ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Мобільна цифрова система голосових сповіщень для ОС Android на основі нейромережевої моделі синтезу мовлення»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Артем РУДИК
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-41

Артем РУДИК

Керівник: д-р філософії Микола ГЕРЦЮК
д-р техн. наук, проф.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Рудику Артему Всеволодовичу

1. Тема кваліфікаційної роботи: «Мобільна цифрова система голосових сповіщень для ОС Android на основі нейромережевої моделі синтезу мовлення»
керівник кваліфікаційної роботи д-р філософії Микола ГЕРЦЮК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № _51_.

2. Строк подання кваліфікаційної роботи «___» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості, система для трансформації текстових сповіщень у голосові для ОС Android.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Проаналізувати предметну область сповіщень смартфонів.
2. Проаналізувати існуючі рішення озвучення сповіщень для Android.
3. Сформувати вимоги до системи з урахуванням попереднього аналізу, користувача та ОС.
4. Вибрати та обґрунтувати інструменти і технології для реалізації системи.
5. Спроєктувати архітектуру системи й алгоритми обробки сповіщень та їх озвучення.

6. Реалізувати систему із підтримкою черги озвучення, визначення мови та синтезу мовлення для Android.
7. Провести тестування обробки та озвучення сповіщень TTS-сервісами.

5. Перелік графічного матеріалу: *презентація*

1. Актуальність
2. Мета та завдання
3. Об’єкт, предмет та методи дослідження
4. Аналіз аналогів
5. Вимоги до системи
6. Проєктування архітектури системи
7. Проєктування алгоритму роботи системи
8. Програмна реалізація. Інтерфейс
9. Функціональне тестування
10. Нефункціональне тестування
11. Демонстрація роботи додатка
12. Висновки
13. Апробація результатів дослідження

6. Дата видачі завдання «_20_» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	01.02-05.02.2026	
2	Огляд предметної області	05.02-15.02.2026	
3	Аналіз та дослідження існуючих аналогів	15.05-28.05.2026	
4	Проектування архітектури та алгоритмів	01.03-10.03.2026	
5	Програмна реалізація системи	10.03-31.03.2026	
6	Тестування застосунку та	01.04-14.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	14.04-30.04.2026	
8	Розробка демонстраційних матеріалів	01.05-08.05.2026	
9	Попередній захист роботи	09.05-12.05.2026	

Здобувач вищої освіти _____ Артем РУДИК
(підпис)

Керівник
кваліфікаційної роботи _____ Микола ГЕРЦЮК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 100 стор., 3 табл., 33 рис., 34 джерел.

Мета роботи – створення системи як альтернативного способу отримання інформації, що передається через сповіщення сучасних смартфонів.

Об'єкт дослідження – процес обробки та передачі текстових даних у мобільному середовищі.

Предмет дослідження – методи та алгоритми обробки й озвучення текстових сповіщень у мобільних системах Android.

Короткий зміст роботи: У роботі проаналізовано предметну оласть сповіщень в середовищі ОС Android. Розглянуто та проаналізовано існуючі програмні рішення для озвучення сповіщень. На основі попереднього етапу сформовано вимоги до системи з врахуванням потреб користувача та специфіки ОС Android. Обґрунтовано та обрано необхідні для реалізації технології та інструменти: Kotlin, Android studio, ML Kit, Google TTS, Gemini 2.5 Flash TTS. Спроєктовано архітектуру та алгоритми необхідні для роботи системи. Реалізовано систему обробки та озвучення сповіщень з детальним описом проведеної роботи. Проведено функціональне та нефункціональне тестування, що виявило дефекти Bluetooth-фільтра та детектора мови. Надана якісна оцінка з демонстрацією ефективності системи та проблем її експлуатації.

Сферою використання застосунку є повсякденне використання мобільних пристроїв з ОС Android у випадках, коли пряма взаємодія з екраном смартфона є неможливою, обмеженою, незручною або небезпечною.

КЛЮЧОВІ СЛОВА: TTS, ФІЛЬТРАЦІЯ, ANDROID, ЗАДАЧА, ML KIT, VERTEX AI, GEMINI, МОДЕЛЬ, КОМПОНЕНТ, МОДУЛЬ, СЕРВІС, ОЗВУЧЕННЯ, МОВА, ГОЛОС, СПОВІЩЕННЯ.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасному світі мобільні пристрої стали основним джерелом інформації, що надходить постійно і у самих різних формах. Однією з таких форм є сповіщення – короткий звіт про те яка інформація прийшла, звідки і коли. Без сповіщень, неможливо було б уявити своєчасну реакцію на події. Проте, для доступу до цієї інформації, необхідно мати прямий доступ до екрана смартфону.

Існує досить багато сценаріїв коли отримати цей доступ не просто незручно, але і нереально. У повсякденному житті, телефон як правило знаходиться в кишені або поблизу, що потребує використання рук для перегляду вмісту. Прогулянки, заняття спортом або водіння автівки створює необхідність відволіктися на перегляд сповіщення, що веде до дискомфорту, ускладнює виконання основних дій або навіть створює небезпечні ситуації.

Саме тому виникає необхідність в іншому способі отримання даної інформації. Проаналізувавши інші від очей органи чуття, легко дійти висновку, що найкращим чином буде передавати дані у вигляді аудіо. Отож, з’являється потреба в системі, що буде перетворювати текстові сповіщення у потік мовлення, який надасть доступ до інформації без задіяння рук.

Тому метою роботи є підвищення ефективності взаємодії користувача з мобільним пристроєм шляхом розробки мобільної цифрової системи голосових сповіщень для ОС Android на основі нейромережевого синтезу мовлення.

Виходячи з мети, завданнями до бакалаврської роботи є:

- Проаналізувати предметну область сповіщень смартфонів;
- Проаналізувати існуючі рішення озвучення сповіщень для Android;
- Сформувати вимоги до системи з урахуванням попереднього аналізу, користувача та ОС;
- Вибрати та обґрунтувати інструменти і технології для реалізації системи;
- Спроєктувати архітектуру системи й алгоритми обробки сповіщень та їх озвучення;

– Реалізувати систему із підтримкою черги озвучення, визначення мови та синтезу мовлення для Android;
– Провести тестування обробки та озвучення сповіщень TTS-сервісами.

Об'єкт дослідження – процес обробки та передачі текстових даних у мобільному середовищі.

Предмет дослідження – методи та алгоритми обробки й озвучення текстових сповіщень у мобільних системах Android.

Методи дослідження. У роботі використано методи порівняння, аналізу та узагальнення існуючих підходів до реалізації систем озвучення сповіщень а також методи проєктування цифрових систем. Для обробки текстових даних застосовано алгоритмічні методи, зокрема методи фільтрації, визначення мови та озвучення на основі налаштувань користувача. Для реалізації синтезу мовлення використано методи роботи з системними та хмарними TTS-сервісами. Оцінювання якості роботи системи здійснювалось шляхом тестування у різних сценаріях використання.

Практична новизна одержаних результатів. У ході дослідження був створений підхід до озвучення текстових сповіщень для Android, який використовує, нейромережеву та генеративну модель синтезу мовлення, а також адаптивне визначення мови для окремих частин повідомлення. Даний підхід дозволяє гнучко обробляти сповіщення та налаштовувати озвучення відповідно до потреб користувача, що підвищує зручність сприйняття інформації.

Практична значущість одержаних результатів. можливість використання розробленого програмного рішення в реальних умовах. Подальші дослідження можуть бути спрямовані на розширення функціоналу системи, зокрема реалізацію інтерактивної взаємодії з повідомленнями або ж збільшення асортименту налаштувань та кастомізації, розробку моделі визначення мови з урахуванням одночасно фонограм та n-грам.

Апробація результатів та публікації. Основні положення і результати бакалаврської роботи доповідались на науково практичних конференціях, що

проходили на базі Державного університету інформаційно-комунікаційних технологій:

1. Рудик А. В., Герцюк М. М. Обґрунтування вибору моделей TTS для озвучення сповіщень на ОС Android // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація», 1–3 травня 2026 року, м. Київ. Збірник тез. (подано до друку).
2. Рудик А. В., Герцюк М. М. Розробка алгоритму озвучення сповіщень на мобільних пристроях на базі ОС Android // VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, м. Київ. Збірник тез. С. 368.

1. ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

Цей розділ присвячений теоретичному огляду та аналізу предметної області та існуючих програмних рішень для вирішення проблеми. Розглянуто властивості, структуру, види сповіщень, а також обмеження існуючих TTS-систем. Окрім цього, описано обрані технології та інструменти, для розробки системи озвучення повідомлень на ОС Android.

1.1 Поняття та властивості сповіщень

Розуміння сутності сповіщень є ключовим для визначення вимог до системи озвучення. Сповіщення – це елемент інтерфейсу, повідомлення яке з’являється за межами додатків із метою передати певну інформацію. Зазвичай, це повідомлення від іншої людини, новинна розсилка, статус певної дії, додатку, чи надання важливої інформації, проте зміст не обмежений конкретними типами інформації. Стилізація в основному базується на основі комплекту дизайнерських шаблонів від Android, але не обмежена ними і допускає власні творчі рішення. Для ілюстрації різноманітності та гнучкості візуальної кастомізації сповіщень (на прикладі медіаплеєра Spotify) наведено рисунки 1.1–1.3. Для детального розуміння їх функціональності та впливу на досвід користувача доцільно розглянути основні сценарії взаємодії зі сповіщеннями, їх структуру та додаткові класифікації.

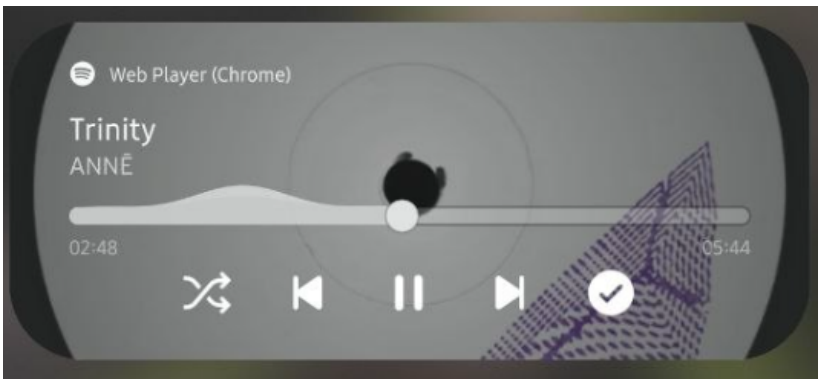


Рис. 1.1 Сповіщення зі змістом медіаплеєра Spotify

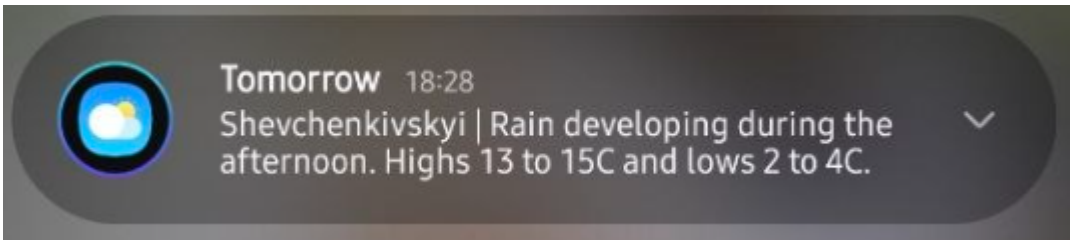


Рис. 1.2 Сповіщення зі звичайним текстовим змістом

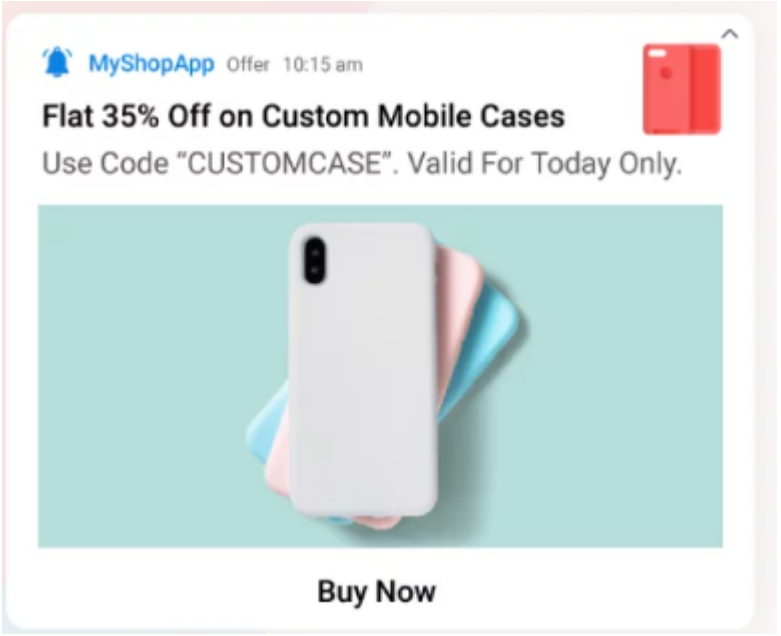


Рис. 1.3 Сповіщення з змістом зображення

При натисканні на сповіщення є декілька можливих сценаріїв, залежно від того, які елементи присутні і куди тисне користувач. Перший – нічого не відбувається, просто інформаційне сповіщення, це доволі рідкісний випадок у сучасному світі. Другий – цей сценарій витіснив перший, тут телефон запускає додаток і конкретне його вікно. Третій – зазвичай властивий месенджерам чи іншим додаткам для спілкування, де відкривається поле для вводу повідомлення відповіді. Четвертий – також стосується спілкування, але на цей раз пропонується розумна відповідь одним натисканням кнопки. П’ятий – здійснюється управління медіа за допомогою контролера. Шостий – виконується інша, специфічна дія, наприклад зупинка таймера. Сьомий – взаємодія із дзвінком. Усі вищеперераховані сценарії можуть комбінуватись залежно від потреб. До того ж

сповіщення можна змахувати і видаляти, а також розгортати, якщо вони містять багато тексту чи зображення.

Структурно сповіщення складаються з наступних елементів: Пакетні дані, контент, функціональні елементи, технічні прапорці, канали і важливість. В пакетних даних у нас назва додатку, час отримання та ідентифікатор сповіщення, інколи ще певні ярлики. В контенті знаходиться автор та основний текст сповіщення, також можуть бути присутні й інші елементи. Функціональні елементи як зрозуміло з назви це реалізація вищеописаних сценаріїв взаємодії з додатком. Технічні прапорці відповідають за те як сповіщення взаємодіє із Android, наприклад необхідно, аби сповіщення не очищалось при натисканні відповідної кнопки і для цього ставиться відповідний прапорець. Канали і важливість відповідають за те до якої категорії належить сповіщення і наскільки воно важливе. Категорій є доволі багато: для пошти, для повідомлень, дзвінків, реклами, будильника і т.д.. Рівнів важливості усього 4 і від них залежить, що користувач побачить та почує: терміновий – звук та спливаюче вікно, високий – звук зі спливаючою іконкою, середній – сама спливаюча іконка, низький – сповіщення просто з’являється в статусі.

Сповіщення також можна розділити за часом життя. Є тимчасові – які можна змахнути і видалити у разі їх появи. Є постійні – позбавитись їх не получиться до виконання певної дії чи закриття додатку.

Ще можна розрізняти сповіщення за стилем і джерелом, проте це не релевантно для даної роботи.

Як висновок, сповіщення є невід’ємним елементом взаємодії користувача зі смартфоном та встановленими додатками, який може містити різні типи інформації та функціональні елементи і має комплексну структуру. Розуміння цих особливостей дозволить ефективно декомпозувати об’єкт сповіщення, визначити необхідні налаштування, а також створювати власні сповіщення [3] [4].

1.2 Огляд та аналіз існуючих рішень

В цьому підрозділі будуть розглянуті та проаналізовані програмні рішення, які вже вирішують проблему, та буде розглянутий їх функціонал, переваги і недоліки. Дані дії допоможуть зрозуміти наявні проблеми і дозволять скласти список вимог для нашої системи. На розгляд були обрані 6 застосунків із Google Play Market: Notification Reader (Argon Dev) [5], Notification Reader (Malcolm Bryant) [6], Notification Reader (DFound) [7], Audify [8], Voice Notify [9], Notivoca [10].

Також тут буде звучати словосполучення “озвучувати n-мовним голосом”. Під цим мається на увазі не просто використати конкретний голос і прочитати слова, а використати модель TTS, яка натренована на озвучці цієї мови. Тобто модель використовує правила і фонетику мови, а не лише слова. Тому якщо дати моделі мови “X”, текст мовою “Y”, то вимова буде не коректною згідно правил мови “Y” і буде звучати невнятно і незрозуміло. Подібний ефект можна побачити коли люди не знаючи мови намагаються вимовляти її слова.

Тестуватись додатки будуть використовуючи програму Fake Notifications [11], що дозволяє самому створювати сповіщення. Визначення технологій та їх використання буде проводитись зміною змісту сповіщення. Різні пропорції мов в різних частинах сповіщення дозволять виявити алгоритм роботи програм. По вимові тексту можна зрозуміти яку мову обрав TTS, а також яку частину сповіщення він озвучує і провести відповідно аналіз.

Notification Reader (Argon Dev) (див. рисунок 1.4). Програма змішує усі дані сповіщення, а саме пакетні дані і контент в один текстовий блок, після чого дає на озвучку Google TTS. Дане рішення є простим, але водночас і проблемним, адже визначення мови віддається TTS, а враховуючи що частина даних це англійський текст, то озвучування інших мов буде працювати по різному щоразу. Короткі сповіщення будуть озвучуватись англомовним голосом, через що зміст буде незрозумілими, а довгі – з більшою ймовірністю будуть озвучуватись уже правильним голосом.

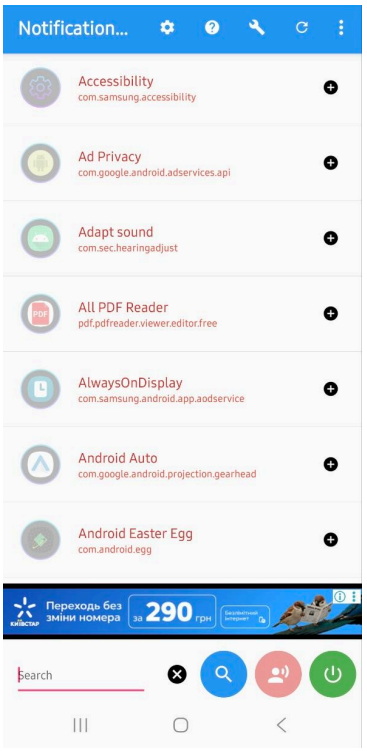


Рис. 1.4 Notification Reader (Argon Dev)

Недоліки:

- незручний і застарілий дизайн;
- не оптимізоване використання заряду акумулятора;
- відсутня кастомізація;
- налаштувань практично немає;
- всередині додатку розміщена реклама;
- може зламатись при тривалій роботі.

Переваги:

- можна обрати додатки чиї сповіщення будуть озвучуватись;
- додаток безкоштовний.

Notification Reader (Malcolm Bryant) (див. рисунок 1.5). Даний додаток знаходиться в ранньому доступі, тому велика частина функціоналу відсутня. Реалізація озвучки ідентична до попереднього програмного рішення.

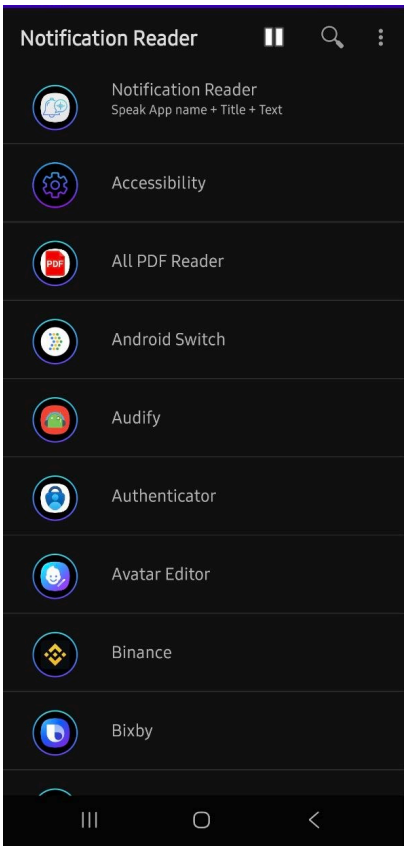


Рис. 1.5 Notification Reader (Malcolm Bryant)

Недоліки:

- увімкнення супроводжується голосовим озвученням;
- не оптимізоване використання заряду акумулятора;
- налаштування не працюють;
- налаштування необхідних додатків не зручне;
- може зламатись при тривалій роботі.

Переваги:

- налаштувань і кастомізації доволі багато;
- можна обрати додатки чиї сповіщення будуть озвучуватись;
- додаток безкоштовний.

Notification Reader (DFound) (див. рисунок 1.6). Як і попередні додатки, також збирає в один блок, але вказує одну єдину мову для озвучки. Через це іншомовні слова повністю пропускаються. Також всередині додатку є можливість протестувати голос, проте тест не симулює реальну поведінку із сповіщеннями.

Додатком заявлена підтримка різних TTS, проте TTS від Samsung не підтримується, щодо інших постачальників, то для їх перевірки потрібен інший смартфон.

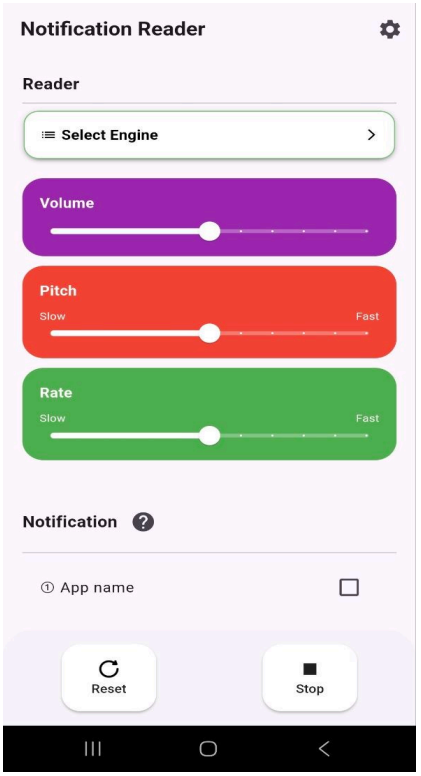


Рис. 1.6 Notification Reader (DFound)

Недоліки:

- присутня реклама;
- налаштувань мало і вони не працюють;
- по стандарту швидкість голоса занадто висока;
- зміна гучності не працює;
- легко може зламатись в процесі.

Переваги:

- вибір додатків, чиї сповіщення будуть озвучуватись;
- візуально приємний дизайн;
- додаток безкоштовний.

Audify (див. рисунок 1.7). На відміну від попередніх програмних рішень, розділяє частини сповіщення і віддає їх окремо TTS. Щоправда, TTS не

призначений для визначення мови, за умови використання різномовних слів, або коли речення що може бути прочитане декількома мовами, вимова буде неправильна. Як приклад, для речення “Я щойно checkout-нув новий branch у Git” TTS використає англомовний голос, хоча для нас очевидно, що тут українська із вживанням англійських слів, тому було б правильно обрати саме україномовний голос.

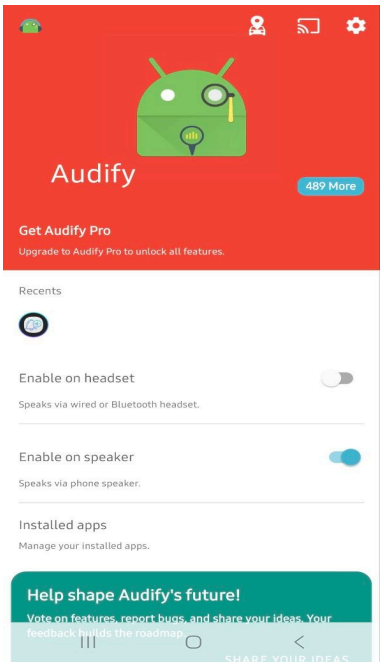


Рис. 1.7 Audify

Недоліки:

- присутня реклама;
- обмеження озвучення без підписки;
- не працює якщо у користувача обраний інший TTS;
- не оптимізоване використання акумулятора;
- може зламатись при тривалій роботі;
- по замовчуванню увімкнена озвучка усіх додатків.

Переваги:

- візуально приємний дизайн;
- багато налаштувань;

– посередня кастомізація.

Voice Notify (див. рисунок 1.8). Як і Notification Reader (Argon Dev) та Notification Reader (Malcolm Bryant), це програмне рішення збирає всі дані сповіщення в один блок і віддає TTS.

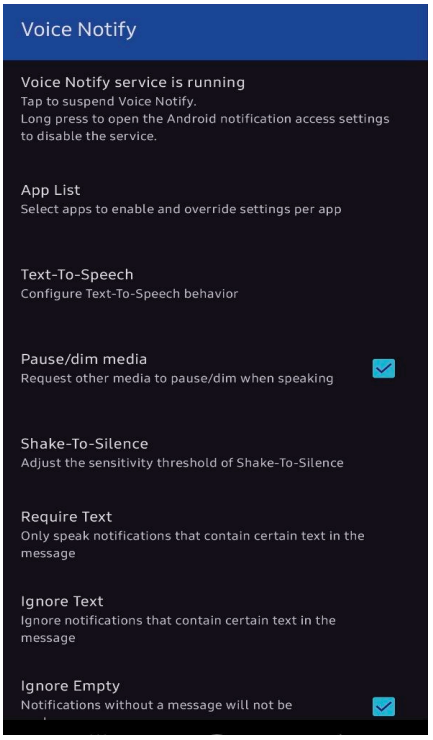


Рис. 1.8 Voice Notify

Недоліки:

– дизайн – це набір списків;
– по замовчуванню увімкнена озвучка усіх додатків;
– збирає дані сповіщень.

Переваги:

– багато налаштувань і кастомізації;
– додаток безкоштовний.

Notivoca (див. рисунок 1.9). Дана програма працює аналогічно до попередніх програмних рішень і також само передає всі дані сповіщення TTS в одному блоці.

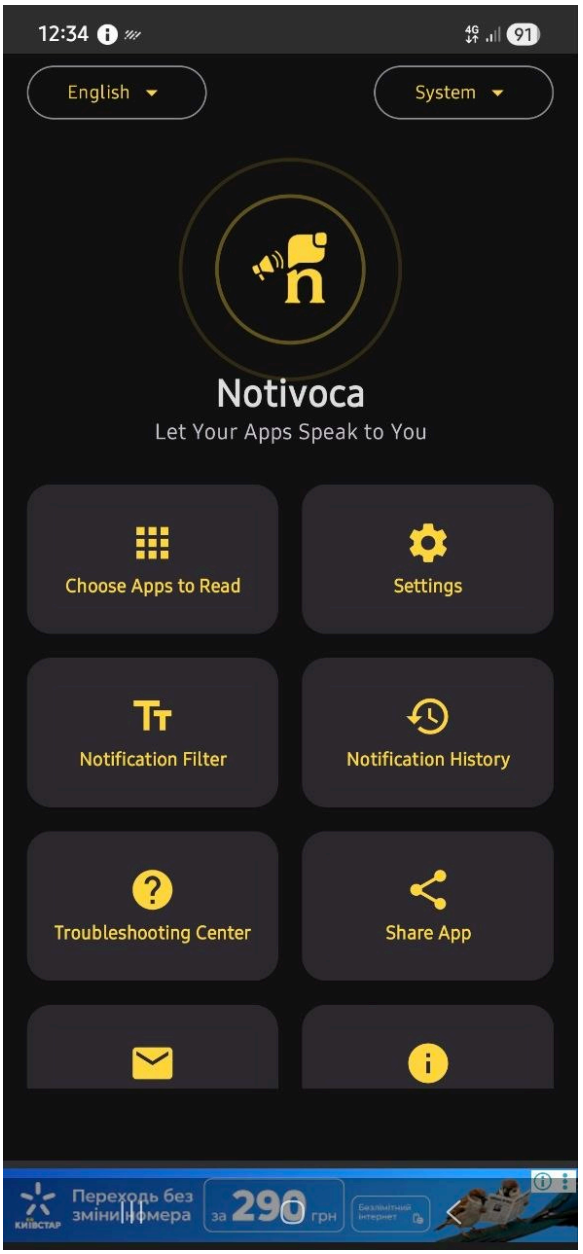


Рис. 1.9 Notivoca

Недоліки:

- може зламатись при тривалій роботі;
- збирає дані сповіщень;
- в додатку присутня реклама.

Переваги:

- приємний дизайн;
- дуже багато налаштувань і кастомізації;

– додаток безкоштовний.

Узагальнене порівняння попередньо розглянутих програмних рішень наведено в таблиці 1.1.

Таблиця 1.1

Зведена таблиця порівняння

Аналог	Визначення мови	Обробка сповіщення	Якість інтерфейсу	Автономність і стабільність	Кількість налаштувань та кастомізації	Конфіденційність	Гнучка фільтрація	Монетизація та реклама
Notification Reader (Argon Dev)	TTS	Монолітна	Низька	Низька	Низька	Невизначено	Так	Безкоштовний, з рекламою
Notification Reader (Malcolm Bryant)	TTS	Монолітна	Середня	Низька	Висока (налаштування не працюють)	Невизначено	Так	Безкоштовний
Notification Reader (DFound)	TTS	Монолітна	Висока	Низька	Низька (налаштування не працюють)	Невизначено	Так	Безкоштовний, з рекламою
Audify	TTS	Фрагментована	Висока	Низька	Висока	Невизначено	Так	Freemium
Voice Notify	TTS	Монолітна	Висока	Середня	Висока	Наявні ризики витоку даних	Так	Безкоштовний
Notivoca	TTS	Монолітна	Висока	Низька	Висока	Наявні ризики витоку даних	Так	Безкоштовний, з рекламою

Розглянувши існуючі програмні рішення можна зрозуміти наявні проблеми і шляхи їх вирішення.

Існуючі додатки використовують TTS для визначення мови, що є вкрай неправильно. Технологія не призначена для таких цілей і тому виконує поставлену задачудоволі поверхнево. Рекомендується делегувати це завдання детектору мови. Основними принципами роботи детектору є n-грами, комбінації символів які використовуються в певній послідовності для визначення ймовірності тої чи іншої мови. Такий підхід сильно підвищує точність визначення мови, у порівнянні з TTS.

Обробка всього сповіщення як єдиного текстового блоку негативно впливає на точність визначення мови детектором мови, оскільки метадані часто містять

англомовні елементи, тоді як текст – ні. У результаті це може призводити до озвучки тексту англомовним голосом, навіть коли мала б бути інша мова. З метою уникнення цього, варто передавати і озвучувати сповіщення частинами.

Більшість додатків мають низьку ефективність при роботі з апаратною частиною телефону, що спричиняє швидше розряджання акумулятора, а також нестабільну роботу самого додатку. Для запобігання даних проблем, оптимізація використання ресурсів та інформація про активність смартфону є доцільними.

Обов’язковою умовою є користувачу вибору, що саме озвучувати. Наприклад, озвучка зарядки не несе жодної користі, адже користувач власноруч поставив смартфон заряджатись і уже знає про цю дію. Особливо враховуючи, що умовна зарядка це не одне сповіщення, а нескінченна черга сповіщень, що призводить до засмічення аудіопотоку. Пропонується подібно до розглянутих програмних рішень, створити список де можна налаштувати, сповіщення якого додатка будуть озвучувати.

Необхідні налаштування та кастомізація, аби користувач міг адаптувати додаток під власні потреби і користуватись без незручностей. Рекомендується надати можливість налаштувати голос, швидкість мови, висоту голоса та інші параметри, що будуть позитивно впливати на досвід користування.

В розглянутих додатках, у більшості немає можливості перевірити їх роботу. Аби взнати чи було усе налаштовано правильно доведеться чекати реального сповіщення, або використовувати сторонню програму. Отож, необхідно реалізувати створення власного, тестового сповіщення.

Варто зазначити, що хороший дизайн піднімає ефективність використання додатку. Відповідно, варто намагатись дотримуватись сучасних стандартів користувацького дизайну.

2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА АЛГОРИТМІВ СИСТЕМИ

У цьому розділі здійснюється безпосереднє проєктування архітектури системи та створення її алгоритмічної бази, а також підбір необхідних інструментів для розробки. Мета етапу полягає у формалізації структури мобільного додатка та використовуваних технологій.

2.1 Складання вимог

На основі проведеного аналізу існуючих програмних рішень та виявлених проблем, а також на основі технічних потреб, можна визначити низку вимог до нашої системи. Дані вимоги спрямовані на визначення меж розроблюваного.

Функціональні вимоги визначають які дії додаток повинен виконувати. До них належать:

- перехоплення сповіщень. Додаток повинен у фоновому режимі перехоплювати сповіщення і розділяти дані до озвучки;
- фільтрація. Сповіщення які не відповідають заданим умовам, зокрема налаштуванням користувача не повинні озвучуватись. Серед технічних, обов’язкових фільтрів наступні:
    1) сповіщення є дублікатом в межах 5 секунд;
    2) сповіщення – це звіт до групи сповіщень;
    3) користувач знаходиться в дзвінку;
    4) пустий текст.

  серед користувацьких, необов’язкових фільтрів наступні:
    1) телефон розблоковано;
    2) не підключений Bluetooth-пристрій;
    3) сповіщення від додатку, що не в білому списку;
    4) системні сповіщення.

- нормалізація тексту. Перед визначенням мови, система повинна прибрати слова які можуть заважати, читатись погано і повернути їх після визначення, у нормалізованому, читабельному для TTS стані у відповідній мові;
- визначення мови. Визначення мови повинно відбуватися за допомогою спеціального детектора, для забезпечення високої точності і правильного вибору голосу;
- мультимодальний синтез мовлення. Має бути реалізовано синтез мовлення за допомогою стандартного TTS, а також більш просунутого TTS на основі генеративної моделі ШІ;
- налаштування. Для адаптації додатку під конкретні потреби, має бути створений відповідний інтерфейс. Користувачу мають бути доступні наступні налаштування:
    1) налаштування фільтрів;
    2) налаштування обраних мов для озвучки;
    3) перемикання між TTS;
    4) налаштування TTS.
- тестування – повинна бути реалізована можливість тестувати TTS, за допомогою створення штучного сповіщення.

Для демонстрації та кращого розуміння загальних функціональних можливостей була створена відповідна діаграма прецедентів (див. рисунок 2.1), а для процесу обробки сповіщень наведена діаграма активностей (див. рисунок 2.2).

Нефункціональні вимоги визначають як додаток повинен виконувати вищевказані дії. До них належать:

- енергоефективність. Додаток повинен обмежено використовувати ресурси пристрою (до 0.1% заряду на годину). Для озвучки повинен резервуватись процесор, а після завершення – звільнятись;
- низька затримка. Час між отриманням сповіщення і озвучкою не повинен перевищувати 2 секунди;
- конфідеційність. Робота зі сповіщенням повинна відбуватись локально, де це можливо. Дані не повинні зберігатись на сторонніх серверах.

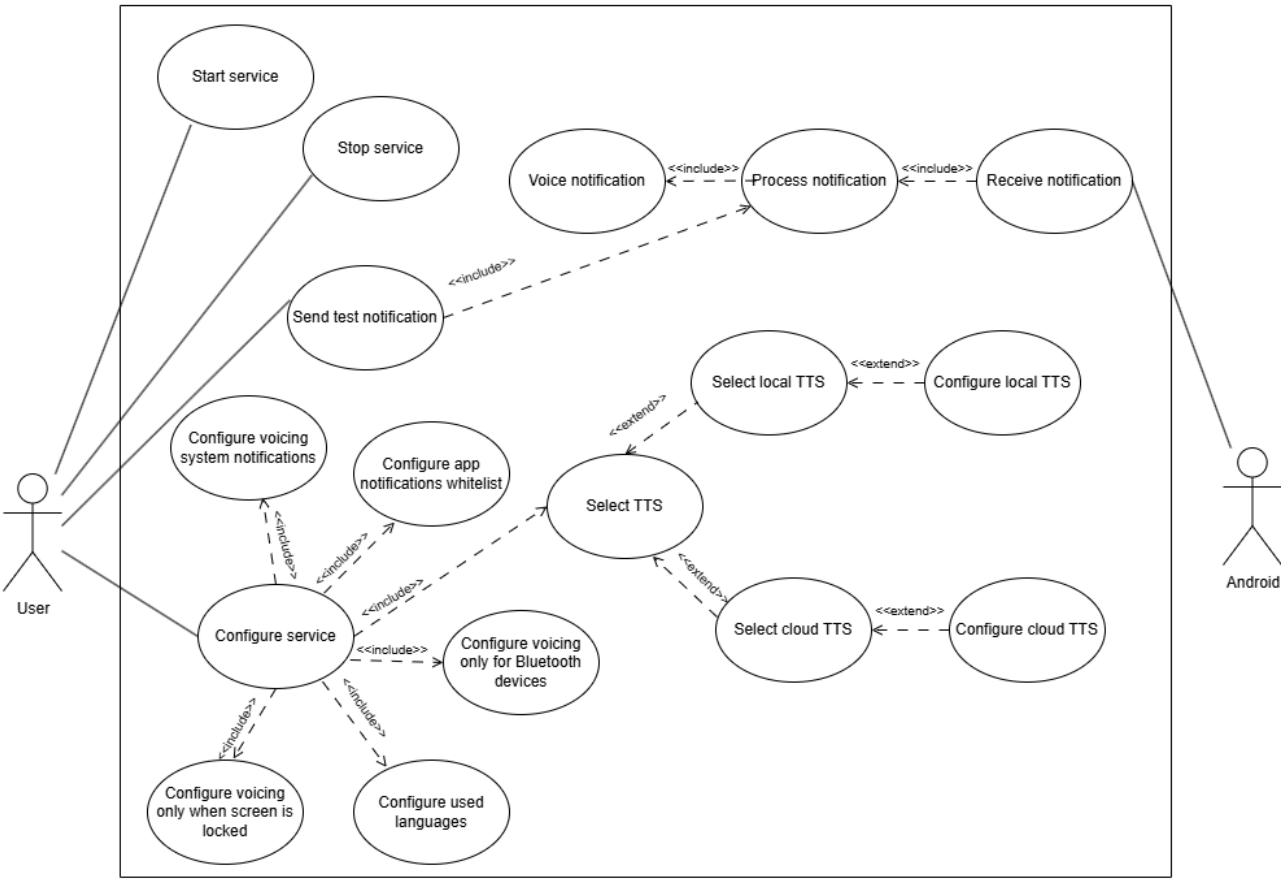


Рис. 2.1 Діаграма прецедентів системи

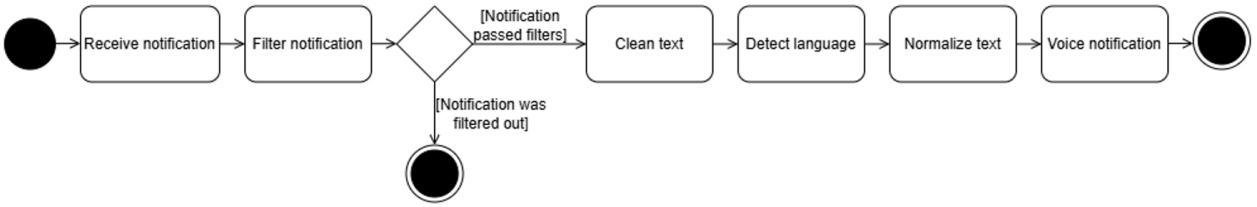


Рис. 2.2 Діаграма активностей обробки сповіщень

Таким чином, визначений перелік функціональних та нефункціональних вимог формує основу для подальшого проєктування архітектури системи.

2.2 Обґрунтування вибору технологій та інструментів

Для реалізації визначених вимог та забезпечення стабільної роботи сервісу у фоновому режимі, необхідно обрати стек технологій та інструментів який дозволить це зробити.

Почнемо з того, що ми робимо додаток для платформи Android, а отже найкраще використовувати офіційні мови, Kotlin [12], або Java. Було вирішено використовувати Kotlin, адже він лаконічніший і безпечніший, до того ж, він повністю сумісний з Java, а це значить, що ми нічого не втрачаємо роблячи вибір на його користь.

Наступним кроком необхідно обрати середовище для розробки (IDE). На вибір є Android Studio [13], Intellij IDEA, VS Code. Серед усіх варіантів для розробки додатків для ОС Android, найкращим варіантом є Android Studio. У нього вбудований емулятор, він дозволяє переглядати інтерфейс при написанні, має спеціалізовані дебаггери і моніторинг ресурсів, тоді як аналогами потрібно ставити додаткові плагіни і шукати обхідні шляхи. Варто зазначити, що моніторинг ресурсів є досить важливою перевагою, адже дозволить перевірити, що додаток працює з залізом ефективно.

Подальшим етапом буде вибір детектора мови. Тут є три шляхи: Google ML Kit [14], Lingua, Compact Language Detector v3 (CLD). CLD був відкинутий одразу, адже він написаний на C++ і для імплементації в проекті доведеться використовувати механізми JNI та NDK, що суттєво ускладнює архітектуру і підтримку не надаючи сильних переваг. Подальший вибір був складніший, адже Lingua точніше за ML Kit визначає мову, проте і використовує більше ресурсів, що веде до зниження енергоефективності. Точність визначення до 1-2 слів знадобиться рідко, адже більшість сповіщень змістовніші, а ось витрачати ресурси на визначення прийдеться однаково завжди. Тому, зваживши всі за і проти, було вирішено обрати ML Kit.

Далі вибір локального TTS. Без вагань було обрано рішення від Google. Аналогів у нього практично немає, він доступний на всіх смартфонах Android і

має найкращі голоси. За відсутності на пристрої, його в будь-який момент можна завантажити з магазину [15].

Наступний на черзі вибір генеративного TTS. Спочатку було вирішено обрати Sherpa-ONNX [16], безкоштовне рішення з відкритим кодом, що підтримує встановлення власних голосових моделей, яке буде працювати офлайн, хоча і ціною меншої енергоефективності. Але при глибшому дослідженні виявилось, що встановлення і робота з ним при наявній документації неможлива, хоча заявляється, що зробити це дуже просто. До того ж українських голосів мало, їх якість жахлива, а як їх встановлювати, аби не шкодити досвіду користувача – невідомо. Тому надалі розглядались лише хмарні рішення: Gemini 2.5 Flash TTS [17], Azure AI Speech [18] та ElevenLabs [19]. Із недоліків, вони потребують інтернет з'єднання і правильної роботи з потоком аудіо, але загалом це краще у всіх інших аспектах, зокрема голоса, енергоефективність, якість, кількість мов. ElevenLabs пропонує найвищу якість голоса і максимальну його гнучкість, проте ціна за це обмеження в 10 000 символів на місяць і затримка більше 1 с. Після виходу за межі доведеться платити близько 8 000 грн за 1 000 000 символів, що є доволі багато. Azure у свою чергу має меншу затримку у 500 мс. Його голоса звучать добре, але проблема в тому що вони звучать не зовсім природньо і налаштовуються через уже визначенні стилі, що буде складно реалізувати в коді. Щодо вартості, то Azure дає 500 000 символів на місяць, чого повністю має вистачити на розробку, тоді як вихід за ліміт буде вартувати лише близько 1 000 грн за 1 000 000 символів. Gemini в свою чергу набагато краще працює з голосами ніж Azure, але гірше ніж ElevenLabs, затримка в середньому до 300 мс. До того ж, Gemini найщедріший, і дає аж 1 000 000 символів на місяць, вихід за ліміт вартує до 200 грн за 1 000 000 символів. Щоправда у нього є додаткове обмеження в 1 500 запитів на добу, але цього спокійно вистачить для розробки та тестування. В результаті було обрано Gemini 2.5 Flash TTS, так як він надає високу якість голосів, низьку затримку і більше можливостей для роботи без вкладання додаткових коштів [1].

Оскільки проєкт досить великий, для відслідковування змін варто використовувати систему контролю версій. Тому, був обраний Github. Він уже вбудований в Android Studio і завдяки студентським привілеям буде кращим рішенням порівняно з GitLab.

Тепер перейдемо до необхідних пристроїв. Android Studio звичайно підтримує емуляцію смартфонів на ОС Android і це дозволить провести певні тести, проте для повноцінного тестування додатку в реальних умовах він не підходить. Перевірка енергоефективності в емуляторі є неможливою. Він використовує процесор та інтернет трафік комп'ютера, тоді як смартфон буде використовувати значно слабший свій процесор і власні модулі для отримання трафіка. Також в емуляторі не можна користуватись Bluetooth-пристроями, коли нам це потрібно для перевірки відповідного фільтру. Отож, будуть використані пристрої, що є під рукою, а саме: Samsung S24 (Android 16.0), Samsung Galaxy A41 (Android 12.0), JBL Charge 3, Samsung Buds 3 pro, Corsair Void Max Wireless.

2.3 Загальна архітектура системи

Для забезпечення гнучкості додатку та зручності розробки було застосовано підхід з орієнтацією на сервісну логіку. На створеній схемі (див. рисунок 2.1) відображено графічне представлення архітектури додатка.

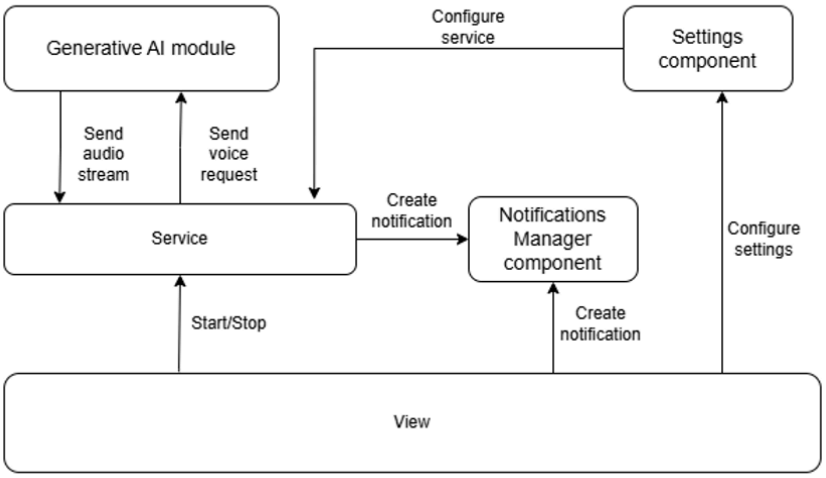


Рис. 2.1 Схема архітектури додатка

Основним вузлом буде фоновий сервіс, який буде працювати безперервно, незалежно від інтерфейсу, що бачить користувач. Такий підхід дозволя завжди моніторити вхідні сповіщення та обробляти їх. Варто зазначити, що детектор мови та TTS – це бібліотечні, вбудовані модулі, тому вони існують як частина сервісу.

Для роботи сервіс використовує окремі модулі та компоненти:

- Generative AI module – це як зрозуміло з назви, модуль для роботи з генеративним штучним інтелектом. Він буде відповідати за взаємодію з хмарним сервісом і повертати потокове аудіо озвучення тексту;
- Notification Manager component – це допоміжний інструмент, який в свою чергу необхідний для налаштування каналів та створення власних сповіщень;
- Settings component – це компонент, що відповідає за збереження всіх налаштувань, параметрів фільтрації, конфігурацію ШІ. Згодом ці дані компонент передає сервісу для прийняття рішень.

Користувач взаємодіє із системою через View, який забезпечує інтерфейс для управління життєвим циклом сервісу та редагування налаштувань.

2.4 Алгоритм обробки сповіщень

Для розуміння дій, які повинен виконувати сервіс і в якій послідовності він повинен їх виконувати, був розроблений відповідний алгоритм. На діаграмі (див. рисунок 2.2) відображено графічне представлення алгоритму обробки сповіщень.

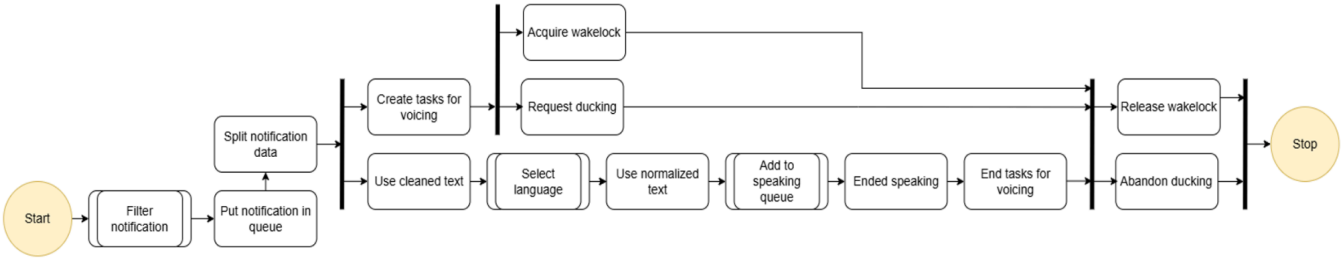


Рис. 2.2 Блок-схема алгоритму обробки сповіщень

Варто наперед зазначити, що фільтрація, озвучення і виявлення мови це окремі алгоритми, які були згорнуті і будуть розкриті дальше. Даний алгоритм приходить в дію при отриманні сповіщення і закінчується коли TTS закінчує говорити.

В даному алгоритмі спочатку виконується фільтрація з урахуванням системних та користувацьких фільтрів. Далі сповіщення додається до черги, що необхідно для випадків коли приходить декілька сповіщень одночасно. Таким чином ми їх будемо обробляти почергово. Якщо сповіщення пройшло фільтр, то воно розділяється на дві частини: заголовок і текст. Для кожної частини створюється задача, як лічильник який дозволить слідкувати за процесом виконання. Коли були додані нові задачі, віддаються команди смартфону, аби він не вимикав процесор і приглушив медіа, щоб користувач міг почути озвучення. Наступним кроком зміст очищається від нечитабельних слів, таких як посилання, номери телефону, пошта, емодзі. Очищений зміст використовується для визначення мови. Якщо ж не очищати зміст, то нечитабельні слова будуть заважати виявляти мову. Потім нормалізується текст. Нечитабельні слова перетворюються у читабельний формат залежно від виявленої мови. Наприклад було посилання в англійському тексті, воно при нормалізації трансформується у “link”. Після цього текст додається у чергу до озвучки. При закінченні озвучки, задачі закриваються. Відповідно при закритті усіх задач звук перестає приглушатись і дозволяється вимикати процесор [2].

2.5 Алгоритм фільтрації

Даний алгоритм перевіряє чи досягнуті умови для оголошення сповіщення. На блок-схемі(див. рисунок 2.3) відображено графічне представлення алгоритму.

Процес обробки кожного сповіщення відбувається за принципом конвеєра. Обробка відбудеться лише якщо сповіщення дійде до кінця. Інакше якщо сповіщення зупиняється хоча б на одному фільтрі то подальша робота з ним зупиняється.

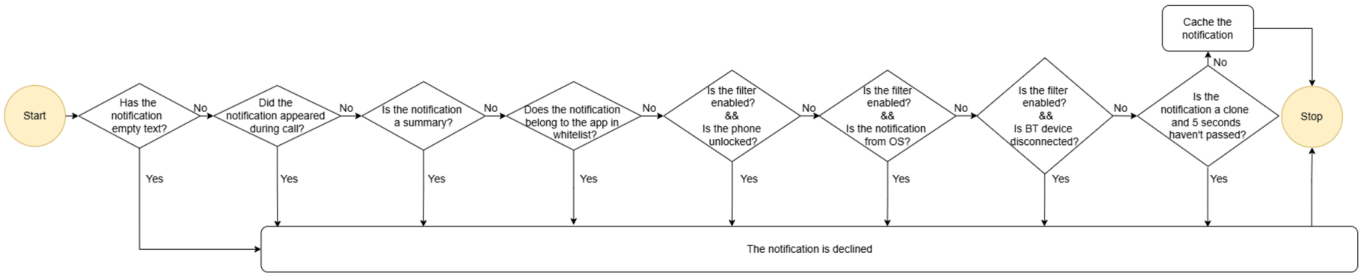


Рис. 2.3 Блок-схема алгоритму фільтрації

Тут реалізовані всі фільтри вказані у вимогах. Важливо те, що системні фільтри обов'язкові і через них сповіщення проходить завжди, тоді як проходження користувацьких залежить від параметрів налаштованих користувачем.

Також важливий нюанс, що при проходженні фільтрів в кінці записується сповіщення в кеш, що дозволяє ідентифікувати дублікати. Запис обов'язково повинен відбуватися саме в кінці. Інакше, якщо робити це в початку, то зайвий раз будуть витрачатись ресурси на запис сповіщення в кеш, хоча система його не озвучувала. Щодо фільтрів, то порядок не важливий, адже так чи інакше сповіщення повинно пройти всі етапи.

2.6 Алгоритм вибору мови

Самостійне використання детектора мови значно піднімає якість вибору мови для TTS, але цього недостатньо, особливо враховуючи, що користувач може налаштовувати мови. Для таких цілей був розроблений відповідний алгоритм, який допоможе пояснити логіку вибору мови (див. рисунок 2.4).

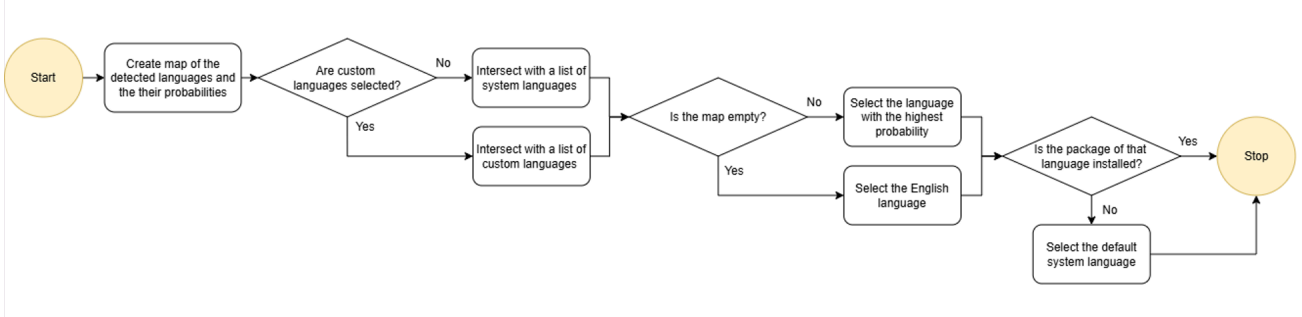


Рис. 2.4 Блок-схема алгоритму вибору мови

Даний алгоритм реалізує послідовний процес прийняття рішень, який спочатку з допомогою детектора аналізує текст по його n-грамам і створює повний перелік можливих мов та їх ймовірності. Наступним кроком перелік детектора перетинається зі списком мов користувача. Залежно від налаштувань це або список його системних мов, або список обраних мов. У випадку коли при перетині перелік виявився порожнім, обирається англійська мова як запасний, універсальний варіант. В разі, якщо при пересічені у нас залишились мови, обирається та, що має найбільшу ймовірність. В кінці стоїть технічний запобіжник. Якщо вийшло так, що TTS не може найти голос мови, то обирається користувацька мова за замовчуванням.

2.7 Алгоритм озвучення

Процес перетворення текстового сповіщення в аудіопотік є багатоетапною і складною процедурою, що вимагає чіткого порядку. Враховуючи ці особливості, було розроблено відповідний алгоритм, який визначає логіку озвучення залежно від налаштувань користувача (див. рисунок 2.5).

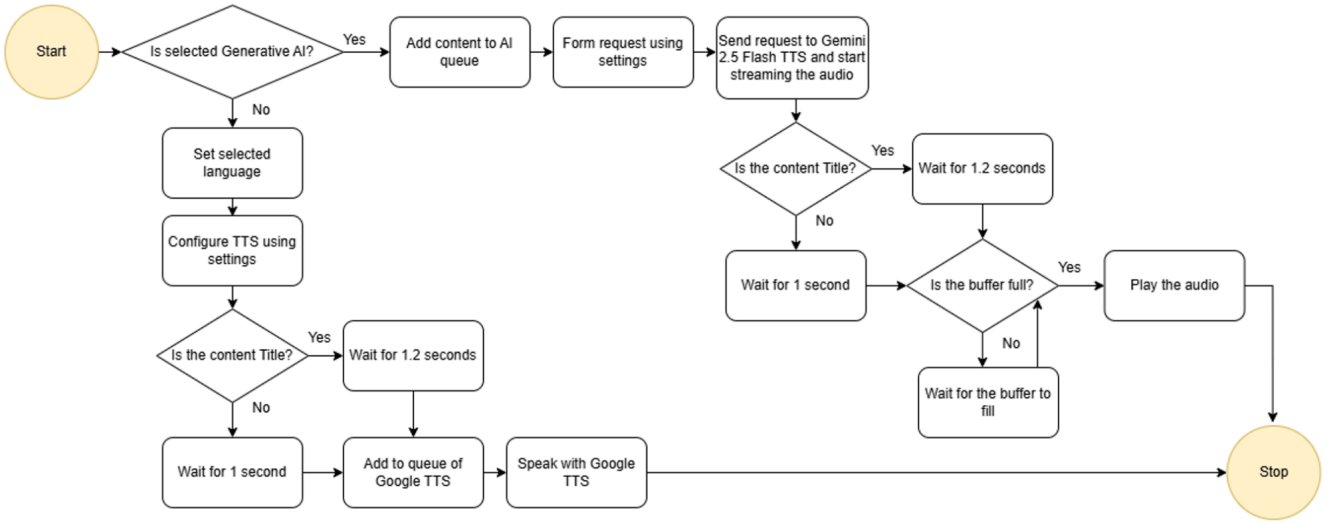


Рис. 2.5 Блок-схема алгоритму озвучення

Залежно від обраного користувачем TTS, у системі реалізовано два різні підходи: один для генеративного синтезу мовлення, інший — для локального

системного. Для початку розберемо шлях роботи для локального.

Завдяки роботі детектора, чітко встановлюється мова для TTS. Далі налаштовується TTS: голос, висота, а також швидкість мовлення. Наступним кроком, залежно від того який контент прийшов, виставляється затримка перед мовленням. Якщо це заголовок, то затримка 1.2 секунди, аби звук сповіщення встиг програтись і не заважав слухати озвучення. Якщо це текст, то виставляється затримка в 1 секунду, аби чітко розділити мовлення із заголовком. Після цього, затримка зі змістом передається у чергу TTS. Як тільки TTS може говорити, він почергово озвучує всі елементи в черзі, в тому числі і затримку (затримка озвучена як пауза, мовчання).

З хмарним сервісом робота складніша, адже тут уже необхідно працювати з передачею даних в мережі інтернет. Тому розберемо детальніше, що саме потрібно зробити для роботи з генеративним TTS.

Додавання змісту в чергу знаходиться на початку. Такий підхід необхідний для почергової обробки на всіх етапах. Оскільки у смартфона може бути слабкий інтернет, то відповідно не можна просто почати працювати з усіма сповіщеннями одразу і завантажувати їх аудіо, адже це призведе до глобальних проблем з затримкою. Трафік один, тому чим більше озвучень буде завантажуватись за раз тим сильніше доведеться ділити його. Після цього, потрібно сформувати запит до хмарного сервісу. Необхідно надати авторизаційні дані, мову, голос та промпт. Варто зазначити пару важливих моментів. Перше, генеративний TTS дуже добре працює з різними мовами в одному тексті, тому можна загалом і не вказувати мову, він сам здогадається, але для більшої точності вимови, краще її уточнювати. Друге, промпт може бути доволі гнучким, адже це LLM модель і від цього залежить яке аудіо вона згенерує. Окрім основного тексту повідомлення, промпт може містити інструкції для TTS: додавання певних фраз, слів-паразитів чи емоційних вигуків. Також можна керувати стилем мовлення, що дозволяє задавати певні характеристики голосу чи інтонаційне забарвлення, наприклад, переможна промова або нейтральна дикторська вимова. Водночас, це лише мала частка можливостей, які можна реалізувати за допомогою промпта, і повний їх потенціал

складно передбачити. Наразі все це налаштування здійснюється в межах одного промпта і може приводити до неочікуваних результатів, одна загалом відповідь повинна залишатися стабільною. Наступним кроком, відправляється запит і починається стрімінг потоку аудіо даних. Без стрімінгу довелося б чекати доки TTS повністю сформує аудіо доріжку, після цього завантажувати її і тільки після повного завантаження програвати її. Завдяки стрімінгу можна завантажувати аудіо в процесі його формування і програвати аудіо майже відразу. До того ж, пропонується використовувати буфер, при наповненні якого можна одразу почати програвати озвучення і в процесі, дозавантажити недостаючу частину даних. Загалом, це дозволить досягти низької затримки. Повільне наповнення буфера буде створювати розриви в аудіо, але об'єм даних доволі низький, тому навіть у 3G не повинно бути проблем зі завантаженням. Згодом, коли завантаження вже розпочалося, як і у випадку з локальним TTS, виконується коротке очікування з тією ж метою та за тією ж логікою. Як тільки час очікування завершився, програється аудіо.

2.8 Алгоритми запуску та зупинки сервісу

Із урахуванням попередніх алгоритмів, простий запуск і зупинка сервісу є неможливими. Їх ігнорування веде до негативних сценаріїв, що вплинуть на енергоефективність мобільного пристрою та користувацький досвід. Саме тому було розроблено алгоритм запуску сервісу (див. рисунок 2.6) та алгоритм його зупинки (див. рисунок 2.7).

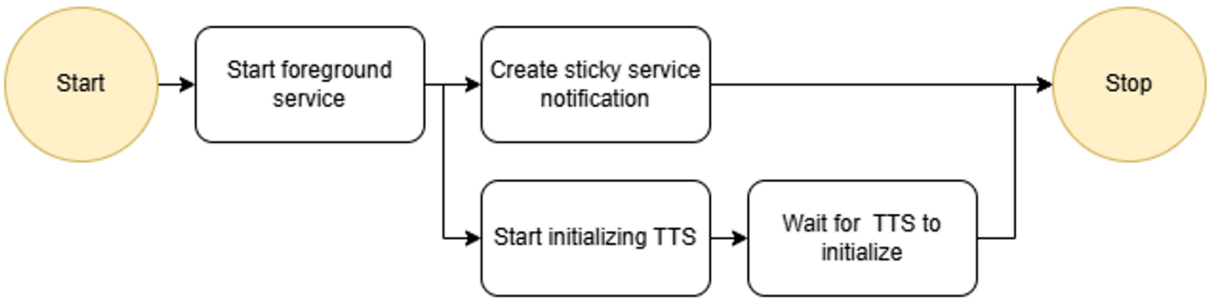


Рис. 2.6 Блок-схема запуску сервісу

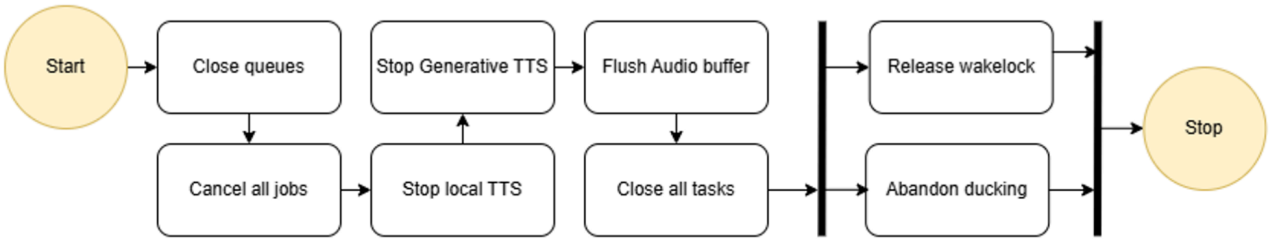


Рис. 2.7 Блок-схема зупинки сервісу

Алгоритм запуску буде спрацьовувати у двох випадках: коли користувач вручну запускає сервіс, а також коли він уже був активний і телефон перезавантажується. Якщо з першим сценарієм усе зрозуміло, то другий це автоматичне відновлення ОС андроїд після циклу вимкнення-увімкнення. Розберемо детальніше сам алгоритм.

Спочатку запускається увесь фоновий сервіс. Згодом починається ініціалізація локального TTS. В цей же час створюється постійне сповіщення для користувача, аби він був усвідомлений, що сервіс наразі працює. При повній ініціалізації локального TTS, сервіс може приступати до роботи зі сповіщеннями. Варто зазначити, що незалежно від обраного користувачем синтезу, необхідно забезпечити постійну готовність локального, оскільки перемикання між типами можуть призводити до повторної ініціалізації та нестабільної його роботи. Тому, локальний TTS ініціалізується один раз і завжди напоготові.

Щодо алгоритму вимкнення, то він спрацьовує лише коли користувач зупиняє сервіс через додаток, або через менеджер телефону. Тут важливо зупинити усі процеси які виконувала система, ще й так, аби при запуску нічого не зламати.

На початку закриваються усі черги, аби не починати обробку нових сповіщень в процесі вимкнення. Після цього зупиняються усі роботи, що в свою чергу припиняє роботу з наявними даними. Тут же зупиняється і стрімінг, аби перестати споживати інтернет трафік та завантажувати аудіо. Потім зупиняється та закривається локальний TTS і його озвучення. Згодом повторюється те ж саме, але вже для генеративного TTS. Наступним кроком очищається аудіо буфер

генеративного TTS. Необхідно позбутись залишкових даних. Далі закриваються усі задачі, які були відкриті для озвучки змістів. Закривши задачі перестає приглушатись звук і процесору дозволяється йти у сон. Останній неявний крок, який виконується автоматично ОС Android – це знищення сервісу, його посилань та очистка пам’яті.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОПИС РОБОТИ СИСТЕМИ

У даному розділі наведено опис програмної реалізації розробленого застосунку та принципів його роботи. Розглянуто структуру системи, особливості реалізації модулів, а також механізми обробки сповіщень, синтезу мовлення та взаємодії з користувачем.

3.1 Загальна структура системи

Відповідно до спроєктованої архітектури, система складається з декількох виконавчих компонентів та прошарку користувацького інтерфейсу. Забезпечена гнучкість, модульність та низька залежність окремих модулів додатка. Детальну структуру наведено на рисунку 3.1.

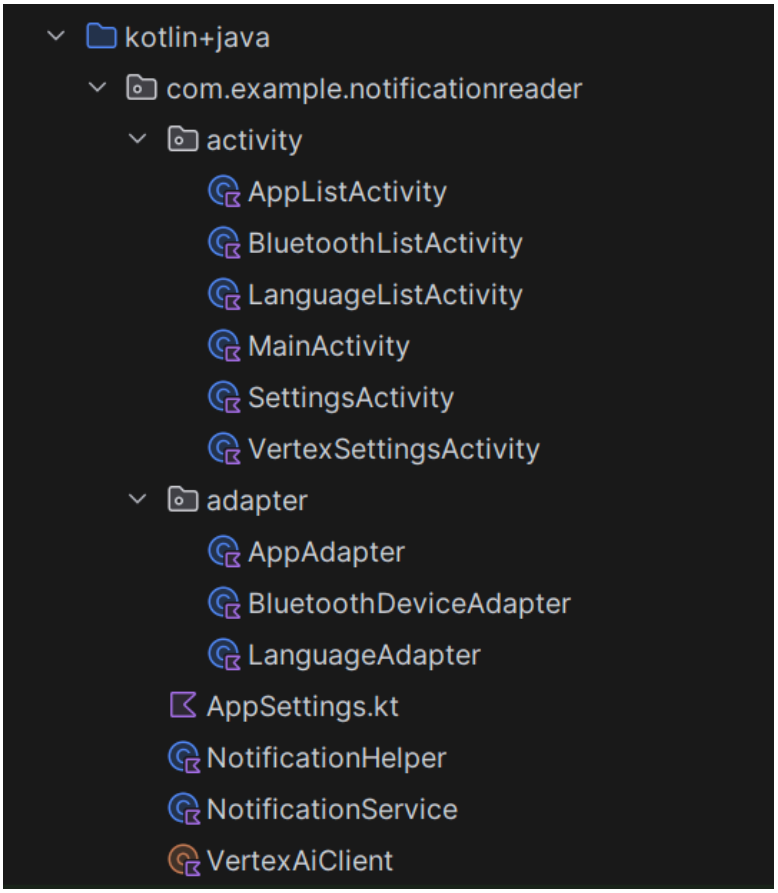


Рис. 3.1 Структура системи

Рівень представлення, або ж рівень користувацького інтерфейсу реалізований через Activity та їх Adapter. Як і планувалося, він взаємодіє із сервісом, налаштуваннями та менеджером сповіщень. Нижче наведено стислий опис основних елементів інтерфейсу:

- MainActivity – це головний екран, який зустрічає користувача. Дозволяє запустити сервіс, протестувати його і містить перехід до налаштувань;
- SettingsActivity – це екран який містить усі наявні в додатку налаштування. Дає можливість змінити конфігурацію в межах одного екрана наскільки це можливо і містить переходи до специфічних налаштувань;
- AppListActivity – це екран керування додатками, чиї сповіщення можуть бути озвучені. Використовує для реалізації списку AppAdapter;
- BluetoothListActivity – це екран для вибору пристроїв при підключенні до яких дозволяється озвучення. Використовує BluetoothDeviceAdapter для зв’язку з апаратним рівнем;
- LanguageListActivity – це екран налаштування де користувач може підібрати мови, що будуть використовуватись для озвучення. Робота зі списком мов забезпечується через LanguageAdapter;
- VertexSettingsActivity – це екран зі специфічними для генеративного TTS налаштуваннями.

Рівень логіки цілком і повністю відповідає раніше спроєктованій архітектурі й забезпечують функціональність системи. Ключовими компонентами цього рівня виступають:

- AppSettings – це компонент управління конфігурацією. Відповідає за постійне збереження усіх налаштувань додатка;
- NotificationService – це центральний вузол, ядро додатка. Перехоплює сповіщення, обробляє його та озвучує;
- NotificationHelper – це допоміжний менеджер, який інкапсулює логіку взаємодії із сповіщеннями. Необхідний для створення власних сповіщень;

- VertexAiClient – це модуль-клієнт, що забезпечує взаємодію з хмарним сервісом VertexAI. Він відправляє запит та завантажує аудіо озвучення від генеративного TTS.

3.2 Рівень представлення

Рівень представлення є важливою ланкою у взаємодії між користувачем та функціоналом додатка. Тому, у межах даного підрозділу буде проведено детальний огляд його компонентів.

3.2.1 Огляд компоненту MainActivity

MainActivity – це головний екран який зустрічає користувача. Отож, він повинен надати миттєвий доступ до основного функціоналу та інструмент для перевірки. Від цього сильно залежить перше враження та чи буде людина використовувати додаток взагалі. Зовнішній вигляд розробленого інтерфейсу представлено на рисунку 3.2.

По центру розміщується велика кнопка увімкнення та вимкнення сервісу. До того ж вона одразу відображає статус. Нижче розміщена картка для тестування додатку. При натисканні на “Test” створюється тестове сповіщення, зміст якого залежить від введеного. Якщо не ввести заголовок і текст для тестового сповіщення, то замість них автоматично підставиться “Title” та “Notification text”. Справа зверху розміщена кнопка яка переводить до екрана налаштувань SettingsActivity.

Варто зазначити, що для активації сервісу має бути досягнута низка обов'язкових умов, інакше додаток буде допомагати користувачеві виконати їх:

- додаток повинен мати особливий доступ для прослуховування сповіщень. при відсутності доступу, користувач переспрямовується до відповідного розділу налаштувань;
- для роботи додаток повинен мати доступ на створення, публікацію сповіщень. Інакше відкривається конфігурація цього параметру;

– при використанні локального TTS, повинен бути встановлений пакет Google TTS. В іншому випадку, відбудеться пересилання на сторінку синтезатора в Play Market.

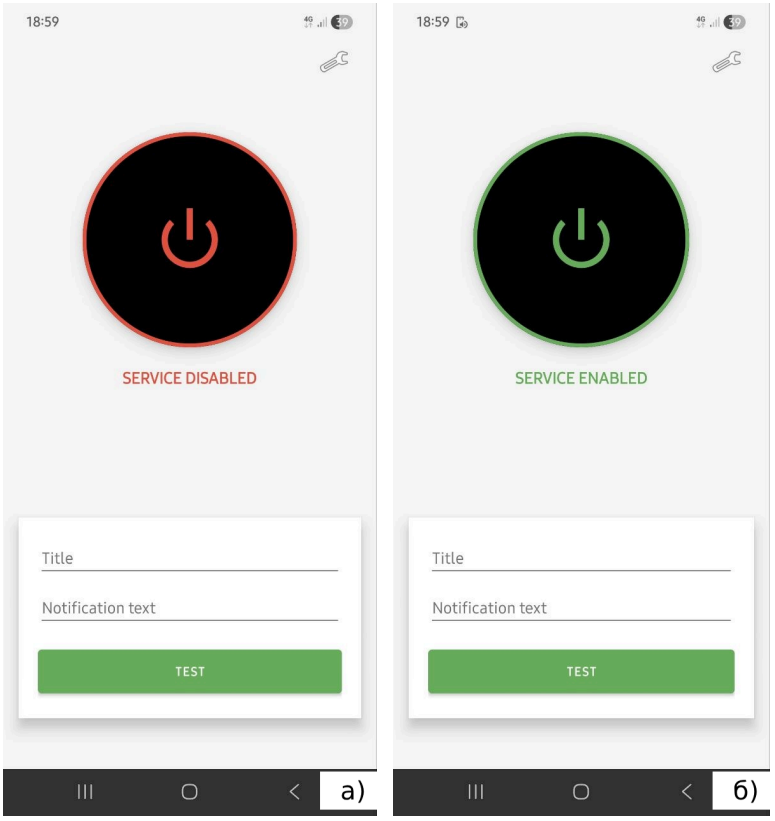


Рис. 3.2 Інтерфейс MainActivity при вимкненому сервісі (а) та при увімкненому сервісі (б)

3.2.2 Огляд компоненту SettingsActivity

SettingsActivity – це ключовий екран для конфігурації додатку. Для логічного розділення налаштувань та кастомізації було вирішено розбити налаштування на категорії. Перша категорія – це загальні налаштування фільтрів та мов. Друга категорія – це конфігурація голосу. Переглянути зовнішній вигляд екрана можна на рисунку 3.3 (розгорнутий скриншот).

Усі параметри конфігурації були зібрані у зручний список у межах власних категорії. В кінці категорії голосу були розміщені повзунки для зміни висоти голосу та швидкості мовлення, а також кнопка, яка в разі чого допоможе повернути значення за замовчуванням. Для всіх опцій які можна вмикати та вимикати, прикріплені перемикачі, а якщо доступна глибша конфігурація, то

перемикач відділений роздільником. Натискання зліва від тумблера, а за його відсутності, на елемент, переводить на окремий екран, що дозволяє керувати цим налаштуванням:

- Allowed Applications – AppListActivity;
- Custom Languages – LanguageListActivity;
- Bluetooth Only – BluetoothListActivity;
- Vertex AI Voice – VertexSettingsActivity
- Install Voice Data – com.android.settings.TTS_SETTINGS.

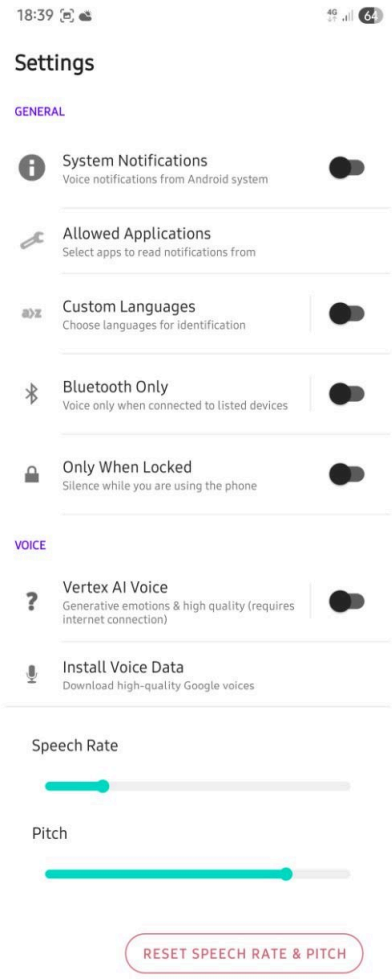


Рис. 3.3 Інтерфейс SettingsActivity

Варто звернути увагу на те, що для активації опції “Bluetooth Only” користувач повинен надати додатку дозвіл на доступ до списку пристроїв поблизу. Для цього викликається системне меню запиту дозволів Android (System Permission Dialog).

Починаючи з Android 11.0, після двох відмов у наданні дозволу операційна система блокує подальший виклик системного вікна запиту [20]. Для таких випадків було реалізовано альтернативне, власне діалогове вікно. Воно надає користувачу можливість перейти до відповідного розділу налаштувань та вручну надати доступ.

На старіших версіях Android блокування системного запиту може відбутися після першої відмови, якщо користувач обере опцію “Don’t ask again”. У такому випадку також використовується альтернативне рішення. Візуалізацію системного меню та розробленого діалогового вікна наведено на рисунку 3.4.

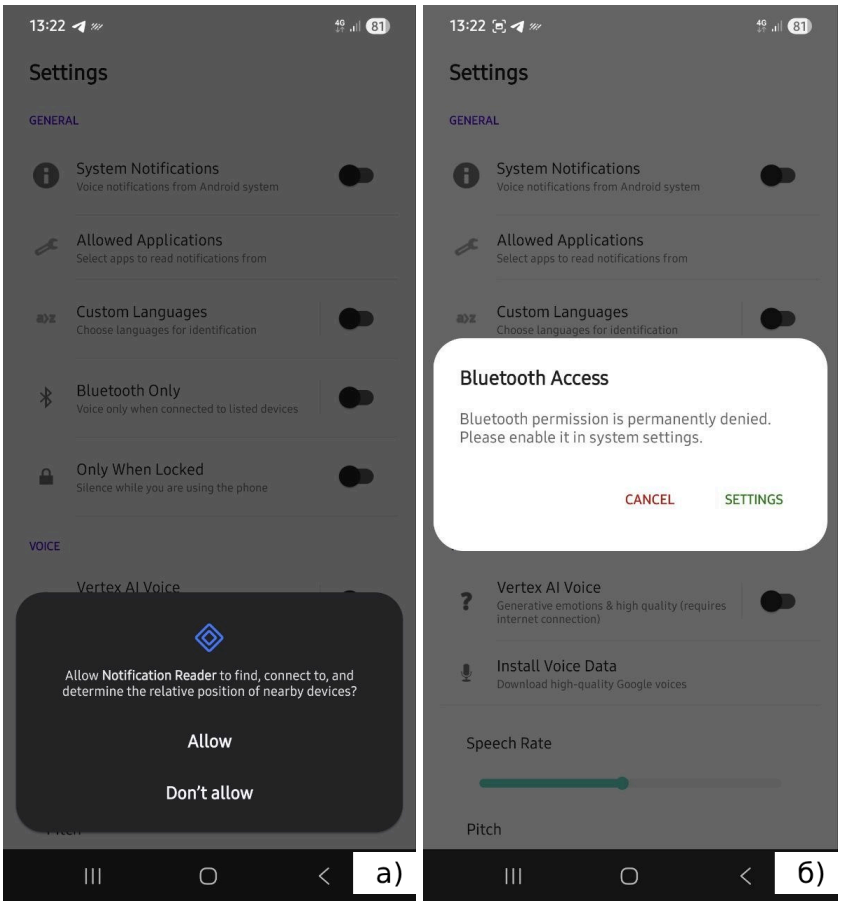


Рис. 3.4 Порівняння системного меню (а) та власного діалогового вікна (б)

Налаштування голосу є релевантними виключно для локального TTS, тоді як конфігурація генеративного здійснюється суто на окремо виділеному екрані. Для запобігання неоднозначності у впливі налаштувань, вибір генеративної моделі робить параметри локального TTS неактивними. Для порівняння станів

інтерфейсу наведена ілюстрація вимкненого та увімкненого генеративного TTS на рисунку 3.5.

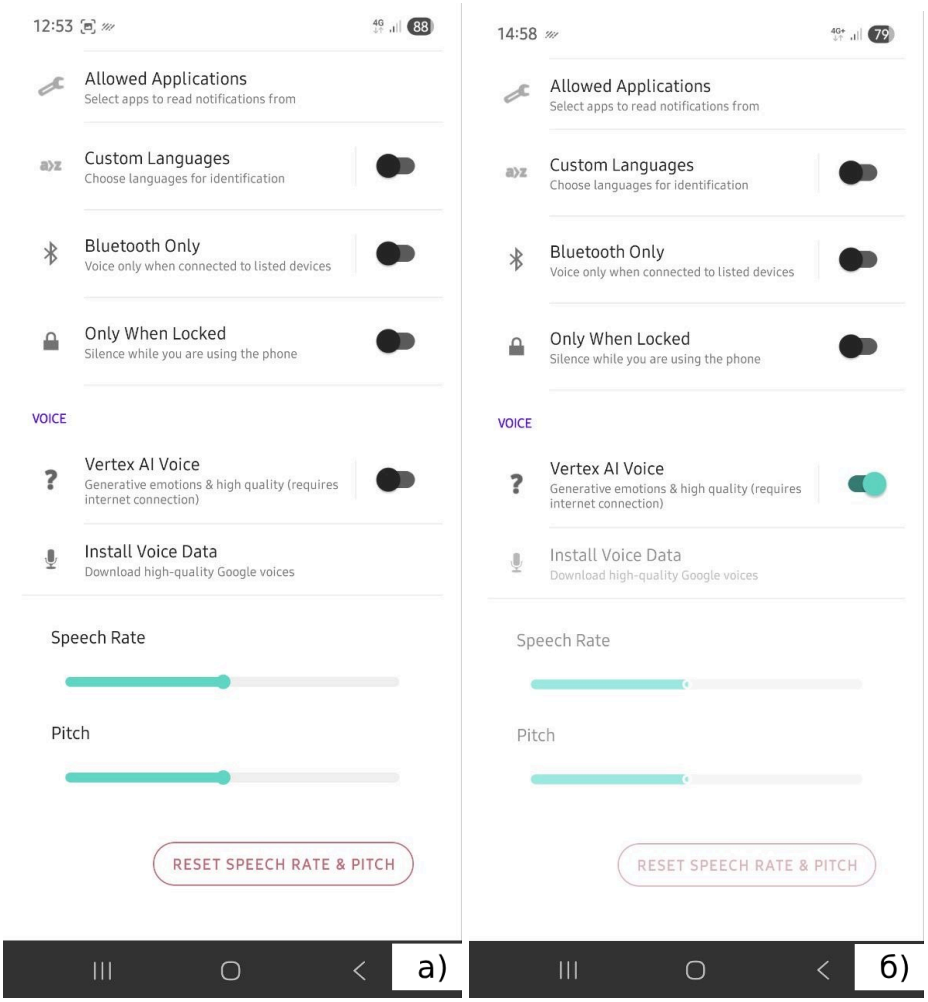


Рис. 3.5 Порівняння вимкненого (а) й увімкненого (б) генеративного TTS

Опція Install Voice Data перенаправляє користувача в системні налаштування Google TTS, де він може для кожної мови обрати голос який йому подобається і відразу ж там його прослухати. Так користувач зможе чути однакові голоса в різних додатках що використовують локальний синтезатор, уникаючи дисонансу.

3.2.3 Огляд компоненту AppListActivity

AppListActivity – це екран для налаштувань додатків чиї сповіщення будуть озвучуватись. Він містить великий список усіх додатків встановлених

користувачем і поле для пошуку. Кожен елемент списку має перемикач для керування і водночас відображення статусу.

Для оптимізації розміру і якості списку було застосовано фільтрацію: присутні лише не системні додатки, що можуть надсилати сповіщення і які користувач взагалі може запустити. Єдиним винятком серед застосунків є “Погода”, оскільки це окремий додаток, що був інтегрований в систему заради дозволів та енергоефективності, до того ж його сповіщення мають високу цінність для користувача. Інші важливі програми, як-от “Календар”, “Годинник” чи “Карти” потрапляють до списку автоматично, не вимагаючи специфічної обробки, оскільки оновлюються через магазин і завдяки цьому можуть бути розпізнані.

Як результат фільтрації, на пристрої Samsung S24 вдалося скоротити список з 184 додатків до 85, а на Samsung Galaxy A41 – з 147 до 70. Як результат, вийшло зменшити кількість елементів приблизно вдвічі, що пропорційно підвищує ефективність навігації. Для ілюстрації інтерфейсу та часткового порівняння списків із системним наведено рисунок 3.6.

Пошук являє собою динамічну фільтрацію . Для її реалізації зроблено два списки: повний список (після фільтрації додатків) і відфільтрований список. Повний список потрібен лише для формування відфільтрованого списка, який користувач і бачить. Нижче представлено код функції фільтрації:

```
fun filter(query: String) {
    filteredApps = if (query.isEmpty()) {
        allApps
    } else {
        allApps.filter { app ->
            val label = pm.getApplicationLabel(app).toString()
            label.contains(query, ignoreCase = true)
        }
    }
    notifyDataSetChanged()
}
```

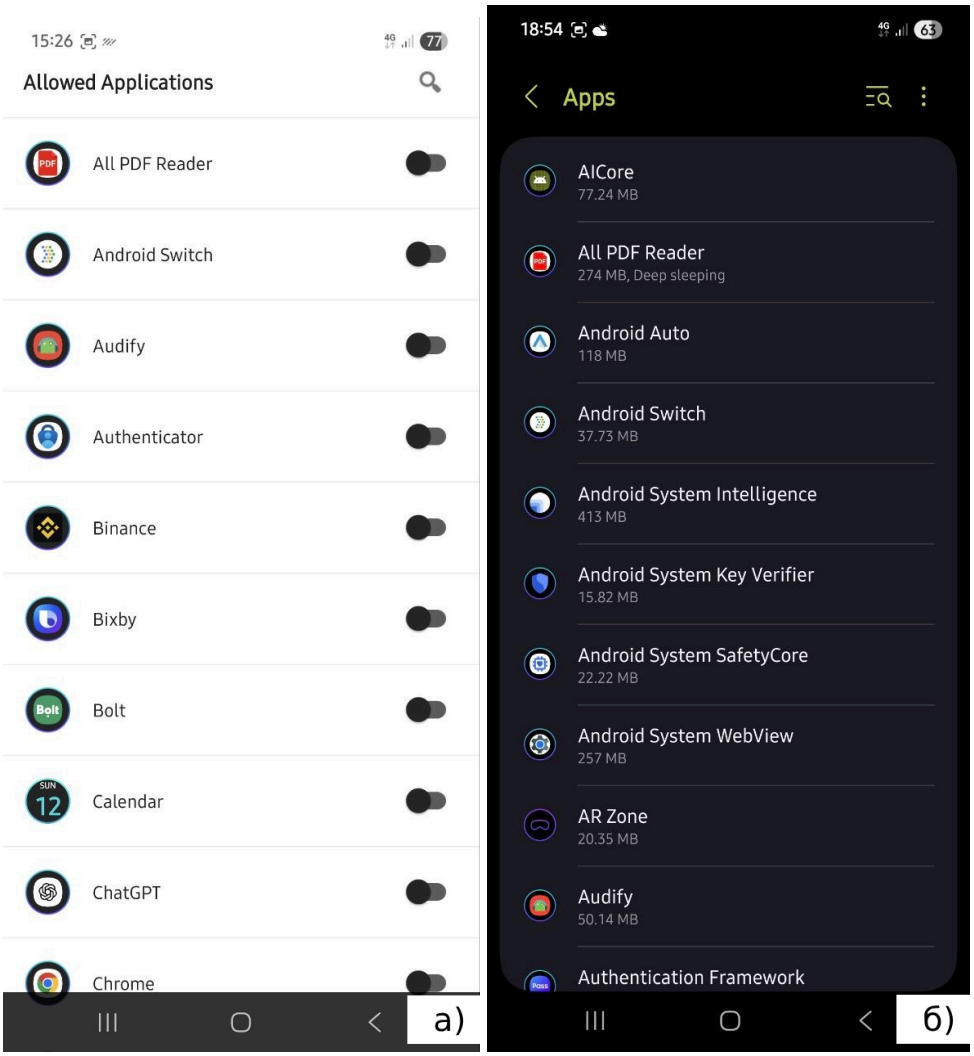


Рис. 3.6 Порівняння інтерфейсу AppListActivity (а) із системним списком додатків (б)

Адаптований TextWatcher відстежує зміни у полі ввода і при їх появі викликає фільтрацію. Фільтрація відбувається по наявності символів у назві додатка, розташування неважливе. Якщо ввести умовний “gram”, то в результаті в списку будуть “Telegram”, “Instagram” та інші додатки, що містять цю частку.

3.2.4 Огляд компоненту LanguageListActivity

LanguageListActivity – це екран для конфігурації мов, що будуть використовуватись для озвучки. Дане налаштування необхідне у випадках, коли необхідно детальніше уточнити мови, або ж використовувати мови що не встановленні за замовчуванням у системі. Інтерфейс екрана зображено на рисунку 3.8.

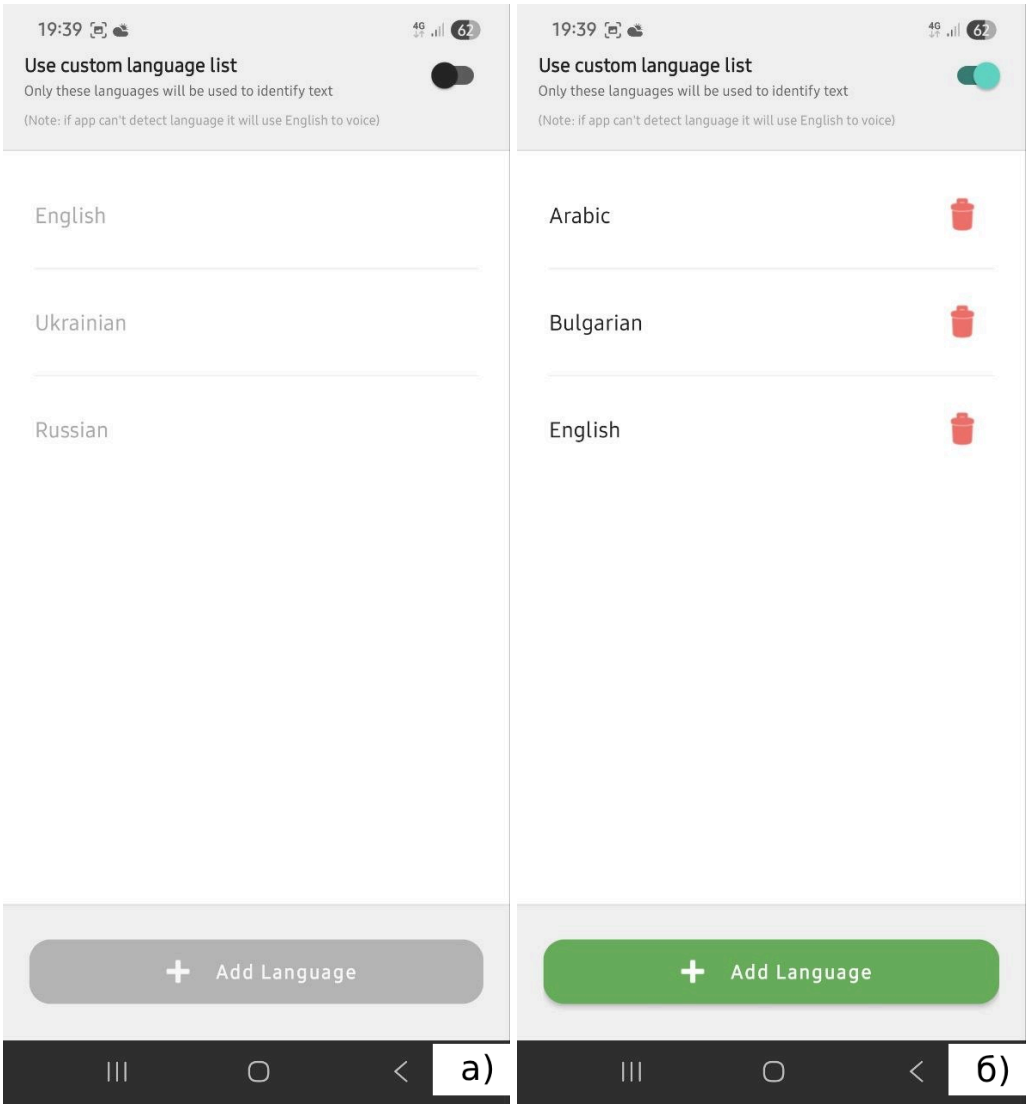


Рис. 3.8 Інтерфейс LanguageListActivity у вимкненому стані (а) та у ввімкненому стані (б)

Коли це налаштування вимкнене, то відображається і використовується список системних мов через Resources.getSystem().configuration.locales. Його можна змінити лише через системні налаштування.

При ввімкненні налаштування відображається і використовується список мов, який користувач може налаштувати безпосередньо всередині додатка. Зверху наведені додаткові підказки по принципу роботи цієї конфігурації.

Керування списком здійснюється за допомогою відповідних елементів. Кнопка з іконкою смітника прибирає мову зі списка. Кнопка “Add Languages” відкриває діалогове вікно зі списком доступних локалей (див. рисунок 3.9). Натискання на мову додає її до переліку використовуваних.

Рис. 3.9 Зовнішній вигляд діалогового вікна з мовами

Особливістю переліку є механізм фільтрації мов. У списку відображаються лише ті локалі, які підтримуються TTS. Оскільки основний сервіс може бути вимкненим на момент налаштування, то у LanguageListActivity реалізовано власну

ініціалізацію TTS. Такий підхід дозволяє виконувати запити до двигуна напряму, для перевірки сумісності мов незалежно від стану сервісу додатка.

3.2.5 Огляд компоненту BluetoothListActivity

BluetoothListActivity – це екран для конфігурації пристроїв лише при наявності з'єднання з якими будуть озвучуватись сповіщення. Інтерфейс компоненту зображено на рисунку 3.10.

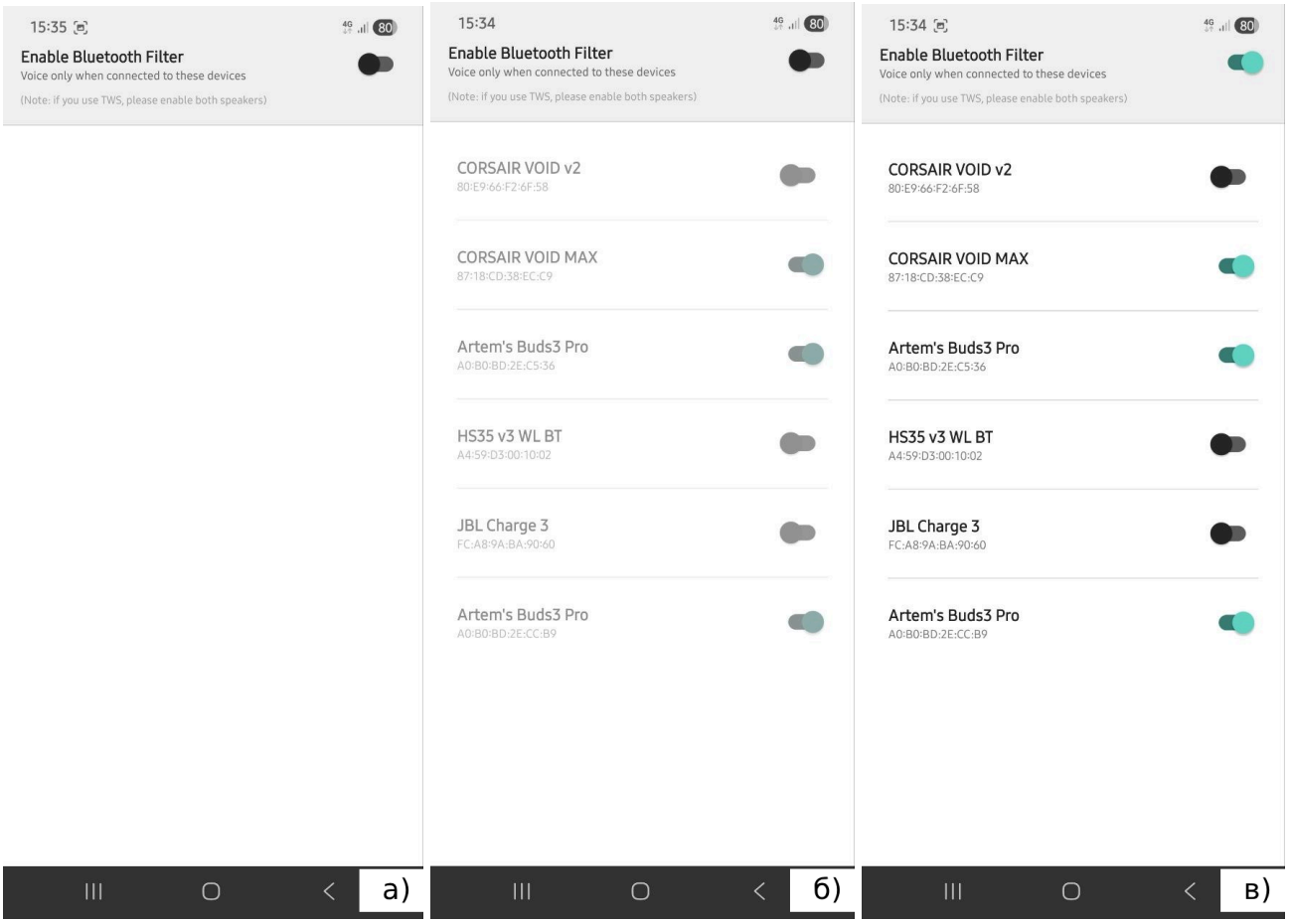


Рис. 3.10 Інтерфейс BluetoothListActivity при відсутності доступу (а), у вимкненому стані (б) та у ввімкненому стані (в)

Як і в SettingsActivity, тут реалізований запит на доступ до списку пристроїв поблизу. Відповідно, аналог на випадок відмов також присутній. В коді це 100% копія того що є в SettingsActivity.

При відсутності доступу, список буде порожнім і відповідно подальша конфігурація неможлива. Якщо дозвіл наданий, але сам параметр вимкнений, то

користувачу буде відображатись неактивний список пристроїв та їх налаштування. При ввімкненні параметру, список стає активним і доступним для редагування. Оскільки навушники-вкладиші можуть ідентифікуватись як два окремих Bluetooth-пристрої, то була додана підказка, що вмикати потрібно обидва девайса.

3.2.6 Огляд компоненту VertexSettingsActivity

VertexSettingsActivity – це екран для конфігурації генеративного TTS. Він дозволяє обрати голос та налаштувати промпт щодо стилю мовлення. Візуалізацію даного екрана наведено на рисунку 3.11.

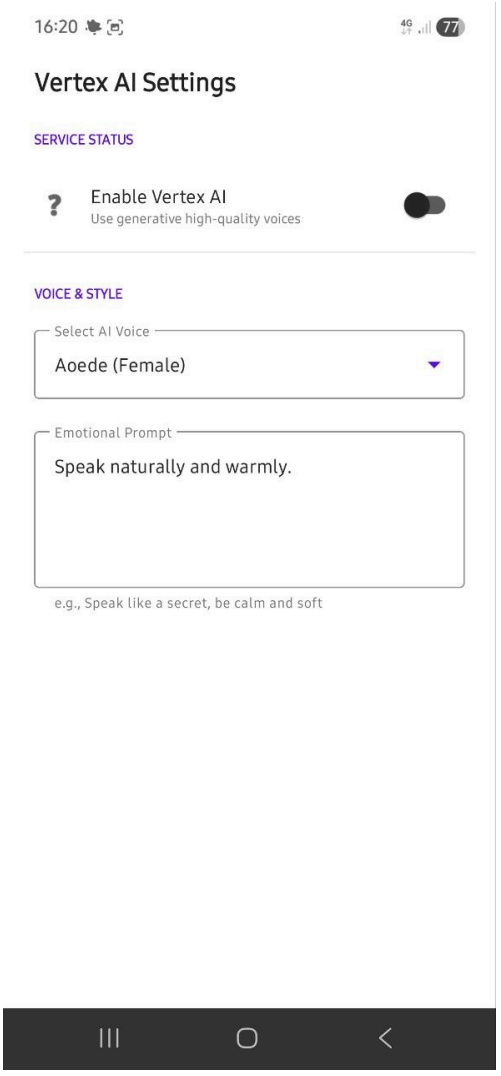


Рис. 3.11 Інтерфейс VertexSettingsActivity

Процес ознайомлення і конфігурації може зайняти багато часу в користувача. Тому на відміну від попередніх екранів було вирішено не блокувати налаштування за увімкненням параметру.

На вибір користувачу доступні 30 голосів (17 чоловічих та 13 жіночих). Усі вони зібрані в одному елементі, при натисканні на який відкривається повний список наявних опцій.

Промпт можна задавати через поле вводу. Воно має обмеження у 500 символів, адже нам необхідно забезпечити низьку затримку. Чим довший промпт, тим важче відправити запит і тим більше часу моделі потрібно на його обґрунтування та опрацювання. Стосовно інших обмежень, то їх нема, користувач може вводити абсолютно будь-що і експериментувати із цим налаштуванням.

Для зберігання змін у полі в змінну, використаний TextWatcher. Аби не оновлювати значення при введені кожного символу, він відслідковує зміни і перевіряє чи пройшла секунда з моменту останнього вводу. Таким чином заощаджуються ресурси процесора.

3.3 Рівень логіки

Рівень логіки відповідає за реалізацію функціоналу всередині додатка. У межах даного підрозділу буде проведено огляд компонентів та модулів, що забезпечують обробку даних.

3.3.1 Огляд компоненту AppSettings

AppSettings – це компонент, що зберігає усі налаштування, а також надає певні допоміжні методи: фільтрація без перевірки на дублікат та перевірка чи дозволено вмикати сервіс.

Розпочнемо огляд зі зберігаємих налаштувань, за що вони відповідають у системі та де і як їх можна змінити. Усі параметри наведені в таблиці 3.1.

Таблиця 3.1

Перелік зберігаємих налаштувань

Назва змінної	Тип	Призначення	Елемент керування	За замовчуванням
IsServiceEnabled	Boolean	Увімкнення та зупинка сервісу озвучення	Велика кнопка-перемикач у MainActivity	False
readSystem	Boolean	Налаштування фільтру озвучення системних сповіщень	Перемикач System Notifications у SettingsActivity	False (фільтр ввімкнено)
useCustomLanguages	Boolean	Перемикання використовуваних мов між системними та користувацькими	Перемикач Custom Languages у SettingsActivity, LanguageListActivtiy	False
customLanguages	Set<String>	Список використовуваних користувацьких мов	Кнопка зі смітником для видалення та кнопка “Add Languages” для додавання у LanguageListActivity	{“en”}
onlyWhenLocked	Boolean	Налаштування фільтру озвучення при вимкненому екрані	Перемикач Only When Locked у SettingsActivity	False
onlyBluetooth	Boolean	Налаштування фільтру озвучення лише при підключеному Bluetooth-пристрою	Перемикач Bluetooth Only та Bluetooth Filter у SettingsActivity й BluetoothListActivity відповідно	False
allowedBluetoothDevices	Set<String>	Налаштування Bluetooth-пристроїв для озвучення	Перемикачі біля пристроїв у BluetoothListActivity	{}
useVertexAI	Boolean	Перемикання між локальним та генеративним TTS	Велика кнопка-перемикач у MainActivity	False (локальний)

Продовження таблиці 3.1

Назва змінної	Тип	Призначення	Елемент керування	За замовчуванням
vertexPrompt	String	Промпт щодо стилю мовлення генеративного TTS	Поле вводу Emotional prompt у VertexSettingsActivity	Speak naturally and warmly.
vertexSpeaker	String	Голос генеративного TTS	Елемент Select AI Voice у VertexSettingsActivity	Aoede
speechRate	Float	Налаштування швидкості мовлення локально TTS	Повзунок Speech Rate у SettingsActivity	1f
speechPitch	Float	Налаштування висоти голосу локального TTS	Повзунок Pitch у SettingsActivity	1f

Даний компонент також допомагає у фільтрації і відповідає за декларацію конвеєра усіх фільтрів, окрім дублікатів. Допоміжна функція наведена у додатку Б.

Для конвеєра фільтрів був використаний паттерн pipeline. Дане рішення краще за чергу з if, адже у нас багато фільтрів і складна логіка. Так буде простіше тестувати і масштабувати функцію при потребі.

На початку ініціалізуються сервіси необхідні для деяких фільтрів, аби при єдиній ініціалізації об’єкта конвеєра не викликати їх багато разів всередині фільтрів і не завантажувати систему надлишковими процесами.

Щодо фільтрів, то було вирішено розмістити їх у порядку від найдешевших до найдорожчих по використанню ресурсів. Повернення true всередині фільтра означає, що його успішно пройдено; false відповідно навпаки – не пройдено.

Фільтр пустого тексту. Отримується контентний текст сповіщення і повертається зворотне значення від isNullOrBlank. Дана функція повертає true, коли значення пусте, null, або складається з одних пропусків;

Фільтр звітів. Перевіряється чи містить сповіщення хоч якісь прапорці, якщо так, то перевіряється прапорець FLAG_GROUP_SUMMARY. Прапорець це

число у якого біт встановлено 1 тому для отримання булевого значення порівнюється з 0 і записується у isSummary. В кінці повертається зворотне від isSummary значення;

Фільтр під час дзвінків. Повертається зворотне значення від логічної суми режимів. MODE_IN_CALL – це режим звичайних дзвінків. MODE_IN_COMMUNICATIONS – це режим дзвінків через інтернет, зокрема через додатки на кшталт Viber, Telegram, Discord, Google Meet;

Фільтр під час використання телефону . Спочатку перевіряється чи ввімкнене налаштування, якщо воно вимкнене то повертається true і здійснюється вихід з цього фільтра. В іншому випадку повертається стан екрану (ввімкнений чи вимкнений) або повертається стан замку (розблокований чи заблокований), для проходження достатньо одного true;

Фільтр системних сповіщень та сповіщень додатків. Здійснюється перевірка білого списку дозволених користувачем додатків, якщо відправник сповіщення там є, то повертається true. В іншому випадку перевіряється чи це системний додаток через прапорці і пакети. Якщо системний, то повертаємо булеве значення налаштування фільтра readSystem;

Фільтр Bluetooth-пристроїв. Перевіряється чи увімкнене налаштування. Якщо ввімкнене, то отримується список спарованих пристроїв, Наступним кроком перевіряється чи є такі у списку дозволених і чи підключений цей пристрій. Підключення перевіряється через рефлексію метода isConnected.

Загалом для перевірки підключення краще імплементувати BluetoothProfile.ServiceListener або BroadcastReceiver. Проблема в тому, що їх імплементація вимагає багато коду, асинхронності, а ще знижує енергоефективність, тому було обрано простіше рішення з рефлексією, яке позбавлене цих недоліків.

3.3.2 Огляд модуля VertexAiClient

VertexAiClient – це модуль, що відповідає за взаємодію з хмарним сервісом. Якщо точніше, то він авторизується, формує запит і обробляє потік вихідних даних.

Для роботи з необхідним сервісом, спочатку необхідно зареєструватись на Google Cloud платформі та підключити необхідний Vertex AI API [21] [22] (див. рисунок 3.12). Наступним кроком потрібно створити сервісний акаунт та налаштувати його роль як користувача, в даному випадку це VertexAIUser. Також вказуємо package додатку та id, аби використання API було можливим лише в нашому додатку. Якщо цього не зробити, будь-хто може вкрасти доступ до API та використовувати його в своїх цілях. Згодом, генеруємо JSON для авторизації і поміщаємо його в файли проєкту.

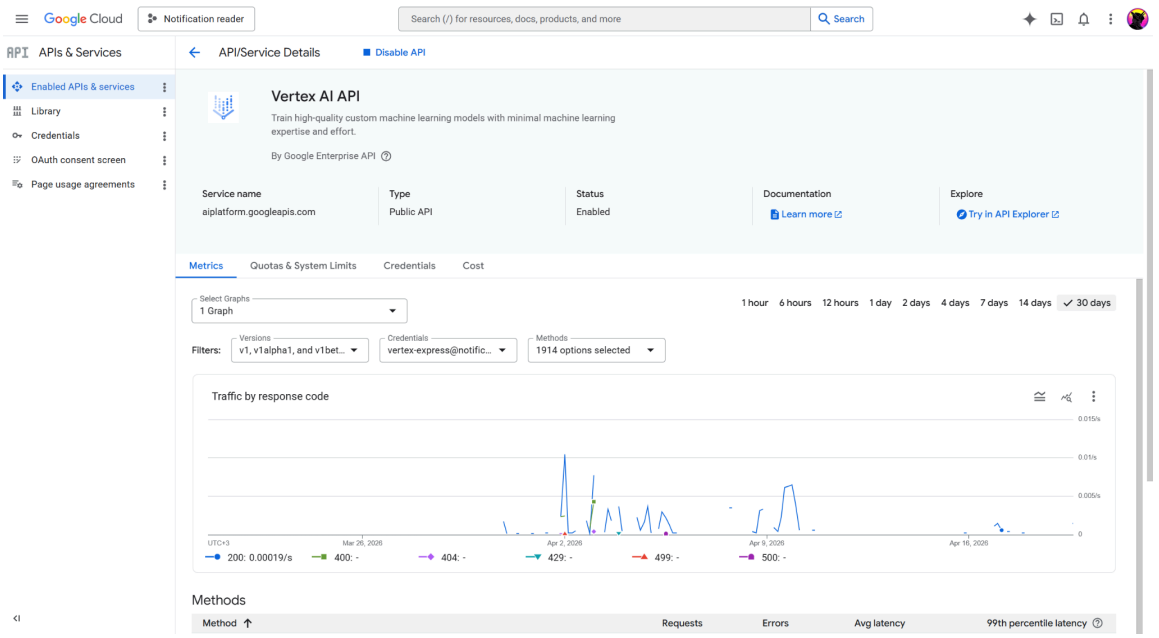


Рис. 3.12 Vertex AI API

Для роботи використовуємо об’єкт, він оголошує клас і одразу створює його єдиний екземпляр (патерн Singleton). Для підключення вказуємо ідентифікатор проєкту, регіон, модель і створюємо функцію для авторизації getAccessToken() (див. додаток Б). Регіон в даному випадку це “europe-central2”, адже він найближчий по локації до України і має доступ до моделі Gemini 2.5 Flash TTS.

Створення запиту відбувається через функцію fetchAudioStreaming() (див. додаток Б).

Для реалізації стрімінгу використовується потік для введення і виведення Dispatchers.IO та вказується лямбда-функція onBytesReceived: (ByteArray) -> Unit для передачі часток даних.

Далі формується структура запиту. Попередньо визначена функція допомагає отримати токен для авторизації, якщо наявний не актуальний. Далі вказується адреса для запиту, де є локація, ідентифікатор проєкту і модель. В кінці обов’язково вказується streamGenerateContent, аби модель повертала дані саме частками в процесі генерації.

В змісті запиту вказано роль користувача та текст. Текст розбитий на інструкції, стиль і відповідно сам текст. В інструкціях лише вказівка озвучувати текст один раз, бо при тестуванні було виявлено, що модель може працювати нестабільно і повторювати слова. Стиль задається користувачем через промпт у налаштуваннях. Текст відповідно береться зі сповіщень.

Наступним кроком вказана конфігурація для моделі. Тут визначається, що формат відповіді має бути саме аудіо, адже модель може повертати різноманітні види даних. Також тут налаштовується голос і мова.

Після того як запит сформований створюється по ньому HTTP-запит через Request.Builder та відправляється за допомогою client.newCall(request).execute. Одразу після відправки запиту перевіряється чи немає ніяких помилок.

Якщо ж все добре отримується потік байтів і створюється для нього читач JsonReader, який потрібен для перетворення сирих байтів у читабельний формат UTF-8. Для цих байтів створюється масив всередині якого викликається цикл while із функцією parseResponseObjectStreaming() для парсинга (див. додаток Б).

Відповідь API Vertex AI містить складну вкладену структуру, тому для роботи з даними потрібен потоковий парсер. Написана функція використовує низькорівневий ручний парсинг, який є швидким і максимально ефективним для пам’яті. Вона спускається по дереву JSON, шукає data, декодує та повертає через раніше оголошену лямбда-функцію. Аналогічні рішення як GSON чи бібліотека

Moshi, або ж Jackson мають недоліки у порівнянні. GSON потребує чекати повного завантаження даних, тоді як бібліотеки додають ваги додатку і зайвих залежностей.

3.3.3 Огляд компоненту NotificationHelper

NotificationHelper – це компонент що відповідає за створення сповіщень. Він необхідний для тестування додатку, а також відображення статусу роботи сервісу поза межами системи.

Починаючи з Android 8.0, сповіщення не можна просто відправити на телефон, їх необхідно надсилати за каналами. Тому спершу ніж слати сповіщення, створюються канали за допомогою відповідної функції createChannels() [23] (див. додаток Б).

Для сервісу створюється канал з низькою важливістю і уточнюється, що звуку при надсиланні не повинно бути, тоді як для тестових сповіщень – просто з середньою важливістю.

Після того як були створені канали, уже можна створювати сповіщення використовуючи відповідні функції createServiceNotification() та sendTestNotification() (див. додаток Б).

Створення сповіщення у сервісному каналі має наступні особливості:

- setContentIntent() дозволяє при натисканні на сповіщення переходити на головний екран додатку;
- setOngoing(true) і setAutoCancel(false) – це пара яка забезпечує живучість сповіщення. Так його не можна змахнути;
- setLocalOnly(true) гарантує, що сповіщення не буде дублюватись на смарт годиннику;
- setCategory(Notification.CATEGORY_SERVICE) позначає сповіщення як службове, що дозволяє ОС краще керувати ресурсами;
- setForegroundServiceBehavior(). Android може не показувати сповіщення сервісу протягом певного періоду, тому ми вказуємо, що сповіщення повинно відображатись одразу;

- setVisibility(NotificationCompat.VISIBILITY_PUBLIC) – це дозвіл на показ на екрані блокування.

Щодо тестового сповіщення, то як таких особливостей у нього нема, за винятком що створюється воно по іншому на відміну від сервісного. Сервісне – створюється з допомогою startForeground(), завдяки чому його цикл життя тісно пов’язаний з самим сервісом. По завершенню сервісу, зникне і сповіщення. Тоді як тестове створюється через notify(), що є стандартним способом створення більшості сповіщень в системі.

3.3.4 Огляд модуля NotificationService

NotificationService – це головний модуль, контролер, або ж оркестратор системи. Він відповідає за фоновий сервіс, що перехоплює сповіщення та озвучує їх.

Спочатку, коли користувач запускає сервіс, він ініціалізується в onCreate(), по готовності викликається onInit(), а при оновленні даних системи викликається onStartCommand() (див. додаток Б).

В onCreate() створюється об’єкт локального TTS, визначаються налаштування роботи, створюється конвеєр фільтрів, а також створюється сповіщення, що відображає статус сервісу.

onInit() оновлює змінну статусу локального TTS, що знадобиться пізніше та запускає його прослуховувач, аби знати коли закінчиться мовлення.

onStartCommand() відповідає за оновлення сервісу. Дана функція викликається кожен раз коли користувач змінює налаштування, чим оновлює сервіс з новою конфігурацією. Даний метод також стосується і вимкнення сервісу, але замість оновлення, сервіс вбивається з допомогою requestUnbind(), stopForeground() та stopSelf(). Окремо існує і STOP_TTS команда, яка зупиняє синтезатор. Наразі вона існує лише як перший крок перед зупинкою сервісу. Логіка в MainActivity така, що при вимкненні, спочатку змінюється значення isServiceEnabled, потім надсилається STOP_TTS, а згодом – UPDATE_STATUS. В

майбутньому функціонал команди STOP_TTS можна буде розширити наприклад, щоб TTS замовкав при свайпі сповіщення, трусінні телефону чи іншій дії.

Наступним кроком після запуску сервісу, коли приходить сповіщення спрацьовує функція onNotificationPosted() (див. додаток Б), що дісталась у спадок від класу NotificationListenerService.

Дана функція виконує наступне:

а) валідація стану. Якщо сервіс якимось чином опинився вимкненим, або ж користувач не повинен мати змоги його увімкнути (не надані необхідні дозволи, або не встановлений TTS), то сервіс примусово вимикається. Якщо сповіщення null, або локальний синтезатор не готовий, то сповіщення пропускається. Далі ми фільтруємо сповіщення сервісу, аби не озвучувати увімкнення, адже користувач сам натиснув кнопку і відповідно уже усвідомлює статус сервіса;

б) фільтрація. Створюється і перевіряється булева змінна, що містить результат фільтрації у конвеєрі. Згодом окремо перевіряється чи є сповіщення дублікатом;

в) обробка. Створюється черга із сповіщень та запускається її асинхронна обробка.

Перевірка чи є сповіщення дублікатом здійснюється у відповідній функції isDuplicate() (див. додаток Б).

Даний метод використовує вбудовану бібліотеку Android LruCache. Вона дозволяє створювати готовий LinkedHashMap для зберігання об’єктів в тимчасовій пам’яті з певним лімітом. При виході за межі ліміту, найстаріший об’єкт видаляється.

Функція спершу отримує теперішній час і складає ключ із назви додатка, заголовку і тексту. Далі отримує записаний час за ключем і якщо він існує то звіряється різниця в часі. Якщо різниця менше 5000 мс то сповіщення є дублікатом. В іншому випадку, записується ключ і теперішній час в кеш. Також варто зазначити, що ліміт кешу було встановлено всього на 10 пар для збереження

ресурсів, адже для більшості користувачів отримати 11 сповіщень в проміжку 5 секунд де в кінці присутній дублікат є нереальним завданням.

Після того, як сповіщення потрапило до черги, вона обробляється через функцію processInputQueue() (див. додаток Б). Якщо даний метод уже запущений, то він просто опрацює нові дані в черзі. Поки він працює, він асинхронно обробляє сповіщення у черзі через цикл. Для кожного заголовку та тексту він створює задачу та запускає метод озвучення.

Задачі відповідають за отримання ресурсів процесора та фокусу на період озвучення. За допомогою функцій startSpeechTask() та endSpeechTask() (див. додаток Б), змінюється значення змінної activeTtsTasks, яка представляє собою лічильник. Аби лічильник змінювався завжди коректно, використовується synchronized(this), який гарантує невтручання інших потоків під час змін. Тобто startSpeechTask() та endSpeechTask() не можуть спрацювати одночасно і встають у чергу. Метод по закінченню задач в кінці перевіряє чи залишились активні задачі і якщо таких немає, то лише тоді звільняються ресурси та аудіофокус.

Методи які реалізують роботу з фокусом приглушають саме медіа контент через setUsage(AudioAttributes.USAGE_MEDIA). Якщо медіа з іншого додатку запуститься під час озвучення, то цей додаток забере аудіофокус собі.

Методи які реалізують роботу з ресурсами використовують synchronized(this), як додатковий захист від втручання. При отриманні ресурсів також встановлюється тривалість використання ресурсів, а саме 10 хв. У випаду, якщо додаток випадково заглючить, то по закінченню таймера, ОС автоматично поверне собі ресурси.

Після того як задача на озвучення була створена, запускається саме озвучення через identifyAndSpeak() (див. додаток Б). При помилці, функція закриє задачу.

Спочатку створюється, якщо можливо, очищена копія тексту за допомогою getCleanTextForML() (див. додаток Б). Даний метод видаляє слова за допомогою регулярного виразу. Функція прибирає лише посилання, проте легко масштабується.

Згодом для очищеного тексту виконується асинхронний виклик Google ML Kit. Варто зазначити, що ресурси смартфону обмежені, особливо для роботи з нейромережами, тому функції виконаються почергово. Таке явище відбувається через те що Google ML Kit, як і більшість API [24] в сервісах Google Play, реалізований за допомогою Task API і використовує лише один потік.

Після визначення мов, функція отримує список дозволених мов. Залежно від налаштувань це або системні мови, або користувацькі мови. Оскільки системні мови не відомі заздалегідь, то формується Set і заповнюється ними.

ML Kit результати збирає у список, де кожен елемент це мова та ймовірність. З таким форматом не зручно працювати, тому список трансформується у Map.

Згодом список дозволених та визначених мов пересікаються. Якщо перетин порожній, то обирається англійська. Інакше з цього перетину обирається мова що має найбільшу ймовірність.

Потім перевіряється чи доступна мова за допомогою LANG_AVAILABLE та LANG_COUNTRY_AVAILABLE. Якщо не доступна, то обирається стандартана системна мова.

Наступним кроком, текст нормалізується за допомогою функції cleanContent() (див. додаток Б). Дана функція працює подібно до getCleanTextForML(), але заміняє не на пусті рядки, а на конкретні слова, що відповідають обраній мові. Слова для різних мов зберігаються у відповідних ресурсах.

Коли текст уже повністю оброблений, він передається на озвучення TTS. Почнемо з гілки локального. Синтезатору встановлюється мова та налаштування (Pitch і SpeechRate). Після цього створюється унікальний маркер для озвучення на основі хешу. Даний етап важливий, адже TTS використовує ідентифікатори для моніторингу внутрішньої черги та зворотного зв’язку. Унікальність не важлива, адже порядковість визначається послідовністю викликів, проте може бути потрібна для дебагінгу і прозорішої роботи. Хеш дешево отримується і має відносну унікальність, чого має бути достатньо. Згодом TTS додає в чергу і

програє паузу. Якщо це заголовок, то вона триває 1 200 мс, якщо ні, то – 1 000 мс. Пауза має відповідний префікс, аби вона не змінювала лічильник задач. Потім в чергу додається сам текст.

По закінченню мовлення, прослуховувач setupTtsProgressListener(), (див. додаток Б) що створюється при ініціалізації TTS, відслідковує це, і залежно від того чи є префікс PAUSE, закриває задачу, на чому і закінчується гілка локального синтезатора.

Гілка генеративного TTS спрацьовує, коли увімкнене відповідне налаштування. Виконується метод addToAiQueue(), (див. додаток Б) який реалізує роботу з чергою аналогічно до onNotificationPosted, але з іншими об’єктами.

Також функція обробки черги ensureQueueIsRunning() (див. додаток Б) працює подібно до processInputQueue(), але вона складніша. На початку – перевірка чи не запущений уже метод. Згодом для кожного об’єкта через цикл запускається функція модуля VertexAiClient, яка відправляє запит і завантажує аудіо. Даний метод схожий на цикл, проте спрацьовує лише при отриманні нових даних з мережі. Через if на початку один раз створюється доріжка, пауза, а також запускається режим програвання. Пауза блокує потік і відповідно виконання функції, проте, вона не блокує завантаження, адже мережева карта пристрою продовжує працювати і накопичує дані у системному буфері. Тривалість паузи ідентична до локального TTS. Після виконання блоку if, в доріжку записуються нові дані завантажені функцією. Якщо даних достатньо, аудіоплеєр почне грати аудіо. Як тільки write записує у буфер доріжки всі дані, то спрацьовує блок finally який знищує доріжку і закінчує задачу. Варто зазначити, що запис у буфер неможливий без його спустошення, тобто програвання гарантоване, але завершення запису означає вихід з функції і вбивство доріжки. Проте, буфер достатньо малий, аби доріжка встигла дограти до того як знищиться.

Аудіодоріжка реалізується через функцію createStreamingAudioTrack() (див. додаток Б). Всередині визначається мінімально допустимий буфер за наступними параметрами: частота – 24 000 Гц, кількість каналів – один, формат – сирі аудіодані (PCM) 16 біт. Вони були визначені за конфігурацією даних, що повертає

модель, інакше, замість мовлення можна було б почути лише шум. Наступним кроком викликається низькорівневе API AudioTrack. З його допомогою будується доріжка з наступними налаштуваннями:

- setAudioAttributes():
    1) setUsage(USAGE_ASSISTANT) – визначається взаємодія з аудіо як з ШІ помічником, що дозволить уникнути випадкового самоприглушення та дозволить користувачу окремо налаштувати гучність озвучення;
    2) setContentType(CONTENT_TYPE_SPEECH) – вказується програвання мови, це позитивно впливає на обробку аудіо системою та взаємодію з іншими звуками;
- setAudioFormat() – вказуються параметри аудіо;
- setBufferSizeInBytes(max(minBuffer * 2, initialChunkSize * 4)) – встановлюється буфер на основі мінімально допустимого, або від розміру першого вхідного пакету залежно від того яке значення більше;
- setTransferMode(AudioTrack.MODE_STREAM) – вказується режим потокового програвання, що дозволяє коректно працювати із поступовим завантаженням.

Зупинка сервісу відбувається за допомогою onDestroy() (див. додаток Б). Спочатку закриваються та очищаються усі черги. Згодом скасовуються всі асинхронні задачі. Далі зупиняється та вбивається TTS і доріжка. Після цього закриваються всі задачі та відпускається фокус з ресурсами. Наостанок викликається базовий onDestroy(), який завершує роботу з сервісом.

4. ТЕСТУВАННЯ СИСТЕМИ

Даний розділ містить результати тестування розробленої системи у різноманітних сценаріях. Додатково оцінено ефективність запропонованого рішення в певних метриках.

4.1 Функціональне тестування

Для перевірки роботоздатності був прописаний список необхідних перевірок. Також, аби повністю покрити всі сценарії роботи, було вирішено розглядати й різні середовища, а саме, доступні фізичні пристрої Samsung A41 (Android 12.0), Samsung S24 (Android 16.0) та емулятори Android 8.0 і 16.0. Вибір емуляції Android 8.0 обґрунтований мінімально допустимим SDK 26 додатка, який відповідає цій версії. Емуляція Android 16.0 потрібна для перевірки роботи на сучасній версії та порівнянні роботи з пристроєм. Проте, як відомо, емулятори мають обмеження і повністю протестувати функціонал з їх допомогою неможливо [25]. Для демонстрації тестів та можливості проведення наведено таблицю В.1 у додатку В.

Щодо тестів позначених знаком “-”, то відсутність можливості їх проведення обумовлена наступними факторами:

- робота в режимі енергозбереження. Хоча фактично його можна увімкнути в емуляторі, на практиці це дає ефект близький до нуля, адже емулятор використовує обчислювальні ресурси комп’ютера. Тому, навіть використовуючи цей режим, емулятор буде все ще користуватися ресурсами, які не доступні на більшості пристроїв навіть з вимкненим енергозбереженням;
- фільтр дзвінків (телефонний дзвінок). В емуляторі є функціонал для перевірки роботи з дзвінком. Проте, по-перше, він не має емуляції Baseband-процесора [26] для обробки дзвінків. А по-дргуе, на пристрої для

виведення звуку використовується Audio HAL (Hardware Abstraction Layer) [27], який працює на рівні драйверів, коли в емуляторі це просто пряма робота з аудіокартою комп'ютера.

- функціонал пов'язаний з Bluetooth. Емулятор не оснащений Bluetooth-модулем і відповідно не може підключатись до пристроїв.
- озвучення (локальний TTS, але в системі обраний інший). Зазвичай телефони поставляються з TTS від виробника. Емулятор в свою чергу має лише Google TTS. Встановлення додаткового TTS є технічно складне.
- озвучення у беззвучному режимі. Подібно до фільтра дзвінків, даний функціонал керується Audio HAL, який відсутній в емуляторі.

За результатами функціонального тестування встановлено, що система коректно реалізує встановлені вимоги без критичних дефектів. Проте, були знайдені низьковпливові дефекти.

Одним з таких виявилась взаємодія з Bluetooth-пристроями. Наразі озвучення відбувається при наявності підключення і це працює з будь-якими пристроями. Однак, при використанні аудіопристроїв, при встановленні підключення, немає гарантії, що було підключено аудіо пристрою. Тому, якщо Bluetooth-пристрій, дасть збій, то озвучення буде відбуватися через динаміки телефону, що веде до небажаного озвучення. Виправити дефект можна додавши додаткові перевірки, але це може накласти обмеження на не аудіопристрої.

Щодо іншого дефекту, то він стосується детектора мови. ML Kit визначає мову за n-грамами, проте, він не враховує фонетику. А це означає, що мова для тексту написаного транслітом визначиться некоректно. Виправлення даного дефекту без суттєвих змін неможливе, адже це обмеження самого детектору, яке щоб обійти, необхідно навчити власну модель яка буде враховувати як n-грами, так і фонетику.

4.2 Нефункціональне тестування та оцінка ефективності

4.2.1 Тестування стабільності

Повертаючись до існуючих рішень можна пригадати, що в них була така властивість, що з деяким часом вони переставали працювати. Тому для перевірки вірності рішення даної проблеми необхідно провести тестування стабільності. Для цього використовувались фізичні пристрої Samsung A41 та Samsung S24, емулятори не оснащені акумулятором та живляться від комп'ютера і тому не можуть відтворити відповідні сценарії.

Тестування виконувалось наступним чином. На одному пристрої обраний генеративний TTS, на іншому локальний. Телефони сконфігуровані так, що отримують приблизно одні і ті ж самі сповіщення. Вони залишаються в простої на 3 дні з одним циклом підзарядки. В кінці кожного дня кожному пристрої надсилається тестове сповіщення для перевірки працездатності сервіса. По закінченню трьох днів, на смартфонах TTS міняються місцями, тепер на тому де був генеративний, працює локальний і навпаки. Наступні 3 дня аналогічні до попередніх.

За результатами шестиденного тестування стабільності не було виявлено жодного випадку зупинки роботи сервісу. Система успішно пройшла перевірку в умовах простою та низького заряду. Отримані результати свідчать, що використання резервування ресурсів ОС, дозволяє запобігти сценаріїв зупинки роботи, які характерні існуючим рішенням.

4.2.2 Тестування енергоефективності

Для оцінки впливу реалізованого додатка на автономність пристрою було проведено тестування енергоефективності. Використані були Samsung A41 та інструмент Android studio – Profiler [28], що дозволяє моніторити ресурси телефону. Щоправда, оскільки відбувається робота саме з аудіо, то саме за відтворення відповідальний DSP (Digital Signal Processor) [29], активність та

використання якого наразі не існує можливість перевірити. Проте, відомо що подібні компоненти споживають ресурсів у рази менше.

Для тестування були обрані три сценарії: робота системи у фоновому режимі без озвучення, озвучення сповіщення за допомогою локального TTS, озвучення сповіщення за допомогою генеративного TTS. Заради точності тестів, було підключено телефон до Android Studio через Wi-Fi мережу, відключено від зарядних пристроїв та був вимкнений екран. Також, для кожного сценарію, смартфон перед початком перевірок було вирішено залишити на 30 секунд з вимкненим екраном для стабілізації фонових процесів. У сценаріях озвучення використаний месенджер Discord та сповіщення зі заголовком “KnightElmojr” й текстом “Some text”. TTS сконфігуровані за замовчуванням.

Результати тестування фонової роботи (див. рисунок 4.1) показують, що система дійсно більшість часу (~93%) знаходиться в режимі очікування, оскільки потік спить. На рисунку можна побачити маленькі стрибки у використанні процесора до ~1%. Дане явище відбувається через використання ОС механізму binder IPC, який в свою чергу, перевіряє, чи живий сервіс надсилаючи мікро-транзакції.

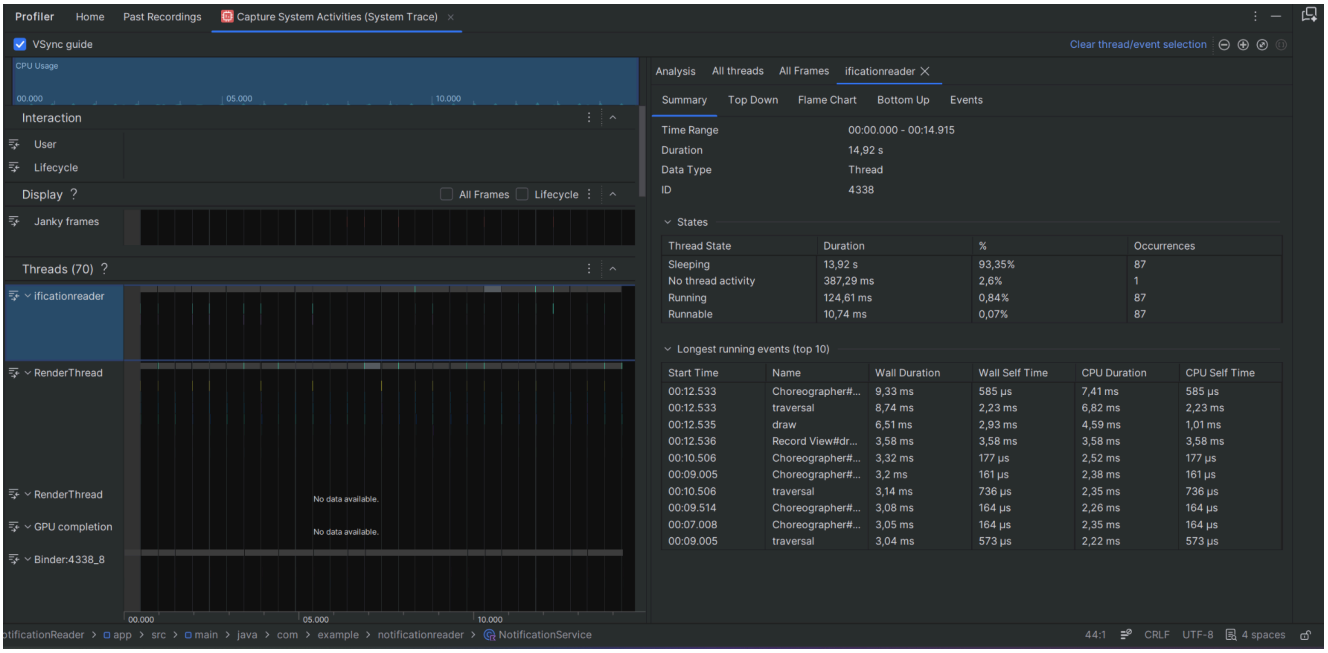


Рис. 4.1 Результати тестування сценарію фонової роботи

При активній роботі, тобто озвученні за допомогою локального TTS (див. рисунок 4.2) спостерігається невеликий імпульс у навантаженні на процесор. Пікове значення навантаження досягає максимум до 15%, тоді як середнє значення в межах 5-10%. Можна побачити, що весь процес обробки та передача в TTS займає близько 60 мс, що означає, такі відсотки навантаження на процесор є насправді досить малими.

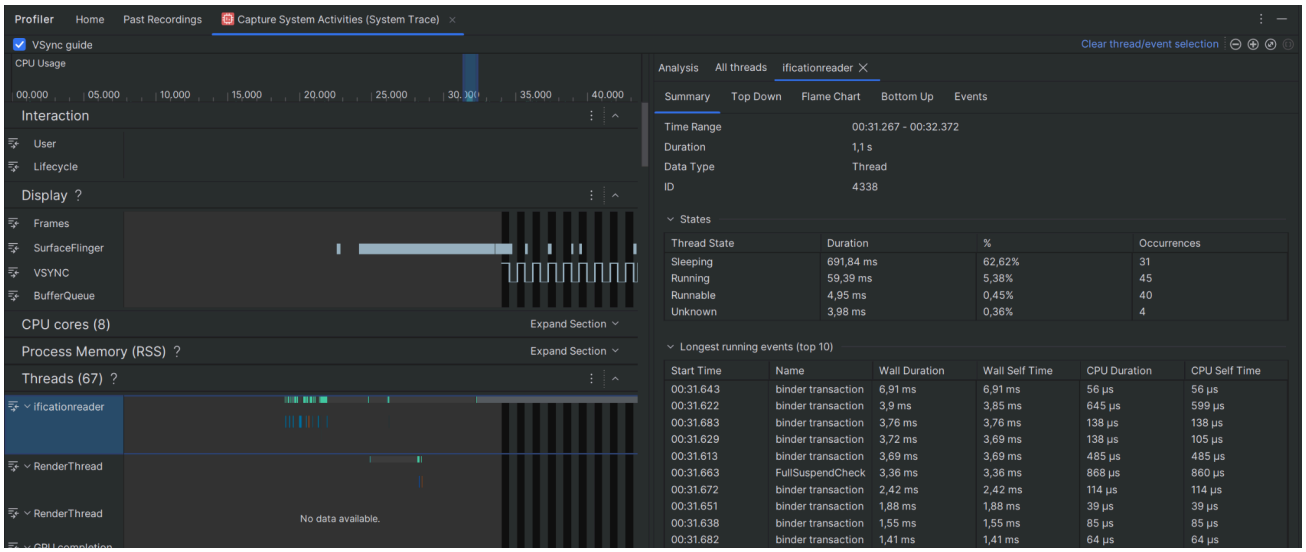


Рис. 4.2 Результати тестування сценарію озвучення локального TTS

У сценарії озвучення за допомогою генеративного TTS (див. рисунок 4.3) розподіляє ресурси рівномірніше. Як зазначалось у розділі реалізації, для запису аудіоданих у буфер, він має звільнитися від попередніх. Дане звільнення і запис добре видно після виконання основних операцій. Пікове навантаження сягає до 15%, коли середнє значення – до 5%.

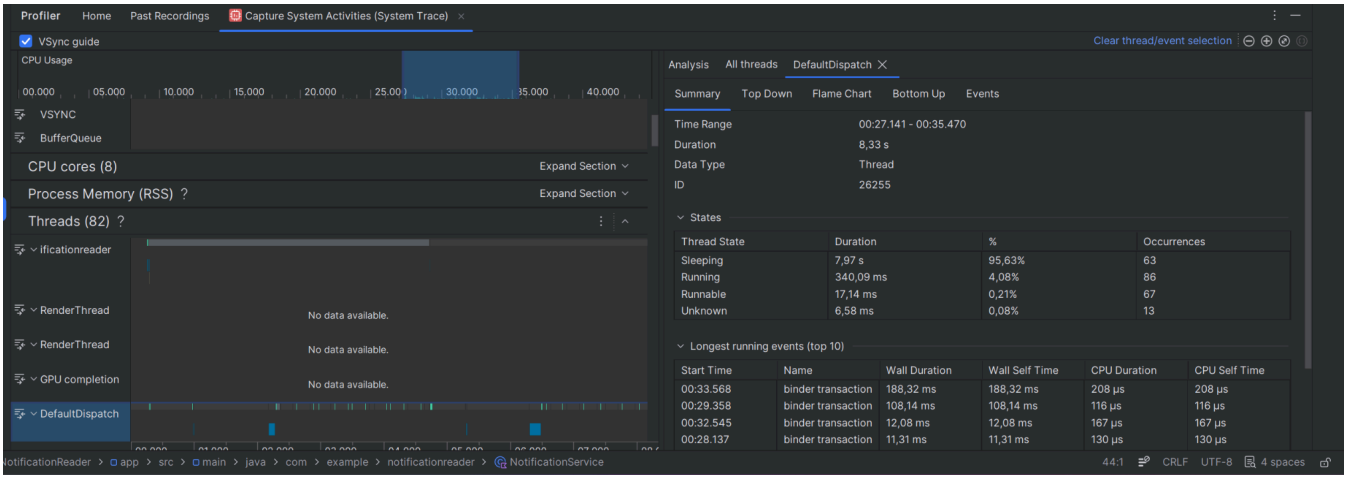


Рис. 4.3 Результати тестування сценарію озвучення генеративного TTS

Для точних розрахунків використання заряду акумулятора необхідний доступ до навантаження на DSP та інформація щодо його споживання та споживання центрального процесора. Оскільки дана інформація недоступна, були зроблені приблизні розрахунки у порівнянні з додатком Youtube. Дослідивши відгуки, в середньому у звичайного користувача активне використання Youtube втрачає 10% заряду за годину [30] [31] [32]. Заміри навантаження центрального процесора показали, що даний додаток використовує близько 35% ресурсів. Тобто, завантаження 1%/год дорівнює близько 0.28% заряду акумулятора. З цього випливає наступна формула:

$$C_{drain} = L \times t \times K_e \quad (4.1)$$

де Cdrain – втрата заряду в годину;

L – навантаження під час роботи;

t – час роботи;

Ke – коефіцієнт енерговитрат (фіксоване значення 0.28).

Транзакція на перевірку життя сервісу здійснюється в проміжках 1% від загального часу, тобто 0.06 годин, а її навантаження 1% тобто в годину вона споживає 0.0168% заряду.

Робота процесора з озвученням локального TTS триває 60 мс. За середнє значення навантаження візьмемо 7%. Згідно з одним дослідженням у юнаків в день приходить близько 240 сповіщень [33]. Згідно з іншою статистикою в середньому людина отримує орієнтовно 200 сповіщень на день [34]. Тобто справедливо сказати, що звичайному користувачу приходить приблизно 10 сповіщень в годину. Таким чином процесор буде працювати 600 мс в проміжку однієї години, що є 0.00016 годин. Підставивши значення у формулу отримуємо 0.0003136% заряду в годину.

Генеративний TTS виконував роботу близько 350 мс, що при множенні на сповіщення дає 3.5 с, а це 0.001 год. Проте, він під час роботи багато чекає, тому 5% середнього значення можна звести до 2%, для заповнення очікування і

рівномірності. Як результат, генеративний TTS витрачає 0.00056% заряду в годину.

Підсумовуючи витрати на TTS та фонову роботу, використання локального TTS вартує 0.0171136% заряду в годину, тоді як – генеративного 0.01736%. Варто зазначити, що розрахунки є досить приблизними і реальні сповіщення, як і їх кількість можуть сильно відрізнятись, відповідно і витрати на TTS будуть інші.

4.3 Приймальне тестування

У даному підрозділі наведено якісну оцінку ефективності системи на основі практичного застосування. Основна увага приділена зручності отримання інформації та впливу на поведінку користувача.

В процесі тривалого практичного використання встановлено, що система зменшує необхідність прямої взаємодії з екраном приблизно на 70% (залежно від конкретного випадку цифра варіюється). Основну перешкоду на шляху до досягнення 100%, становлять сповіщення месенджерів. Зазвичай, при отриманні повідомлення від знайомого, виникає потреба відповісти, а додаток, в свою чергу, не надає такий функціонал. Однак час на реагування на повідомлення зменшується, адже доступ до змісту отримується раніше.

В більшості випадків, озвучення дозволяє чітко зрозуміти зміст сповіщення. Якість озвучення сильно залежна від тексту. Тому система найкраще працює зі структурованими сповіщеннями від додатків, тоді як з повідомленнями, що містять неформальний текст (сленг, скорочення, абревіатури), працює посередньо. Сучасним моделям потрібно більше контексту для розпізнавання та озвучення маловідомих акронімів і скорочень. Також, вони рідко розставляють неправильну пунктуацію та інтонацію в озвученні, через що, текст може звучати дивно та незрозуміло.

ВИСНОВКИ

У ході виконання бакалаврської роботи, було розроблено мобільну цифрову систему голосових сповіщень для ОС Android на основі нейромережевої моделі синтезу мовлення, яка забезпечує доступ до інформації у сповіщеннях без взаємодії з екраном телефону. У даній роботі отримано наступні результати:

1. Проведено аналіз предметної області, зокрема розглянуто поняття сповіщень, їх функціональність та особливості обробки.
2. Проаналізовано існуючі програмні рішення озвучення сповіщень, визначено їхні особливості, переваги та недоліки, а також сформовано основні проблеми та можливі шляхи їх вирішення.
3. Сформовано вимоги до системи на основі аналізу аналогів та з врахуванням специфіки мобільних пристроїв і потреб користувача.
4. Обрано технології та інструменти для реалізації системи: Kotlin, Android studio, Google ML Kit, Google TTS, Gemini 2.5 Flash TTS. Обґрунтовано, чому вони підходять краще за аналоги.
5. Спроєктовано архітектуру та алгоритми системи, зокрема обробки сповіщень, фільтрації, вибору мови, озвучення, запуску та зупинки сервісу.
6. Реалізовано систему, що відповідає вимогам та рішенням етапу проєктування. Продемонстровано всі екрани та описано їх: MainActivity, SettingsActivity, LanguageListActivity, BluetoothListActivity, AppListActivity, VertexSettingsActivity. Наведено детальний опис компонентів та модулів: AppSettings, NotificationHelper, VertexAiClient, NotificationService.
7. Проведено тестування системи у різноманітних сценаріях використання. Виявлено незначні дефекти стосовно Bluetooth-фільтра та детектора мови. Висвітлено проблеми системи, що можуть виникнути під час її експлуатації, зокрема проблема месенджерів та неформального тексту.

Таким чином, розроблена система повністю виконує поставлені завдання, а виявлені під час тестування особливості обробки тексту формують чітке підґрунтя для подальшого вдосконалення алгоритмів додатка у майбутньому.

ПЕРЕЛІК ПОСИЛАНЬ

1. Рудик А. В., Герцюк М. М. Обґрунтування вибору моделей TTS для озвучення сповіщень на ОС Android // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація», 1–3 травня 2026 року, м. Київ. Збірник тез. (подано до друку).
2. Рудик А. В., Герцюк М. М. Розробка алгоритму озвучення сповіщень на мобільних пристроях на базі ОС Android // VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, м. Київ. Збірник тез. С. 368.
3. Notifications | Mobile | Android Developers. Android Developers. URL: https://developer.android.com/design/ui/mobile/guides/home-screen/notifications (дата звернення: 24.04.2026).
4. Notification | API reference | Android Developers. Android Developers. URL: https://developer.android.com/reference/android/app/Notification (дата звернення: 24.04.2026).
5. Dev A. Notification Reader – Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=com.argonremote.notificationreader&hl=en-GB (дата звернення: 24.04.2026).
6. Bryant M. Notification Reader - Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=org.freepoc.notificationreader&hl=en (дата звернення: 24.04.2026).
7. DFound. Notification Reader - Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=com.dfound.notification.reader.notification_reader&hl=en-US (дата звернення: 24.04.2026).

8. Codeseed. Audify Notification Announcer - Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=in.codeseed.audify&hl=en-US (дата звернення: 24.04.2026).
9. voice notify - Android Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/search?q=voice%20notify&c=apps (дата звернення: 24.04.2026).
10. notivoca - Android Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/search?q=notivoca&c=apps&hl=en-US (дата звернення: 24.04.2026).
11. Technology Z. Fake Notifications - Apps on Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=zopsoft.com.zerofall&hl=en (дата звернення: 24.04.2026).
12. Kotlin for android | kotlin. Kotlin Help. URL: https://kotlinlang.org/docs/android-overview.html (дата звернення: 24.04.2026).
13. Download android studio & app tools - android developers. Android Developers. URL: https://developer.android.com/studio (дата звернення: 24.04.2026).
14. ML kit | google for developers. Google for Developers. URL: https://developers.google.com/ml-kit (дата звернення: 24.04.2026).
15. LLC G. Speech Recognition & Synthesis – Додатки в Google Play. Android Apps on Google Play. URL: https://play.google.com/store/apps/details?id=com.google.android.tts&hl=uk (дата звернення: 24.04.2026).
16. GitHub - k2-fsa/sherpa-onnx: Speech-to-text, text-to-speech, speaker diarization, speech enhancement, source separation, and VAD using next-gen Kaldi with onnxruntime without Internet connection. Support embedded

systems, Android, iOS, HarmonyOS, Raspberry Pi, RISC-V, RK NPU, Axera NPU, Ascend NPU, x86_64 servers, websocket server/client, support 12 programming languages. GitHub. URL: https://github.com/k2-fsa/sherpa-onnx (дата звернення: 24.04.2026).

17. Gemini-TTS | cloud text-to-speech | google cloud documentation. Google Cloud Documentation. URL: https://docs.cloud.google.com/text-to-speech/docs/gemini-tts (дата звернення: 24.04.2026).
18. Text to speech overview - Speech service - Foundry Tools. Microsoft Learn: Build with answers in reach. URL: https://learn.microsoft.com/en-us/azure/ai-services/speech-service/text-to-speech (дата звернення: 24.04.2026).
19. Free Text To Speech Online with Lifelike AI Voices. ElevenLabs. URL: https://elevenlabs.io/text-to-speech (дата звернення: 24.04.2026).
20. Request runtime permissions | Privacy | Android Developers. Android Developers. URL: https://developer.android.com/training/permissions/requesting (дата звернення: 24.04.2026).
21. Get a Google Cloud API key | Generative AI on Vertex AI | Google Cloud Documentation. Google Cloud Documentation. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/start/api-keys?usertype=expressmode (дата звернення: 24.04.2026).
22. Quickstart: Send text prompts to Gemini using Agent Platform Studio | Generative AI on Vertex AI | Google Cloud Documentation. Google Cloud Documentation. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/start/quickstarts/quickstart (дата звернення: 24.04.2026).
23. Create and manage notification channels | Views | Android Developers. Android Developers. URL: https://developer.android.com/develop/ui/views/notifications/channels (дата

звернення: 24.04.2026).
24. The Task APIs | Google Play services | Google for Developers. Google for Developers. URL: https://developers.google.com/android/guides/tasks (дата звернення: 24.04.2026).
25. Advanced emulator usage | Android Studio | Android Developers. Android Developers. URL: https://developer.android.com/studio/run/advanced-emulator-usage (дата звернення: 24.04.2026).
26. Contributors to Wikimedia projects. Baseband processor - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Baseband_processor (дата звернення: 24.04.2026).
27. Audio HAL | Android Open Source Project. Android Open Source Project. URL: https://source.android.com/docs/core/audio/implement (дата звернення: 24.04.2026).
28. Profile your app performance | Android Studio | Android Developers. Android Developers. URL: https://developer.android.com/studio/profile (дата звернення: 24.04.2026).
29. Contributors to Wikimedia projects. Digital signal processor - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Digital_signal_processor#Overview (дата звернення: 24.04.2026).
30. Panoptech. Anyone's YouTube significantly draining battery?. Reddit. URL: https://www.reddit.com/r/GalaxyS24Ultra/comments/1av8t4p/anyones_youtube_significantly_draining_battery/ (дата звернення: 24.04.2026).
31. Community S. YouTube Battery Drain. URL: https://us.community.samsung.com/t5/Discussions/YouTube-Battery-Drain/td-p/3118553 (дата звернення: 24.04.2026).
32. Is 10% battery drop normal for 50-minute YouTube video?. Facebook. URL: https://www.facebook.com/groups/396715689802791/posts/59891672624935

2/ (дата звернення: 24.04.2026).

33. Study: Average teen received more than 200 app notifications a day. Michigan Medicine. URL: https://www.michiganmedicine.org/health-lab/study-average-teen-received-more-200-app-notifications-day (дата звернення: 25.04.2026).
34. Cell Phone Usage Stats 2026: Americans Check Their Phones 186 Times a Day. Reviews.org. URL: https://www.reviews.org/mobile/cell-phone-addiction/ (дата звернення: 24.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра Технологій цифрового розвитку

Мобільна цифрова система голосових сповіщень для ОС Android на основі нейромережевої моделі синтезу мовлення

Виконав:
студент 4 курсу
групи ТЦР-41
Рудик А. В.

Керівник роботи:
д-р філософії,
доцент кафедри ТЦР
Герцюк М.М.

Київ 2026

Мета, об’єкт та предмет дослідження

Мета – підвищення ефективності взаємодії користувача з мобільним пристроєм шляхом розробки мобільної цифрової системи голосових сповіщень для ОС Android на основі нейромережевого синтезу мовлення.

Об’єкт дослідження – процес обробки та передачі текстових даних у мобільному середовищі.

Предмет дослідження – методи та алгоритми обробки й озвучення текстових сповіщень у мобільних системах Android.

Завдання

1. Проаналізувати предметну область сповіщень смартфонів.

2. Проаналізувати існуючі рішення озвучення сповіщень для Android.

3. Сформувати вимоги до системи з урахуванням попереднього аналізу, користувача та ОС.

4. Вибрати та обґрунтувати інструменти і технології для реалізації системи.

5. Спроєктувати архітектуру системи й алгоритми обробки текстових повідомлень та їх озвучення.

6. Реалізувати систему із підтримкою черги озвучення, визначення мови та синтезу мовлення для Android.

7. Провести тестування обробки та озвучення сповіщень TTS-сервісами.

Аналіз аналогів

Був проведений аналіз існуючих програмних рішень: Notification Reader (Argon Dev), Notification Reader (Malcolm Bryant), Notification Reader (DFound), Audify, Voice Notify, Notivoca.

Результати наступні:

- Існуючі додатки для визначення мови використовують TTS, що призводить до некоректного вибору мови та значного зниження якості озвучення;
- Сповіщення необхідно передавати частинами для точного підбору мови для кожної з них;
- Аналоги не оптимізовано використовують ресурси пристрою, що веде до швидшої втрати заряду акумулятором та нестабільної роботи додатку;
- Необхідні налаштування, кастомізація та зручний дизайн для підвищення якості користувацького досвіду;
- Аналоги не містять інструментів для перевірки власної роботи.

Вимоги до системи

Функціональні:

- перехоплення сповіщень. Система повинна у фоновому режимі перехоплювати сповіщення і розділяти дані до озвучки;
- фільтрація. Сповіщення які не відповідають налаштуванням користувача та загальним правилам не повинні озвучуватись;
- нормалізація тексту. Система повинна прибирати нечитабельні слова та повертати їх після визначення мови у нормалізованому вигляді;
- визначення мови. Визначення мови повинно відбуватися за допомогою спеціального детектора;
- мультимодальний синтез мовлення. Має бути реалізовано синтез мовлення з використанням стандартного та генеративного TTS;
- налаштування. Користувачу мають бути доступні налаштування фільтрів, мов озвучення та TTS;
- тестування. Повинен бути реалізований функціонал для перевірки системи шляхом створення штучного сповіщення.

Нефункціональні:

- енергоефективність. Система повинна обмежено використовувати ресурси (до 0.1% заряду на годину), а для озвучки повинен резервуватись процесор;
- низька затримка . Час між отриманням сповіщення і його озвученням не повинен перевищувати 2 секунди;
- конфідеційність. Дані повинні оброблятися локально, де це можливо, і не повинні зберігатись на серверах.

Проєктування архітектури системи

Система складається з наступних компонентів:

- Service;
- Settings component;
- Generative AI module;
- Notifications Manager component;
- View.

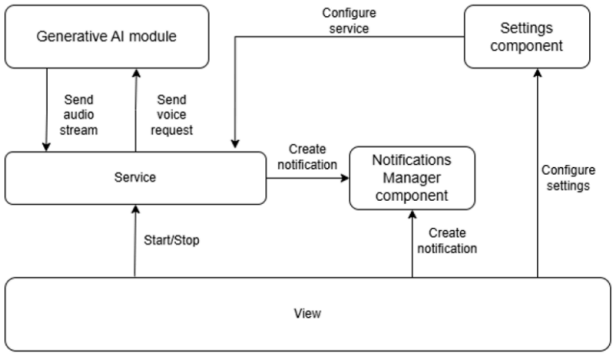
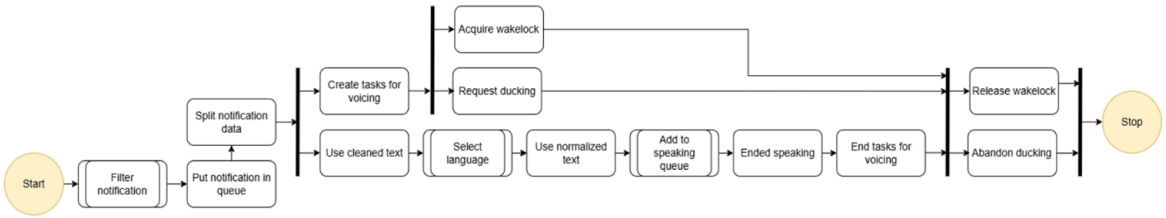


Схема архітектури системи

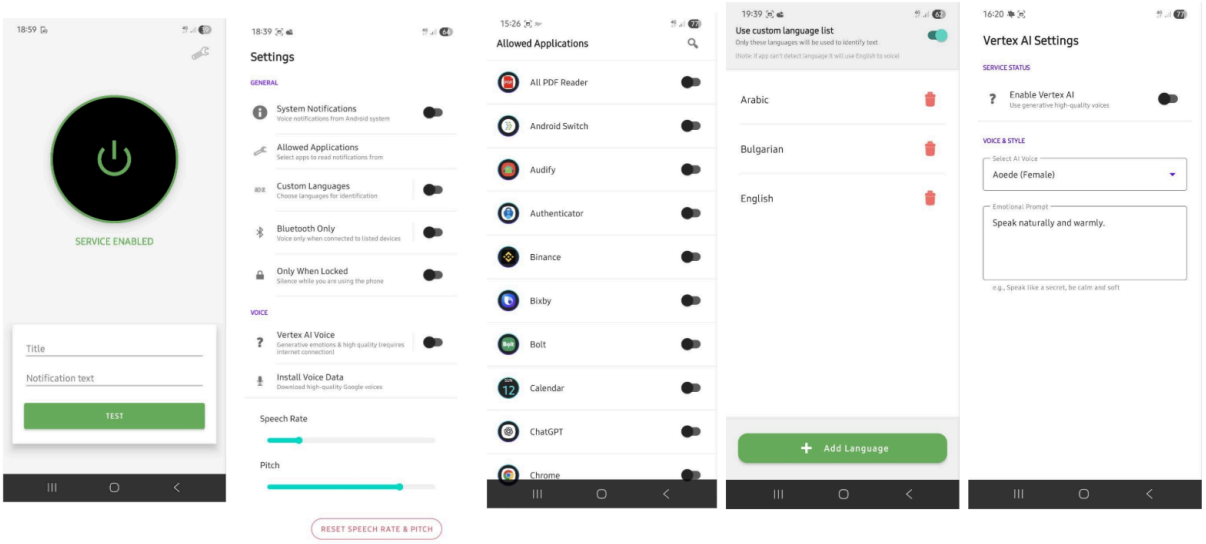
Проєктування алгоритму роботи системи

Алгоритм спрацьовує при отриманні сповіщення і завершується після закінчення озвучення. Спочатку виконується фільтрація з урахуванням налаштувань, після чого сповіщення додається до черги для обробки. Далі, розділяється на заголовок і текст, для яких створюються задачі. Наступним кроком зміст очищається від нечитабельних елементів, взизначається мова та текст нормалізується відповідно до неї. Після цього текст додається до черги озвучення, а після завершення озвучення задачі закриваються.



Блок-схема алгоритму обробки сповіщень

Програмна реалізація. Інтерфейс



Функціональне тестування

Було проведене функціональне тестування системи у 4 різних середовищах: емулятор Android 8.0, емулятор Android 16.0, Samsung A41 (Android 14.0), Samsung S24 (Android 16.0).

За результатами тестування було визначено два незначних дефекти:

- Фільтр Bluetooth-пристроїв не перевіряє чи підключено аудіо пристрою;
- Детектор мови ML Kit не враховує фонетику тексту.

Нефункціональне та приймальне тестування

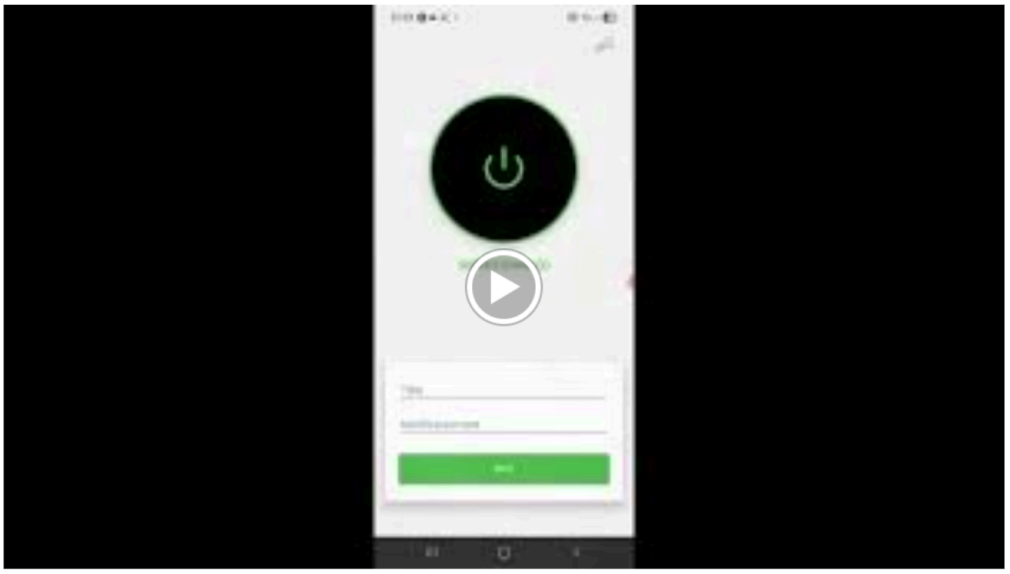
Було проведене нефункціональне тестування стабільності, енергоефективності та була надана якісна оцінка ефективності.

Перевірено і підтверджено, що система протягом тривалого використання (6 днів) не перестає працювати, як це відбувається з аналогами.

Проведено вимірювання використання ресурсів системи та оцінено приблизні витрати заряду акумулятора, які не перевищують 0.02% на годину.

Надана якісна оцінка ефективності: система практично повністю усуває потребу контакту з екраном смартфона, в більшості випадків озвучення дозволяє чітко зрозуміти зміст. Основною перешкодою виявились месенджери та неформальний текст (сленг, скорочення, абревіатури).

Демонстрація роботи додатка



Апробація результатів дослідження

Рудик А. В., Герцюк М. М. Обґрунтування вибору моделей TTS для озвучення сповіщень на ОС Android // Всеукраїнська науково-технічна конференція «Технології цифрового розвитку: програмні системи та децентралізація», 1–3 травня 2026 року, м. Київ. Збірник тез. (подано до друку).

Рудик А. В., Герцюк М. М. Розробка алгоритму озвучення сповіщень на мобільних пристроях на базі ОС Android // VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, м. Київ. Збірник тез. С. 368.

Висновки

У роботі отримано наступні результати:

1. Проведено аналіз предметної області сповіщень смартфонів та визначено особливості їх обробки.
2. Проаналізовано існуючі рішення озвучення сповіщень для Android і виявлено їх основні недоліки.
3. Сформовано функціональні та нефункціональні вимоги до системи обробки сповіщень.
4. Обрано та обґрунтовано інструменти та технології для реалізації системи.
5. Розроблено архітектуру та алгоритми обробки, фільтрації, визначення мови й озвучення сповіщень.
6. Реалізовано Android-систему із підтримкою черги озвучення, детектора мови та декількох TTS.
7. Проведено тестування системи у різних сценаріях використання та оцінено стабільність і якість озвучення.

Дякую за увагу!

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ ПРОГРАМНИХ МОДУЛІВ

AppSettings

```
package com.example.notificationreader

import android.annotation.SuppressLint
import android.app.KeyguardManager
import android.app.Notification
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothManager
import android.content.Context
import android.media.AudioManager
import android.os.PowerManager
import
android.service.notification.StatusBarNotification
import android.speech.tts.TextToSpeech


interface NotificationFilter {
    fun shouldProceed(sbn: StatusBarNotification,
settings: AppSettings): Boolean
}

class AppSettings(private val context: Context) {
    private val prefs =
context.getSharedPreferences("Settings",
Context.MODE_PRIVATE)

    // On/Off Switch
    var isServiceEnabled: Boolean
        get() = prefs.getBoolean("service_enabled",
false)
        set(value) =
prefs.edit().putBoolean("service_enabled",
value).apply()

    // GENERAL SETTINGS

    // System Notifications Switch
    var readSystem: Boolean
        get() =
prefs.getBoolean("read_system_notifications",
false)
        set(value) =
prefs.edit().putBoolean("read_system_notifications"
, value).apply()

    // Custom Languages Switch
    var useCustomLanguages: Boolean
        get() = prefs.getBoolean("use_custom_langs",
false)
        set(value) =
prefs.edit().putBoolean("use_custom_langs",
value).apply()

    // List of Custom Languages
    var customLanguages: Set<String>
        get() =
prefs.getStringSet("selected_languages",
setOf("en")) ?: setOf()
        set(value) =
prefs.edit().putStringSet("selected_languages",
value).apply()

    // List of Allowed Applications
    var allowedApps: Set<String>
        get() = prefs.getStringSet("allowed_apps",
setOf(context.packageName)) ?:
setOf(context.packageName)
        set(value) =
prefs.edit().putStringSet("allowed_apps",
value).apply()

    // Only when Locked Switch
    var onlyWhenLocked: Boolean
        get() = prefs.getBoolean("only_when_locked",
false)
        set(value) =
prefs.edit().putBoolean("only_when_locked",
value).apply()

    // BLUETOOTH SETTINGS

    // Bluetooth Only Switch
    var onlyBluetooth: Boolean
        get() = prefs.getBoolean("only_bluetooth",
false)
        set(value) =
prefs.edit().putBoolean("only_bluetooth",
value).apply()

    // List of Allowed Bluetooth Devices
    var allowedBluetoothDevices: Set<String>
```

```
        get() =
prefs.getStringSet("allowed_bt_devices", setOf()) ?:
setOf()
        set(value) =
prefs.edit().putStringSet("allowed_bt_devices",
value).apply()

    // VOICE SETTINGS

    // Main switch Vertex AI
    var useVertexAI: Boolean
        get() = prefs.getBoolean("use_vertex_ai",
false)
        set(value) =
prefs.edit().putBoolean("use_vertex_ai",
value).apply()

    // Prompt for Gemini (mood/emotion)
    var vertexPrompt: String
        get() = prefs.getString("vertex_prompt",
"Speak naturally and warmly.") ?: "Speak naturally
and warmly."
        set(value) =
prefs.edit().putString("vertex_prompt",
value).apply()

    // vertexSpeaker
    var vertexSpeaker: String
        get() = prefs.getString("vertex_speaker",
"Aoede") ?: "Aoede"
        set(value) =
prefs.edit().putString("vertex_speaker",
value).apply()

    // Speech rate
    var speechRate: Float
        get() = prefs.getFloat("speech_rate", 1f)
        set(value) =
prefs.edit().putFloat("speech_rate", value).apply()

    // Speech pitch
    var speechPitch: Float
        get() = prefs.getFloat("speech_pitch", 1f)
        set(value) =
prefs.edit().putFloat("speech_pitch", value).apply()

    // Build Pipeline
    fun getProcessingPipeline():
List<NotificationFilter> {
        val pipeline =
mutableListOf<NotificationFilter>()

        val audioManager =
context.getSystemService(Context.AUDIO_SERVI
CE) as AudioManager
        val powerManager =
context.getSystemService(Context.POWER_SERVI
CE) as PowerManager
        val keyguardManager =
context.getSystemService(Context.KEYGUARD_S
ERVICE) as KeyguardManager
        val bluetoothManager =
context.getSystemService(Context.BLUETOOTH_
SERVICE) as BluetoothManager

        // 1. Filter empty text
        pipeline.add(object : NotificationFilter {
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {
                val text =
sbn.notification.extras.getCharSequence(Notificatio
n.EXTRA_TEXT)
                return !text.isNullOrBlank()
            }
        })

        // 2. Filter notifications summaries
        pipeline.add(object : NotificationFilter {
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {
                val isSummary = (sbn.notification.flags
and Notification.FLAG_GROUP_SUMMARY) !=
0
                return !isSummary
            }
        })

        // 3. Filter during calls
        pipeline.add(object : NotificationFilter {
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {
                val mode = audioManager.mode
                return mode !=
AudioManager.MODE_IN_CALL && mode !=
AudioManager.MODE_IN_COMMUNICATION
            }
```

```
        })

        // 4. Filter when phone is not in use
        pipeline.add(object : NotificationFilter {
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {
                if (!settings.onlyWhenLocked) return true
                return
keyguardManager.isKeyguardLocked ||
!powerManager.isInteractive
            }
        })

        // 5. System and App filter combined
        pipeline.add(object : NotificationFilter {
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {

                // If enabled in whitelist, skip filtering
                val pkg = sbn.packageName
                if (settings.allowedApps.contains(pkg)) {
                    return true
                }

                // If app belongs to system, check if the
setting enabled
                val pm = context.packageManager
                val appInfo = try {
pm.getApplicationInfo(pkg, 0) } catch (e:
Exception) { null }
                val isSystem = appInfo?.let {
                    (it.flags and
android.content.pm.ApplicationInfo.FLAG_SYSTE
M) != 0 &&
                            (it.flags and
android.content.pm.ApplicationInfo.FLAG_UPDA
TED_SYSTEM_APP) == 0
                } ?: (pkg == "android" || pkg ==
"com.android.systemui")

                return if (isSystem) {
                    settings.readSystem
                } else {
                    false
                }
            }
        })

        // 6. Bluetooth Filter
        pipeline.add(object : NotificationFilter {
            @SuppressLint("MissingPermission")
            override fun shouldProceed(sbn:
StatusBarNotification, settings: AppSettings):
Boolean {
                if (!settings.onlyBluetooth) return true
                val adapter = bluetoothManager.adapter ?:
return false
                val pairedDevices =
adapter.bondedDevices
                val allowed =
settings.allowedBluetoothDevices

                return pairedDevices.any {
allowed.contains(it.address) &&
isDeviceConnected(it) }
            }

            //Helping function
            private fun isDeviceConnected(device:
BluetoothDevice): Boolean {
                return try {
                    val method =
device.javaClass.getMethod("isConnected")
                    method.invoke(device) as Boolean
                } catch (e: Exception) {
                    false
                }
            }
        })

        return pipeline
    }

    fun applyToTts(tts: TextToSpeech?) {
        tts?.setSpeechRate(speechRate)
        tts?.setPitch(speechPitch)
    }

    // Check if Google TTS installed
    fun isGoogleTtsInstalled(): Boolean {
        return try {

context.packageManager.getPackageInfo("com.goo
gle.android.tts", 0)
            true
        } catch (e: Exception) {
            false
        }
```

```
    }

    // Check if App has access to Notifications
    fun isNotificationListenerEnabled(): Boolean {
        val pkgName = context.packageName
        val flat =
android.provider.Settings.Secure.getString(
            context.contentResolver,
            "enabled_notification_listeners"
        )
        return flat?.contains(pkgName) == true
    }

    // Check if Notifications are allowed
    fun areNotificationsEnabled(): Boolean {
        val nm =
context.getSystemService(Context.NOTIFICATIO
N_SERVICE) as android.app.NotificationManager
        return nm.areNotificationsEnabled()
    }

    // Verify that all needed things are done and we
can work
    fun canEnableService(): Boolean {
        if (!isNotificationListenerEnabled()) return
false

        if (!useVertexAI && !isGoogleTtsInstalled())
return false

        if (!areNotificationsEnabled()) return false

        return true
    }
}
```

NotificationHelper

```
package com.example.notificationreader

import android.app.Notification
import android.app.NotificationChannel
import android.app.NotificationManager
import android.app.PendingIntent
import android.content.Context
import android.content.Intent
import android.os.Build
import androidx.core.app.NotificationCompat
import
com.example.notificationreader.activity.MainActivi
ty

class NotificationHelper(private val context:
Context) {

   companion object {
      const val SERVICE_CHANNEL_ID =
"service_status_channel"
      const val TEST_CHANNEL_ID =
"test_notifications_channel"
      const val SERVICE_NOTIFICATION_ID = 1
      const val TEST_NOTIFICATION_ID = 101
   }

   private val notificationManager =

context.getSystemService(Context.NOTIFICATIO
N_SERVICE) as NotificationManager

   init {
      createChannels()
   }

   private fun createChannels() {
      if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.O) {
         val serviceChannel = NotificationChannel(
            SERVICE_CHANNEL_ID,
            "Service status",

NotificationManager.IMPORTANCE_LOW
         ).apply {
            description = "Notifications for service
status"
            setSound(null, null)
         }
         val testChannel = NotificationChannel(
            TEST_CHANNEL_ID,
            "Test notifications",

NotificationManager.IMPORTANCE_DEFAULT
         )


notificationManager.createNotificationChannel(serv
iceChannel)

notificationManager.createNotificationChannel(test
Channel)
      }
   }

   fun createServiceNotification(): Notification {
      return NotificationCompat.Builder(context,
SERVICE_CHANNEL_ID)

.setSmallIcon(R.drawable.outline_mobile_speaker_
24)
         .setContentTitle("Notification Reader")
         .setContentText("Service is ready for
voicing")
         //Link to MainActivity
         .setContentIntent(PendingIntent.getActivity(
            context, 0,
            Intent(context,
MainActivity::class.java).apply {
               flags =
Intent.FLAG_ACTIVITY_NEW_TASK or
Intent.FLAG_ACTIVITY_CLEAR_TASK
            },
            PendingIntent.FLAG_IMMUTABLE or
PendingIntent.FLAG_UPDATE_CURRENT
         ))
         .setOngoing(true)
         .setAutoCancel(false)
         .setLocalOnly(true)

.setCategory(Notification.CATEGORY_SERVICE)

.setForegroundServiceBehavior(NotificationCompa
t.FOREGROUND_SERVICE_IMMEDIATE)

.setVisibility(NotificationCompat.VISIBILITY_PU
BLIC)
```

```
            .build()
    }

    fun sendTestNotification(title: String, content:
String) {
        val notification =
NotificationCompat.Builder(context,
TEST_CHANNEL_ID)

.setSmallIcon(android.R.drawable.ic_dialog_info)
            .setContentTitle(title)
            .setContentText(content)
            .setAutoCancel(true)
            .build()

notificationManager.notify(TEST_NOTIFICATION
_ID, notification)
    }
}
```

VertexAiClient

```
package com.example.notificationreader

import android.content.Context
import android.util.Base64
import android.util.JsonReader
import android.util.Log
import com.google.auth.oauth2.GoogleCredentials
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import
okhttp3.MediaType.Companion.toMediaType
import okhttp3.OkHttpClient
import okhttp3.Request
import
okhttp3.RequestBody.Companion.toRequestBody
import org.json.JSONArray
import org.json.JSONObject
import java.io.ByteArrayOutputStream
import java.io.InputStreamReader
import java.util.Locale

object VertexAiClient {
    private val client = OkHttpClient()
    private const val PROJECT_ID =
"notification-reader-491815"
    private const val LOCATION =
"europe-central2"
    private const val MODEL_ID =
"gemini-2.5-flash-tts"
    private var credentials: GoogleCredentials? =
null

    private fun getAccessToken(context: Context):
String {
        if (credentials == null) {
            credentials =
GoogleCredentials.fromStream(context.resources.o
penRawResource(R.raw.vertex_key))

.createScoped(listOf("https://www.googleapis.com/
auth/cloud-platform"))
        }
        credentials!!.refreshIfExpired()
        return credentials!!.accessToken.tokenValue
    }

    suspend fun fetchAudioStreaming(
        text: String,
        locale: Locale,
        context: Context,
        settings: AppSettings,
        onBytesReceived: (ByteArray) -> Unit
        ) = withContext(Dispatchers.IO) {
        try {
            val token = getAccessToken(context)
            val url =
"https://$LOCATION-aiplatform.googleapis.com/v
1beta1/projects/$PROJECT_ID/locations/$LOCATI
ON/publishers/google/models/$MODEL_ID:stream
GenerateContent"
            val jsonRequest = JSONObject().apply {
                put("contents", JSONArray().apply {
                    put(JSONObject().apply {
                        put("role", "user")
                        put("parts", JSONArray().apply {
                            put(JSONObject().apply {
                                put("text", "[INSTRUCTIONS:
Voice only once]
[STYLE:${settings.vertexPrompt}] [TEXT:$text")
                            })
                        })
                    })
                })
                put("generationConfig",
JSONObject().apply {
                    put("responseModalities",
JSONArray().apply { put("AUDIO") })
                    put("speechConfig",
JSONObject().apply {
                        put("languageCode",
locale.toLanguageTag())
                        put("voiceConfig",
JSONObject().apply {
                            put("prebuiltVoiceConfig",
JSONObject().apply {
                                put("voiceName",
settings.vertexSpeaker)
                            })
                        })
                    })
                })
            }

            val request = Request.Builder()
                .url(url)
```

```
            .addHeader("Authorization", "Bearer
$token")

.post(jsonRequest.toString().toRequestBody("applic
ation/json".toMediaType()))
            .build()

        client.newCall(request).execute().use {
response ->
            if (!response.isSuccessful) {
                Log.e("VERTEX_REST", "Response
not successful: ${response.code}")
                return@withContext
            }

            val inputStream =
response.body?.byteStream() ?:
return@withContext
            val reader =
JsonReader(InputStreamReader(inputStream,
"UTF-8"))
            reader.beginArray()
            while (reader.hasNext()) {
                parseResponseObjectStreaming(reader,
onBytesReceived)
            }
            reader.endArray()
        }
    } catch (e: Exception) {
        Log.e("VERTEX_REST", "Streaming error:
${e.message}")
    }
  }

  private fun
parseResponseObjectStreaming(reader: JsonReader,
onBytesReceived: (ByteArray) -> Unit) {
    reader.beginObject()
    while (reader.hasNext()) {
        when (reader.nextName()) {
            "candidates" -> {
                reader.beginArray()
                while (reader.hasNext()) {
                    reader.beginObject()
                    while (reader.hasNext()) {
                        if (reader.nextName() ==
"content") {
                            reader.beginObject()
                            while (reader.hasNext()) {
                                if (reader.nextName() ==
"parts") {
                                    reader.beginArray()
                                    while (reader.hasNext()) {
                                        reader.beginObject()
                                        while (reader.hasNext())
{
                                            if (reader.nextName()
== "inlineData") {

reader.beginObject()
                                                while
(reader.hasNext()) {
                                                    if
(reader.nextName() == "data") {
                                                        val base64 =
reader.nextString()
                                                        val bytes =
Base64.decode(base64, Base64.DEFAULT)

onBytesReceived(bytes)
                                                    } else
reader.skipValue()
                                                }
                                                reader.endObject()
                                            } else
reader.skipValue()
                                        }
                                        reader.endObject()
                                    }
                                    reader.endArray()
                                } else reader.skipValue()
                            }
                            reader.endObject()
                        } else reader.skipValue()
                    }
                    reader.endObject()
                }
                reader.endArray()
            }
            else -> reader.skipValue()
        }
    }
    reader.endObject()
  }
}
```

NotificationService

```
package com.example.notificationreader

import android.app.Notification
import android.content.ComponentName
import android.content.Context
import android.content.Intent
import android.content.res.Configuration
import android.content.res.Resources
import android.media.AudioAttributes
import android.media.AudioFocusRequest
import android.media.AudioFormat
import android.media.AudioManager
import android.media.AudioTrack
import android.os.Build
import android.os.PowerManager
import
android.service.notification.NotificationListenerSer
vice
import
android.service.notification.StatusBarNotification
import android.speech.tts.TextToSpeech
import
android.speech.tts.UtteranceProgressListener
import android.util.Log
import
com.google.mlkit.nl.languageid.LanguageIdentifica
tion
import java.util.Locale
import android.util.LruCache
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.Channel
import kotlin.math.max

class NotificationService :
NotificationListenerService(),
TextToSpeech.OnInitListener {

    private var tts: TextToSpeech? = null
    private var isTtsReady = false
    private val languageIdentifier =
LanguageIdentification.getClient()
    private lateinit var pipeline:
List<NotificationFilter>
    private data class RawNotification(
        val title: String,
        val text: String,
        val sbn: StatusBarNotification,
        val settings: AppSettings
    )

    private val notificationInputChannel =
Channel<RawNotification>(Channel.UNLIMITED)

    private var isInputProcessing = false
    private val duplicateCache = LruCache<String,
Long>(10)

    private val audioManager by lazy {
getSystemService(Context.AUDIO_SERVICE) as
AudioManager }
    private var focusRequest: AudioFocusRequest? =
null

    private val speechChannel =
Channel<AiTask>(Channel.UNLIMITED)
    private var isQueueStarted = false

    private val serviceJob = SupervisorJob()
    private val serviceScope =
CoroutineScope(Dispatchers.Main + serviceJob)

    private val powerManager by lazy {
getSystemService(Context.POWER_SERVICE) as
PowerManager }

    private var currentAiAudioTrack: AudioTrack? =
null

    private val wakeLock by lazy {

powerManager.newWakeLock(PowerManager.PAR
TIAL_WAKE_LOCK,
"NotificationReader:SpeechWakeLock")
    }

    private var activeTtsTasks = 0

    override fun onCreate() {
        super.onCreate()
        tts = TextToSpeech(this, this)

        //Return notification when booted
        val settings = AppSettings(this)
        pipeline = settings.getProcessingPipeline()
        if (settings.isServiceEnabled) {
            val helper = NotificationHelper(this)
```

```
startForeground(NotificationHelper.SERVICE_NOTIFICATION_ID,
helper.createServiceNotification())
        }
    }

    override fun onInit(status: Int) {
        if (status == TextToSpeech.SUCCESS) {
            isTtsReady = true
            applyAudioAttributesToLocalTts()
            setupTtsProgressListener()
        }
    }

    // Create|Delete service
    override fun onStartCommand(intent: Intent?,
flags: Int, startId: Int): Int {
        val settings = AppSettings(this)
        val helper = NotificationHelper(this)
        when (intent?.action) {
            "UPDATE_STATUS" -> {
                if (settings.isServiceEnabled) {
                    requestRebind(ComponentName(this,
NotificationService::class.java))

startForeground(NotificationHelper.SERVICE_NOTIFICATION_ID,
helper.createServiceNotification())
                } else {
                    requestUnbind()

stopForeground(STOP_FOREGROUND_REMOVE)
                    stopSelf()
                }
            }
            "STOP_TTS" -> tts?.stop()
        }
        return START_STICKY
    }

    // Local TTS listener for tracking tasks
    private fun setupTtsProgressListener() {
        tts?.setOnUtteranceProgressListener(object :
UtteranceProgressListener() {
            override fun onStart(utteranceId: String?) {
                Log.d("TTS_DEBUG", "Started
speaking: $utteranceId")
            }
            override fun onDone(utteranceId: String?) {
                if (utteranceId?.startsWith("PAUSE") ==
true) return
                Log.d("TTS_DEBUG", "Finished:
$utteranceId")
                endSpeechTask()
            }

            @Deprecated("Deprecated in Java")
            override fun onError(utteranceId: String?) {
                Log.e("TTS_DEBUG", "Error speaking:
$utteranceId")
                endSpeechTask()
            }
        })
    }

    // When service is enable via button
    override fun onNotificationPosted(sbn:
StatusBarNotification?) {
        Log.d("SERVICE_TEST", "Detected
notification from: ${sbn?.packageName}")
        super.onNotificationPosted(sbn)
        val settings = AppSettings(this)

        if (!settings.isServiceEnabled ||
!settings.canEnableService()) {
            if (settings.isServiceEnabled) {
                settings.isServiceEnabled = false
            }

stopForeground(STOP_FOREGROUND_REMOVE)
            stopSelf()
            return
        }

        if (sbn == null || !isTtsReady) return
        if (sbn.id ==
NotificationHelper.SERVICE_NOTIFICATION_ID
&&
            sbn.packageName == this.packageName) {
            return
        }

        val extras = sbn.notification.extras
        val title =
extras.getString(Notification.EXTRA_TITLE) ?: ""
```

```
        val text =
extras.getCharSequence(Notification.EXTRA_TEX
T)?.toString() ?: ""
        val pkg = sbn.packageName

        val canSpeak = pipeline.all {
it.shouldProceed(sbn, settings) }

        if (!canSpeak){
            Log.d("TTS_DEBUG", "Notification from
$pkg blocked (filter)")
            return
        }
        if (isDuplicate(sbn, title, text)) {
            Log.d("DUPLICATE_DEBUG",
"Notification from $pkg blocked (duplicate)")
            return
        }
        Log.d("TTS_DEBUG", "Notification received.
isTtsReady: $isTtsReady")

        // Add notification to queue
        serviceScope.launch {

notificationInputChannel.send(RawNotification(titl
e, text, sbn, settings))
            processInputQueue()
        }
    }

    // Process notifications in queue
    private fun processInputQueue() {
        if (isInputProcessing) return
        isInputProcessing = true
        serviceScope.launch {
            for (raw in notificationInputChannel) {
                if (raw.title.isNotEmpty()) {
                    startSpeechTask()
                    identifyAndSpeak(raw.title, isTitle =
true)
                }
                if (raw.text.isNotEmpty()) {
                    startSpeechTask()
                    identifyAndSpeak(raw.text, isTitle =
false)
                }
            }
            isInputProcessing = false
        }
    }

    //Return notification if it was swiped
    override fun onNotificationRemoved(sbn:
StatusBarNotification?) {
        super.onNotificationRemoved(sbn)
        if (sbn?.id ==
NotificationHelper.SERVICE_NOTIFICATION_ID
&&
            sbn?.packageName == this.packageName) {

            Log.d("SERVICE_DEBUG", "Service
notification was swiped, returning it back")

            val settings = AppSettings(this)
            val helper = NotificationHelper(this)

            if (settings.isServiceEnabled) {
                startForeground(

NotificationHelper.SERVICE_NOTIFICATION_ID
,
                    helper.createServiceNotification()
                )
            }
        }
    }

    // Identifies language and makes TTS speak
    private fun identifyAndSpeak(content: String,
isTitle: Boolean, onComplete: (() -> Unit)? = null) {
        val textToIdentify =
getCleanTextForML(content)

languageIdentifier.identifyPossibleLanguages(textT
oIdentify)
            .addOnSuccessListener {
identifiedLanguages ->
                // 1. Get languages: either custom or
system
                val settings = AppSettings(this)
                val preferredCodes = if
(settings.useCustomLanguages) {
                    settings.customLanguages
                } else {
                    val userLocales =
Resources.getSystem().configuration.locales
                    val codes = mutableSetOf<String>()
                    for (i in 0 until userLocales.size()) {
codes.add(userLocales[i].language) }
                    codes
```

```
            }

            // 2. Create map of languages and
possibilities
            val confidenceMap =
identifiedLanguages.associate { it.languageTag to
it.confidence }
            val type = if (isTitle) "TITLE" else
"TEXT"
            Log.d("TTS_DEBUG", "[$type]
Preffered codes: $preferredCodes")
            Log.d("TTS_DEBUG", "[$type] All
possibilities: $confidenceMap")

            // 3. Intersect map with user languages
            val filteredMap = confidenceMap.filter {
preferredCodes.contains(it.key) }

            // 4. Select language from map with
highest probability
            val finalLanguageCode =
filteredMap.maxByOrNull { it.value }?.key
                ?: "en" // if nothing selected, select
english language

            var finalLocale =
Locale.forLanguageTag(finalLanguageCode)
            Log.i("TTS_DEBUG", "[$type]
Available: $filteredMap | Selected:
$finalLanguageCode")

            // 5. Check if language available
            val result =
tts?.isLanguageAvailable(finalLocale)
            if (result !=
TextToSpeech.LANG_AVAILABLE && result !=
TextToSpeech.LANG_COUNTRY_AVAILABLE)
{
                Log.e("TTS_DEBUG", "Language
$finalLanguageCode is NOT supported. Falling
back to default.")
                finalLocale = Locale.getDefault()
            }

            // 6. Normalize text
            val processedContent =
cleanContent(content, finalLocale)

            // 7. Speak

            // VertexAI TTS
            if (settings.useVertexAI) {
                addToAiQueue(processedContent,
finalLocale, isTitle, onComplete)
            }

            else{
            // Usual TTS

            tts?.setLanguage(finalLocale)
                settings.applyToTts(tts)

            // Unique marker
            val uniqueId =
content.hashCode().toString()

            tts?.playSilentUtterance(if (isTitle) 1200
else 1000, TextToSpeech.QUEUE_ADD,
"PAUSE_$uniqueId")
            tts?.speak(processedContent,
TextToSpeech.QUEUE_ADD, null, uniqueId)

        }}
        .addOnFailureListener { e ->
            Log.e("TTS_DEBUG", "Error ML Kit:
${e.message}")
            endSpeechTask()
        }
    }

    override fun onDestroy() {
        Log.e("FATAL_STOP", "!!! ONDESTROY
!!!")
        notificationInputChannel.cancel()
        speechChannel.cancel()
        serviceJob.cancel()
        serviceScope.cancel()



        tts?.stop()
        tts?.shutdown()

        synchronized(this) {
            currentAiAudioTrack?.let { track ->
                try {
                    track.stop()
                    track.release()
                } catch (e: Exception) {
                    Log.e("SERVICE_DEBUG", "Force
release failed: ${e.message}")
```

```
            }
        }
        currentAiAudioTrack = null
    }

    synchronized(this) {
        if (activeTtsTasks > 0) {
            activeTtsTasks = 0
            abandonDucking()
            releaseWakeLock()
        }
    }

    super.onDestroy()
}
// Function to detect duplicate notifications
private fun isDuplicate(sbn:
StatusBarNotification, title: String, text: String):
Boolean {
    val pkg = sbn.packageName
    val currentTime = System.currentTimeMillis()
    val contentKey = "$pkg|$title|$text"
    val lastTime = duplicateCache.get(contentKey)
    if (lastTime != null) {
        val timeDiff = currentTime - lastTime
        if (timeDiff < 5000) {
            return true
        }
    }
    duplicateCache.put(contentKey, currentTime)
    return false
}

/* Cleans text from non-verbal tokens for better
language identification
-link
 */
private fun getCleanTextForML(text: String):
String {
    val urlPattern =
android.util.Patterns.WEB_URL.toRegex()

    // Delete links
    val cleanText = text.replace(urlPattern,
"").trim()

    // If nothing left return original
    return cleanText.ifEmpty { text }
}

/* Converts non-verbal tokens into localized
spoken words:
-link
*/
private fun cleanContent(content: String, locale:
Locale): String {
    val config =
Configuration(resources.configuration)
    config.setLocale(locale)
    val localizedContext =
createConfigurationContext(config)
    val replacement = try {
        localizedContext.getString(R.string.link)
    } catch (e: Exception) {
        "link"
    }
    val urlPattern =
android.util.Patterns.WEB_URL.toRegex()
    return content.replace(urlPattern, replacement)
}

// Add and process queue of AI TTS audio tracks
private fun addToAiQueue(text: String, locale:
Locale, isTitle: Boolean, onDone: (() -> Unit)? =
null) {
    serviceScope.launch {
        speechChannel.send(AiTask(text, locale,
isTitle, onDone))
        ensureQueueIsRunning()
    }
}

// Additional object for queue
data class AiTask(
    val text: String,
    val locale: Locale,
    val isTitle: Boolean,
    val onDone: (() -> Unit)?
)

// Keep queue in shape
private fun ensureQueueIsRunning() {
    if (isQueueStarted) return
    isQueueStarted = true

    serviceScope.launch {
        try {
            for (task in speechChannel) {
                val (text, locale, isTitle, onDone) = task
                try {
```

```
VertexAiClient.fetchAudioStreaming(
                    text, locale,
this@NotificationService,
AppSettings(this@NotificationService)
                ) { bytes ->
                    if (!coroutineContext.isActive) {
                        throw
CancellationException("Service is stopping")
                    }
                    if (currentAiAudioTrack == null)
{
                        currentAiAudioTrack =
createStreamingAudioTrack(bytes.size)
                        runBlocking {
                            if (isTitle) delay(1200)
                            else delay(1000)
                        }
                        currentAiAudioTrack?.play()
                    }

currentAiAudioTrack?.write(bytes, 0, bytes.size)
                }

            } catch (e: Exception) {
                Log.e("AI_DEBUG", "Streaming
error: ${e.message}")
            } finally {
                try{
                    currentAiAudioTrack?.stop()
                    currentAiAudioTrack?.release()
                    currentAiAudioTrack = null
                }
                catch (e: Exception){
                    Log.e("AI_DEBUG", "Streaming
error: ${e.message}")
                    currentAiAudioTrack = null
                }
                endSpeechTask()
                onDone?.invoke()
            }
        }
    } finally {
        isQueueStarted = false
    }
  }
}

  // Make it possible to play LINEAR16 format of
AI TTS audio
  private fun
createStreamingAudioTrack(initialChunkSize: Int):
AudioTrack {
    val sampleRate = 24000
    val minBuffer =
AudioTrack.getMinBufferSize(
      sampleRate,
      AudioFormat.CHANNEL_OUT_MONO,
      AudioFormat.ENCODING_PCM_16BIT
    )

    return AudioTrack.Builder()

.setAudioAttributes(AudioAttributes.Builder()

.setUsage(AudioAttributes.USAGE_ASSISTANT)

.setContentType(AudioAttributes.CONTENT_TYP
E_SPEECH)
          .build())
        .setAudioFormat(AudioFormat.Builder()

.setEncoding(AudioFormat.ENCODING_PCM_16
BIT)
          .setSampleRate(sampleRate)

.setChannelMask(AudioFormat.CHANNEL_OUT_
MONO)
          .build())
        .setBufferSizeInBytes(max(minBuffer * 2,
initialChunkSize * 4))

.setTransferMode(AudioTrack.MODE_STREAM)
        .build()
  }

  // Audio Ducking
  private fun requestDucking(): Boolean {
    return if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.O) {
        focusRequest =
AudioFocusRequest.Builder(AudioManager.AUDI
OFOCUS_GAIN_TRANSIENT_MAY_DUCK)
          .setAudioAttributes(
            AudioAttributes.Builder()

.setUsage(AudioAttributes.USAGE_MEDIA)

.setContentType(AudioAttributes.CONTENT_TYP
E_SPEECH)
```

```
                .build())
            .build()

audioManager.requestAudioFocus(focusRequest!!)
==
AudioManager.AUDIOFOCUS_REQUEST_GRA
NTED
        } else {
            @Suppress("DEPRECATION")
            audioManager.requestAudioFocus(null,
AudioManager.STREAM_MUSIC,
AudioManager.AUDIOFOCUS_GAIN_TRANSIE
NT_MAY_DUCK) ==
AudioManager.AUDIOFOCUS_REQUEST_GRA
NTED
        }
    }

    private fun abandonDucking() {
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.O) {
            focusRequest?.let {
audioManager.abandonAudioFocusRequest(it) }
        } else {
            @Suppress("DEPRECATION")
            audioManager.abandonAudioFocus(null)
        }
    }

    // Wakelocks
    private fun acquireWakeLock() {
        synchronized(this) {
            if (!wakeLock.isHeld) {
                wakeLock.acquire(10 * 60 * 1000L) // 10
Minutes
                Log.d("POWER_DEBUG", "WakeLock
ACQUIRED")
            }
        }
    }

    private fun releaseWakeLock() {
        synchronized(this) {
            if (wakeLock.isHeld) {
                wakeLock.release()
                Log.d("POWER_DEBUG", "WakeLock
RELEASED")
            }
        }
    }

    // Task management
    private fun startSpeechTask() {
        synchronized(this) {
            activeTtsTasks++
            Log.d("TTS_DEBUG", "Task started.
Current tasks: $activeTtsTasks")
            requestDucking()
            acquireWakeLock()
        }
    }

    private fun endSpeechTask() {
        synchronized(this) {
            if (activeTtsTasks > 0) {
                activeTtsTasks--
                Log.d("TTS_DEBUG", "Task ended.
Remaining tasks: $activeTtsTasks")
            } else {
                Log.w("TTS_DEBUG", "endSpeechTask
called, but counter was already 0")
            }

            if (activeTtsTasks == 0) {
                abandonDucking()
                releaseWakeLock()
                Log.d("TTS_DEBUG", "All tasks
finished. Ducking released.")
            }
        }
    }
    private fun applyAudioAttributesToLocalTts() {
        if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.LOLLIPOP) {
            val attr = AudioAttributes.Builder()

.setUsage(AudioAttributes.USAGE_ASSISTANT)

.setContentType(AudioAttributes.CONTENT_TYP
E_SPEECH)
                .build()

            tts?.setAudioAttributes(attr)
        }
    }

}
```

ДОДАТОК В. ПЕРЕЛІК ТЕСТІВ

Таблиця В.1

Перелік тестів

Тест (перевірка)	Емулятор	Пристрій
Увімкнення без дозволів	+	+
Увімкнення з дозволами	+	+
Повернення дозволів після увімкнення	+	+
Озвучення (локальний TTS)	+	+
Озвучення (локальний TTS, але в системі обраний інший)	-	+
Озвучення (генеративний TTS)	+	+
Озвучення без інтернету (генеративний TTS)	+	+
Сервісне сповіщення	+	+
Тестове сповіщення пусте	+	+
Тестове сповіщення змінене	+	+
Вимкнення сервісу під час озвучення (локальний TTS)	+	+
Вимкнення сервісу під час озвучення (генеративний TTS)	+	+
Перезапуск смартфона	+	+
Вимкнення додатку (сервіс все ще увімкнений)	+	+
Вибір мови (одна системна мова)	+	+
Вибір мови (дві системні мови)	+	+
Вибір мови (одна системна мова, друга відсутня в системі)	+	+
Вибір мови (одна мова відсутня в системі)	+	+
Фільтр заблокованого екрану (екран вимкнено)	+	+

Продовження таблиці В.1

Тест (перевірка)	Емулятор	Пристрій
Фільтр заблокованого екрану (екран ввімкнено, телефон заблоковано)	+	+
Фільтр Bluetooth-пристроїв	-	+
ФІльтр підсумків	+	+
Пошук у AppListActivity	+	+
Керування користувацькими мовами	+	+
Вибір мови (одна користувацька мова)	+	+
Вибір мови (дві користувацькі мови)	+	+
Вибір мови (одна користувацька мова, друга відсутня в списку)	+	+
Вибір мови (одна мова відсутня в списку)	+	+
Вибір мови (одна мова відсутня в списку і пакет англійської не встановлено)	+	+
Керування Bluetooth-пристроями з дозволом	-	+
Керування Bluetooth-пристроями без дозволу	-	+
Керування Bluetooth-пристроями з вимкненим Bluetooth	-	+
Зміна голосу локального TTS	+	+
Зміна швидкості мовлення локального TTS	+	+
Зміна висоти голосу локального TTS	+	+
Скидання до заводських значень швидкості мовлення та висоти голосу локального TTS	+	+
Озвучення у беззвучному режимі	-	+
Зміна голосу генеративного TTS	+	+
Зміна стилю генеративного TTS	+	+
Значення за замовчуванням	+	+