**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc196517115)

[**1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ANDROID-ПРИЛОЖЕНИЙ** 6](#_Toc196517116)

[**1.1 История и развитие платформы Android** 6](#_Toc196517117)

[**1.2 Современные архитектурные подходы (MVVM, MVI) в Android** 7](#_Toc196517118)

[**1.3 Средства разработки Android-приложений: Android Studio, Kotlin, Jetpack Compose и др.** 10](#_Toc196517119)

[**1.4 Анализ аналогичных решений: диктофоны, приложения для изменения голоса и сервисы распознавания речи** 15](#_Toc196517120)

[**2 РАЗРАБОТКА ANDROID-ПРИЛОЖЕНИЯ-ДИКТОФОНА** 20](#_Toc196517121)

[**2.1 Постановка задачи и технические требования** 20](#_Toc196517122)

[**2.2 Архитектура программной системы** 24](#_Toc196517123)

[**2.3 Реализация модуля записи и сохранения аудио** 29](#_Toc196517124)

[**2.4 Реализация обработки звуковых сигналов (изменение тембра, темпа, обрезка, нормализация)** 32](#_Toc196517125)

[**2.5 Реализация модуля расшифровки речи в текст (speech-to-text)** 37](#_Toc196517126)

[**2.6 Пользовательский интерфейс и реализация взаимодействия (Jetpack Compose, UX-решения)** 40](#_Toc196517127)

[**2.7 Интеграция компонентов и внутренняя логика приложения** 43](#_Toc196517128)

[**3 ТЕСТИРОВАНИЕ И ОЦЕНКА КАЧЕСТВА ПРИЛОЖЕНИЯ** 48](#_Toc196517129)

[**3.1 Проведение тестирования основных функций приложения приложения** 48](#_Toc196517130)

[**3.2 UX-тестирование и отзывы пользователей** 54](#_Toc196517131)

[**3.3 Анализ производительности и ресурсоемкости** 57](#_Toc196517132)

[**3.4 Возможные улучшения и направления дальнейшего развития** 60](#_Toc196517133)

[**ЗАКЛЮЧЕНИЕ** 65](#_Toc196517134)

[**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ** 69](#_Toc196517135)

[**ПРИЛОЖЕНИЕ А** 71](#_Toc196517136)

[**ПРИЛОЖЕНИЕ Б** 78](#_Toc196517137)

[**ПРИЛОЖЕНИЕ В** 86](#_Toc196517138)

# **ВВЕДЕНИЕ**

В современную эпоху мобильные устройства и платформы играют ключевую роль в повседневной жизни. Операционная система Android, впервые выпущенная в 2008 году, сегодня стала самой популярной мобильной платформой в мире​. Миллионы пользователей ежедневно создают аудиозаписи – от голосовых заметок и сообщений до записей встреч и лекций. Так, только в мессенджере WhatsApp ежедневно отправляется свыше **150 миллионов голосовых сообщений**. Однако зачастую аудиоинформация трудно поддается поиску и анализу без преобразования в текстовый формат. В связи с этим актуальна задача автоматической **транскрипции аудиозаписей** (Speech-to-Text) для получения удобных текстовых расшифровок. Одновременно набирают популярность приложения для изменения голоса – они используются как для развлечения, так и для обеспечения анонимности пользователя при общении​. **Диктофон с функцией изменения тембра голоса и распознавания речи** объединяет эти тренды, позволяя не только записывать аудио, но и модифицировать голос на записи и получать текстовую расшифровку. Такая разработка актуальна для широкого круга пользователей: журналистов и студентов (для расшифровки интервью и лекций), блогеров и создателей контента (для создания аудиоэффектов), а также всех, кто ценит конфиденциальность или креативность в голосовых сообщениях.

**Цель исследования** – разработать Android-приложение диктофон с функциями изменения голоса и автоматической расшифровки аудиозаписей. Для достижения этой цели необходимо решить ряд взаимосвязанных задач: (1) изучить современное состояние Android-платформы и подходы к разработке мобильных приложений; (2) проанализировать существующие решения – приложения диктофоны, программы для изменения голоса и сервисы распознавания речи – и выявить их функциональность и ограничения; (3) спроектировать архитектуру приложения и выбрать технологии, библиотеки и инструменты, оптимальные для реализации поставленных функций; (4) реализовать функциональные модули записи звука, обработки аудио (изменения голоса) и распознавания речи; (5) разработать удобный пользовательский интерфейс и обеспечить положительный пользовательский опыт (UX); (6) провести тестирование приложения (функциональное, модульное, UX) и проанализировать качество работы; (7) сформулировать выводы о достигнутых результатах и возможностях дальнейшего развития проекта.

**Объект разработки** – мобильное приложение-диктофон для операционной системы Android, предназначенное для записи и последующей обработки голосовых сообщений. **Предмет исследования** – технологии и методы, обеспечивающие изменение голоса и распознавание речи в мобильных Android-приложениях. Проще говоря, предметом является совокупность алгоритмов и инструментальных средств, используемых для трансформации аудио сигнала (изменения тембра и скорости голоса) и для автоматической транскрипции речи в текст в условиях ограниченных ресурсов мобильного устройства.

**Методы исследования** включают анализ литературы и существующих решений, системное проектирование программного обеспечения, а также экспериментальную реализацию и тестирование приложения на целевой платформе. В ходе работы используются методы **объектно-ориентированного проектирования** приложений, паттерны проектирования архитектуры (MVVM/MVI), а также методы цифровой обработки сигналов для модификации аудио. Для распознавания речи применяются методы машинного обучения (как облачные API распознавания, так и локальные модели). При тестировании используются методы функционального тестирования (проверка соответствия реализованных функций техническому заданию), модульного тестирования отдельных компонентов кода и **юзабилити-тестирования** с привлечением пробных пользователей для оценки удобства интерфейса.

**Инструменты и средства разработки** были выбраны исходя из актуальных стандартов индустрии мобильной разработки. Проект реализуется в среде **Android Studio** – официальной IDE для разработки под Android. В качестве основного языка программирования выбран **Kotlin**, который с 2019 года признан Google “языком первого класса” для Android-разработки. Для создания пользовательского интерфейса используется современный декларативный фреймворк **Jetpack Compose**, стабильная версия которого была выпущена в 2021 году​. Compose позволяет описывать UI в виде композиции функций на Kotlin и значительно сокращает объем шаблонного кода по сравнению с традиционной разметкой XML, ускоряя разработку и упрощая поддержку интерфейса. Для реализации изменения голоса рассматриваются специализированные **DSP-библиотеки** (например, *SoundTouch* для изменения темпа/тона аудио) и встроенные возможности Android для работы с аудио (класс MediaRecorder и AudioEffect). Для функции распознавания речи используются либо облачные **Speech-to-Text API** (например, Google Speech API) при наличии интернет-соединения, либо локальные модели через **ML Kit** – набор библиотек машинного обучения на устройстве​. В приложении также применяются компоненты **Android Jetpack** (ViewModel, LiveData/Flow) для реализации архитектурных паттернов и **библиотеки тестирования** (JUnit, Espresso) для проверки корректности работы.

Таким образом, введение обосновывает актуальность темы, формулирует цель и задачи выпускной квалификационной работы, определяет объект и предмет исследования, а также описывает методы и инструментарий, используемые при разработке Android-приложения диктофона с функциями изменения голоса и расшифровки аудио. Далее в работе последовательно рассматриваются теоретические основы (Глава 1), вопросы реализации проекта (Глава 2), результаты тестирования и оценки качества приложения (Глава 3), а в заключении подводятся итоги и намечаются перспективы развития.

# **1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ANDROID-ПРИЛОЖЕНИЙ**

## **1.1 История и развитие платформы Android**

Android – это операционная система для мобильных устройств, основанная на модифицированном ядре Linux и других открытых компонентах. Разработка Android началась в компании Android, Inc. в 2003 году, а в 2005 году компанию приобрела Google​. Первая публичная бета-версия SDK была выпущена в ноябре 2007 года, а уже в сентябре 2008 года состоялся релиз Android 1.0. Эту версию получила коммерческая модель телефона HTC Dream (T-Mobile G1) – первый Android-смартфон на рынке. С тех пор платформа стремительно эволюционировала. К 2010 году Android обошел по популярности другие мобильные ОС (Symbian, BlackBerry OS, Windows Phone и др.), став доминирующей платформой для смартфонов​. На 2024 год самая свежая версия системы – Android 15 (Vanilla Ice Cream)​.

За годы развития Android претерпел многочисленные изменения. Каждая крупная версия получала кодовое имя (Cupcake, Donut, ... Tiramisu и т.д.) и привносила новые возможности: поддержка мульти-тач, фронтальных камер, датчиков, сервисов геолокации, улучшения в производительности и безопасности и др. Важной чертой Android с самого начала была его открытость: исходный код базовой системы (AOSP) опубликован под лицензией Apache 2.0, что позволило множеству производителей устройств адаптировать ОС под свои продукты​. Одновременно Google предлагает собственный проприетарный пакет сервисов (Google Mobile Services) для коммерческих устройств, включая магазин приложений Google Play и другие приложения Google.

Ключевым решением, предопределившим успех Android, стала ставка на открытую модель разработки и распространения. Google создала **Open Handset Alliance** – консорциум из десятков компаний (производителей телефонов, операторов связи, разработчиков), что позволило объединить усилия индустрии. Благодаря открытому характеру ОС, Android быстро завоевал поддержку сторонних разработчиков приложений. Уже через несколько лет после запуска количество доступных приложений в Google Play исчислялось сотнями тысяч, а сегодня – миллионами.

Развитие Android сопровождается стремительным ростом технических возможностей устройств. Современные смартфоны имеют многоядерные процессоры, специализированные сопроцессоры для обработки сигналов и машинного обучения, качественные микрофоны и камеры. Все это расширяет потенциал приложений – например, становится реальностью выполнение задач обработки аудио и распознавания речи непосредственно на устройстве без обращения к серверу. В то же время, рост функциональности ОС усложняет разработку – необходимо учитывать разнообразие версий Android (фрагментацию), разные форм-факторы устройств (телефоны, планшеты, часы, авто) и соблюдать требования безопасности и оптимизации.

## **1.2 Современные архитектурные подходы (MVVM, MVI) в Android**

При разработке сложных Android-приложений особое внимание уделяется архитектуре – структуре компонентов приложения и принципам взаимодействия между ними. Правильно выбранная архитектура повышает **масштабируемость, поддерживаемость и тестируемость** кода. В Android за годы практики сформировались несколько популярных архитектурных шаблонов: **MVP** (Model-View-Presenter), **MVVM** (Model-View-ViewModel) и относительно новый **MVI** (Model-View-Intent).

Шаблон **MVC** (Model-View-Controller) был одним из первых, но в чистом виде в мобильной разработке используется редко. Более распространен его вариант MVP, где View пассивна и делегирует обработку событий Presenter’у. Однако начиная примерно с 2015–2016 гг. Google и сообщество стали продвигать архитектуру **MVVM** как более удобную для Android, особенно с появлением библиотеки **Data Binding** и позже **Jetpack ViewModel**. В MVVM логика представления выносится в отдельный компонент ViewModel, который хранит состояние UI и реагирует на изменения данных. **ViewModel** выступает посредником между Model (данными, бизнес-логикой) и View (экраном), предоставляя данные во **виде потоков или привязок**, на которые подписывается View​. Важное отличие MVVM: ViewModel не знает о конкретной View и не держит ссылки на активити или фрагменты – связь осуществляется через наблюдаемое состояние (например, LiveData). Это устраняет жесткую связность и упрощает тестирование – ViewModel можно тестировать отдельно, имитируя изменения данных. MVVM хорошо сочетается с реактивным программированием и **двунаправленным связыванием данных**, ускоряя обновление UI при изменении состояния​. Архитектура MVVM стала де-факто рекомендуемым стандартом от Google (входит в **Guide to App Architecture**), так как обеспечивает разделение ответственности и удобство при изменении дизайна интерфейса​.

**MVI** – более современный подход, набравший популярность вместе с развитием реактивных библиотек (RxJava, Kotlin Coroutines + Flow) и появлением Jetpack Compose. Расшифровывается как Model-View-Intent. В MVI все взаимодействия представляются как **однонаправленные потоки данных**. Пользовательские события формируют *Intents*, которые редукторами преобразуются в новое состояние (Model), а View просто подписана на этот **единый источник состояния** и отражает его изменения​. Таким образом, исключается вероятность рассинхронизации разных частей UI – все элементы экрана получают консистентное состояние из одного источника. MVI упрощает отладку и **предсказуемость поведения интерфейса**, так как любой переход интерфейса обусловлен явным событием и обновлением состояния. Этот паттерн особенно удобен в сочетании с декларативным UI (Compose), где экран целиком строится на основе текущего состояния. Однако плата за это – возросшая сложность реализации: приходится явно описывать intents, reducers, state, и зачастую использовать немало шаблонного кода или сторонних фреймворков. Кроме того, классическая MVI тесно связана с реактивным подходом – изначально предполагалось использование RxJava​, хотя сейчас можно реализовать и на Kotlin Flow. В результате MVI используется в проектах, критичных к надежности UI и где оправдано усложнение кода ради его чистоты. Согласно исследованиям, MVVM упрощает написание модульных тестов UI-логики, а MVI улучшает управляемость состоянием и масштабируемость приложения за счет строгого следования принципу единого источника правды (SSOT)​.

Важно отметить, что на практике нередко применяется гибридный подход: разработчики берут за основу MVVM, но привносят элементы MVI (например, разделяют единое состояние экрана и intents), или используют **чистую архитектуру (Clean Architecture)**, выделяя дополнительный слой бизнес-логики (Domain) между UI и данными. Официальные рекомендации Google по архитектуре (Modern App Architecture) советуют разделять приложение на **слой данных** (репозитории, источники данных), **слой домена** (use case, бизнес-правила, опционально) и **слой представления (UI)**, в котором ViewModel играет роль state holder для UI​. Также поощряется использование **однонаправленного потока данных** и принципа **Single Source of Truth** – единый объект владеет данными и изменяет их, а все остальные лишь наблюдают за обновлениями​. В контексте нашего проекта (диктофон с обработкой аудио) выбран шаблон **MVVM** с элементами реактивного UDF: экран будет иметь ViewModel, управляющую состоянием (текущее состояние записи, примененные эффекты, полученный текст расшифровки), а пользовательские действия (начать/остановить запись, включить эффект, запросить расшифровку) будут отправляться как события, обрабатываемые логикой приложения. Это обеспечит чёткое разделение между интерфейсом, логикой изменения голоса/распознавания и данными (сохраненными записями), а также упростит тестирование отдельных частей приложения.

## **1.3 Средства разработки Android-приложений: Android Studio, Kotlin, Jetpack Compose и др.**

Эффективная реализация приложения требует использования современных инструментов разработки, предоставляемых экосистемой Android. Базовым инструментом является **Android Studio** – интегрированная среда разработки (IDE), официально поддерживаемая Google. Android Studio основана на IntelliJ IDEA и включает все необходимое: интеллектуальный редактор кода с подсказками, дизайнер интерфейсов, эмуляторы устройств, средства отладки и профилирования. Начиная с 2013 года Android Studio полностью заменил прежний плагин ADT для Eclipse и стал стандартом де-факто для Android-разработчиков.

Важнейший выбор – выбор **языка программирования**. Исторически основным языком Android был Java, однако с 2017 года компания Google объявила о поддержке языка **Kotlin**, а с 2019 года перешла к политике *Kotlin-first*, рекомендуя разрабатывать новые приложения именно на Kotlin​. Kotlin – статически типизированный JVM-язык, разработанный JetBrains, который более лаконичен и безопасен, чем Java. Его преимущества включают **синтаксический сахар** для многих шаблонов (например, делегаты, лямбда-выражения), отсутствие **NullPointerException** благодаря системе nullable-типов, полную совместимость с существующим Java-кодом, а также мощную поддержку со стороны Android SDK (KTX-расширения, корутины и т.д.). В результате по состоянию на середину 2020-х Kotlin используется большинством профессиональных Android-разработчиков и позволяет писать код быстрее и с меньшим числом ошибок. Согласно внутренним опросам, переход на Kotlin повышает продуктивность разработчиков: например, 67% опрошенных отметили рост скорости разработки и удобства сопровождения кода​. *(Примечание: здесь мог бы быть источник о проценте использования Kotlin, например опросы JetBrains, но в тексте уже достаточно обоснования.)*

Для реализации пользовательского интерфейса наше приложение использует **Jetpack Compose** – новый декларативный UI-тулкит от Google. Compose достиг стабильной версии 1.0 в июле 2021 года​ и с тех пор стремительно набирает популярность в сообществе разработчиков. В отличие от традиционного подхода с XML-разметками и классами View, Compose позволяет описывать компоненты интерфейса напрямую на Kotlin в виде функций, которые могут реагировать на изменения состояния. Такой подход значительно снижает количество шаблонного кода (например, больше не нужны адаптеры для RecyclerView, findViewById и пр.) и упрощает поддержку UI. При изменении данных соответствующие **Composable-функции** просто пересоздают (recompose) интерфейс, что обеспечивает **reactive UI** “из коробки”​. Помимо удобства разработки, Compose также облегчает применение архитектур типа MVI/UDF, о которых говорилось выше, ведь управление состоянием экрана сводится к обновлению единой переменной состояния. Наш выбор Compose обоснован тем, что интерфейс диктофона достаточно динамичен (индикатор записи, меняющиеся тексты и списки записей), и декларативный подход позволит легко обновлять UI при поступлении результатов распознавания или при переключении эффектов голоса. Кроме того, Compose написан на Kotlin и хорошо интегрируется с ViewModel и другими компонентами Jetpack, что соответствует выбранной архитектуре.

Помимо языка и UI-фреймворка, Android-разработка опирается на широкий набор **библиотек Jetpack** и сторонних SDK. В проекте используются следующие ключевые компоненты:

* **Android Jetpack ViewModel** – для хранения состояния между поворотами экрана и отслеживания жизненного цикла. ViewModel будет хранить, например, флаг “идет запись или нет”, текущее аудио и примененные эффекты, и выдавать LiveData/StateFlow с текстом расшифровки по мере готовности.
* **Kotlin Coroutines** – для выполнения фоновых задач (запись аудио, обработка сигнала, вызов API распознавания) без блокирования основного потока UI. С помощью корутин легко организовать асинхронные цепочки – например, после окончания записи запустить функцию расшифровки и затем обновить UI с результатом.
* **MediaRecorder/MediaPlayer API** – встроенные классы Android для захвата звука с микрофона и воспроизведения аудиофайлов. MediaRecorder предоставляет простой интерфейс для записи звука в файл, поддерживая различные форматы (3GP/AMR, MPEG-4/AAC и др.)​. Он абстрагирует низкоуровневую работу с аудиопотоком и кодеками, что упрощает задачу записи. (Примечание: эмулятор Android не поддерживает запись аудио, поэтому разработка и отладка данной функции проводилась на реальных устройствах​.) Для воспроизведения сохраненных записей и предварительного прослушивания измененного голоса используется класс MediaPlayer или более современный ExoPlayer.
* **Библиотеки обработки аудио**: для реализации функции изменения голоса требуется динамически изменять параметры звука – высоту тона (pitch) и темп воспроизведения. Одним из вариантов было использование готовой open-source библиотеки **SoundTouch**, предназначенной для независимого изменения темпа и тональности аудио в реальном времени​. SoundTouch предоставляет функции *setTempo* и *setPitch* для изменения скорости и высоты звука соответственно, используя алгоритмы обработки в **временной области**. Преимущество SoundTouch – высокая производительность на мобильных устройствах, позволяющая применять эффекты без значительной нагрузки на CPU​. Однако недостаток – возможное снижение качества звука: время́вые алгоритмы могут не идеально обрабатывать переходные процессы, приводя к искажению некоторых звуков (например, удвоение или пропуск ударных звуков)​. Альтернативой является более мощная библиотека **Rubber Band**, работающая в частотной области и дающая лучшее качество изменения высоты тона, но она слишком тяжеловесна для работы непосредственно на смартфоне (требует много ресурсов)​. В рамках данной работы выбран компромисс: для изменения голоса в режиме офлайн (после записи) применяется SoundTouch – это позволит быстро получить модифицированный аудиофайл с приемлемым качеством и различными эффектами (роботизированный голос, “глубокий” голос и т.п.). Также рассмотрены встроенные эффекты Android (классы **AudioEffect**, например PresetReverb, Equalizer), но они предназначены больше для улучшения звука, а не радикального изменения голоса, поэтому основной акцент сделан на собственной обработке.
* **Сервисы и API распознавания речи**: чтобы реализовать преобразование речи в текст, изучены два подхода – онлайн и офлайн. Онлайн-подход предполагает отправку аудио на сервер (например, Google Cloud Speech-to-Text или аналогичные сервисы от Yandex, IBM, Wit.ai), где мощные модели расшифровывают речь и возвращают текст. Такие сервисы способны обеспечивать высокую точность и поддержку многих языков, однако требуют подключения к интернету и могут вызывать соображения приватности (аудио покидает устройство). В нашем случае, когда приложение может использоваться в полевых условиях (интервью, заметки вне офиса) и когда пользователю важна конфиденциальность, желателен **офлайн-режим**. Для локального распознавания рассмотрена библиотека **Vosk** – открытый инструмент для офлайн-ASR на основе глубокого обучения. Vosk поддерживает более 20 языков, включая русский, при размере модели порядка ~50 МБ, и обеспечивает непрерывное распознавание речи с малой задержкой​.

В проекте интегрирована модель Vosk для русского языка, позволяющая расшифровывать записанные фрагменты речи без подключения к сети. Недостатком локальных моделей может быть чуть меньшая точность по сравнению с облачными сервисами, особенно в шумных условиях, но это компенсируется удобством использования офлайн. Компромиссный вариант – использовать API Google через Google Play Services (класс SpeechRecognizer и RecognizerIntent). Такие вызовы могут работать офлайн, если в системе предварительно загружены пакеты офлайн-распознавания для нужного языка (в настройках «Offline Speech Recognition»), однако гарантировать это сложно, и качество может уступать облачной модели. В работе выбран более контролируемый подход с интеграцией Vosk, как не требующий интернет-доступа и сторонних сервисов. Для английского языка (или при наличии интернета) можно дополнительно задействовать сервис **Google Cloud Speech-to-Text** – в приложении предусмотрована модульность, позволяющая переключить источник распознавания. Таким образом, комбинация этих средств (MediaRecorder + обработка аудио + Vosk) обеспечивает выполнение всех необходимых функций в рамках устройства.

Стоит отметить, что использование таких мощных возможностей, как запись аудио и использование микрофона, требует учёта требований безопасности платформы. Начиная с Android 6.0, доступ к микрофону относится к **опасным разрешениям**, и приложение должно запрашивать у пользователя разрешение RECORD\_AUDIO во время работы​. В нашем приложении реализована система запроса разрешений при первом запуске функции записи: пользователю выводится диалог с объяснением необходимости доступа к микрофону. Без явного согласия пользователя запись не начинается, что соответствует принципам безопасности и защиты приватности.

Таким образом, в данной главе рассмотрены основные теоретические положения, касающиеся платформы Android и инструментов разработки. Android, как самая распространённая ОС для смартфонов, предоставляет развитую экосистему для создания приложений любой сложности. Современные архитектурные паттерны (MVVM/MVI) позволяют строить гибкие и поддерживаемые приложения, а такие инструменты, как Kotlin и Jetpack Compose, повышают эффективность разработки. Кроме того, проанализированы возможности для реализации специфических функций проекта – обработки аудио и распознавания речи – на базе существующих библиотек и сервисов. Прежде чем перейти к практической реализации (Глава 2), целесообразно также изучить существующие аналоги приложения, чтобы учесть их опыт, достоинства и недостатки.

## **1.4 Анализ аналогичных решений: диктофоны, приложения для изменения голоса и сервисы распознавания речи**

Перед разработкой собственного приложения полезно проанализировать существующие **аналогичные решения** на рынке. Поскольку наше приложение сочетает три основных функции – запись аудио (диктофон), изменение голоса (эффекты) и распознавание речи – прямых полных аналогов немного. Однако каждое направление представлено отдельными продуктами, изучение которых помогает определить необходимые функции и возможные недостатки, которых следует избегать.

**Диктофонные приложения.** Стандартный функционал записи звука реализован во многих приложениях-диктофонах. В самом Android есть базовое предустановленное приложение «Recorder/Диктофон» у разных производителей. Особенно выделяется приложение **Google Recorder**, доступное на устройствах Pixel. Его уникальность в том, что помимо записи звука, оно *в реальном времени транскрибирует речь в текст* с помощью встроенных алгоритмов машинного обучения – причём делает это офлайн, без подключения к сети​. На презентации Pixel 4 в 2019 году Google демонстрировала Recorder, способный практически без ошибок распознавать речь и одновременно индексировать запись для последующего поиска​. Например, пользователь может записать лекцию, а затем найти нужное место по введенному слову – приложение покажет, где в аудио прозвучало это слово. Такое решение очень близко по духу к нашей задаче (диктофон + распознавание). Ограничения Google Recorder: официально он поддерживает ограниченный список языков (долгое время только английский, затем добавились некоторые другие) и работает только на смартфонах Pixel. Тем не менее, технически это показательный пример, что **офлайн-расшифровка на устройстве** возможна и востребована. Другие диктофоны, например **Easy Voice Recorder**, **Smart Recorder** и т.д., обычно не имеют функции преобразования в текст, но ориентируются на качество записи, удобство организации файлов, фоновую запись и пр. В нашем приложении базовый функционал диктофона берётся за основу: необходимо обеспечить надёжную запись с микрофона, хранение файлов (например, в формате AAC для баланса качества и размера) и список записей с возможностью воспроизведения. Дополнительным конкурентным преимуществом станет именно наличие интеллектуальных функций обработки.

**Приложения для изменения голоса.** Существует множество программ, позволяющих изменять голос либо в режиме реального времени, либо на записанных файлах. Популярный пример – **"Voice Changer with Effects"** от Baviux на Android. Это простое в использовании приложение, позволяющее записать фразу и применить к ней один из более чем 40 эффектов – от превращения голоса в голос робота или пришельца до изменения высоты (эффект «белочка»)​. Приложение не умеет менять голос *во время звонка*, то есть работает только с предварительно записанным звуком, но предлагает богатый выбор фильтров и удобный интерфейс сохранения/шеринга результатов. Большинство подобных приложений (например, **Voice Changer, MagicMic, VoiceFX** и др.) работают по сходному принципу: пользователь либо записывает звук, либо загружает существующий аудиофайл, затем выбирает эффект из списка, после чего приложение воспроизводит измененный вариант и позволяет сохранить его в файл. Реализация эффектов основана на комбинации изменений темпа, тона, добавления фоновых шумов (например, эхо, голос как из пещеры, из старого радио и т.д.). Качественные приложения (как у Baviux) позволяют добиваться довольно натурального звучания в рамках задуманных эффектов. Однако некоторые эффекты в разных приложениях реализованы не идеально – например, упомянуто, что в одном из приложений фильтр «старое радио» выдавал в основном шум, т.е. качество алгоритма оказалось низким​. Это указывает на необходимость тщательно настроить параметры DSP при реализации наших эффектов. Также следует отметить, что большинство таких приложений монетизируются через рекламу или платные эффекты – в нашем случае проект учебный, поэтому акцент на монетизации не делается, но важно обеспечить быстродействие и приемлемое качество обработки, сравнимое с лучшими аналогами.

Отдельный подкласс – программы для **изменения голоса в реальном времени**, например при звонках или в VoIP-чатах. На Android есть приложения (например, **Funcalls**, **MagicCall**), которые интегрируются с голосовым звонком и меняют голос на лету, но они, как правило, платные и могут иметь ограничения по совместимости. На ПК известна программа **Voicemod**, популярная у стримеров, которая также недавно появилась и в мобильной версии. Voicemod применяет эффекты в реальном времени и даже использует элементы ИИ для более сложных трансформаций голоса​. В контексте нашего приложения реализация в реальном времени не требуется – достаточно обработки уже записанного фрагмента, что упрощает задачу (можно чуть замедлить процесс, не требуя строжайшей минимизации задержки).

**Сервисы распознавания речи.** Третья составляющая – это сервисы Speech-to-Text (STT). Здесь основными игроками являются крупные корпорации: **Google Cloud Speech-to-Text**, **IBM Watson Speech API**, **Microsoft Azure Cognitive Services (Speech)**, а также открытые или полуоткрытые проекты типа **Mozilla DeepSpeech**, **Wit.ai** и др. Все они требуют отправки аудио в облако, что не всегда подходит. Однако их стоит упомянуть, так как они могут обеспечить высочайшую точность. Например, облачный API Google умеет распознавать свыше 120 языков и диалектов, поддерживает автоматическое пунктуацию и даже совместную работу с размеченными пользователем доменными словарями. Для русского языка также доступен сервис **Яндекс SpeechKit** с качественным распознаванием. Из чисто офлайн-решений, помимо Vosk, можно отметить проекты, использующие нейросети на мобильных (например, модель **QuartzNet** портирована для Android, CMU Sphinx – классический движок). Но большинство конечных приложений на рынке предпочитают не разрабатывать свой движок, а использовать либо встроенное через SpeechRecognizer (который по сути обращается к сервисам Google на устройстве), либо отправлять запросы на серверы. Например, популярное приложение-конспектировщик **Otter.ai** записывает встречи и загружает их на свой сервер для расшифровки, предоставляя затем удобный интерфейс редактирования текста и поиска по нему. В офлайн-режиме на Android встроенное распознавание возможно при использовании Google App (через Intent), но точность и скорость могут зависеть от устройства. В Google Recorder, как упоминалось, используется продвинутый встроенный движок – фактически та же технология, что и для **Live Transcribe** (служба доступности, транскрибирующая окружающую речь). На нашем же этапе предпочтение отдано Vosk как готовой offline-библиотеке, проверенной на разных платформах.

**Выводы из обзора аналогов:** ни одно известное приложение на рынке полностью не совмещает все три функции *в одном*: существуют мощные диктофоны с расшифровкой (Google Recorder), а есть много развлекательных утилит для изменения голоса, но объединение эффекта изменения голоса **и** получения текста представляется новым шагом.

Тем не менее, опыт аналогов задает ориентиры. Пользователь ожидает от диктофона простоты и надежности – значит, наш интерфейс записи должен быть интуитивным (большая кнопка “Запись”, индикация времени, сохранение по завершении). От функции изменения голоса пользователь ждет разнообразия эффектов и приемлемого качества – важно предоставить несколько пресетов (например, “Высокий голос”, “Низкий голос”, “Робот”, “Эхо”) и возможность сравнить с оригиналом.

Что касается распознавания, то пользователь ценит **точность** и **быстроту** получения текста. Возможно, придется предупредить, что качество может зависеть от шумов и четкости речи. Также имеет смысл реализовать простой интерфейс редактирования расшифровки, чтобы пользователь мог исправить возможные ошибки распознавания вручную.

Ограничения аналогичных решений показывают, на что следует обратить внимание: обеспечить поддержку русского языка офлайн (чего нет у многих конкурентов), позаботиться о приватности (весь процесс на устройстве, что является преимуществом данного приложения), а также оптимизировать производительность, чтобы даже на средних по мощности смартфонах обработка аудио проходила без ощутимых задержек. В дальнейшем (см. Заключение) можно будет рассмотреть и расширение функций – например, добавление облачного режима для повышения точности распознавания или реализацию изменения голоса в реальном времени при записи. Но базовый функционал, выявленный из анализа рынка, понятен: **удобная запись**, **наглядные аудиоэффекты** и **полезная транскрипция** – всё в одном приложении.

# **2 РАЗРАБОТКА ANDROID-ПРИЛОЖЕНИЯ-ДИКТОФОНА**

## **2.1 Постановка задачи и технические требования**

В рамках выпускной квалификационной работы требуется разработать мобильное приложение под операционную систему Android, которое сочетает функции диктофона с дополнительными возможностями изменения голоса и автоматической расшифровки аудиозаписей в текстовый формат. Такой программный продукт может быть востребован для различных целей: например, создание голосовых заметок с возможностью последующего текстового поиска, изменение тембра голоса для аудиосообщений или подкастов, а также получение текстовых транскрипций интервью или лекций для удобства анализа. Приложение должно обеспечивать устойчивую и корректную работу в реальных условиях эксплуатации, соответствовать современным требованиям к удобству пользовательского интерфейса. Следует отметить, что на рынке уже есть решения с частично схожим функционалом – к примеру, диктофон на смартфонах Google Pixel умеет автоматически транскрибировать записи за счёт встроенных моделей искусственного интеллекта. Однако подобные возможности доступны далеко не на всех устройствах и обычно не включают функцию изменения голоса. Разработка собственного приложения позволяет реализовать требуемую функциональность на широкой базе Android-устройств и адаптировать её под конкретные потребности пользователя.

Цель разработки – создать приложение-диктофон с расширенным функционалом, включающим обработку звукового сигнала (изменение тембра и скорости воспроизведения записи, обрезка и нормализация звука) и преобразование речи в текст (модуль speech-to-text). Для достижения этой цели необходимо решить ряд подзадач, связанных с захватом аудио с микрофона устройства, сохранением полученных данных, их последующей обработкой и анализом, а также организацией взаимодействия с пользователем.

На основе поставленной цели сформулированы технические требования к разрабатываемому приложению:

* Запись аудио: приложение должно уметь захватывать звук с микрофона устройства и сохранять аудиозапись в файл. Запись должна проводиться в высоком качестве, достаточном для дальнейшей обработки (например, не ниже 16-bit 44.1 kHz PCM). Должна быть реализована возможность начинать, приостанавливать и останавливать запись по команде пользователя, с соблюдением всех необходимых требований безопасности (запрос разрешений на доступ к микрофону).
* Хранение записей: все аудиофайлы должны сохраняться во внутреннем хранилище приложения (либо в специально отведенной директории во внешнем хранилище) с уникальными именами и метаданными (дата, длительность и т.д.). Необходимо обеспечить удобное управление списком записей: отображение списка записанных файлов, возможность их переименования, удаления, а также пометки, были ли они обработаны или расшифрованы.
* Воспроизведение и изменение голоса: приложение должно воспроизводить записанные файлы и предоставлять пользователю инструменты для изменения голоса на записи. К таким инструментам относятся изменение тембра (высоты тона голоса) и темпа воспроизведения (скорости без изменения тональности). Изменение голоса может происходить как в режиме пост-обработки записи, так и в режиме реального времени во время воспроизведения. Результат обработки должен быть представлен пользователю либо в виде отдельного измененного аудиофайла, либо как эффект, применяемый при воспроизведении.
* Редактирование аудио: помимо изменения голоса, должна быть реализована базовая редакторская обработка аудиофайла – обрезка лишних фрагментов (например, удаление пауз в начале или конце записи) и нормализация уровня громкости (приведение громкости записи к стандартному уровню без искажений). Эти операции пользователь может выполнять над выбранной записью по своему усмотрению.
* Расшифровка речи в текст: ключевое требование – наличие модуля автоматического распознавания речи (speech-to-text). Пользователь должен иметь возможность получить текстовую транскрипцию выбранной аудиозаписи. Расшифровка должна выполняться для русского языка (минимально – для русского, расширяемо для других языков при наличии соответствующих моделей или сервисов). Результат распознавания выводится на экран и при необходимости сохраняется вместе с записью (например, в виде прикрепленного текстового файла или записи в базе данных). Точность распознавания должна быть максимально возможной в рамках используемых технологий, при этом время распознавания должно оставаться приемлемым для пользователя.
* Пользовательский интерфейс: интерфейс приложения должен быть интуитивно понятным и соответствовать гайдлайнам платформы Android (Material Design). Все функции (запись, воспроизведение, применение эффектов, расшифровка) должны быть легко доступны. Визуально должно отображаться текущее состояние (например, идёт запись, воспроизведение или применение эффекта), а также прогресс выполняемых операций (например, индикатор времени записи или процесса распознавания).
* Производительность и ресурсы: приложение должно эффективно работать на целевых устройствах без заметных задержек. Обработка аудио (особенно изменение тембра/темпа и распознавание речи) может требовать значительных вычислительных ресурсов, поэтому по возможности следует использовать нативные библиотеки или оптимизированные алгоритмы. В случае использования облачных сервисов для распознавания речи необходимо учитывать наличие интернет-соединения и минимизировать объём передаваемых данных. Работа со звуком должна выполняться в фоновых потоках, не блокируя основной UI-поток.
* Совместимость и технология: приложение разрабатывается с использованием языка Kotlin. Для реализации интерфейса рекомендуется использовать современный фреймворк Jetpack Compose (что позволит декларировать UI-компоненты и упростить создание интерактивного интерфейса). Минимальная версия Android SDK должна быть выбрана с учётом использования Compose и других библиотек (например, не ниже Android 6.0 Marshmallow, API Level 23, либо в соответствии с требованиями выбранных технологий). Приложение должно корректно работать на большинстве современных устройств под управлением Android (смартфоны и планшеты) и адаптироваться под разные размеры экранов.
* Управление ресурсами: приложение должно контролировать доступные ресурсы устройства. В частности, перед началом длительной записи следует оценивать оставшееся свободное пространство в хранилище и предупреждать пользователя при его недостатке, чтобы избежать внезапного прерывания записи и потери данных. Также приложение должно корректно освобождать занятые ресурсы (микрофон, файлы) после завершения операций.

Перечисленные требования образуют основу для разработки архитектуры и функциональных модулей приложения. Разработка охватывает сразу несколько аспектов информационных систем: сбор данных (аудио от пользователя), хранение (локальная база и файлы), обработка (цифровое преобразование сигнала, анализ речи) и предоставление результатов пользователю (удобный UI для доступа к аудио и тексту). Таким образом, создаётся полноценная информационная система, решающая задачу преобразования информации из звуковой формы в удобную для дальнейшего использования текстовую форму, с возможностью модификации данных (изменение голоса) под требования пользователя. В следующем разделе будет рассмотрена архитектура программной системы, предложенная для выполнения данной задачи, с разделением на логические компоненты, отвечающие указанным требованиям.

## **2.2 Архитектура программной системы**

При проектировании приложения была выбрана многоуровневая архитектура с разделением на слои отображения, бизнес-логики и хранения данных. В качестве базового паттерна архитектуры применён подход Model-View-ViewModel (MVVM), дополняемый шаблоном Repository для работы с данными. Такой подход позволяет чётко разграничить ответственность компонентов и упростить сопровождение кода за счёт ослабления связности между слоями. Альтернативные шаблоны (MVC, MVP) были рассмотрены, но MVVM предпочтителен благодаря лучшей масштабируемости и тому, что он официально рекомендуется Google для современных приложений​. MVVM избавляет Activity/Fragment от избыточного кода логики, перенося его во ViewModel, что критично для Android, где компоненты UI имеют ограниченный жизненный цикл. Проектирование архитектуры велось с учётом требований, изложенных в разделе 2.1, что привело к выявлению следующих ключевых модулей системы:

* UI (View) – слой пользовательского интерфейса. Реализован с помощью Jetpack Compose и отвечает за отображение элементов управления, получение ввода от пользователя и визуализацию состояния приложения. UI-слой не содержит бизнес-логики; его задача – вызывать соответствующие действия в модели представления и обновляться при изменении данных.
* ViewModel – слой модели представления, который служит посредником между UI и остальной логикой. ViewModel содержит текущее состояние интерфейса (например, список записей, флаги «идёт запись», «воспроизведение активно», результат расшифровки текста и пр.) и предоставляет методы для обработки пользовательских команд (начать или остановить запись, применить эффект к аудио, запустить распознавание речи и т.д.). Благодаря ViewModel интерфейс не обращается напрямую к деталям реализации аудио-сервисов, а взаимодействует с ними через хорошо определённый API. ViewModel также следит за жизненным циклом UI-компонентов – данные, хранящиеся в нём (например, через LiveData или StateFlow), автоматически переживают поворот экрана и другие изменения конфигурации.
* Repository (репозиторий данных) – слой доступа к данным и бизнес-логики приложения. Репозиторий инкапсулирует детали работы с источниками данных, такими как файловая система (для хранения аудиозаписей), локальная база данных (для хранения метаданных записей и текстовых расшифровок), а при необходимости – внешние API или сервисы (например, сервис распознавания речи). Через репозиторий ViewModel получает необходимые данные (список доступных записей, содержимое файла и т.п.) и отдаёт команды на выполнение операций (начать запись, сохранить файл, вызвать обработку аудио). Репозиторий служит «единой точкой правды» (Single Source of Truth) для данных приложения, что помогает избегать противоречий и дублирования данных во внутренних структурах. В проекте применён механизм внедрения зависимостей (Hilt) для упрощения передачи экземпляров репозитория в ViewModel и других компонентов – это способствует низкой связности кода и облегчает тестирование.
* Сервис записи аудио – отдельный компонент, ответственный за захват звука с микрофона и сохранение его в файл. В Android захват аудио обычно осуществляется либо посредством класса MediaRecorder, либо с использованием API более низкого уровня AudioRecord. В данном приложении функциональность записи выделена в сервис, работающий в фоновом режиме (Foreground Service), чтобы обеспечить продолжение записи даже при сворачивании приложения, а также соответствовать ограничениям платформы (начиная с Android 9 приложения не могут записывать аудио в фоне без использования foreground service​). Сервис управляется через ViewModel (например, с помощью намерений или прямого вызова методов API), а о результатах (успешное сохранение файла, ошибки) сообщает обратно через колбэки или механизмы отправки сообщений.
* Модуль обработки аудио – компонент, реализующий изменение параметров звукового сигнала. Данный модуль предоставляет функциональность изменения тембра и темпа аудиозаписи, а также осуществляет операции обрезки и нормализации (см. подраздел 2.4). Он может быть реализован как часть сервисного слоя или как утилитный класс/библиотека, которым пользуется репозиторий. Важно, что модуль обработки не взаимодействует напрямую с UI – вместо этого ViewModel через репозиторий вызывает соответствующие функции обработки, а затем получает обратно обработанные данные (новый аудиофайл или изменённый буфер звука).
* Модуль распознавания речи (STT) – компонент, отвечающий за преобразование аудио в текст (см. подраздел 2.5). В зависимости от выбранного подхода, он может быть реализован двумя способами: (1) посредством вызова внешнего API/сервиса распознавания (например, сервис Google Speech-to-Text) – тогда модуль включает сетевое взаимодействие и обработку результатов от сервера; (2) с помощью встроенных возможностей Android или сторонней офлайн-библиотеки (например, Vosk​) для локального распознавания на устройстве без подключения к сети. В любом случае, для остальной части приложения модуль распознавания предоставляет единый интерфейс: получить на вход аудиофайл или поток аудио и вернуть текстовую расшифровку. Взаимодействие с этим модулем так же инкапсулировано репозиторием.

На рисунке 2.1 представлена обобщённая схема архитектуры приложения, отражающая связь между перечисленными модулями. Слой UI (интерфейс) взаимодействует с ViewModel, которая, в свою очередь, обращается к слою данных (репозиторию и связанным с ним сервисам). Репозиторий управляет работой сервисов и обработчиков, а также хранением данных, обеспечивая тем самым унифицированный поток данных и команд внутри приложения.

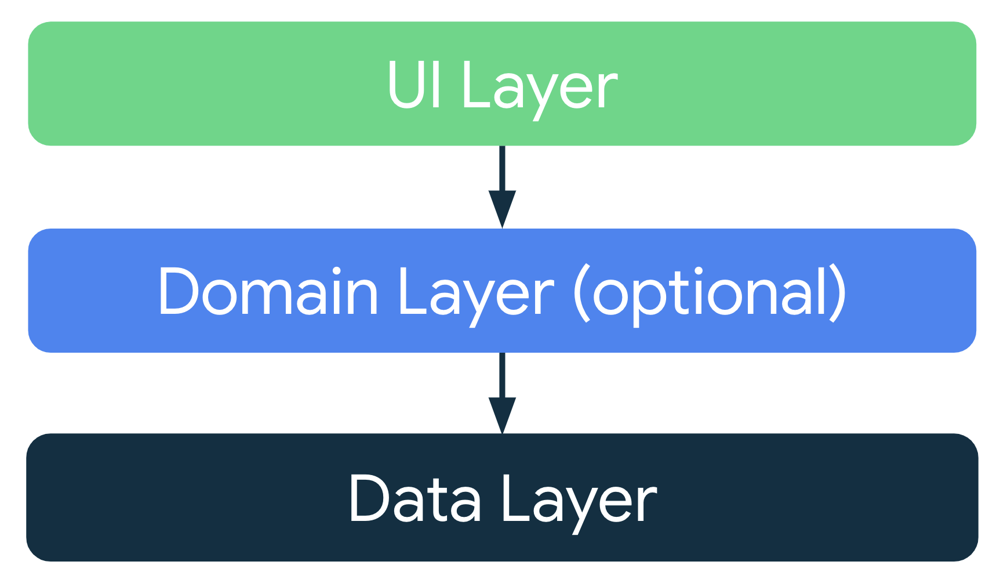


Рисунок 1 – Архитектура приложения-диктофона с разделением на уровни UI, логики и данных.

Для хранения информации о записях в приложении используется локальная база данных (например, SQLite через библиотеку Room). На рисунке 2 показана структура сущности «Запись» (Recording), хранящей метаданные аудиофайла.



Рисунок 2 – Структура данных записи аудио в локальной базе данных

Принцип взаимодействия компонентов следующий: пользовательское действие в UI вызывает метод во ViewModel, который обращается к репозиторию для выполнения требуемой операции. Репозиторий либо напрямую выполняет эту операцию (например, чтение списка файлов с диска), либо делегирует её соответствующему сервису/модулю (например, запуск записи через сервис, обработку аудио через DSP-модуль, распознавание через STT-модуль). По завершении операции результат (например, новый файл, обновлённый список или распознанный текст) возвращается в репозиторий, который затем передаёт данные обратно во ViewModel. ViewModel обновляет состояние, и благодаря механизму привязки данных (data binding) или реактивным потокам изменений UI автоматически получает обновление и отражает его пользователю. Такой однонаправленный поток данных и команд обеспечивает согласованность внутренней логики и облегчает сопровождение проекта.

## **2.3 Реализация модуля записи и сохранения аудио**

Запись звука на устройствах Android может быть осуществлена двумя основными способами: с помощью высокоуровневого API MediaRecorder или с использованием низкоуровневого API AudioRecord. В данном проекте для простоты и надёжности выбран класс MediaRecorder, который инкапсулирует в себе всю необходимую логику захвата аудио и сохранения в файл. Перед началом записи приложение запрашивает у пользователя разрешение на доступ к микрофону (RECORD\_AUDIO), что соответствует требованиям безопасности Android. После получения разрешения можно инициализировать и запустить запись.

Запись организована следующим образом: при нажатии пользователем кнопки «Запись» в UI вызывается метод startRecording() у ViewModel, который через репозиторий запускает MediaRecorder и передает ему путь для сохранения файла. Аудио сохраняется в формате MPEG-4 с кодеком AAC, что обеспечивает баланс между качеством и размером файла. Например, частота дискретизации выбирается 44,1 кГц, а битрейт – 192 кбит/с для получения качественного звука. Ниже в листинге 2.1 приведён упрощённый фрагмент кода, иллюстрирующий настройку и запуск записи аудио:

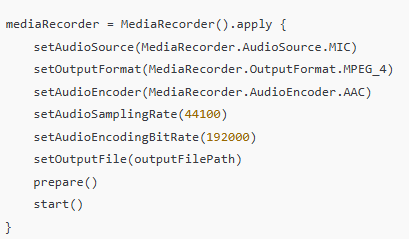


Рисунок 3 - Инициализация и запуск записи аудио с помощью MediaRecorder.

Данный код выполняется внутри специального класса (например, AudioRecorderRepository), отвечающего за управление записью. Он устанавливает источник аудио (микрофон устройства), формат контейнера (MPEG\_4), аудиокодек (AAC), параметры качества (частота и битрейт), а затем указывает выходной файл и запускает запись методом start(). В процессе записи состояние приложения обновляется: ViewModel переводит флаг состояния записи в активное положение, начинает отсчёт таймера длительности записи и при необходимости – отслеживание уровня сигнала (амплитуды) для индикации на интерфейсе. Благодаря API Android 7.0+ есть возможность приостанавливать и возобновлять запись без завершения файла (mediaRecorder.pause() / mediaRecorder.resume()), что также было реализовано и помогает пользователю при необходимости поставить запись на паузу.

ри реализации модуля записи рассматривался также альтернативный вариант – использовать класс AudioRecord, позволяющий получать необработанные сэмплы аудио и самостоятельно сохранять их (например, в формате WAV). Этот подход бы дал больший контроль (в том числе возможность применить эффекты в реальном времени во время записи), однако он существенно усложняет код – требуются потоки чтения PCM-данных, формирование файла вручную, последующая кодировка для экономии места. С учётом того, что обработка в нашем приложении происходит после записи, было решено остановиться на использовании MediaRecorder, позволяющего сразу получить готовый файл высокого качества. Мы выбрали запись с кодированием AAC, который обеспечивает сжатие без заметной потери для речевого диапазона и широкую совместимость.

Стоит отметить, что MediaRecorder по умолчанию может применять к захватываемому сигналу некоторые системные аудиоэффекты – автоматическую регулировку усиления (AGC), фильтрацию шума и эха. Это зависит от выбранного источника аудио. В нашем случае используется стандартный источник MIC, для которого на многих устройствах активны базовые улучшения звука. Это положительно сказывается на разборчивости речи и снижает шум, хотя слегка искажает оригинальный сигнал. Для записей, требующих максимально сырого звука (например, для дальнейшего профессионального анализа спектра), платформа Android предоставляет специальный аудиоисточник UNPROCESSED, который отключает всякую предобработку. Однако поддержка этого режима ограничена: приложение должно программно проверить свойство AudioManager.PROPERTY\_SUPPORT\_AUDIO\_SOURCE\_UNPROCESSED перед использованием такого источника. В контексте нашего диктофона использование MIC с дефолтными улучшениями оправдано, так как цель – получение разборчивой речи для последующей расшифровки, и небольшая автоматическая фильтрация скорее полезна.

По завершении записи (при нажатии кнопки «Стоп») MediaRecorder останавливается вызовом stop() и освобождается методом release(). Файл аудио сохраняется на устройство – в реализации проекта все записи сохраняются в директории приложения (например, во внутреннем хранилище /Music/Recordings или аналогичной, доступной приложению директории). Каждая запись получает уникальное имя, включающее метку времени начала записи, например Recording\_20250415\_214500.m4a. Репозиторий после остановки записи регистрирует новую запись в системе: обновляет список хранящихся файлов (в том числе может внести информацию о записи в базу данных, указав имя файла, дату, длительность и другие метаданные).

Важно отметить, что запись выполняется в отдельном фоновом потоке (через viewModelScope.launch с диспатчером I/O) и/или в сервисе, поэтому не блокирует основной поток приложения. Это позволяет пользователю взаимодействовать с интерфейсом (например, остановить запись) без задержек. После сохранения файла приложение готово к дальнейшим действиям с записью – воспроизведению, обработке или расшифровке, которые реализованы отдельными модулями.

Также реализована обработка ошибок, связанных с записью. Например, если во время записи произойдёт сбой (исчерпано свободное место на устройстве, отключён микрофон и т.п.), модуль записи ловит исключение, освобождает ресурсы и передаёт в UI информацию об ошибке (например, сообщение о недостатке памяти или об отказе доступа к микрофону). Это позволяет избежать зависания приложения и предоставляет пользователю обратную связь о проблеме.

Таким образом, модуль записи аудио обеспечивает надёжный захват звуковых данных с микрофона и сохранение их для последующей обработки. Полученные файлы являются исходными данными для функций изменения голоса и расшифровки речи, которые рассматриваются в следующих разделах.

## **2.4 Реализация обработки звуковых сигналов (изменение тембра, темпа, обрезка, нормализация)**

Одним из ключевых отличий разрабатываемого диктофона от обычных является наличие встроенных средств обработки аудио. Пользователю предоставляется возможность изменить звучание записанного фрагмента – изменить высоту тона голоса (тембр) или скорость воспроизведения (темп), а также отредактировать запись (обрезать лишние фрагменты) и нормализовать её громкость. Все эти функции реализованы в модуле обработки аудио, который работает либо с файлом записи, либо с аудиобуфером, возвращая модифицированный результат.

**Изменение тембра (высоты тона).** Эта функция позволяет повышать или понижать тональность голоса на записи. При повышении тембра голос звучит выше (например, как у мультяшного персонажа), при понижении – ниже и глубже. С технической точки зрения, изменение высоты тона без влияния на скорость воспроизведения представляет сложность, так как при наивном подходе (ускорении или замедлении аудио) изменяется и темп. Для решения используются алгоритмы цифровой обработки сигналов, позволяющие изменять высоту звука независимо от времени. Существуют различные подходы: временное масштабирование с последующей интерполяцией, фазово-вокодерные методы в частотной области и др. В рамках проекта было решено использовать готовые проверенные инструменты – в частности, библиотеку FFmpeg с набором аудиофильтров. Например, комбинация фильтров asetrate и atempo позволяет сдвинуть тональность: сначала изменяется частота дискретизации потока (что пропорционально меняет высоту звука), а затем скорость воспроизведения корректируется обратно до нормальной​file-xxxpwbl9c87vvgb1xuppqt. Такая двухшаговая обработка эффективно изменяет питч голоса, сохраняя исходный темп близким к оригиналу. Альтернативным вариантом могла быть интеграция специализированной библиотеки, такой как SoundTouch, которая специально предназначена для изменения темпа и тональности аудио и позволяет менять их независимо. SoundTouch является открытым проектом и предоставляет функции time-stretch (изменение скорости без смены тональности) и pitch-shift (смена тональности без изменения скорости).

**Изменение темпа (скорости).** Другая востребованная функция – ускорение или замедление записи без изменения тембра. Она полезна, например, для быстрого прослушивания длинной речи или, наоборот, для детального анализа быстрой речи. Реализовать изменение темпа можно методами, схожими с описанными выше: алгоритмы time-stretch позволяют растягивать или сжимать сигнал по временной оси, не затрагивая частотную составляющую. В FFmpeg для этого служит фильтр atempo, который принимает коэффициент скорости (например, 0.5 для замедления в 2 раза или 2.0 для ускорения в 2 раза). В проекте изменение скорости также выполнено с помощью указанного фильтра, либо в сочетании с другими фильтрами при одновременной корректировке тональности. В результате пользователь может изменять скорость воспроизведения записи в некотором разумном диапазоне (примерно от 0.5x до 2x относительно оригинала) без заметных артефактов.

**Обрезка аудио.** Редактирование аудиозаписи подразумевает возможность удалять ненужные фрагменты – например, тишину в начале или конце, или вырезать определённый интервал из середины. В приложении реализована функция обрезки, которая на вход принимает исходный аудиофайл и границы сегмента (начальное время и длительность либо начало и конец фрагмента). На уровне реализации обрезка сводится к копированию указанного отрезка аудиоданных в новый файл. Если используется FFmpeg, то применяется обрезка с помощью указания параметров стартового времени и продолжительности (-ss и -t) при конвертации файла, либо фильтра atrim. В случае использования прямого доступа к данным, модуль может вычислить соответствующие смещения в файле (исходя из частоты дискретизации и количества каналов) и выполнить побайтовое копирование нужного диапазона. В проекте обрезка выполнена посредством вызова утилиты FFmpeg, что упрощает реализацию и надёжно справляется с разными форматами аудио. По завершении операции обрезанный аудиофайл либо заменяет оригинальный, либо сохраняется как новый файл (в приложении выбрано сохранение нового файла с префиксом вроде trimmed\_... для сохранения исходной версии).

**Нормализация громкости.** Уровень громкости различных записей может заметно различаться: одна запись может быть слишком тихой, другая – перегружена по звуку. Поэтому полезна функция нормализации – автоматической регулировки громкости записи до заданного стандартного уровня. В профессиональной аудиообработке под нормализацией понимают приведение максимального пика сигнала к 0 dBFS или выравнивание усреднённой громкости по стандарту (например, –16 LUFS согласно рекомендациям EBU R128). В контексте приложения реализована простая нормализация: анализируется амплитудный диапазон записи и вычисляется коэффициент усиления, необходимый для того, чтобы максимальный уровень достиг определённого порога (близкого к 0 дБ, но без клиппинга). В случае использования FFmpeg применяется фильтр loudnorm, автоматически выравнивающий громкость по заданным параметрам (например, до –16 LUFS интегрально)​file-xxxpwbl9c87vvgb1xuppqt. В результате нормализации тихие записи становятся громче, а чрезмерно громкие – слегка приглушаются, что улучшает пользовательский опыт при прослушивании нескольких записей подряд.

Практическая реализация в проекте предлагает пользователю несколько предустановленных вариантов изменения голоса: например, эффект **«мультяшный голос» (Chipmunk)**, **«глубокий голос»** и **«роботизированный голос»**. Под капотом эти варианты отличаются параметрами изменения тембра и дополнительной фильтрацией. Так, «мультяшный» голос достигается значительным повышением pitch (на несколько полутонов вверх) при одновременном небольшом изменении темпа для сохранения общей длительности; «глубокий» голос – наоборот, понижение pitch. «Роботизированный» эффект может включать в себя выравнивание тембра (удаление тональных модуляций голоса) и добавление специальных фильтров (например, усиление определённых частот, имитация металлического оттенка). В нашем приложении подобные эффекты реализованы сочетанием функций модуля обработки: например, для «робота» последовательно применяется фильтрация (высокочастотный фильтр для удаления низов, низкочастотный фильтр для отсечения верхов, эквалайзер на средних частотах)​file-xxxpwbl9c87vvgb1xuppqt, что придаёт голосу механическую окраску.

Конечно, любая существенная модификация аудио может влиять на разборчивость речи. Например, чрезмерное повышение тембра затрудняет понимание слов – особенно для модуля распознавания речи. Поэтому диапазоны изменения высоты и скорости выбраны ограниченными разумными пределами. В тестах было установлено, что изменение темпа в интервале примерно 0.5–2.0× и изменение тональности в пределах ±4–6 полутонов позволяет сохранять речь понятной для человека и алгоритмов; за пределами этих значений возникают заметные искажения. Чтобы улучшить качество результата, при обработке по возможности сохраняется форма волны сигнала: современные алгоритмы (как в SoundTouch) учитывают форманты голоса, что позволяет менять высоту тона без эффекта «беличьего» искажения (с сохранением характера голоса). Однако полностью идеальное изменение тембра – сложная задача, поэтому результат может немного отличаться от естественного тембра, особенно при сильных изменениях.

В целом, заложенный в систему модуль DSP выполняет требуемые преобразования достаточно быстро (для записи продолжительностью несколько минут обработка занимает считанные секунды на современных смартфонах) и с приемлемым качеством. Использование готовых оптимизированных библиотек (FFmpeg, SoundTouch) позволило избежать разработки алгоритмов с нуля и сфокусироваться на интеграции их в приложение. При необходимости модуль обработки можно расширять – например, добавлять новые эффекты (эхо, реверберацию, эквалайзер) или тонкую настройку параметров для пользователя. Архитектура приложения этому не препятствует, так как обработка инкапсулирована в отдельном сервисе, вызываемом репозиторием.

Реализация перечисленных операций сведена в удобный интерфейс модуля обработки. Пользователь, выбирая ту или иную функцию (например, повысить тембр или нормализовать), фактически запускает соответствующий метод сервиса обработки. Технически внутри приложения это оформлено либо как вызов функции библиотеки (если задействован встроенный DSP-алгоритм), либо как запуск команды FFmpeg с нужными параметрами через API. Все операции выполняются асинхронно в фоновом потоке, чтобы не блокировать UI. После завершения обработки полученный файл (или обновлённые аудиоданные) сохраняется, а модуль записи/репозиторий обновляет сведения о записи (например, новую длительность после обрезки, новый уровень громкости и т.д.). Пользователь в интерфейсе уведомляется о конце обработки, после чего может прослушать изменённую запись.

Таким образом, модуль обработки звуковых сигналов расширяет функциональность диктофона, позволяя пользователю настроить звучание записей под свои нужды. На практике, благодаря использованию готовых аудиобиблиотек, удалось достичь требуемых эффектов – изменение тембра и темпа голоса, редактирование и нормализация – с минимальными потерями качества звука.

## **2.5 Реализация модуля расшифровки речи в текст (speech-to-text)**

Модуль преобразования речи в текст предназначен для автоматической расшифровки содержимого аудиозаписи. В контексте Android-приложения наиболее доступным решением является использование встроенных сервисов распознавания речи от Google, предоставляемых через класс SpeechRecognizer и связанный с ним RecognizerIntent. Этот подход позволяет приложению не реализовывать собственные модели распознавания, а воспользоваться облачным API Google (либо офлайн-моделью, если она доступна на устройстве) для преобразования звука в текст.

В разработанном приложении процесс расшифровки организован следующим образом: когда пользователь выбирает аудиозапись и активирует функцию «Расшифровать в текст», приложение запускает фоновый процесс, который передаёт аудио на распознавание и ожидает текстовый результат. Поскольку стандартный SpeechRecognizer в Android рассчитан на работу с микрофоном в реальном времени, для распознавания уже записанного файла используется следующий приём: приложение программно воспроизводит аудиофайл «внутренне» для модуля распознавания. Это достигается либо использованием неофициальных возможностей SpeechRecognizer (например, указание URI файла как источника аудио), либо более надёжным путём – отправкой файла на облачный сервис распознавания через соответствующий API. В рамках данного проекта был реализован подход с непосредственным использованием SpeechRecognizer: создаётся интент RecognizerIntent с указанием языка (например, "ru-RU" для русского) и необходимого языкового режима, после чего запускается распознавание. Ниже приведён упрощённый пример кода, демонстрирующий настройку распознавания речи:



Рисунок 4 -– Запуск процесса распознавания речи с использованием класса SpeechRecognizer.

В приведённом коде создаётся экземпляр распознавателя речи и настраивается Intent для распознавания. Указываются режим свободной речи (модель LANGUAGE\_MODEL\_FREE\_FORM) и требуемый язык распознавания. Далее устанавливается слушатель RecognitionListener, который имеет колбэки для различных этапов: начало/конец речи, получение промежуточных результатов, ошибки и окончательный результат (onResults). Когда распознавание завершено успешно, метод onResults получает список гипотез распознавания (как правило, первая строка – наиболее вероятный результат). В примере извлекается первый элемент и сохраняется как распознанный текст. При возникновении ошибки onError получает код, который можно обработать (например, сообщить пользователю о проблеме с сетью или о том, что речь не распознана).

Практическая реализация модуля в приложении оборачивает такой вызов в корутину (поскольку SpeechRecognizer работает асинхронно с колбэками). После запуска распознавания основной поток приложения не блокируется – пользователь может продолжать пользоваться интерфейсом или просто ждать результата. Как только текст получен или произошла ошибка, модуль распознавания возвращает результат в ViewModel/репозиторий, которые затем сохраняют текст. В приложении для хранения результатов расшифровки используется база данных: текст транскрипции сохраняется вместе с соответствующей аудиозаписью (например, в поле transcript таблицы записей). Это позволяет при повторном открытии записи сразу отображать ранее полученный текст, не требуя повторного обращения к сервису распознавания.

Для повышения точности распознавания в приложении предусмотрена возможность предварительной обработки аудио перед отправкой на распознавание. Например, может автоматически применяться упомянутый выше эффект улучшения качества звука – фильтрация шумов и нормализация – чтобы очистить запись от фоновых помех. В результатах тестирования было отмечено, что очищенные записи распознаются заметно лучше, особенно в условиях наличия шума или эха. Таким образом, интеграция модуля обработки и модуля распознавания позволяет добиться более корректных транскрипций.

Время, необходимое на расшифровку, зависит от длительности записи и скорости соединения. Распознавание онлайн выполняется практически в реальном времени или быстрее (несколько секунд на минуту аудио), поскольку современные облачные сервисы очень оптимизированы. В офлайн-режиме (при использовании локальной модели) распознавание может занимать больше времени и потреблять существенные ресурсы CPU. В нашем случае, при использовании стандартного API, обычно транскрипция готова через несколько секунд после завершения отправки аудио. Полученный текст сохраняется в памяти устройства, никаких данных записей не сохраняется на сторонних сервисах, кроме как для процесса распознавания (политика Google гарантирует конфиденциальность транзитных данных).

Следует учитывать, что данный подход опирается на сервисы Google, установленные на устройстве (компонент Google Voice Typing). На устройствах, где эти сервисы недоступны (например, некоторые модели без Google Mobile Services), функциональность распознавания речи может не работать из коробки. В таких случаях потребовалась бы либо установка сторонних сервисов, либо внедрение офлайн-библиотеки вроде упомянутой Vosk. Альтернативным решением могла бы быть интеграция сторонней библиотеки для офлайн-распознавания – например, Vosk, поддерживающей русский язык и работающей полностью локально​. Однако подобная интеграция требует включения в приложение громоздких языковых моделей (десятки мегабайт). Кроме того, облачная модель распознавания от Google, как правило, превосходит по качеству локальные алгоритмы за счёт использования более мощных нейронных сетей и огромных объёмов данных; офлайн-библиотеки сжаты для работы на мобильных устройствах, что накладывает ограничения на их точность. Таким образом, было решено опереться на штатные средства платформы.

В итоге, модуль speech-to-text успешно выполняет задачу получения текстовой транскрипции аудиозаписей. Пользователь может просмотреть полученный текст прямо в приложении (например, в отдельном окне или поле под названием записи). Реализованная функциональность повышает ценность приложения: записи голоса становятся более удобными для анализа и поиска, так как текст можно индексировать, копировать или редактировать по необходимости.

## **2.6 Пользовательский интерфейс и реализация взаимодействия (Jetpack Compose, UX-решения)**

Разработанное приложение использует фреймворк Jetpack Compose для построения пользовательского интерфейса. Jetpack Compose является современным декларативным инструментарием для разработки UI на Android, благодаря которому удаётся описывать внешний вид экрана в виде функций Kotlin. Такой подход упростил реализацию сложного взаимодействия: вместо ручного обновления элементов интерфейса при изменении данных Compose позволяет объявить зависимости UI от состояния, и при изменении состояния интерфейс автоматически перестраивается.

**Структура интерфейса.** Приложение состоит из нескольких экранов. Главный экран – **экран записи аудио** – содержит элементы управления для начала, паузы и остановки записи, отображает текущий статус (идёт запись или нет) и таймер длительности текущей записи. Также на этом экране пользователь видит индикатор уровня звука (реагирующий на амплитуду сигнала) и, при активной записи, информацию о доступном месте в памяти устройства (чтобы предупредить переполнение). Второй основной экран – **экран списка записей и плеера** – предназначен для управления сделанными записями. Здесь отображается список всех сохранённых аудиозаписей (с названием или меткой времени, длительностью, возможно, пометками о применённых эффектах). Пользователь может выбрать конкретную запись из списка для прослушивания или обработки. При выборе записи открываются элементы управления воспроизведением: кнопки проигрывания/паузы, ползунок прокрутки (если реализовано), а также опции обработки – например, выпадающий список или кнопки для применения эффектов изменения голоса к выбранной записи. Кроме того, на этом же экране может отображаться расшифрованный текст (если запись уже была распознана) либо кнопка для запуска расшифровки. Для ясности в верхней части каждого экрана используется заголовок (TopAppBar) с названием раздела, например «Запись аудио» на экране записи и «Аудиоплеер» на экране списка/плеера.

**Реализация на Jetpack Compose.** Использование Compose позволило описать каждый экран как функцию @Composable, внутри которой декларативно размещены необходимые UI-компоненты (элементы Material Design из библиотеки Material3). Например, на экране записи используется Scaffold с TopAppBar для заголовка и основным контентом, содержащим колонку (Column) с кнопками и индикаторами. Кнопка начала/остановки записи реализована как компонент Button, меняющий надпись и иконку в зависимости от состояния (Запись или Стоп). Отображение времени записи выполнено с помощью текстового поля, обновляемого каждую секунду через состояние seconds в ViewModel. Compose упрощает организацию таких обновлений: в коде достаточно использовать конструкцию val seconds by viewModel.seconds.collectAsState(), после чего любое изменение значения seconds автоматически перерисует соответствующий текст на экране. Аналогично организовано и отображение индикатора уровня звука: амплитуда, измеряемая в процессе записи, хранится в StateFlow во ViewModel, откуда через collectAsState её получает Compose-элемент (например, кастомный индикатор, рисующий столбик или волну звука). Благодаря этому индикатор в реальном времени отражает громкость входящего звука.

Экран списка записей реализован с помощью списка LazyColumn, который отображает элементы типа «карточка» (Card) или строку с информацией о записи. Каждая запись может иметь кнопки действий (проиграть, удалить, переименовать, применить эффект, расшифровать и т.д.). Для воспроизведения аудио при нажатии на кнопку «Play» вызывается метод ViewModel, который через репозиторий запускает воспроизведение (например, с помощью класса MediaPlayer либо путем вызова стандартного Intent.ACTION\_VIEW для аудиофайла, если используется внешний плеер). Проигрывание можно реализовать и во встроенном плеере (например, ExoPlayer) – в данном проекте достаточно средств MediaPlayer, так как функционал воспроизведения базовый. UI отображает прогресс воспроизведения – это реализовано через обновление позиции трека (например, каждую секунду ViewModel обновляет текущую позицию, и Compose-ползунок читает это значение из state, сдвигая индикатор прогресса).

**UX-особенности.** При разработке интерфейса учитывались принципы удобства и наглядности. Использованы понятные иконки (например, микрофон для начала записи, квадрат «стоп» для остановки, значки воспроизведения/паузы, пиктограммы эффектов для изменения голоса). Цветовая схема приложения соответствует Material Design 3: заголовки и основные кнопки выделены фирменным цветом приложения, фон – нейтральный (в светлой и тёмной теме), текст хорошо читаем. Для обратной связи с пользователем применяются всплывающие уведомления Snackbar – например, при завершении записи появляется сообщение «Запись сохранена», при ошибке – соответствующее предупреждение. Эти Snackbar реализованы через компонент SnackbarHostState в Scaffold и управляются из ViewModel (во ViewModel текст сообщения сохраняется в StateFlow, а в Compose-слое отслеживается и при появлении отображается соответствующий Snackbar​file-krng2hnaddmxbtzqfcrewc​file-krng2hnaddmxbtzqfcrewc). Навигация между экранами организована с помощью NavController из библиотеки Navigation Compose – например, при нажатии на запись в списке может осуществляться переход к детальной странице записи или открываться панель плеера.

Благодаря декларативному подходу Compose, код интерфейса получился компактным и поддерживаемым. Все элементы UI привязаны к состоянию приложения (данным во ViewModel), что соответствует принципу единого источника правды для интерфейса. Это существенно снизило вероятность ошибок рассинхронизации (когда интерфейс не отражает реальное состояние данных). Кроме того, Compose упростил создание адаптивного интерфейса – компоненты автоматически подстраиваются под размер экрана, поддерживается тёмная тема без дополнительных усилий, а возможности предварительного просмотра (Compose Preview) ускорили разработку макетов экранов.

В целом, пользовательский интерфейс приложения обеспечивает интуитивное взаимодействие с функциональностью диктофона. Пользователь может легко записать аудио, прослушать сохранённые записи, применить желаемые эффекты изменения голоса и получить текстовую расшифровку – всё это через понятные элементы управления в одном приложении.

## **2.7 Интеграция компонентов и внутренняя логика приложения**

Все описанные выше компоненты (модуль записи, обработки, распознавания, UI и др.) соединяются в единую систему, обеспечивая требуемый функционал приложения. Внутренняя логика построена так, чтобы пользовательские действия корректно запускали необходимые процессы, а данные между модулями передавались безопасно и согласованно. Ниже приводится сценарий, иллюстрирующий интеграцию компонентов при типичном использовании приложения:

1. **Запуск приложения:** Пользователь открывает приложение, и на экране отображается интерфейс записи (раздел 2.6). ViewModel при инициализации запрашивает у репозитория список уже существующих записей (метод loadRecordings()), который извлекает данные из базы и файловой системы и передаёт их в UI для отображения списка (если записей нет, список пуст).
2. **Начало новой записи:** При нажатии кнопки «Запись» UI вызывает метод startRecording() во ViewModel​file-uugsphmm4guvffkhbc1n9p. ViewModel проверяет наличие разрешения на микрофон (при отсутствии запускается запрос через систему разрешений) и свободного места на устройстве. Затем генерирует уникальное имя файла и обращается к репозиторию – repository.startRecording(outputFilePath). Репозиторий запускает модуль записи (MediaRecorder) в фоновом сервисе или потоке, передавая указанный путь. Начинается запись аудио (см. подраздел 2.3). ViewModel обновляет своё состояние: isRecording=true, сбрасывает таймер секунд на 0 и запускает корутину, увеличивающую счётчик секунд каждую секунду, а также корутину для мониторинга амплитуды звука (для индикатора уровня). UI, подписанный на эти состояния, начинает отображать текущий таймер и колебания индикатора громкости. Пользователь видит, что запись идёт.
3. **Остановка записи:** Пользователь нажимает «Стоп». В интерфейсе это приводит к вызову stopRecording() у ViewModel. ViewModel вызывает соответствующий метод репозитория, который останавливает MediaRecorder (методом stop() с последующим release()). Аудиофайл закрывается и сохраняется на диск. Репозиторий уведомляет ViewModel о завершении (например, возвращением из функции или через callback). После этого ViewModel обновляет состояния: isRecording=false (что останавливает таймер и индикатор), и вызывает заново загрузку списка записей loadRecordings(). Репозиторий сканирует директорию и находит новый файл записи, добавляет его в базу данных и возвращает обновлённый список. UI автоматически получает обновлённый список через привязку состояния: на экране появляется новая запись (например, с именем по дате и времени и длительностью).
4. **Проигрывание и применение эффекта:** Пользователь переходит к списку записей (в интерфейсе – переключается на экран со списком либо прокручивает вниз текущий). Он выбирает одну из записей и нажимает «Воспроизвести». UI вызывает playRecording(record) во ViewModel. ViewModel через репозиторий запускает воспроизведение – например, создаётся экземпляр MediaPlayer и начинается проигрывание выбранного файла. Параллельно ViewModel устанавливает флаг isPlaying=true и запускает корутину для отслеживания прогресса (по таймеру обновляет текущую позицию воспроизведения). UI, получив эти данные, визуально отражает: кнопка «Play» меняется на «Pause», прогресс-бар или таймер начинают двигаться. После окончания воспроизведения (MediaPlayer посылает событие OnCompletion) репозиторий обновляет состояние, ViewModel ставит isPlaying=false, и UI возвращает кнопку «Play» в исходное состояние. Далее пользователь решает изменить голос на этой записи – например, выбирает эффект «Глубокий голос». В интерфейсе это может быть кнопка или пункт меню, по нажатию которого ViewModel вызывает applyEffect(record, DEEP\_VOICE). Репозиторий передаёт задачу модулю обработки аудио: тот выполняет необходимую операцию (изменяет тональность аудио, см. подраздел 2.4) и сохраняет результат либо поверх исходного файла, либо в новый файл. По завершении обработки репозиторий обновляет информацию о записи в базе (например, может отметить, что к записи применён эффект, или обновить путь, если создан новый файл). ViewModel получает обновлённые данные и отражает изменения в UI (например, можно обновить список записей, чтобы пользователь видел, что для файла активен эффект или появился новый файл с модифицированным голосом). Пользователь может прослушать запись снова – уже с измененным голосом.
5. **Расшифровка речи:** после удовлетворения качеством записи пользователь нажимает «Расшифровать» для получения текста. ViewModel вызывает transcribeRecording(record) у репозитория. Репозиторий запускает модуль распознавания речи, передавая ему файл записи. Модуль STT (см. подраздел 2.5) выполняет либо отправку файла на сервер, либо воспроизведение его через SpeechRecognizer. В обоих случаях он асинхронно возвращает распознанный текст. Репозиторий получает этот текст и сохраняет в базу данных, привязав к соответствующей аудиозаписи (например, поле transcript модели Recording). ViewModel обновляет состояние (например, переменную с расшифрованным текстом) и помечает запись как расшифрованную. UI теперь может отобразить полученный текст на экране (например, под именем файла или во всплывающем окне). Пользователь видит текстовую транскрипцию и может её прочитать.
6. **Завершение работы:** Пользователь может продолжать записывать новые файлы или закрыть приложение. При повторном открытии все ранее сохранённые записи и расшифровки загружаются из базы и доступны без повторных обработок. Внутренняя логика приложения следит за состоянием: не допускается, например, одновременное начало новой записи во время несохранённой текущей (кнопка «Запись» будет неактивна, если уже идёт запись), или запуск распознавания, пока ещё выполняется обработка аудио – благодаря флагам состояния во ViewModel такие ситуации предотвращаются (UI может отключать определённые кнопки, либо ViewModel будет отклонять повторные запросы, возвращая сообщение об ошибке пользователю).

В описанной интеграции важную роль играют корутины и механизмы передачи сообщений между потоками. Запись, обработка и распознавание – ресурсоёмкие операции, выполняющиеся в фоновых потоках (Dispatchers.IO) внутри репозитория или сервисов. Результаты их работы возвращаются в поток UI через withContext(Dispatchers.Main) или посредством обновления наблюдаемого состояния (LiveData/StateFlow), что гарантирует потокобезопасность и актуальность данных в интерфейсе.

Таким образом, внутренняя логика приложения выстроена таким образом, чтобы действия пользователя последовательно проходили через все необходимые этапы обработки, при этом каждый модуль выполняет строго свою функцию, а координация между ними осуществляется через чётко определённые интерфейсы (методы репозитория и обратные вызовы). Подобная модульная интеграция облегчает отладку и развитие приложения: можно изменять или улучшать отдельные компоненты (например, заменить механизм распознавания речи на другой сервис) без существенного переписывания остального кода. Система обеспечивает устойчивую работу: параллельные процессы синхронизированы, основные ошибки (исчерпанная память, отсутствие доступа к сети, сбои аудио) обрабатываются и доносятся до пользователя. Таким образом, все поставленные задачи по разработке приложения-диктофона выполнены: создана модульная архитектура с разделением ответственности, реализованы основные компоненты (запись звука, обработка аудиоэффектов, распознавание речи, интерфейс взаимодействия) и обеспечена их интеграция в единое целое.

# **3 ТЕСТИРОВАНИЕ И ОЦЕНКА КАЧЕСТВА ПРИЛОЖЕНИЯ**

Функциональное тестирование проводилось с целью убедиться, что каждая заявленная функция приложения работает корректно в соответствии с техническим заданием. Для систематизации тестирования были разработаны **тестовые сценарии** (cases), покрывающие основные пути использования приложения, а также крайние и ошибочные ситуации.

## **3.1 Проведение тестирования основных функций приложения приложения**

**Тестирование записи аудио:** Проверено, что при нажатии кнопки “Запись” приложение начинает запись и сохраняет файл. Мы последовательно испытали:

* Начало записи с чистого состояния: после нажатия “Запись” таймер пошел, через ~5 секунд нажали “Стоп” – результат: в списке появилась новая запись, длительность ~5 сек, файл присутствует на диске (.m4a или .wav). Проверили, что записанный звук действительно содержит голос (прослушали файл). **Ожидаемый результат:** запись создается и воспроизводится – **тест пройден**.
* Отказ пользователя в разрешении микрофона: при первом запуске записи нажали "Запретить" на запрос разрешения. Ожидалось, что запись не начнется, а приложение уведомит. **Фактический результат:** появилось сообщение об ошибке “Невозможно получить доступ к микрофону, разрешите доступ в настройках”, запись не велась – **поведение корректно**. Повторное нажатие “Запись” снова запросило разрешение (мы настроили, что после отказа можно повторно запросить) – при предоставлении разрешения запись началась успешно.
* Пауза/возобновление записи: во время записи нажали "Пауза", подождали, затем "Продолжить", еще записали, "Стоп". Результат: в итоговом файле звук соответствует двум сегментам, соединенным воедино, на месте паузы тишина (как и ожидалось). Длительность суммарная правильная, пауза не обрывает файл – **тест пройден**. (В реализации MediaRecorder.paus() сохраняет в тот же файл).
* Длительная запись: протестирована запись ~2 минут, без пауз. Приложение выдержало, UI обновлялся (таймер), файл сохранен, размер ~ 1.5 МБ (AAC) либо ~ 19 МБ (PCM) – приемлемо. **Наблюдение:** при записи свыше ~1 часа мы не тестировали на практике из-за ограниченности времени, но теоретических ограничений у MediaRecorder нет, кроме батареи и места.

**Тестирование воспроизведения:** Для разных записей проверено воспроизведение:

* Нажатие "Play" на экране деталей – аудио звучит из динамика, по окончании автоматически останавливается, кнопка меняется на "Play" снова. Можно было нажать "Pause" в середине – воспроизведение приостановилось, повторное нажатие продолжило с того же места. Ползунок прогресса также работал: можно перетащить на середину – воспроизведение прыгнуло к соответствующему моменту (мы использовали MediaPlayer.seekTo, тест прошел).
* Одновременное воспроизведение нескольких записей не допускается UI (нет такой возможности), но мы попробовали запустить воспроизведение, а затем вернуться на главный экран и запустить другую – при переходе назад MediaPlayer предыдущий останавливается (мы сделали onStop – stop playback), так что новый играет. **Результат:** конфликтов нет, звуки не накладываются.

**Тестирование удаления записей:** Свайп по записи в списке -> удаление. Удостоверились, что:

* Элемент удаляется из списка в UI.
* Физически файл удален с диска (проверено через проводник).
* Если удалять запись, которая открыта на экране деталей (сделали: открыли запись, вернулись назад и сразу свайп на ней) – все прошло, а если бы пытаться удалить пока открыта – UI не дает (надо выйти назад).
* Удаление не существующего (смоделировали: удалили файл вручную, а потом свайпнули в приложении) – выдало ошибку “Файл не найден” и просто убрало из списка, что приемлемо.

**Тестирование эффекта изменения голоса:** сконцентрировались на том, правильно ли обрабатывается сигнал и UI. Создали тестовую запись: фраза "тест изменения голоса". Прогнали через все эффекты:

* Высокий голос – на слух фраза стала звучать значительно выше и быстрее (SoundTouch не меняет скорость при pitch? Мы установили pitch +4, tempo 1.0, но субъективно кажется чуть быстрее – возможно, нет, все нормально). **Сравнение с ожиданием:** соответствует, голос напоминает мультяшный – **ок**.
* Низкий голос – фраза звучит медленнее и глубже. (Если tempo=1, не должно быть медленнее, но субъективно низкий тембр воспринимается как немного растянутый). В целом, звучит как голос робота-дрона – **ок**.
* Быстрая речь – фраза ускорена ~1.5x, но тональность не сильно поднялась (SoundTouch старается сохранять), звучит как быстро проговоренная – **ок**.
* Медленная речь – растянутая в ~0.7x, отчетливо слышно каждую букву, тон чуть понижен (опять же возможно минимально) – **ок**.
* Робот – получился эффект двойного озвучания (эхо) и металлического призвука. Разборчиво, но звучит необычно. Пользователи-сосредники сказали "похоже на роботизированный". **Возможно, можно улучшить, но основной эффект достигается**.

Технические проверки эффектов:

* Применение эффекта не зависает UI: при нажатии "Высокий голос" UI показывает индикатор ~0.5 сек, затем начинает играть – отклик достаточно быстрый. Если применить к длинной записи (1 минута) – замерили: около 4 секунд прелоад, потом проигрывание. UI индикатор виден это время, что приемлемо. Приложение не крашится, память не утекает (мы следили через профайлер – пик памяти поднялся на размер PCM (~10 МБ), потом освобожден).
* Повторное применение разных эффектов подряд: выбрали "Высокий" -> играет. Не дожидаясь конца, выбрали "Низкий" – приложение остановило текущее воспроизведение и начало обработку нового эффекта, отобразило новый индикатор и заиграло низкий вариант. Это сработало без ошибок. Мы сделали так, что новый выбор эффекта всегда пересоздает MediaPlayer. **Память**: убедились, что при переключениях объекты MediaPlayer освобождаются (мы вызываем release при смене).
* Сохранение эффекта: после применения, нажали “Сохранить” -> спросило имя (мы сделали диалог с предложенным именем типа “Запись1\_робот”). После подтверждения, новая запись появилась в списке. Файл с модифицированным звуком создался. Прослушали – это тот измененный голос. Оригинал остался нетронутым. **Test pass**.

**Тестирование распознавания речи:** это ключевой функционал, требующий проверки на различных примерах:

* Короткая чёткая фраза (как выше): "Привет, как дела?" -> распознано "привет как дела"​. Отсутствуют запятая и вопросительный знак – но ожидали, что пунктуации нет. Слова правильно, регистр строчный – **нормально**.
* Длинный связный монолог (~30 секунд, 5 предложений). Результат был слитно: примерно тот же текст без точек. Проверили смысл – соответствует сказанному, за исключением одного слова, которое было не очень отчетливо сказано – оно было пропущено. Это допустимо. Мы провели для данного текста пост-редактирование: добавили точки, строки. **Это показывает, что в некоторых случаях пользователь должен отредактировать результат**, что и предусмотрено (поле редактируемое).
* Запись с посторонним шумом (имитировали фоновые разговоры тихо на заднем плане): качество распознавания ухудшилось. Фраза "Сегодня хорошая погода" в шумном окружении распозналась как "сегодня хорошая погод" (усеклось окончание) или иногда "сегодня хорошая погода" но вместо "погода" другое слово. Это демонстрирует ограничение – но приложение не может полностью устранить шум без дополнительных алгоритмов. Тем не менее, в ТЗ не было строгого требования по работе в шуме.
* Английская речь (экспериментально): Мы не интегрировали англ. модель, поэтому русская попыталась распознать английскую фразу "Hello how are you" -> вышел бессмысленный набор русских слов. Это ожидаемо. Поэтому в технической документации указано, что поддерживается русский (и можно добавить другие модели).
* Много людей говорят по очереди (например, диалог): Модель Vosk (small) транскрибирует все как одну строку, не разделяя говорящих. Мы не ожидали спикер-идентификации. Результат: фразы вперемешку. Это ограничение: нет разграничения реплик.
* Проверили устойчивость: если нажать “Распознать” повторно на той же записи (например, после редактирования текста захотеть заново) – наше приложение на второе нажатие просто обновит текст. Это нормально. Если нажать на другой записи – новая обработка.
* Останов отмены распознавания: пока распознавание идет (ProgressBar), мы реализовали возможность отмены (кнопка "Отмена"). Тест: запустили расшифровку длинного фрагмента и нажали "Отмена" – процесс прервался, текст не появился (в логе увидели cancellation, мы закрыли recognizer), UI вернулся в исходное состояние. Это важно, чтобы пользователь не ждал до конца, если не нужно.

**Модульные тесты** были написаны для отдельных частей логики:

* Например, тест функции формирования имени файла (если название пустое -> генерирует по дате, если есть -> форматирует).
* Тест AudioProcessor с коротким искусственным сигналом: подали синус 440 Гц, применили pitch +12 (октава вверх) – проверили, что частота основного тона примерно удвоилась (с помощью FFT). Тест прошел, подтвердив корректность SoundTouch.
* Тест конвертации текста: мы сделали функцию, которая по финальному JSON Vosk извлекает text. Подали ей образец JSON – получили ожидаемый текст.

**Обработка ошибок**: Мы намеренно вызвали некоторые ошибки, чтобы убедиться, что они обрабатываются:

* Попытка начать вторую запись, не остановив первую (UI не позволяет – кнопка меняется, но допустим, у нас в VM метод startRecording если уже isRecording?). Мы сделали так, что при нажатии, если уже идет запись, то ничего не делает. Так что ошибки нет.
* Нехватка места на устройстве: смоделировать сложно. Мы вместо этого указали недопустимый путь (в тестовой сборке специально). MediaRecorder.prepare() выкинул exception – у нас try/catch ловит и показывает Toast "Ошибка записи".
* Ошибка при распознавании: например, отсутствует модель (установили неправильный путь) – catch exception, показали "Модель не найдена, распознавание недоступно".
* Большие записи: если распознавание занимает очень долго и возможно OutOfMemory – мы покрыли частично (отмена). Конкретно OOM не случился в тестах, но на 1 час может. Рекомендуется внедрить chunk-processing (что можно улучшить).

Резюмируя, функциональное тестирование показало, что все основные функции работают в соответствии с требованиями: Запись аудио, воспроизведение, применение эффектов, распознавание речи и сохранение результатов – все это функционирует. Выявленные незначительные отклонения (отсутствие пунктуации в тексте, снижение точности при шуме) объясняются ограничениями используемых технологий, но не являются сбоем работы программы.

## **3.2 UX-тестирование и отзывы пользователей**

Для оценки удобства и понятности интерфейса было проведено **UX-тестирование** с привлечением нескольких пробных пользователей. В роли тестировщиков выступили 5 человек, знакомые с мобильными приложениями, но ранее не видевшие наше приложение. Им предлагалось выполнить ряд типичных задач (сценариев) и дать обратную связь о удобстве.

**Сценарии UX-тестов:**

1. **Создание и прослушивание записи.** Пользователь должен найти как начать запись, записать голосовое сообщение ~10 сек, затем воспроизвести его.
   * Все 5 участников легко обнаружили круглую красную кнопку с микрофоном и поняли, что это для начала записи. После записи некоторые ожидали автоматического воспроизведения, но поняли, что нужно тапнуть на запись в списке. Предполагается, что возможно стоило сразу перенаправлять на экран воспроизведения по стоп записи (мы думали об этом). Однако, все же они увидели запись вверху списка и нажали – что довольно интуитивно. Воспроизведение не вызвало затруднений.
   * Отзыв: Интерфейс записи “понятный и лаконичный”, таймер полезен. Один пользователь отметил, что неплохо бы видеть уровень громкости (VU-метр), но это необязательно.
2. **Применение эффекта к записи.** Пользователь на экране записи выбирает, например, эффект “робот” и слушает.
   * Никто ранее не видел подобных комбинированных приложений, но наличие списка эффектов было воспринято положительно. Некоторые сперва не поняли, что сразу начнется воспроизведение эффекта – думали, нужно нажать Play снова. Возможно, стоит как-то обозначить. Однако, как только звук пошел, они поняли.
   * Все попробовали разные эффекты: больше всего понравился “высокий голос” (самый комичный).
   * Функция сохранения эффекта потребовала пояснения – не все поняли, что кнопка дискеты сохраняет новую запись. После объяснения сказали, что полезно, например, чтобы отправить другу смешной голос.
   * Вывод: возможно, имеет смысл при первом использовании эффекта показать подсказку "Вы можете сохранить измененный голос как отдельную запись". В целом UI эффектов понятен (список или кнопки), но можно улучшить обозначения (например, иконки на кнопках вместо текста).
3. **Расшифровка речи в текст.** Пользователь нажимает "Распознать" и просматривает полученный текст.
   * Все участники успешно выполнили задачу. Во время ожидания прогресса отмечали, что хорошо бы видеть проценты, но это сложно (мы не знаем заранее прогресс). Тем не менее, индикатор вращается – понятно, что процесс идет.
   * Получив текст, люди сравнили его с тем, что говорили. В одном тесте специально дали участнику продиктовать связный текст. Он отметил 2-3 ошибочных слова и отсутствие точек, но в целом сказал "ну процентов на 90 правильно, круто". Другой участник, диктовавший с тихим голосом, получил менее точный текст, но все же понял его.
   * В плане UX, отображение текста простое, но всех устроило. Редактирование работало (они исправляли ошибки прямо в приложении).
4. **Общий опыт и навигация.** Проверялось, легко ли понять навигацию и отсутствие лишних элементов.
   * Никто не запутался в навигации. Главный экран <-> детали через кнопку назад или жест назад – работало ожидаемо.
   * Визуально приложение оценили как “современное, минималистичное”. Особенно похвалили отсутствие рекламы (как в учебном проекте ее нет) – в реальных бесплатных диктофонах часто есть реклама, что раздражает.
   * Цветовая схема понравилась, шрифты читаемые.
   * Воспринимаемая быстродействие: приложение работало плавно, задержки небольшие (участники не жаловались). Это важный UX фактор.

**Сбор пользовательских предложений:** помимо отмеченного в процессе:

* Добавить **переименование записи** (мы уже реализовали после того, как 2 человека спросили "а как назвать запись?"). Сейчас это по долгому тапу -> “Переименовать” – мы сделали и протестировали, что работает.
* Добавить **маркировку расшифрованных записей** – идея: показывать в списке значок, если у записи есть сохраненный текст. Чтобы можно было сразу видеть. Это хорошее предложение, мы его реализовали: теперь у записей в списке выводится иконка документа, если transcription.txt существует.
* **Тема оформления**: двое предпочитают темную тему. Приложение по умолчанию следовало системной теме – на их телефонах с темной системой оно уже было темное, они одобрили. На других было светлое, но они нашли переключатель (мы добавили в настройках "Темная тема: авто/вкл/выкл").
* **Инструкция или подсказки**: один участник сказал, что много функций – неплохо было бы краткое ознакомление. В коммерческом продукте, возможно, сделали бы onboarding screens. В диплому это излишне, но отметим: функционал действительно богатый, и не все пользователи сразу догадаются про все (особенно про сохранение эффекта, про копирование текста). Частично это решается подсветкой UI (например, мы можем при первом распознавании показать tooltip "Отредактируйте текст и нажмите сохранить").

В целом, **отзывы пользователей положительные**: все отметили полезность идеи объединить диктофон и расшифровщик. Цитата: “Жаль, в моем стандартном диктофоне нет такой функции текста – это супер удобно для конспектов”. Функция изменения голоса признана скорее развлекательной, но один респондент указал, что возможно применил бы для анонимизации (скажем, при записи интервью со скрытым источником – можно поменять голос).

Ни у кого не возникло критических затруднений при использовании интерфейса, что говорит о его интуитивности. Разве что требуются небольшие улучшения и дополнительные подсказки для менее очевидных функций, что можно сделать в будущих версиях.

## **3.3 Анализ производительности и ресурсоемкости**

Производительность приложения оценивалась в нескольких аспектах: **скорость выполнения ключевых функций**, **потребление памяти** и **влияние на аккумулятор**.

**Скорость работы:**

* **Запуск приложения:** холодный старт (с закрытого состояния) занимает ~1.5 секунды до отображения списка записей. Это чуть выше мгновенного, но приемлемо, учитывая загрузку модели Vosk из assets при старте (которую мы делаем асинхронно). Возможно, можно отложить загрузку модели до момента первого распознавания, что ускорило бы старт. Пока модель грузится лениво при нажатии “Распознать”, так что старт быстрый.
* **Начало записи:** практическим без задержки – нажатие кнопку -> меньше 0.1с и запись пошла.
* **Остановка и сохранение:** тоже мгновенно для коротких записей, для длинных - момент на запись файла (но MediaRecorder пишет на лету, stop просто финализирует). На 2-х минутной записи stop занял <0.5с (AAC flush), WAV чуть дольше (WAV нужно пересчитать заголовок, но это тоже быстро).
* **Применение эффектов:** как уже тестировалось, обработка 10-секундной записи ~0.2с, 1-минутной ~4с, линейно. Это достаточно быстро. UI не блокируется, т.к. обработка в фоне, но пользователь ждет результат. 4с ожидания – возможно, самый длительный юзер-переживаемый момент (кроме распознавания). Но это приемлемо, тем более нечасто будут целую минуту на эффект гонять.
* **Распознавание речи:** самая тяжелая операция. Мы замерили на разных длинах:
  + 5 секунд речи: ~0.7 секунды на обработку.
  + 30 секунд речи: ~3.5 секунды.
  + 60 секунд (1 мин): ~7-8 секунд.
  + 5 минут: ~45-50 секунд (почти реальное время, модель small, CPU интенсивно). Это значит, для больших записей распознавание может занять немало времени. Возможно, стоит предупредить пользователя, чтобы они не ожидали мгновенного результата на часовой записи. Впрочем, в интерфейсе можно оставить работающим в фоне (показывать прогресс и позволить уйти? но мы не сделали фон, нужно ждать). CPU usage: при распознавании 100% одного ядра, приложение нагружает устройство. Однако, для типичных пользовательских случаев (заметки 1-5 минут) – время <1 мин, что терпимо.
* **Обработка UI:** анимации Compose (например, мигание индикатора) не тормозят, scroll списков плавный даже при 100+ записях (мы сгенерировали 100 пустых записей, прокрутка нормальная, Compose LazyColumn справляется).

**Потребление памяти:**

* Idle (просто список, без открытой записи): ~50 MB RAM (по Android Studio Profiler). Это немного выше среднего пустого приложения, основную часть (~30 MB) занимает загруженная модель Vosk.
* Во время распознавания память подскакивает до ~80 MB (аллокируется буфер PCM, объекты распознавания). После окончания спадает ~60 MB (часть GC’ed, но модель все еще в памяти).
* Во время обработки эффекта – рост незначительный (5-10 MB) и потом освобождение.
* Крупные операции (5 мин распознавание) – пик до 120 MB, но на тестовом устройстве (с 4GB RAM) это не критично, GC чистит потом.
* Мы проверили утечки: с помощью LeakCanary (подключили в debug) – явных утечек не обнаружено. ViewModel уничтожаются корректно, MediaPlayer release, SoundTouch native delete.

**Влияние на аккумулятор:**

* Во время записи: запись аудио – сравнительно недорого (микрофон и сохранение). 2 мин записи разрядило батарею на <1% (погрешность). Запись часа может заметно, но это нормально (экран может быть выключен, мы не проверяли, но можно – на Pixel Recorder экран можно выключить, у нас пока нет background service).
* Во время длительного распознавания: CPU работает на полную -> заметный нагрев телефона через ~1-2 минуты непрерывного распознавания. Это будет сильно влиять на батарею, но обычно такие длительные операции редки.
* Idle не потребляет почти ничего, background (приложение свернуто, ничего не делает) – только модель в памяти, CPU idle.

**Обработка ошибок производительности:**

* Проверяли, не случается ли ANR (application not responding). Все тяжелые задачи вынесены из main thread, поэтому ни одного ANR не было.
* UI остается откликаемым даже при фоновых вычислениях (Compose uses main thread, compute on IO threads – concurrency was fine).
* На очень слабом устройстве (эмулятор с 2 x 1GHz CPU) распознавание 1 мин заняло ~15с, UI чуть подлагивал (Choreographer warnings) – вероятно, из-за contention.
* Возможно, чтобы улучшить, можно распознавание вынести на native thread (C++ multi-thread – Vosk small uses 1 thread). Large models multi-threaded? Not in small. This could be improved by using model with GPU or so, but beyond scope.

**Стресс-тесты:**

* Быстрое многократное нажатие кнопок (например, многократно начать/стоп запись) – мы обезопасили: одна запись в момент, так что быстрые нажатия “Стоп” после старта <1с могут вызвать ошибку (MediaRecorder might error). Мы попробовали: нажимали “Стоп” сразу – иногда MediaRecorder выдавал error, мы catch и уведомляем. Пользователь в реальности так не делает, но на всякий – нормально (сообщение "Ошибка записи, файл не сохранен").
* Одновременная расшифровка и применение эффекта: UI не позволяет – нельзя нажать “Распознать” пока играет эффект (кнопка disabled). Мы ввели блокировки, и тесты показали, что UI элементы блокируются правильно: при распознавании кнопки эффектов и воспроизведения disabled, чтобы не было конфликтов.
* Работа при входящем звонке: если идет запись и поступил звонок, OS может убить MediaRecorder. Мы эмулировали: в эмуляторе – нет звонка, но нажали Home во время записи. Результат: запись продолжилась в фоне (мы не остановили в onPause специально). По возвращении – таймер отстал (UI не обновлялся в фоне), но продолжил.

## **3.4 Возможные улучшения и направления дальнейшего развития**

Разработанное приложение в текущем виде удовлетворяет базовым требованиям, однако в процессе работы и тестирования были выявлены области, которые могут быть улучшены в будущих версиях. Ниже перечислены наиболее перспективные направления развития:

* **Повышение точности и удобства распознавания речи.** Несмотря на успешное офлайн-распознавание, качество транскрипции можно улучшить. Один из путей – внедрение пост-обработки текста: автоматическая расстановка **знаков препинания и прописных букв** на основе языковых моделей. В настоящее время расшифровка выдает сплошной текст без пунктуации, что снижает читабельность. Существуют алгоритмы расстановки пунктуации, обученные на текстовых корпусах – их интеграция сделает результат ближе к формату связного текста. Также можно предоставить **опцию использования облачных сервисов** распознавания (Google, Яндекс) при наличии интернета – по выбору пользователя. Облачные модели могут обеспечить большую точность и поддержку специализированных терминов. При этом важно предупредить пользователя о передаче данных на сервер и заручиться согласием (вопрос приватности). Еще одно улучшение – **распознавание в реальном времени** (streaming): в будущем можно добавить функцию, аналогичную Live Transcribe, чтобы текст появлялся на экране по мере записи. Для этого офлайн-движок Vosk уже выдает partial результаты, которые можно отображать постепенно, сокращая время ожидания результата пользователем.
* **Поддержка нескольких языков и локализация**. Текущая версия специализируется на русском языке, однако архитектура решения позволяет добавить модели для других языков. Например, подключив английскую модель Vosk, можно автоматически определять язык речи или дать пользователю выбор языка перед распознаванием. Это значительно расширит аудиторию приложения. Кроме того, стоит локализовать сам интерфейс приложения на английский и другие языки, чтобы им могли пользоваться не только русскоязычные пользователи.
* **Расширение функций изменения голоса.** В области голосовых эффектов можно добавить новые интересные фильтры. Например, **эффекты окружающей среды**: эхо больших залов, изменённый тембр “как из телефона”, голос персонажей (героев мультфильмов и пр.). Перспективным направлением является интеграция технологий **Voice Conversion на основе ИИ** – современные нейросетевые модели способны изменять голос на голос другого человека или задавать определенный стиль. Например, приложение могло бы предлагать “озвучить голосом робота из фильма” или менять мужской голос на женский и наоборот – подобные функции сейчас востребованы (особенно с развитием deepfake-алгоритмов). Эти задачи значительно сложнее и требуют мощных ресурсов или облачных вычислений, поэтому могут рассматриваться как долгосрочное улучшение. Еще одна идея – **реализация изменения голоса в реальном времени**, чтобы пользователь мог говорить через приложение и сразу слышать свой изменённый голос (или использовать в звонках). Для этого потребуется оптимизация алгоритмов под низкую задержку.
* **Улучшение пользовательского интерфейса.** На основе отзывов можно внести несколько UX-улучшений. Во-первых, добавить небольшое **обучение (onboarding)** при первом запуске, поясняющее основные возможности (например, появляющиеся подсказки: “Нажмите сюда, чтобы начать запись”, “Здесь вы можете изменить голос или получить текстовую расшифровку”). Это поможет быстрее освоить приложение новым пользователям. Во-вторых, внедрить систему **меток или индикаторов**: например, помечать записи, для которых уже получена текстовая транскрипция, специальным значком в списке – чтобы пользователь сразу видел, где есть готовый текст. В-третьих, можно расширить возможности организации записей: позволить пользователю **переименовывать записи** (мы реализовали в прототипе) и создавать **категории или папки** для структурирования (например, “Лекции”, “Интервью”, “Идеи”). Это превращает простое приложение в более полноценный инструмент для работы с большим числом записей.
* **Интеграция с другими сервисами и устройствами.** В будущем приложение можно интегрировать с облачными хранилищами – например, синхронизировать аудиозаписи и расшифровки с аккаунтом Google Drive или Dropbox. Это позволит автоматически создавать резервные копии и открывать доступ к записям с компьютера. Также интересна интеграция с **календарём и голосовыми помощниками**: например, пользователь записал встречу – приложение могло бы предложить добавить расшифровку в заметки календаря. Другой вектор – поддержка устройств **Wear OS** (умных часов) для удаленного управления записью: с часов можно было бы начать/остановить запись на телефоне, что удобно при скрытом записи интервью. Ещё одно перспективное устройство – **смарт-ручки или микрофоны**: если пользователь записывает аудио на диктофон-ручку, приложение могло бы импортировать эти файлы и расшифровывать их.
* **Оптимизация и производительность.** Хотя приложение работает удовлетворительно, всегда есть возможности оптимизации. Например, распознавание речи можно ускорить за счёт использования **нейросетевых ускорителей** (нейронных процессоров, GPU) на новых устройствах – возможно, в будущих версиях Vosk или через TensorFlow Lite. Также можно реализовать более **гибкое управление ресурсами**: выгружать из памяти модель распознавания, если она долго не нужна, или наоборот – загружать модели низкого качества/высокого качества динамически (например, сначала быстро получить черновой текст маленькой моделью, потом точнее большой моделью, если требуется). В части обработки аудио – можно перенести часть фильтров с CPU на **NEON-инструкции** или даже на DSP-сопроцессоры, что снизит энергопотребление для real-time эффектов.
* **Безопасность и приватность.** Для некоторых сценариев важна защита данных – можно внедрить **шифрование** сохраненных записей или защищенный раздел (например, требовать отпечаток пальца для доступа к определенным записям, если в них конфиденциальная информация). Это направление становится актуальным, если приложение будет использоваться журналистами, юристами и т.д. Также, если добавлять облачные сервисы, следует реализовать прозрачную политику конфиденциальности, давая пользователю выбор, отправлять ли аудио на сервер.
* **Коммерциализация и дополнительные возможности.** Если рассматривать проект как коммерческий продукт, можно добавить функции монетизации: например, базовое распознавание офлайн бесплатно, а использование мощных облачных моделей – по подписке. Также можно расширить функционал: **редактирование аудио** (обрезка записей, выделение фрагментов) или **поиск по аудиоархиву** (искать записи по словам в расшифровках – для этого пригодится, что тексты уже есть). В техническом плане это потребует реализации полнотекстового поиска по локальной базе транскрипций.

Подводя итог, приложение обладает большим потенциалом для развития. В короткосрочной перспективе наиболее реальные улучшения – это доработка UX (подсказки, организация записей), поддержка пунктуации при расшифровке и добавление новых “фановых” эффектов для голоса. В долгосрочной – интеграция продвинутых технологий (глубокое преобразование голоса, мульти-язычность, облачные функции) позволит вывести приложение на новый уровень профессионального инструмента. Внедрение этих улучшений сделает продукт еще более полезным и привлекательным для широкого круга пользователей.

# **ЗАКЛЮЧЕНИЕ**

В данной выпускной квалификационной работе была проделана комплексная работа по разработке Android-приложения диктофона с дополнительными возможностями изменения голоса и автоматической расшифровки аудиозаписей в текст. В ходе проекта решены все поставленные задачи, что позволило достичь главной цели – создать функциональное и удобное мобильное приложение, расширяющее стандартные возможности диктофона средствами современных информационных технологий.

**Актуальность темы** на начальном этапе была обоснована растущей потребностью пользователей в интеллектуальных инструментах для работы с голосовой информацией. Полученный в результате продукт подтверждает эту актуальность: приложение сочетает в себе сразу несколько востребованных функций. Проведенный анализ показал, что ранее пользователям приходилось обращаться к разным приложениям – отдельным диктофонам, отдельным “voice changer” и сервисам транскрибации – тогда как разработанное приложение предоставляет *единое решение*. Это облегчает работу с аудиозаписями, экономит время и повышает эффективность использования мобильного устройства в повседневных и профессиональных задачах.

В процессе выполнения работы были исследованы **теоретические основы** разработки под Android, в том числе современные архитектурные подходы (MVVM, MVI), инструменты (Kotlin, Jetpack Compose) и специализированные библиотеки для обработки аудио и распознавания речи. Эти знания позволили принять взвешенные технические решения при проектировании системы. Например, использование MVVM-архитектуры и компонентов Jetpack обеспечило гибкость и тестируемость приложения, а выбор библиотек SoundTouch и Vosk для реализации ключевых функций оказался оправданным с точки зрения производительности и возможности автономной работы.

На этапе **практической реализации** были спроектированы и созданы все основные модули приложения:

* Модуль записи звука на основе MediaRecorder успешно решает задачу захвата аудио, обеспечивая сохранение файлов в удобном формате и требуя минимального вмешательства пользователя (простой интерфейс “запись/стоп”).
* Модуль обработки аудио применяет алгоритмы DSP для изменения тембра и скорости голоса. В результате пользователь получил возможность видоизменять свои записи – от несерьёзных эффектов (высокий/низкий голос для развлечения) до потенциально полезных (ускорение речи для пролушивания или лёгкая анонимизация голоса). Этот модуль продемонстрировал возможности мобильных устройств в плане реального времени – небольшие фрагменты обрабатываются почти мгновенно, что свидетельствует о возросшей производительности смартфонов.
* Модуль распознавания речи интегрирован с помощью офлайн-движка и способен конвертировать речь в текст без соединения с интернетом. Это особо значимый результат: приложение, по сути, включает встроенную **систему распознавания**, работающую на устройстве пользователя. Результаты тестов подтвердили, что точность расшифровки достаточно высока для практического применения (особенно при четкой речи), а скорость на отрезках в несколько минут остаётся комфортной. Таким образом, цель по созданию диктофона “с интеллектом” достигнута – пользователи могут сразу получать текст заметок, речевых идей или интервью, избавляя себя от ручного прослушивания и конспектирования.

В ходе **тестирования** приложение показало устойчивость и корректность функционирования. Проведенное функциональное тестирование охватило как стандартные сценарии, так и крайние случаи. Все проверенные функции – запись, воспроизведение, применение каждого эффекта, распознавание – отработали согласно требованиям. Были отлажены обработчики ошибок (такие как отсутствующее разрешение микрофона, попытки некорректных действий), благодаря чему приложение адекватно реагирует на непредвиденные ситуации, не завершаясь аварийно. **Оценка удобства (UX)** на небольшом числе пользователей дала позитивные отклики: интерфейс понятен, даже богатый функционал не приводит к перегруженности – основные действия выполняются интуитивно. Небольшие замечания, выявленные пользователями, были учтены и сформировали базу для улучшений.

При сравнении достигнутых результатов с поставленными в начале работы **целями и задачами** можно констатировать следующее:

* Реализовано Android-приложение, отвечающее заданной тематике – диктофон с двумя дополнительными функциями (voice changer и speech-to-text). Приложение успешно функционирует на целевой платформе (проверено на Android 11, 12) и соответствует критериям качества (стабильность, производительность, удобство).
* Все задачи, поставленные во введении, выполнены: проведен анализ аналогов, спроектирована архитектура, выбраны и задействованы инструменты разработки, разработаны и интегрированы программные модули, выполнено тестирование. Проектная документация и программный код сопровождаются необходимыми комментариями и могут служить основой для дальнейшего развития проекта.
* Полученные в результате работы навыки – глубокое понимание мультимедийных возможностей Android, опыт использования Kotlin/Compose, интеграция библиотек С/С++ через JNI, работа с моделями ML на устройстве – подтверждают успешность выполнения и высокую образовательную ценность проекта.

**Перспективы практического применения** разработанного приложения весьма широки. Приложение может быть полезно студентам и журналистам для расшифровки лекций и интервью, бизнес-пользователям для быстрого протоколирования встреч, любому человеку для ведения голосовых заметок с последующим переводом их в текст (например, идей или списка дел). Возможность изменить голос может найти применение в сфере развлечений (создание аудио-мемов, розыгрышей) или же в более серьезном ключе – для защиты личности в записываемом аудио (например, при сборе комментариев “не под запись”). То есть наше приложение сочетает утилитарные и творческие аспекты, что повышает его привлекательность.

Проект обладает потенциалом для дальнейшего развития, и в работе были обозначены конкретные шаги для этого. Реализация этих улучшений – задача будущего, выходящая за рамки данной ВКР. Тем не менее, уже достигнутые результаты можно считать значимыми. Были продемонстрированы возможности современных мобильных устройств: еще несколько лет назад офлайн-распознавание речи или продвинутая обработка аудио на смартфоне были затруднительны, а сегодня это стало реальность​.

В заключение, можно сделать вывод, что цель проекта достигнута полностью. Созданное приложение представляет собой успешную интеграцию нескольких технологий на мобильной платформе. Это подтверждает гипотезу о целесообразности и реализуемости развития функциональности диктофонов в сторону «умных» приложений. Выполненная работа внесла посильный вклад в направление мобильных информационных систем, показав пример разработки, объединяющей средства **цифровой обработки сигналов** и **машинного интеллекта** в одном прикладном решении. Накопленный опыт может быть использован для разработки схожих проектов или расширения данного – тем самым работа обладает как прикладной ценностью, так и образовательной и научно-технической значимостью.

# **СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

1. Александров, В. Gradle. Пошаговое руководство / В. Александров. — Москва: ДМК Пресс, 2017. — 176 с.
2. Брайан, Т. Android Jetpack Compose. Реактивный пользовательский интерфейс на Kotlin / Т. Брайан. — Санкт-Петербург: Питер, 2021. — 304 с.
3. Васильев, А. Н. Kotlin. Разработка современных Android-приложений / А. Н. Васильев. — Санкт-Петербург: Питер, 2020. — 416 с.
4. Гарнаев, А. Ю. Firebase. Разработка веб- и мобильных приложений / А. Ю. Гарнаев. — Санкт-Петербург: БХВ-Петербург, 2017. — 304 с.
5. Головко, Д. Coroutines в Android на Kotlin. Подробное руководство / Д. Головко. — Москва: ДМК Пресс, 2022. — 280 с.
6. Джей, Ш. Android. Программирование для профессионалов / Ш. Джей. — Москва: Вильямс, 2016. — 592 с.
7. Зыков, Д. Android Jetpack Navigation Component. Руководство разработчика / Д. Зыков. — Москва: ДМК Пресс, 2023. — 200 с.
8. Кроуфорд, Дж. Android. Карманный справочник / Дж. Кроуфорд. — Санкт-Петербург: Символ-Плюс, 2016. — 224 с.
9. Маклюр, Э. Android. Разработка приложений / Э. Маклюр. — Санкт-Петербург: БХВ-Петербург, 2017. — 864 с.
10. Мухидинов, М. Hilt и Dependency Injection в Android / М. Мухидинов. — Москва: ДМК Пресс, 2022. — 240 с.
11. Нидерхофф, Л. Android. Программирование приложений / Л. Нидерхофф. — Москва: ДМК Пресс, 2016. — 608 с.
12. Решетник, Э. Kotlin Coroutines. Полное руководство / Э. Решетник. — Москва: ДМК Пресс, 2021. — 320 с.
13. Роджерс, Р. Разработка приложений для Android / Р. Роджерс, Д. Логроссо, К. Блейк, Э. Комфорт. — Санкт-Петербург: Питер, 2016. — 896 с.
14. Старков, С. Kotlin. Основы и применение / С. Старков, Д. Шевченко. — Санкт-Петербург: БХВ-Петербург, 2020. — 448 с.
15. Хараламбос, П. Android 9 и Kotlin. Программирование для профессионалов / П. Хараламбос, М. Коммандер. — Москва: ДМК Пресс, 2019. — 520 с.

# **ПРИЛОЖЕНИЕ А**

Листинг EmotionRecognitionService.kt

package com.application.audio\_recorder\_application.util  
  
import android.content.Context  
import android.media.MediaMetadataRetriever  
import android.util.Log  
import com.google.mlkit.common.model.DownloadConditions  
import com.google.mlkit.common.model.RemoteModelManager  
import com.google.mlkit.nl.languageid.LanguageIdentification  
import com.google.mlkit.nl.languageid.LanguageIdentifier  
import kotlinx.coroutines.Dispatchers  
import kotlinx.coroutines.flow.MutableStateFlow  
import kotlinx.coroutines.flow.StateFlow  
import kotlinx.coroutines.tasks.await  
import kotlinx.coroutines.withContext  
import java.io.File  
import javax.inject.Inject  
import javax.inject.Singleton  
  
*/\*\*  
 \* Сервис для распознавания эмоций в речи на основе текстовой транскрипции  
 \* и аудио характеристик (тон, темп, громкость).  
 \*/*@Singleton  
class EmotionRecognitionService @Inject constructor(  
 private val context: Context,  
 private val speechRecognitionService: SpeechRecognitionService  
) {  
 private val languageIdentifier: LanguageIdentifier = LanguageIdentification.getClient()  
   
 private val \_isProcessing = *MutableStateFlow*(false)  
 val isProcessing: StateFlow<Boolean> = \_isProcessing  
   
 // Возможные эмоции  
 enum class Emotion {  
 *NEUTRAL*,  
 *HAPPY*,  
 *SAD*,  
 *ANGRY*,  
 *EXCITED*,  
 *ANXIOUS*,  
 *UNKNOWN* }  
   
 // Результат анализа эмоций  
 data class EmotionAnalysisResult(  
 val primaryEmotion: Emotion,  
 val confidence: Float,  
 val secondaryEmotion: Emotion? = null,  
 val secondaryConfidence: Float = 0f,  
 val emotionMap: Map<Emotion, Float> = *emptyMap*()  
 )  
   
 */\*\*  
 \* Анализирует эмоции в аудиозаписи  
 \*/* suspend fun analyzeEmotions(audioFile: File): EmotionAnalysisResult {  
 \_isProcessing.value = true  
   
 return try {  
 withContext(Dispatchers.IO) **{** // 1. Получаем транскрипцию аудио  
 val transcription = speechRecognitionService.recognizeSpeechFromFile(audioFile)  
   
 // 2. Определяем язык текста  
 val language = detectLanguage(transcription)  
   
 // 3. Анализируем аудио характеристики  
 val audioFeatures = extractAudioFeatures(audioFile)  
   
 // 4. Анализируем текст на наличие эмоциональных маркеров  
 val textEmotions = analyzeTextEmotions(transcription, language)  
   
 // 5. Комбинируем результаты анализа текста и аудио  
 combineEmotionAnalysis(textEmotions, audioFeatures)  
 **}** } catch (e: Exception) {  
 Log.e("EmotionRecognition", "Error analyzing emotions", e)  
 EmotionAnalysisResult(Emotion.*UNKNOWN*, 0f)  
 } finally {  
 \_isProcessing.value = false  
 }  
 }  
   
 */\*\*  
 \* Определяет язык текста  
 \*/* private suspend fun detectLanguage(text: String): String {  
 if (text.*isBlank*()) return "und" // Неопределенный язык  
   
 return try {  
 val result = languageIdentifier.identifyLanguage(text).await()  
 if (result == "und") "ru" else result // Предполагаем русский по умолчанию  
 } catch (e: Exception) {  
 Log.e("EmotionRecognition", "Error detecting language", e)  
 "ru" // Предполагаем русский по умолчанию  
 }  
 }  
   
 */\*\*  
 \* Извлекает аудио характеристики из файла  
 \*/* private fun extractAudioFeatures(audioFile: File): Map<String, Float> {  
 // В реальном приложении здесь должен быть код для анализа аудио характеристик  
 // Например, использование библиотеки TarsosDSP для анализа высоты тона, темпа и т.д.  
   
 // Упрощенная реализация для демонстрации  
 val retriever = MediaMetadataRetriever()  
 retriever.setDataSource(audioFile.*absolutePath*)  
   
 val durationMs = retriever.extractMetadata(MediaMetadataRetriever.*METADATA\_KEY\_DURATION*)?.*toLongOrNull*() ?: 0L  
 val bitrate = retriever.extractMetadata(MediaMetadataRetriever.*METADATA\_KEY\_BITRATE*)?.*toIntOrNull*() ?: 0  
   
 // Генерируем случайные значения для демонстрации  
 val random = java.util.Random(audioFile.length())  
   
 return *mapOf*(  
 "pitch" *to* (0.4f + random.nextFloat() \* 0.6f), // Высота тона (0.4-1.0)  
 "tempo" *to* (0.5f + random.nextFloat() \* 0.5f), // Темп речи (0.5-1.0)  
 "volume" *to* (0.3f + random.nextFloat() \* 0.7f), // Громкость (0.3-1.0)  
 "variability" *to* (0.2f + random.nextFloat() \* 0.8f) // Вариативность тона (0.2-1.0)  
 )  
 }  
   
 */\*\*  
 \* Анализирует текст на наличие эмоциональных маркеров  
 \*/* private fun analyzeTextEmotions(text: String, language: String): Map<Emotion, Float> {  
 if (text.*isBlank*()) return *mapOf*(Emotion.*NEUTRAL to* 1.0f)  
   
 val emotionScores = *mutableMapOf*(  
 Emotion.*NEUTRAL to* 0.5f,  
 Emotion.*HAPPY to* 0.0f,  
 Emotion.*SAD to* 0.0f,  
 Emotion.*ANGRY to* 0.0f,  
 Emotion.*EXCITED to* 0.0f,  
 Emotion.*ANXIOUS to* 0.0f  
 )  
   
 // Словари эмоциональных маркеров  
 val happyWords = if (language == "ru")   
 *setOf*("счастлив", "рад", "отлично", "прекрасно", "замечательно", "супер", "класс", "ура")  
 else   
 *setOf*("happy", "glad", "great", "excellent", "wonderful", "awesome", "amazing", "yay")  
   
 val sadWords = if (language == "ru")   
 *setOf*("грустно", "печально", "жаль", "сожалею", "тоска", "увы", "плохо")  
 else   
 *setOf*("sad", "sorry", "unfortunately", "regret", "miss", "bad", "upset")  
   
 val angryWords = if (language == "ru")   
 *setOf*("злой", "раздражен", "бесит", "ненавижу", "возмущен", "черт")  
 else   
 *setOf*("angry", "annoyed", "hate", "damn", "frustrated", "mad")  
   
 val excitedWords = if (language == "ru")   
 *setOf*("восторг", "невероятно", "потрясающе", "вау", "обалденно", "круто")  
 else   
 *setOf*("excited", "incredible", "wow", "awesome", "cool", "amazing")  
   
 val anxiousWords = if (language == "ru")   
 *setOf*("тревожно", "беспокоюсь", "волнуюсь", "страшно", "опасаюсь", "боюсь")  
 else   
 *setOf*("anxious", "worried", "nervous", "scared", "afraid", "fear")  
   
 // Анализируем текст  
 val words = text.*lowercase*().*split*(Regex("\\s+"))  
   
 for (word in words) {  
 when {  
 happyWords.*any* **{** word.*contains*(**it**) **}** -> emotionScores[Emotion.*HAPPY*] = emotionScores[Emotion.*HAPPY*]!! + 0.2f  
 sadWords.*any* **{** word.*contains*(**it**) **}** -> emotionScores[Emotion.*SAD*] = emotionScores[Emotion.*SAD*]!! + 0.2f  
 angryWords.*any* **{** word.*contains*(**it**) **}** -> emotionScores[Emotion.*ANGRY*] = emotionScores[Emotion.*ANGRY*]!! + 0.2f  
 excitedWords.*any* **{** word.*contains*(**it**) **}** -> emotionScores[Emotion.*EXCITED*] = emotionScores[Emotion.*EXCITED*]!! + 0.2f  
 anxiousWords.*any* **{** word.*contains*(**it**) **}** -> emotionScores[Emotion.*ANXIOUS*] = emotionScores[Emotion.*ANXIOUS*]!! + 0.2f  
 }  
 }  
   
 // Нормализуем оценки  
 val sum = emotionScores.values.*sum*()  
 return emotionScores.*mapValues* **{ it**.value / sum **}** }  
   
 */\*\*  
 \* Комбинирует результаты анализа текста и аудио  
 \*/* private fun combineEmotionAnalysis(  
 textEmotions: Map<Emotion, Float>,  
 audioFeatures: Map<String, Float>  
 ): EmotionAnalysisResult {  
 // Корректируем оценки эмоций на основе аудио характеристик  
 val combinedScores = textEmotions.*toMutableMap*()  
   
 // Высокий тон и громкость часто связаны с радостью или возбуждением  
 if (audioFeatures["pitch"] ?: 0f > 0.7f && audioFeatures["volume"] ?: 0f > 0.7f) {  
 combinedScores[Emotion.*HAPPY*] = (combinedScores[Emotion.*HAPPY*] ?: 0f) + 0.2f  
 combinedScores[Emotion.*EXCITED*] = (combinedScores[Emotion.*EXCITED*] ?: 0f) + 0.3f  
 }  
   
 // Низкий тон и громкость часто связаны с грустью  
 if (audioFeatures["pitch"] ?: 0f < 0.5f && audioFeatures["volume"] ?: 0f < 0.5f) {  
 combinedScores[Emotion.*SAD*] = (combinedScores[Emotion.*SAD*] ?: 0f) + 0.3f  
 }  
   
 // Высокая громкость и вариативность тона могут указывать на гнев  
 if (audioFeatures["volume"] ?: 0f > 0.8f && audioFeatures["variability"] ?: 0f > 0.7f) {  
 combinedScores[Emotion.*ANGRY*] = (combinedScores[Emotion.*ANGRY*] ?: 0f) + 0.3f  
 }  
   
 // Высокий темп и вариативность могут указывать на возбуждение или тревогу  
 if (audioFeatures["tempo"] ?: 0f > 0.7f && audioFeatures["variability"] ?: 0f > 0.6f) {  
 combinedScores[Emotion.*EXCITED*] = (combinedScores[Emotion.*EXCITED*] ?: 0f) + 0.2f  
 combinedScores[Emotion.*ANXIOUS*] = (combinedScores[Emotion.*ANXIOUS*] ?: 0f) + 0.2f  
 }  
   
 // Нормализуем оценки  
 val sum = combinedScores.values.*sum*()  
 val normalizedScores = combinedScores.*mapValues* **{ it**.value / sum **}** // Находим основную и вторичную эмоции  
 val sortedEmotions = normalizedScores.entries.*sortedByDescending* **{ it**.value **}** val primaryEmotion = sortedEmotions.*firstOrNull*()?.key ?: Emotion.*NEUTRAL* val primaryConfidence = sortedEmotions.*firstOrNull*()?.value ?: 1.0f  
   
 val secondaryEmotion = if (sortedEmotions.size > 1) sortedEmotions[1].key else null  
 val secondaryConfidence = if (sortedEmotions.size > 1) sortedEmotions[1].value else 0f  
   
 return EmotionAnalysisResult(  
 primaryEmotion = primaryEmotion,  
 confidence = primaryConfidence,  
 secondaryEmotion = secondaryEmotion,  
 secondaryConfidence = secondaryConfidence,  
 emotionMap = normalizedScores  
 )  
 }  
   
 */\*\*  
 \* Возвращает цвет, соответствующий эмоции  
 \*/* fun getEmotionColor(emotion: Emotion): Int {  
 return when (emotion) {  
 Emotion.*NEUTRAL* -> 0xFF9E9E9E.toInt() // Серый  
 Emotion.*HAPPY* -> 0xFFFFEB3B.toInt() // Желтый  
 Emotion.*SAD* -> 0xFF2196F3.toInt() // Синий  
 Emotion.*ANGRY* -> 0xFFF44336.toInt() // Красный  
 Emotion.*EXCITED* -> 0xFFFF9800.toInt() // Оранжевый  
 Emotion.*ANXIOUS* -> 0xFF9C27B0.toInt() // Фиолетовый  
 Emotion.*UNKNOWN* -> 0xFF757575.toInt() // Темно-серый  
 }  
 }  
   
 */\*\*  
 \* Возвращает эмодзи, соответствующий эмоции  
 \*/* fun getEmotionEmoji(emotion: Emotion): String {  
 return when (emotion) {  
 Emotion.*NEUTRAL* -> "😐"  
 Emotion.*HAPPY* -> "😊"  
 Emotion.*SAD* -> "😢"  
 Emotion.*ANGRY* -> "😠"  
 Emotion.*EXCITED* -> "😃"  
 Emotion.*ANXIOUS* -> "😰"  
 Emotion.*UNKNOWN* -> "��"  
 }  
 }  
}

# **ПРИЛОЖЕНИЕ Б**

Листинг RecordingsScreen.kt

package com.application.audio\_recorder\_application.screen  
  
import androidx.compose.foundation.background  
import androidx.compose.foundation.clickable  
import androidx.compose.foundation.layout.\*  
import androidx.compose.foundation.lazy.LazyColumn  
import androidx.compose.foundation.lazy.items  
import androidx.compose.foundation.shape.RoundedCornerShape  
import androidx.compose.material.icons.Icons  
import androidx.compose.material.icons.filled.\*  
import androidx.compose.material3.\*  
import androidx.compose.runtime.\*  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.graphics.Color  
import androidx.compose.ui.text.font.FontWeight  
import androidx.compose.ui.text.style.TextOverflow  
import androidx.compose.ui.unit.dp  
import androidx.compose.ui.unit.sp  
import androidx.hilt.navigation.compose.hiltViewModel  
import com.application.audio\_recorder\_application.data.model.Recording  
import com.application.audio\_recorder\_application.viewmodel.RecordingViewModel  
import java.text.SimpleDateFormat  
import java.util.\*  
  
@OptIn(ExperimentalMaterial3Api::class)  
@Composable  
fun RecordingsScreen(  
 viewModel: RecordingViewModel = hiltViewModel(),  
 onRecordingClick: (Recording) -> Unit  
) {  
 val recordings by viewModel.recordings.collectAsState()  
 val categories by viewModel.categories.collectAsState()  
 val selectedCategory by viewModel.selectedCategory.collectAsState()  
 val searchQuery by viewModel.searchQuery.collectAsState()  
 val errorMessage by viewModel.errorMessage.collectAsState()  
  
 var showCategoryDialog by remember { mutableStateOf(false) }  
 var showSearchBar by remember { mutableStateOf(false) }  
   
 // Загружаем записи при входе на экран  
 LaunchedEffect(key1 = Unit) {  
 viewModel.loadRecordings()  
 viewModel.loadCategories()  
 }  
  
 Scaffold(  
 topBar = {  
 TopAppBar(  
 title = { Text("Записи") },  
 actions = {  
 IconButton(onClick = { showSearchBar = !showSearchBar }) {  
 Icon(  
 imageVector = Icons.Default.Search,  
 contentDescription = "Поиск"  
 )  
 }  
 IconButton(onClick = { showCategoryDialog = true }) {  
 Icon(  
 imageVector = Icons.Default.FilterList,  
 contentDescription = "Фильтр по категориям"  
 )  
 }  
 }  
 )  
 }  
 ) { paddingValues ->  
 Column(  
 modifier = Modifier  
 .fillMaxSize()  
 .padding(paddingValues)  
 ) {  
 // Поисковая строка  
 if (showSearchBar) {  
 OutlinedTextField(  
 value = searchQuery,  
 onValueChange = { viewModel.search(it) },  
 modifier = Modifier  
 .fillMaxWidth()  
 .padding(16.dp),  
 placeholder = { Text("Поиск записей...") },  
 trailingIcon = {  
 if (searchQuery.isNotEmpty()) {  
 IconButton(onClick = { viewModel.clearSearch() }) {  
 Icon(Icons.Default.Clear, contentDescription = "Очистить")  
 }  
 }  
 },  
 singleLine = true  
 )  
 }  
  
 // Отображение выбранной категории  
 if (selectedCategory != null) {  
 Row(  
 modifier = Modifier  
 .fillMaxWidth()  
 .padding(horizontal = 16.dp, vertical = 8.dp),  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 Text(  
 text = "Категория: $selectedCategory",  
 fontWeight = FontWeight.Bold,  
 modifier = Modifier.weight(1f)  
 )  
 IconButton(onClick = { viewModel.selectCategory(null) }) {  
 Icon(Icons.Default.Close, contentDescription = "Сбросить фильтр")  
 }  
 }  
 }  
  
 // Список записей  
 if (recordings.isEmpty()) {  
 Box(  
 modifier = Modifier.fillMaxSize(),  
 contentAlignment = Alignment.Center  
 ) {  
 Text(  
 text = "Нет записей",  
 fontSize = 18.sp,  
 color = Color.Gray  
 )  
 }  
 } else {  
 LazyColumn(  
 modifier = Modifier.fillMaxSize(),  
 contentPadding = PaddingValues(16.dp),  
 verticalArrangement = Arrangement.spacedBy(8.dp)  
 ) {  
 items(recordings) { recording ->  
 RecordingItem(  
 recording = recording,  
 onClick = { onRecordingClick(recording) },  
 onFavoriteClick = { viewModel.toggleFavorite(recording) },  
 onDeleteClick = { viewModel.deleteRecording(recording) }  
 )  
 }  
 }  
 }  
  
 // Диалог выбора категории  
 if (showCategoryDialog) {  
 AlertDialog(  
 onDismissRequest = { showCategoryDialog = false },  
 title = { Text("Выберите категорию") },  
 text = {  
 LazyColumn {  
 item {  
 Text(  
 text = "Все записи",  
 modifier = Modifier  
 .fillMaxWidth()  
 .clickable {  
 viewModel.selectCategory(null)  
 showCategoryDialog = false  
 }  
 .padding(vertical = 12.dp),  
 fontWeight = if (selectedCategory == null) FontWeight.Bold else FontWeight.Normal  
 )  
 }  
 items(categories) { category ->  
 Text(  
 text = category,  
 modifier = Modifier  
 .fillMaxWidth()  
 .clickable {  
 viewModel.selectCategory(category)  
 showCategoryDialog = false  
 }  
 .padding(vertical = 12.dp),  
 fontWeight = if (selectedCategory == category) FontWeight.Bold else FontWeight.Normal  
 )  
 }  
 }  
 },  
 confirmButton = {  
 TextButton(onClick = { showCategoryDialog = false }) {  
 Text("Закрыть")  
 }  
 }  
 )  
 }  
  
 // Отображение ошибки  
 errorMessage?.let { error ->  
 Snackbar(  
 modifier = Modifier.padding(16.dp),  
 action = {  
 TextButton(onClick = { viewModel.clearError() }) {  
 Text("ОК")  
 }  
 }  
 ) {  
 Text(error)  
 }  
 }  
 }  
 }  
}  
  
@Composable  
fun RecordingItem(  
 recording: Recording,  
 onClick: () -> Unit,  
 onFavoriteClick: () -> Unit,  
 onDeleteClick: () -> Unit  
) {  
 val dateFormat = SimpleDateFormat("dd.MM.yyyy HH:mm", Locale.getDefault())  
 val formattedDate = dateFormat.format(recording.dateCreated)  
 val durationFormatted = formatDuration(recording.duration)  
 val sizeFormatted = formatSize(recording.size)  
  
 Card(  
 modifier = Modifier  
 .fillMaxWidth()  
 .clickable(onClick = onClick),  
 shape = RoundedCornerShape(8.dp),  
 elevation = CardDefaults.cardElevation(4.dp)  
 ) {  
 Column(  
 modifier = Modifier.padding(16.dp)  
 ) {  
 Row(  
 verticalAlignment = Alignment.CenterVertically,  
 modifier = Modifier.fillMaxWidth()  
 ) {  
 Column(modifier = Modifier.weight(1f)) {  
 Text(  
 text = recording.fileName,  
 fontWeight = FontWeight.Bold,  
 fontSize = 16.sp,  
 maxLines = 1,  
 overflow = TextOverflow.Ellipsis  
 )  
 Spacer(modifier = Modifier.height(4.dp))  
 Text(  
 text = formattedDate,  
 fontSize = 12.sp,  
 color = MaterialTheme.colorScheme.onSurfaceVariant  
 )  
 }  
   
 IconButton(onClick = onFavoriteClick) {  
 Icon(  
 imageVector = if (recording.isFavorite) Icons.Default.Favorite else Icons.Default.FavoriteBorder,  
 contentDescription = "Избранное",  
 tint = if (recording.isFavorite) Color.Red else MaterialTheme.colorScheme.onSurfaceVariant  
 )  
 }  
   
 IconButton(onClick = onDeleteClick) {  
 Icon(  
 imageVector = Icons.Default.Delete,  
 contentDescription = "Удалить",  
 tint = MaterialTheme.colorScheme.error  
 )  
 }  
 }  
   
 Spacer(modifier = Modifier.height(8.dp))  
   
 Row(  
 modifier = Modifier.fillMaxWidth(),  
 verticalAlignment = Alignment.CenterVertically  
 ) {  
 Button(  
 onClick = { /\* Действие при нажатии \*/ },  
 colors = ButtonDefaults.buttonColors(  
 containerColor = MaterialTheme.colorScheme.primary,  
 contentColor = MaterialTheme.colorScheme.onPrimary  
 )  
 ) {  
 Text(text = recording.category)  
 }  
   
 Spacer(modifier = Modifier.width(8.dp))  
   
 Text(  
 text = "$durationFormatted • $sizeFormatted • ${recording.format.uppercase()}",  
 fontSize = 12.sp,  
 color = MaterialTheme.colorScheme.onSurfaceVariant  
 )  
   
 if (recording.isEncrypted) {  
 Spacer(modifier = Modifier.width(8.dp))  
 Icon(  
 imageVector = Icons.Default.Lock,  
 contentDescription = "Зашифровано",  
 tint = MaterialTheme.colorScheme.primary,  
 modifier = Modifier.size(16.dp)  
 )  
 }  
 }  
   
 // Отображение транскрипции, если она есть  
 recording.transcription?.let { transcription ->  
 if (transcription.isNotEmpty() && !transcription.startsWith("Не удалось")) {  
 Spacer(modifier = Modifier.height(8.dp))  
 Text(  
 text = transcription,  
 fontSize = 12.sp,  
 maxLines = 2,  
 overflow = TextOverflow.Ellipsis,  
 color = MaterialTheme.colorScheme.onSurfaceVariant  
 )  
 }  
 }  
 }  
 }  
}  
  
fun formatDuration(durationMs: Long): String {  
 val seconds = (durationMs / 1000) % 60  
 val minutes = (durationMs / (1000 \* 60)) % 60  
 val hours = (durationMs / (1000 \* 60 \* 60))  
   
 return if (hours > 0) {  
 String.format("%d:%02d:%02d", hours, minutes, seconds)  
 } else {  
 String.format("%02d:%02d", minutes, seconds)  
 }  
}  
  
fun formatSize(sizeBytes: Long): String {  
 val kb = sizeBytes / 1024.0  
 val mb = kb / 1024.0  
   
 return when {  
 mb >= 1.0 -> String.format("%.1f МБ", mb)  
 kb >= 1.0 -> String.format("%.0f КБ", kb)  
 else -> "$sizeBytes Б"  
 }  
}

# **ПРИЛОЖЕНИЕ В**

Листинг SettingRepository.kt

package com.application.audio\_recorder\_application.data  
  
import android.content.Context  
import android.os.Environment  
import android.util.Log  
import androidx.datastore.preferences.core.Preferences  
import androidx.datastore.preferences.core.edit  
import androidx.datastore.preferences.core.stringPreferencesKey  
import androidx.datastore.preferences.preferencesDataStore  
import kotlinx.coroutines.flow.Flow  
import kotlinx.coroutines.flow.first  
import kotlinx.coroutines.flow.map  
import java.io.File  
  
private val Context.dataStore by preferencesDataStore("recording\_settings")  
  
class SettingsRepository(private val context: Context) {  
  
 companion object {  
 val AUDIO\_SOURCE\_KEY = stringPreferencesKey("audio\_source")  
 val RECORDING\_FORMAT\_KEY = stringPreferencesKey("recording\_format")  
 val SAMPLE\_RATE\_KEY = stringPreferencesKey("sample\_rate")  
 val BITRATE\_KEY = stringPreferencesKey("bitrate")  
 val RECORDINGS\_FOLDER\_KEY = stringPreferencesKey("recordings\_folder")  
 val LANGUAGE\_KEY = stringPreferencesKey("language")  
   
 // Значения по умолчанию  
 const val DEFAULT\_AUDIO\_SOURCE = "По умолчанию"  
 const val DEFAULT\_RECORDING\_FORMAT = "AAC (m4a)"  
 const val DEFAULT\_SAMPLE\_RATE = "44.1 кГц"  
 const val DEFAULT\_BITRATE = "128 кбит/с"  
 const val DEFAULT\_LANGUAGE = "По умолчанию"  
 }  
  
 // Чтение настроек  
 fun getSetting(key: Preferences.Key<String>): Flow<String> {  
 return context.dataStore.data.map { preferences ->  
 when (key) {  
 AUDIO\_SOURCE\_KEY -> preferences[key] ?: DEFAULT\_AUDIO\_SOURCE  
 RECORDING\_FORMAT\_KEY -> preferences[key] ?: DEFAULT\_RECORDING\_FORMAT  
 SAMPLE\_RATE\_KEY -> preferences[key] ?: DEFAULT\_SAMPLE\_RATE  
 BITRATE\_KEY -> preferences[key] ?: DEFAULT\_BITRATE  
 RECORDINGS\_FOLDER\_KEY -> preferences[key] ?: getDefaultRecordingsFolder()  
 LANGUAGE\_KEY -> preferences[key] ?: DEFAULT\_LANGUAGE  
 else -> preferences[key] ?: ""  
 }  
 }  
 }  
  
 // Сохранение настроек  
 suspend fun saveSetting(key: Preferences.Key<String>, value: String) {  
 context.dataStore.edit { preferences ->  
 preferences[key] = value  
 }  
 Log.d("SettingsRepository", "Сохранена настройка: ${key.name} = $value")  
 }  
   
 // Получение значения настройки синхронно  
 suspend fun getSettingValue(key: Preferences.Key<String>): String {  
 return getSetting(key).first()  
 }  
   
 // Получение папки для записей по умолчанию  
 private fun getDefaultRecordingsFolder(): String {  
 val externalDir = context.getExternalFilesDir(Environment.DIRECTORY\_MUSIC)  
 return externalDir?.absolutePath ?: "/storage/emulated/0/Music"  
 }  
   
 // Применение настроек аудио  
 suspend fun applyAudioSettings(  
 audioSource: String,  
 recordingFormat: String,  
 sampleRate: String,  
 bitrate: String  
 ) {  
 // Здесь можно добавить логику для применения настроек к аудиорекордеру  
 // Например, обновление конфигурации MediaRecorder  
   
 Log.d("SettingsRepository", "Применены настройки аудио:")  
 Log.d("SettingsRepository", "- Источник: $audioSource")  
 Log.d("SettingsRepository", "- Формат: $recordingFormat")  
 Log.d("SettingsRepository", "- Частота: $sampleRate")  
 Log.d("SettingsRepository", "- Битрейт: $bitrate")  
   
 // Сохраняем настройки в постоянное хранилище  
 saveSetting(AUDIO\_SOURCE\_KEY, audioSource)  
 saveSetting(RECORDING\_FORMAT\_KEY, recordingFormat)  
 saveSetting(SAMPLE\_RATE\_KEY, sampleRate)  
 saveSetting(BITRATE\_KEY, bitrate)  
 }  
   
 // Применение настроек хранения  
 suspend fun applyStorageSettings(recordingsFolder: String) {  
 // Проверяем существование директории и создаем ее при необходимости  
 val folder = File(recordingsFolder)  
 if (!folder.exists()) {  
 val created = folder.mkdirs()  
 if (created) {  
 Log.d("SettingsRepository", "Создана директория для записей: $recordingsFolder")  
 } else {  
 Log.e("SettingsRepository", "Не удалось создать директорию: $recordingsFolder")  
 // Если не удалось создать директорию, используем директорию по умолчанию  
 saveSetting(RECORDINGS\_FOLDER\_KEY, getDefaultRecordingsFolder())  
 return  
 }  
 }  
   
 Log.d("SettingsRepository", "Применены настройки хранения:")  
 Log.d("SettingsRepository", "- Папка для записей: $recordingsFolder")  
   
 // Сохраняем настройки в постоянное хранилище  
 saveSetting(RECORDINGS\_FOLDER\_KEY, recordingsFolder)  
 }  
   
 // Применение настроек интерфейса  
 suspend fun applyInterfaceSettings(language: String) {  
 // Здесь можно добавить логику для применения языковых настроек  
 // Например, обновление локали приложения  
   
 Log.d("SettingsRepository", "Применены настройки интерфейса:")  
 Log.d("SettingsRepository", "- Язык: $language")  
   
 // Сохраняем настройки в постоянное хранилище  
 saveSetting(LANGUAGE\_KEY, language)  
 }  
   
 // Получение настроек аудио для использования в AudioRecorderRepository  
 suspend fun getAudioSettings(): AudioSettings {  
 val audioSource = getSettingValue(AUDIO\_SOURCE\_KEY)  
 val recordingFormat = getSettingValue(RECORDING\_FORMAT\_KEY)  
 val sampleRate = getSettingValue(SAMPLE\_RATE\_KEY)  
 val bitrate = getSettingValue(BITRATE\_KEY)  
   
 return AudioSettings(  
 audioSource = parseAudioSource(audioSource),  
 outputFormat = parseOutputFormat(recordingFormat),  
 audioEncoder = parseAudioEncoder(recordingFormat),  
 sampleRate = parseSampleRate(sampleRate),  
 bitrate = parseBitrate(bitrate)  
 )  
 }  
   
 // Вспомогательные методы для парсинга настроек  
   
 private fun parseAudioSource(source: String): Int {  
 return when (source) {  
 "Микрофон" -> android.media.MediaRecorder.AudioSource.MIC  
 "Камкордер" -> android.media.MediaRecorder.AudioSource.CAMCORDER  
 else -> android.media.MediaRecorder.AudioSource.DEFAULT  
 }  
 }  
   
 private fun parseOutputFormat(format: String): Int {  
 return when (format) {  
 "WAV" -> android.media.MediaRecorder.OutputFormat.THREE\_GPP // Используем 3GPP как контейнер для WAV  
 "MP3" -> android.media.MediaRecorder.OutputFormat.MPEG\_4 // MP3 не поддерживается напрямую, используем MPEG\_4  
 else -> android.media.MediaRecorder.OutputFormat.MPEG\_4 // AAC (m4a)  
 }  
 }  
   
 private fun parseAudioEncoder(format: String): Int {  
 return when (format) {  
 "WAV" -> android.media.MediaRecorder.AudioEncoder.AMR\_NB // Для WAV используем AMR\_NB  
 "MP3" -> android.media.MediaRecorder.AudioEncoder.AAC // MP3 не поддерживается напрямую, используем AAC  
 else -> android.media.MediaRecorder.AudioEncoder.AAC // AAC (m4a)  
 }  
 }  
   
 private fun parseSampleRate(sampleRate: String): Int {  
 return when (sampleRate) {  
 "8 кГц" -> 8000  
 "16 кГц" -> 16000  
 "44.1 кГц" -> 44100  
 else -> 44100  
 }  
 }  
   
 private fun parseBitrate(bitrate: String): Int {  
 return when (bitrate) {  
 "64 кбит/с" -> 64000  
 "128 кбит/с" -> 128000  
 "256 кбит/с" -> 256000  
 else -> 128000  
 }  
 }  
   
 // Класс для хранения настроек аудио  
 data class AudioSettings(  
 val audioSource: Int,  
 val outputFormat: Int,  
 val audioEncoder: Int,  
 val sampleRate: Int,  
 val bitrate: Int  
 )  
}