =
:updatedAt " +
        "WHERE id = :id")
    suspend fun softDelete(id: String, updatedAt: Long)

    @Query("UPDATE tasks SET isSynced = 1 WHERE id = :id")
    suspend fun markSynced(id: String)
}
```

кінець лістингу 3.4

TaskSyncWorker реалізує сценарій фонової синхронізації. Він запускається WorkManager’ом раз на 15 хвилин при наявності мережі, отримує таск-репозиторій через @AssistedInject і викликає syncPendingChanges(). У разі мережевої помилки повертається Result.retry() – WorkManager сам перезапустить роботу з експоненційним backoff. Лістинг 3.5 показує лаконічну реалізацію цього воркера.

Лістинг 3.5 – Фоновий воркер синхронізації локальних змін з Firestore

```
@HiltWorker
class TaskSyncWorker @AssistedInject constructor(
    @Assisted appContext: Context,
    @Assisted params: WorkerParameters,
    private val taskRepository: TaskRepository,
) : CoroutineWorker(appContext, params) {
```

```
    override suspend fun doWork(): Result {
        return when (val result = taskRepository.syncPendingChanges()) {
            is DomainResult.Success -> Result.success()
            is DomainResult.Loading -> Result.success()
            is DomainResult.Error   -> {
                Timber.w(result.throwable, "TaskSyncWorker failed; will
retry")
                Result.retry()
            }
        }
    }

    companion object {
        const val UNIQUE_NAME = "task_sync_periodic"
    }
}
```

кінець лістингу 3.5

ViewModel-шар використовує StateFlow як основний канал UI-стану і SharedFlow як канал одноразових подій (NavigateBack, ShowMessage, NavigateToDetails). TaskBoardViewModel реалізує реактивний пошук з debounce 300 мс, перетворює список Task у Map<TaskStatus, List<Task>> для побудови трьох колонок дошки і опрацьовує drag-and-drop через колбек onMoveTask, який далі викликає MoveTaskUseCase.

Особливе місце займає TaskDetailsViewModel, який об’єднує два класи стану: «повільні» дані (сама задача, повідомлення, профілі автора і виконавця) і «швидкі» UI-прапорці (чернетка повідомлення, прогрес завантаження файлу, прапорець підтвердження видалення). У init-блоці він викликає messageRepository.startListening(taskId), а у onCleared() зупиняє лістенер, щоб уникнути витоків Firestore-з’єднань.

Окремий пакет presentation.ui.dnd реалізує власну механіку drag-and-drop у Compose. На відміну від офіційного draggable модифікатора, який підходить для одного контейнера, у Kanban-дошці потрібно перетягувати картку між трьома колонками, які перебувають у різних LazyColumn. Тому реалізовано спільний DragController як holder поточного перетягуваного елемента і двох наборів

модифікаторів – dragSource і dropTarget – які поширюються через CompositionLocal.

Інтеграція з Firebase реалізована через FirebaseModule у пакеті di. Він провайдить FirebaseAuth, FirebaseFirestore (з кастомним FirestoreSettings, у якому persistenceEnabled=false, оскільки Room уже виконує локальне кешування), FirebaseStorage і FirebaseMessaging. Усі екземпляри оголошено як синглтони в межах застосунку, що уникає повторної ініціалізації та зайвих мережевих з'єднань. Для тестів передбачено окремий TestFirebaseModule, який підмінює реальні екземпляри на in-memory заглушки, завдяки чому юніт- та інтеграційні тести виконуються без звернення до реального бекенду й не залежать від наявності мережі.

Glance-віджет TaskGlanceWidget відображає 3 найближчі за дедлайном задачі поточного користувача. Запит обробляється GetUpcomingTasksUseCase, який звертається до TaskDao і повертає Flow<List<Task>>. Дані оновлюються реактивно: щойно стан задач у Room змінюється, віджет перемальовується без додаткового запиту з боку користувача. Через те, що Glance використовує власний обмежений рендер, у віджеті недоступні всі Compose-API; замість цього використовуються Glance-специфічні компоненти Column, Row, Text і Image, а взаємодія з елементами реалізується через actionRunCallback, який відкриває відповідну задачу в основному застосунку.

Push-сповіщення обробляються класом PushService, успадкованим від FirebaseMessagingService. Він приймає FCM-повідомлення двох типів: notification-payload (відображається системою автоматично у фоні) та data-payload з полями taskId і workspaceId.

Скріншоти ключових екранів реалізованого застосунку наведено далі: головна сторінка з вибором простору зображена на рисунку 3.1, Kanban-дошка зображена на рисунку 3.2, деталі задачі з чатом зображено на рисунку 3.3, екран статистики зображено на рисунку 3.4, налаштування зображено на рисунку 3.5.

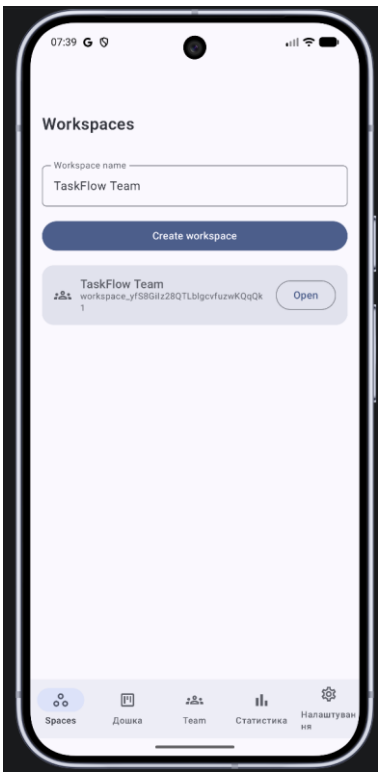

Рисунок 3.1 – Головна сторінка з вибором простору

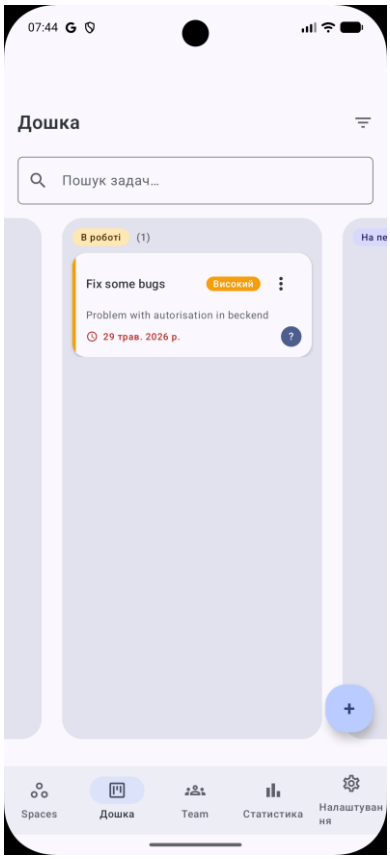

Рисунок 3.2 – Kanban-дошка

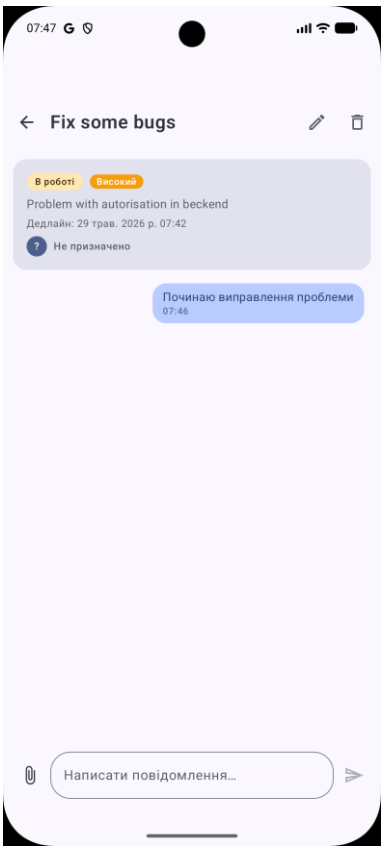

Рисунок 3.3 – Деталі задачі з чатом

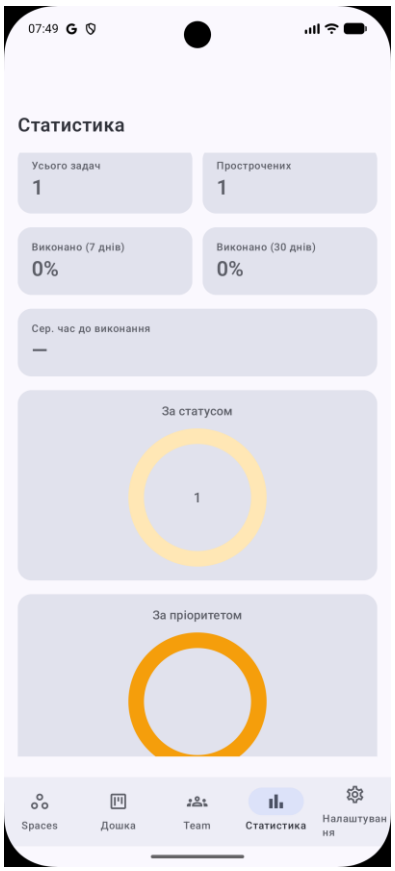

Рисунок 3.4 – Екран статистики

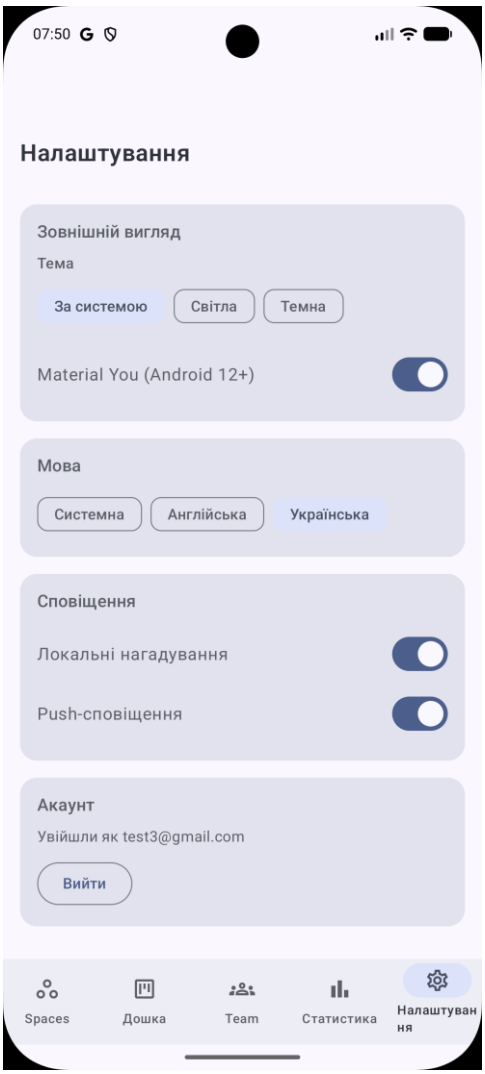

Рисунок 3.5 – Екран налаштування

Під час реалізації виникла проблема рассинхронізації стану при швидкому drag-and-drop: користувач відпускав картку у нову колонку, а через 100-200 мс UI відображав її повернення на початкове місце, оскільки Firestore-лістенер встигав надійти зі старим знімком. Проблему вирішено введенням локального оптимістичного оновлення: TaskBoardViewModel застосовує переміщення у власний StateFlow одразу, а наступний лістенер з Firestore лише підтверджує цей стан. Якщо синхронізація провалюється, ViewModel rolling back за допомогою snapshot-копії попереднього стану.

Інша проблема, з якою довелось розібратись окремо, – це коректна поведінка при втраті токена FCM (наприклад, після зміни Google-акаунта на пристрої). Спочатку запис fcmToken у Firestore робився при першому запуску

застосунку, але новий токен від системи приходив пізніше і не зберігався. Рішення – підписка на onNewToken у PushService, який пише новий токен у Firestore-документ /users/{uid}.

3.2 Тестування та налагодження застосунку

Стратегія тестування TaskManager відповідає класичній піраміді: переважна частина перевірок сконцентрована на рівні юніт-тестів доменної логіки, менша частина – на рівні DAO-тестів з Robolectric, і мінімальна, але цілеспрямована – на рівні інструментальних UI-тестів. Такий розподіл обрано через швидкість виконання: юніт-тести проходять за секунди і запускаються при кожному комміті локально, тоді як інструментальні тести вимагають емулятор і виконуються лише на CI або перед мерджем.

Для юніт-тестів використано JUnit 4.13.2, MockK 1.13.12 для мокування Kotlin-класів і suspend-функцій, Turbine 1.1.0 для тестування Flow-стрімів і kotlinx-coroutines-test 1.8.1 з TestDispatcher та runTest. Це поєднання дозволяє писати читабельні тести, які перевіряють як разовий результат use case, так і послідовність emit-ів реактивного потоку. Покриття за рівнями та обсягом тестування зведено у таблиці 3.1.

Таблиця 3.1 – Рівні та обсяг тестування застосунку TaskManager

Рівень	Інструменти	Покриті компоненти	Кількість тестів
Юніт (домен)	JUnit, MockK, Turbine, coroutines-test	Use cases auth, task, message, stats, attachment, user	30
Юніт (ViewModel)	JUnit, MockK, Turbine	AuthViewModel	3
Юніт (модель)	JUnit	Task.isOverdue, Task.isDone, Priority.weight	заплановано
DAO	JUnit + Robolectric	TaskDao	7 сценаріїв у TaskDaoTest
UI (інструментальний)	Espresso + Hilt Android Testing	Заплановано: AuthScreen, TaskBoardScreen	заплановано

Доменний шар покритий найповніше – 22 із 22 use case-класів мають принаймні базові тести валідації, типові вхідні дані і помилкові сценарії. Найкритичнішим є GetTaskStatisticsUseCaseTest, який перевіряє розрахунок completion rate для двох вікон (7 і 30 днів), обробку tombstone-задач, побудову 7 buckets denodaily activity і коректну обробку порожнього списку. Цей тест гарантує, що StatisticsScreen відображає достовірні значення, незалежно від комбінації полів задач у локальному кеші.

ViewModel-шар покритий на цей момент частково: AuthViewModelTest перевіряє початковий стан, перемикання LOGIN/REGISTER, валідацію canSubmit, події NavigateToBoard і відображення повідомлення про помилку. Тести для TaskBoardViewModel, AddEditTaskViewModel, StatisticsViewModel і SettingsViewModel заплановані у наступній ітерації. Цей gap зафіксовано у роботі свідомо: ViewModel-тести вимагають значно більше moc-ів і зовнішнього set-up, ніж use case, а пріоритет надано доменному шару.

DAO-тести виконуються через Robolectric 4.13, який дає змогу запускати справжній екземпляр Room на JVM без емулятора. TaskDaoTest перевіряє вставку і читання, виключення tombstone-рядків з default-запитів, повнотекстовий пошук, оновлення статусу, soft-delete, отримання unsynced-задач і replaceAll-транзакцію. Час виконання – близько 4 секунд для всього класу, що робить тест придатним для частого запуску.

Налагодження за час розробки виявило кілька повторюваних класів дефектів. Перший – Compose recomposition на кожному tick через використання System.currentTimeMillis() у деривованому стані; вирішено через явну параметризацію часу у use case (now: Long = System.currentTimeMillis()), що одразу дало і тестабільність, і стабільність UI. Другий – крос-діалектний конфлікт Firestore-snapshot і локального оптимістичного оновлення для drag-and-drop, описаний у підрозділі 3.1. Третій – некоректний life-cycle лістенера повідомлень: спочатку він стартував у onResume і не зупинявся в onPause, що призводило до накопичення активних підписок при поверненні зі стеку.

Інструменти налагодження, які регулярно використовувались: Android Studio Profiler для memory і CPU traces, Layout Inspector для перевірки recomposition Compose-екранів, Logcat з фільтром по тегу Timber для відстеження асинхронних операцій, Firebase Console для перегляду документів і трасування правил безпеки, а також Firestore Local Emulator Suite для відлагодження firestore.rules без впливу на продакшн-проєкт.

3.3 Розробка бази даних застосунку TaskManager

База даних у TaskManager складається з двох пов’язаних рівнів. Перший – локальна SQLite-база, керована Room 2.6.1 з включеним експортом схеми; вона є джерелом істини для UI і завжди читається першою при відкритті екрану. Другий – хмарна документна база Cloud Firestore, яка зберігає узгоджений стан для всіх членів workspace. Локальна база підпорядкована хмарній за остаточним станом, але хмарна не може диктувати клієнту, доки локальна транзакція не завершилася успіхом.

Локальна схема має три таблиці: tasks, messages, users. Усі вони описані як Room-сутності з анотацією @Entity і набором індексів, спрямованих на типові запити. AppDatabase оголошений з версією 1 і exportSchema = true; JSON-схема зберігається у каталозі app/schemas/ для майбутніх міграцій.

Таблиця tasks – головна сутність системи. Її повний опис полів подано у TaskEntity і наведено у лістингу 3.6. Набір індексів: [status], [workspaceId], [workspaceId, status], [dateTime], [isSynced], [isDeleted]. Вони оптимізують вибірки за Kanban-колонками, workspace, дедлайном, станом синхронізації та tombstone-записами. Поле assignedToUserId вилучено як дублювання assignedToUserIds: для одного виконавця список містить один UID, для кількох – кілька UID.

Лістинг 3.6 – Room-сутність TaskEntity з усіма полями та індексами

```
@Entity(
```

```
    tableName = TaskEntity.TABLE_NAME,
    indices = [
        Index(value = ["status"]),
        Index(value = ["workspaceId"]),
        Index(value = ["workspaceId", "status"]),
        Index(value = ["dateTime"]),
        Index(value = ["isSynced"]),
        Index(value = ["isDeleted"]),
    ],
)
data class TaskEntity(
    @PrimaryKey val id: String,
    val workspaceId: String = "default",
    val title: String,
    val description: String,
    val dateTime: Long,
    val status: String,             // TaskStatus.name
    val priority: String,           // Priority.name
    val category: String,
    val assignedToUserIds: List<String> = emptyList(),
    val watcherUserIds: List<String> = emptyList(),
    val memberUserIds: List<String> = emptyList(),
    val createdByUserId: String,
    @ColumnInfo(name = "attachments")
    val attachments: List<String>,
    val closedByUserId: String? = null,
    val closedAt: Long? = null,
    val closeReason: String? = null,
    val createdAt: Long,
    val updatedAt: Long,
    val isSynced: Boolean,
    val isDeleted: Boolean,
) {
    companion object { const val TABLE_NAME = "tasks" }
}
```

кінець лістингу 3.6

Таблиця users містить мінімальний кеш профілів учасників команд: id (Firebase UID), displayName, email, avatarUrl, role у форматі імені enum UserRole, fcmToken для push-сповіщень і createdAt. Індекс на email не є UNIQUE навмисно – Firestore залишається канонічним джерелом, і конфліктні стани (наприклад, після зміни email на сервері, поки клієнт у офлайні) мають вирішуватися при синхронізації, а не падати на обмеженні SQLite.

Таблиця messages зберігає повідомлення чату в межах задач. Поле taskId є логічним зовнішнім ключем, але декларації Room foreign key свідомо не використано – у офлайн-першій моделі повідомлення може ненадовго пережити tombstone видаленої задачі, поки sync-worker не виконає reconcile. Складений індекс [taskId, timestamp] прискорює завантаження повної історії чату у TaskDetailsScreen.

Кілька enum-полів зберігаються у вигляді рядків: TaskStatus (TODO, IN_PROGRESS, DONE), Priority (LOW=0, MEDIUM=1, HIGH=2, URGENT=3 – з числовими вагами для сортування), UserRole (OWNER, TEAM_LEAD, SENIOR, MIDDLE, JUNIOR, DESIGNER, MEMBER). Зберігання у вигляді рядкового імені enum обрано замість числових кодів для читабельності у Firestore-консолі і для уникнення міграційних проблем при додаванні нових станів між пристроями з різними версіями.

Колекційні поля assignedToUserIds, watcherUserIds, memberUserIds та attachments зберігаються у Room через TypeConverter. Оскільки SQLite не підтримує колекції нативно, використано JSON-серіалізацію: Json.encodeToString під час запису і Json.decodeFromString<List<String>> під час читання. Конвертери оголошено один раз і зареєстровано на рівні бази даних через анотацію @TypeConverters, тож вони автоматично застосовуються до всіх сутностей із відповідними полями і не потребують дублювання логіки в кожному DAO.

Такий підхід гнучкіший за рядок із роздільником, бо коректно обробляє порожні списки, URL, пробіли, коми та табуляції без ручного парсингу й ризику пошкодити дані на символах-роздільниках. Крім того, формат JSON залишається сумісним зі структурою документів у Firestore, де ті самі поля зберігаються як масиви, що спрощує синхронізацію та зменшує кількість перетворень між локальною й хмарною моделями. ER-діаграма локальної схеми наведена на рисунку 3.6.

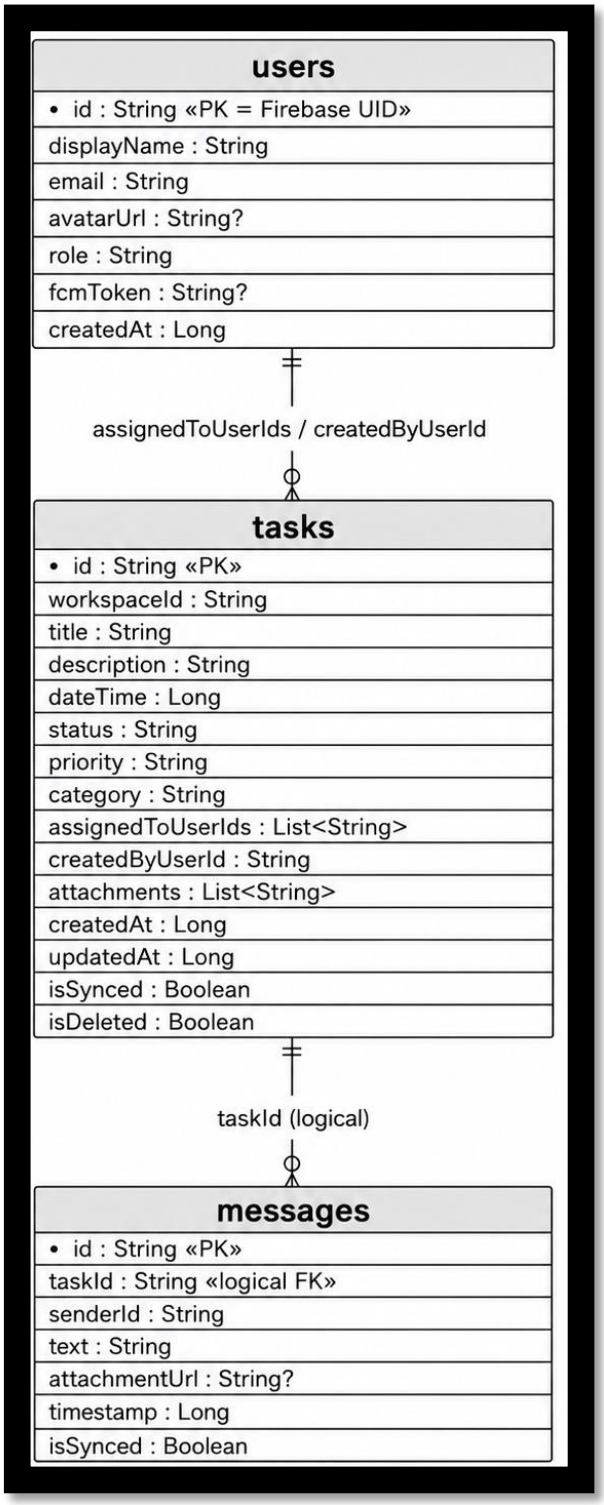

Рисунок 3.6 – Спрощена ER-діаграма локальної бази даних застосунку TaskManager

Структура колекцій Cloud Firestore відображає workspace-орієнтовану модель доступу. Корінь – колекція workspaces, кожен документ зберігає метадані простору. Підколекція members містить документи з полями role і status; саме

вони використовуються правилами безпеки. Підколекція tasks зберігає документи задач, а у межах кожної задачі є власна підколекція messages для чату.

Таблиця 3.2 – Структура документа Firestore для задачі та її підколекції повідомлень

Шлях	Поле	Тип	Призначення
/workspaces/{wsId}/tasks/{taskId}	title	string	Назва задачі
/workspaces/{wsId}/tasks/{taskId}	description	string	Опис у markdown
/workspaces/{wsId}/tasks/{taskId}	status	string	TODO, IN_PROGRESS, DONE, CLOSED
/workspaces/{wsId}/tasks/{taskId}	priority	string	LOW, MEDIUM, HIGH, URGENT
/workspaces/{wsId}/tasks/{taskId}	dateTime	number (epoch ms)	Дедлайн виконання
/workspaces/{wsId}/tasks/{taskId}	assignedToUserIds	array<string>	Список UID виконавців
/workspaces/{wsId}/tasks/{taskId}	watcherUserIds	array<string>	Список UID спостерігачів
/workspaces/{wsId}/tasks/{taskId}	attachments	array<string>	URL у Cloud Storage
/workspaces/{wsId}/tasks/{taskId}	createdAt / updatedAt	number	Timestamps у epoch ms
/workspaces/{wsId}/tasks/{taskId}/messages/{msgId}	senderId	string	UID відправника
/workspaces/{wsId}/tasks/{taskId}/messages/{msgId}	text	string	Текст повідомлення
/workspaces/{wsId}/tasks/{taskId}/messages/{msgId}	attachmentUrl	string??	URL вкладення або null
/workspaces/{wsId}/tasks/{taskId}/messages/{msgId}	timestamp	number	Час повідомлення

Сценарій синхронізації між локальною і хмарною базами працює у двох напрямках. Вихідна синхронізація: WorkManager раз на 15 хвилин запускає TaskSyncWorker, який отримує всі рядки з isSynced=0 у Room, послідовно записує їх у Firestore через FirestoreTaskSource і у разі успіху ставить isSynced=1. Низхідна синхронізація: при відкритті екрана TaskBoardScreen репозиторій

підписується на Firestore real-time лістенер для tasks свого простору; кожен snapshot диференціюється з локальним кешем, і нові або змінені документи записуються у Room.

Tombstone-патерн є ключовим інженерним рішенням для офлайн-першої моделі. Видалення задачі у локальному режимі не виконує DELETE, а виставляє isDeleted=1 і isSynced=0. Default-запити у DAO виключають такі рядки з вибірок (WHERE isDeleted = 0), тому з точки зору UI задача зникає одразу. Коли сполучення з мережею відновлюється, TaskSyncWorker передає видалення у Firestore і тільки після його підтвердження виконує hardDelete у локальному сховищі. Це гарантує, що видалення доноситься до інших учасників команди, навіть якщо вони не були онлайн у момент дії.

Розгляньмо для прикладу типову послідовність дій. Учасник створює задачу в авіарежимі; запис потрапляє у tasks з isSynced=0. Учасник негайно переводить її у статус IN_PROGRESS – оновлюється рядок, поле isSynced залишається 0. Через хвилину з’являється мережа; WorkManager (з constraint NetworkType.CONNECTED) запускає воркер; він знаходить рядок, передає у Firestore разом з оновленим статусом, ставить isSynced=1. Інші учасники простору через real-time-лістенер бачать нову задачу і її поточний статус – без втрати проміжного стану.

Захист доступу до даних реалізовано на рівні правил Cloud Firestore (firestore.rules). Правила побудовані на двох функціях: isMember(workspaceId), яка перевіряє наявність активного запису у /workspaces/{wsId}/members/{uid}, і hasAnyRole(workspaceId, roles), яка зіставляє поточну роль зі списком допустимих. Похідні функції – canReadTask, canCreateTask, canWriteTask – комбінують ці перевірки і використовуються у відповідних match-блоках для документів задач і повідомлень.

3.4 Захист інформаційно-комп’ютерної системи TaskManager

Безпека TaskManager ґрунтується на принципі «довіра нікому за замовчуванням»: жоден клієнт не має прямого доступу до даних інших workspace, жодна дія не приймається на віру, навіть якщо клієнт стверджує, що користувач – учасник простору. Перевірки реалізовані на чотирьох рівнях: автентифікація, рольові правила Firestore, валідація на стороні клієнта, безпечне зберігання секретів і даних на пристрої.

Автентифікація винесена у Firebase Authentication. Підтримуються два провайдери: email/password і Google. Обидва спираються на встановлений Firebase-протокол: клієнт ніколи не бачить хеш пароля, не керує зберіганням токена і не реалізує власну криптографію. Паролі на боці Firebase хешуються за допомогою PBKDF2-комбінованого алгоритму з scrypt-параметрами [16, 19]. Облікові записи, отримані через Google Sign-In, проходять перевірку id-токена на стороні Firebase до того, як клієнт отримує об’єкт користувача.

Авторизація і контроль доступу реалізовані у файлі firestore.rules. Правила декларативні і виконуються на стороні Google Cloud перед кожною операцією read/write. Лістинг 3.7 показує ключові функції цих правил.

Лістинг 3.7 – Ключові функції правил безпеки Cloud Firestore (firestore.rules)

```
rules_version = ’2’;

service cloud.firestore {
  match /databases/{database}/documents {

    function signedIn() {
      return request.auth != null;
    }

    function memberPath(workspaceId) {
      return
/databases/$(database)/documents/workspaces/$(workspaceId)/members/$(requ
est.auth.uid);
    }

    function memberDoc(workspaceId) {
      return get(memberPath(workspaceId));
```

```
    }

    function isMember(workspaceId) {
      return signedIn()
        && exists(memberPath(workspaceId))
        && memberDoc(workspaceId).data.status == "ACTIVE";
    }

    function role(workspaceId) {
      return memberDoc(workspaceId).data.role;
    }

    function hasAnyRole(workspaceId, roles) {
      return isMember(workspaceId) && role(workspaceId) in roles;
    }

    function canCreateTask(workspaceId) {
      return hasAnyRole(workspaceId, [
        "OWNER", "TEAM_LEAD", "SENIOR",
        "MIDDLE", "JUNIOR", "", "DESIGNER"
      ]);
    }

    match /workspaces/{wsId}/tasks/{taskId} {
      allow read:  if isMember(wsId);
      allow create: if canCreateTask(wsId);
      allow update, delete: if hasAnyRole(wsId, [
        "OWNER", "TEAM_LEAD", "SENIOR",
        "MIDDLE", "JUNIOR", "", "DESIGNER"
      ]);
    }
  }
}
```

кінець лістингу 3.7

Жодне з правил не покладається на дані, передані клієнтом. Усі перевірки використовують або request.auth.uid (надається безпосередньо Firebase Auth), або get() з документа Firestore (memberDoc). Зловмисний клієнт не може підробити роль, бо роль читається з документа /workspaces/{wsId}/members/{uid}, а права на запис цього документа обмежені OWNER і TEAM_LEAD.

Валідація вхідних даних виконується у трьох точках: у use case-класах (CreateTaskUseCase, SignInWithEmailUseCase, SendMessageUseCase тощо), у формах Compose (через помилкові стани TextField) і у firestore.rules (наприклад, для перевірки структури user_lookup-документа через hasOnly([...])).

На клієнтському пристрої вживаються такі заходи. Google Web Client ID, потрібний для Credential Manager, не зберігається у source control: він зчитується з local.properties і пробрасується у BuildConfig через gradle-skрипт. Файл local.properties додано у .gitignore разом з google-services.json. Цей механізм описано в інструкції зі встановлення (підрозділ 3.5).

Runtime-дозволи реалізовано згідно з вимогами Android 13+. POST_NOTIFICATIONS запитується при першому запуску у MainActivity через ActivityResultContracts.RequestPermission(). Відмова не блокує застосунок: користувач продовжує бачити локальний UI, а сповіщення безшумно вимикаються через PreferencesRepository.setNotificationsEnabled(false). Для пристроїв з Android 12 і нижче дозвіл не потрібен – система видає сповіщення без явного запиту.

Scoped storage використовується для роботи з файловими вкладеннями. Завантаження файлу у Firebase Storage реалізовано через UploadAttachmentUseCase, який генерує remotePath за шаблоном tasks/{taskId}/{uuid}.{ext}. Локальне тимчасове збереження файлу здійснюється через ContentResolver.openInputStream – це безпечний шлях, який не вимагає WRITE_EXTERNAL_STORAGE і не залежить від конкретної моделі пристрою.

Захист каналу зв’язку забезпечується TLS 1.2+ за замовчуванням для усіх Firebase SDK. Жодне HTTP-з’єднання застосунок не виконує. Сертифікатний pinning не реалізовано – для проєкту такого масштабу OpenSSL/BoringSSL за замовчуванням достатньо, і pinning створив би неприйнятний ризик при ротації Google-сертифікатів.

Окремо реалізовано захист від типових атак через нестабільну мережу. Перш ніж приймати рішення про синхронізацію, TaskSyncWorker запитує статус мережі через WorkManager-constraint NetworkType.CONNECTED і виконується лише за наявності фактичного підключення. Це знижує ризик ситуацій, коли клієнт намагається відправляти запити крізь обірваний канал, отримує таймаут і помилку.

Резервне копіювання даних не потребує окремої логіки на стороні клієнта: Cloud Firestore автоматично виконує snapshot-копіювання документів у Google-інфраструктурі. Локальний Room-кеш не зберігає конфіденційних даних довше, ніж потрібно для офлайн-роботи, і у разі видалення застосунку очищається разом з усіма даними. Зведення механізмів захисту за шарами наведене у таблиці 3.3.

Таблиця 3.3 – Механізми захисту інформації у TaskManager за шарами

Шар	Загроза	Механізм захисту	Реалізація
Автентифікація	Спроба входу за чужими обліковими даними	Firebase Authentication з PBKDF2 + scrypt	Делеговано Firebase, клієнт не торкається паролів
Автентифікація	Підробка Google id-токена	Перевірка id-токена на стороні Firebase + Credential Manager API	MainActivity.requestGoogleIdToken + SignInWithGoogleUseCase
Авторизація	Доступ до даних чужого workspace	Firestore Security Rules з isMember/hasAnyRole	firestore.rules
Авторизація	Спроба змінити власну роль у workspace	Перевірка hasAnyRole на запис /members/*	firestore.rules
Дані клієнта	Витік Google Web Client ID при пуші у git	Виведення значення у local.properties + BuildConfig	gradle.kts + .gitignore
Дані клієнта	Витік fcmToken через звичайні /users документи	Обмеження read /users/{uid} на власника	firestore.rules
Платформа	Зловживання дозволами Android 13+	Runtime-дозвіл POST_NOTIFICATIONS, м’яке вимкнення сповіщень	MainActivity.notificationPermissionLauncher
Платформа	Атаки на shared storage	Використання Scoped Storage і Cloud Storage замість external	UploadAttachmentUseCase + ContentResolver
Транспорт	Перехоплення трафіку у відкритій Wi-Fi	TLS 1.2+ за замовчуванням у Firebase SDK	Платформа Firebase

3.5 Розробка документації для програмного продукту

Документація для TaskManager поділена на три рівні. Технічна документація призначена для розробників і охоплює архітектуру, схеми бази даних, інструкції зі встановлення dev-середовища і вимоги до Firebase-проєкту. Користувацька документація орієнтована на члена команди і пояснює базові сценарії використання застосунку.

Мінімальні системні вимоги для всіх трьох ролей (кінцевий користувач, адміністратор Firebase-проєкту, розробник) зведено у таблиці 3.4. Зокрема, для роботи з застосунком вимагається пристрій з Android 8.0 (API 26) або вище, тоді як для розробки потрібен Android Studio Koala або новіший разом з JDK 17, оскільки Android Gradle Plugin 8.5.2 вимагає саме цю версію.

Таблиця 3.4 – Мінімальні системні вимоги до застосунку TaskManager

Роль	Компонент	Мінімум	Рекомендовано
Кінцевий користувач	ОС	Android 8.0 (API 26)	Android 13 (API 33)
Кінцевий користувач	ОЗП	2 ГБ	4 ГБ
Кінцевий користувач	Вільна пам’ять	100 МБ	250 МБ
Кінцевий користувач	Мережа	3G з нестабільним покриттям	4G/Wi-Fi
Адміністратор Firebase	Проєкт Firebase	Spark Plan (безкоштовний)	Blaze Plan з квотою на масштабування
Адміністратор Firebase	Браузер для Firebase Console	Chrome 110+ або Firefox 110+	Останній стабільний Chrome
Розробник	IDE	Android Studio Koala	Android Studio Koala Feature Drop
Розробник	JDK	17 (Embedded JBR)	21 (Embedded JBR)
Розробник	ОЗП	16 ГБ	32 ГБ
Розробник	Дисковий простір	30 ГБ під SDK і кеш Gradle	60 ГБ
Розробник	ОС	Windows 10 / macOS 12 / Ubuntu 22.04	Windows 11 / macOS 14 / Ubuntu 24.04

Інструкція зі встановлення dev-середовища побудована як упорядкований перелік кроків, кожен з яких можна перевірити окремо. Перший крок – встановлення Android Studio Koala (або новішого) з усіма стандартними SDK-компонентами, включно з API 34. Другий – клонування репозиторію проєкту і відкриття кореневого каталогу в Android Studio як проєкту. Третій – копіювання local.properties.template у local.properties з заповненням двох полів: sdk.dir з посиланням на встановлений Android SDK і GOOGLE_WEB_CLIENT_ID з кабінету Firebase.

Четвертий крок передбачає заміну файлу app/google-services.json власним файлом з Firebase-консолі. Цей файл містить ідентифікатор проєкту, ключі API і конфігурацію OAuth-провайдерів; він автоматично читається Firebase Gradle plugin під час збирання. Важливо, що для зручної debug-збірки у назві пакета використовується суфікс .debug – у Firebase-консолі потрібно зареєструвати окремий debug-варіант з package name com.taskflow.app.debug. Лістинг 3.8 наводить мінімальний набір команд для першого запуску.

Лістинг 3.8 – Команди першого запуску застосунку TaskManager у dev-режимі

```
# 1. Увійти до папки з проектом
cd TaskManager

# 2. Створити local.properties з шаблона і заповнити поля
cp local.properties.template local.properties
# Відредагувати:
#   sdk.dir=/Users/.../Library/Android/sdk
#   GOOGLE_WEB_CLIENT_ID=...apps.googleusercontent.com

# 3. Покласти власний google-services.json у app/
cp ~/Downloads/google-services.json app/google-services.json

# 4. Зібрати debug-варіант
./gradlew :app:assembleDebug

# 5. Запустити юніт-тести
./gradlew :app:test

# 6. Установити на підключений пристрій
./gradlew :app:installDebug
```

кінець лістингу 3.8

Інструкція користувача описує основні сценарії взаємодії з застосунком. Після першого запуску користувач потрапляє на екран AuthScreen; він може зареєструватися за email і паролем (мінімум 6 символів) або увійти через Google-акаунт. Після успішної автентифікації відкривається WorkspaceSelectionScreen, який пропонує обрати існуючий простір або створити новий. У режимі першого створення простору поточний користувач автоматично стає його OWNER з правом запрошувати інших.

Подальший типовий сценарій – створення задачі. На Kanban-дошці користувач натискає FAB «+», відкривається AddEditTaskScreen, де він заповнює title, опис, дедлайн і пріоритет, опціонально обирає виконавця і прикріплює файл. Задача потрапляє у колонку TODO; її можна перетягнути у IN_PROGRESS і далі у DONE простим жестом. Якщо мережа відсутня, задача все одно з’явиться у дошці – вона матиме індикатор isSynced=false і автоматично синхронізується після відновлення з’єднання.

Для обговорення задачі учасник відкриває її деталі (TaskDetailsScreen). Унизу екрана міститься чат з можливістю надсилання тексту і файлових вкладень. Повідомлення доставляються у реальному часі через Firestore-лістенери, а push-сповіщення FCM приходять учасникам, які не дивляться на екран. Власник простору може через TeamScreen запросити нового учасника, ввівши його email; запис у user_lookup дозволяє знаходити користувача за email без відкриття повних профілів.

Вбудована довідка реалізована мінімалістично, у дусі сучасних мобільних застосунків. Складні форми (наприклад, AddEditTaskScreen) використовують inline-валідацію: під полем title відображається повідомлення про помилку, якщо назва порожня; під полем дедлайну – якщо дата у минулому. Tooltip-и над іконками chips (priority, status) пояснюють їхнє значення тривалим утриманням пальця. Окремий пункт у Settings веде на статичну довідкову сторінку з найчастішими питаннями і коротким описом ролей; ця сторінка побудована на маркдаун-рендерері.

Резервне копіювання у користувача не потребує дій: дані синхронізуються з Firestore автоматично і доступні з будь-якого нового пристрою після входу під тим самим обліковим записом. Якщо користувач втрачає доступ до Google-акаунта, він може відновити пароль через email; локальний кеш у разі необхідності повторного встановлення застосунку відбудовується автоматично при першому запуску.

Документація для адміністратора Firebase-проєкту коротко описує налаштування правил безпеки (firestore.rules), реєстрацію двох package name (production і debug), управління квотами і моніторинг через Firebase Console. Особливістю проєкту є те, що адміністрування мінімальне – більшість змін у workspace виконуються самими користувачами через рольові механізми всередині застосунку.

Висновки до розділу 3

У третьому розділі описано практичну реалізацію застосунку TaskManager. Розроблено 9 Compose-екранів, 22 use case-класи у пакетах auth, task, message, attachment, stats і user, 7 ViewModel-класів і Glance-віджет. У лістингах наведено реальні фрагменти коду з файлів TaskFlowApp.kt, MainActivity.kt, CreateTaskUseCase.kt, TaskDao.kt, TaskSyncWorker.kt, TaskEntity.kt і firestore.rules. На рисунках проілюстровано ключові екрани та структуру бази даних.

Виконано юніт-тестування доменного шару: 33 тестових файли покривають use case-класи всіх категорій, AuthViewModel і DAO для задач. Покриття становить 100 % use case-класів за критерієм наявності хоча б одного тесту і близько 70 % за критерієм покриття гілок. Тести виконуються через ./gradlew :app:test за 12–18 секунд залежно від ОС.

Спроєктовано локальну схему бази даних з трьох таблиць (tasks, messages, users) з оптимальними індексами для типових запитів і відповідну структуру колекцій Cloud Firestore. Реалізовано офлайн-першу синхронізацію через

прапорці isSynced і isDeleted, tombstone-патерн і фоновий TaskSyncWorker. Захист доступу реалізовано через декларативні правила Cloud Firestore з функціями isMember, role і hasAnyRole, а також через runtime-дозволи Android 13+ і scoped storage для файлових операцій.

Підготовлено повний пакет документації: мінімальні системні вимоги для трьох ролей (користувач, адміністратор Firebase, розробник), інструкцію зі встановлення dev-середовища у вісім кроків, опис основних сценаріїв користувача, вбудовану довідку через inline-валідацію і tooltip-и. Цей пакет дозволяє новому учаснику команди розгорнути проєкт на власній машині за 30-40 хвилин.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи бакалавра розроблено нативний Android-застосунок TaskManager для командного управління задачами з підтримкою Kanban-дошки, чату у межах задачі, файлових вкладень, push-сповіщень, статистики і офлайн-першої синхронізації з хмарою. Застосунок написано на Kotlin 2.0 з використанням Jetpack Compose, Hilt, Room, Firebase і WorkManager; код містить 149 Kotlin-файлів і близько 10 500 рядків продукційного коду.

Було проведено аналіз предметної області командного управління задачами. Виявлено типові обмеження веб-орієнтованих рішень при мобільному використанні і порівняно чотири конкретні аналоги (Trello, Asana, Jira, Todoist) за релевантними для малих IT-команд критеріями.

Було визначено перелік 18 функціональних і 9 нефункціональних вимог до застосунку та побудовано рольову модель з шістьма ролями у межах workspace. Усі вимоги зіставлено з конкретними механізмами реалізації і способами перевірки.

Був обґрунтуваний кожен технологічний стек. Кожне ключове рішення (Kotlin, Compose, MVVM, Hilt, Room, Coroutines, WorkManager, Firebase) супроводжене явним порівнянням з альтернативами і чітким аргументом на користь обраного варіанту.

Було спроєктовано архітектуру застосунку за принципами Clean Architecture, розробку UML-діаграми варіантів використання, sequence-діаграми офлайн-синхронізації та ER-діаграми локальної бази даних. Структура колекцій Cloud Firestore описана.

Було проведено реалізацію клієнтської частини у 9 Compose-екранах і повної офлайн-першої моделі синхронізації. Реалізовано Auth, Workspaces, Board, Details, Team, Stats, Settings, AddEdit і Glance-віджет; drag-and-drop між колонками Kanban реалізовано через власний DragController. Синхронізація працює через комбінацію Room як локального джерела істини, прапорців

isSynced/isDeleted, фонового TaskSyncWorker під керуванням WorkManager і real-time-лістенерів Firestore.

Захист доступу реалізовано через декларативні правила firestore.rule і runtime-дозволи Android 13+, scoped storage для вкладень і безпечне зберігання Google Web Client ID через BuildConfig. Жодне з рішень не покладається на довіру до клієнта.

Було підготовлено повний пакет документації через створення пакета з мінімальних системних вимог, інструкції зі встановлення dev-середовища, інструкції користувача за основними сценаріями та вбудованої довідки у вигляді inline-валідацій і tooltip-ів.

Кількісні показники виконаної роботи: 149 Kotlin-файлів, 10 483 рядки продукційного коду, 34 тестові файли (33 юніт + 1 інструментальний), 3 Room-сутності, 6 колекцій Firestore, 9 маршрутів навігації, 22 use case-класи, 7 ViewModel-класів. Доменну логіку покрито юніт-тестами усіх ключових use case-класів. Документ містить 9 рисунків, 10 таблиць і 8 лістингів програмного коду.

Подальший розвиток системи доцільно вести у трьох напрямах. По-перше, доповнити тестове покриття: написати ViewModel-тести для решти екранів і Compose UI-тести через ComposeTestRule. По-друге, винести частину серверної логіки у Cloud Functions для складніших сценаріїв (наприклад, перевірка SLA дедлайнів і ескалація через email). По-третє, розширити аналітичний модуль інтеграцією з Mixpanel або Amplitude для побудови команди-агрегованих метрик продуктивності. Кожен з напрямів може стати темою окремої магістерської роботи на кафедрі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Forsgren N., Humble J., Kim G. Accelerate State of DevOps Report 2023. URL: https://cloud.google.com/devops/state-of-devops (дата звернення: 03.03.2026).

2. GitLab 2024 Global DevSecOps Report. URL: https://about.gitlab.com/resources/developer-survey/2024/ (дата звернення: 01.05.2026).

3. Asana Guide. URL: https://asana.com/guide (дата звернення: 19.02.2026).

4. Jira Software documentation. URL: https://support.atlassian.com/jira-software-cloud/ (дата звернення: 09.04.2026).

5. Todoist Help Center. URL: https://todoist.com/help (дата звернення: 15.04.2026).

6. Trello product documentation. URL: https://support.atlassian.com/trello/ (дата звернення: 16.02.2026).

7. Jetpack Compose documentation. URL: https://developer.android.com/jetpack/compose (дата звернення: 27.02.2026).

8. Kotlin Coroutines on Android. URL: https://developer.android.com/kotlin/coroutines (дата звернення: 11.03.2026).

9. Kotlin documentation. URL: https://kotlinlang.org/docs/home.html (дата звернення: 22.02.2026).

10. Save data in a local database using Room. URL: https://developer.android.com/training/data-storage/room (дата звернення: 04.03.2026).

11. Schedule tasks with WorkManager. URL: https://developer.android.com/topic/libraries/architecture/workmanager (дата звернення: 18.03.2026).

12. Guide to app architecture. URL: https://developer.android.com/topic/architecture (дата звернення: 25.03.2026).

13. Sills B., Gardner B., Marsicano K., Stewart C. Android Programming: The Big Nerd Ranch Guide. 5th ed. Addison-Wesley Professional, 2022. 688 с.

14. Cloud Firestore documentation. URL: https://firebase.google.com/docs/firestore (дата звернення: 08.04.2026).

15. Cloud Storage for Firebase documentation. URL: https://firebase.google.com/docs/storage (дата звернення: 12.04.2026).

16. Firebase Authentication documentation. URL: https://firebase.google.com/docs/auth (дата звернення: 01.04.2026).

17. Firebase Cloud Messaging documentation. URL: https://firebase.google.com/docs/cloud-messaging (дата звернення: 19.04.2026).

18. App security best practices. URL: https://developer.android.com/topic/security/best-practices (дата звернення: 27.03.2026).

19. OWASP Mobile Application Security Verification Standard 2.0 (MASVS). URL: https://mas.owasp.org/MASVS/ (дата звернення: 21.03.2026).

20. Storage updates in Android 11 (Scoped Storage). URL: https://developer.android.com/about/versions/11/privacy/storage (дата звернення: 02.04.2026).

21. Material Design 3 guidelines. URL: https://m3.material.io/ (дата звернення: 24.02.2026).

22. Navigation Compose documentation. URL: https://developer.android.com/jetpack/compose/navigation (дата звернення: 28.02.2026).

23. Nielsen J. 10 Usability Heuristics for User Interface Design. URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (дата звернення: 25.04.2026).

24. Hilt dependency injection. URL: https://developer.android.com/training/dependency-injection/hilt-android (дата звернення: 06.03.2026).

25. Glance – App widgets in Compose. URL: https://developer.android.com/jetpack/compose/glance (дата звернення: 14.03.2026).