Project Overview — Lumen: An Offline AI Assistant for the Visually Challenged

Lumen is a fully offline, AI-powered Android application designed to empower visually challenged individuals in their daily lives. By combining on-device computer vision, optical character recognition, and natural language understanding — all optimized for Arm-based mobile hardware — Lumen provides a fast, private, and reliable accessibility companion that works anywhere, anytime, without dependence on the internet.

At its core, Lumen aims to provide **true independence**. Many existing accessibility apps depend on cloud inference, introducing barriers such as slow latency, privacy risks, and unreliable performance in low-connectivity areas. Lumen overcomes all of these by moving advanced AI workloads to the edge, enabling visually impaired users to receive immediate assistance anytime, anywhere.

The app delivers four major modes:

- **Live Object Detection & Dictation**
- **Live Text Detection & Dictation**
- **Document Assistant (Read Aloud + Local Question Answering)**
- **Scene Description**

Each mode is crafted with a **voice-first, minimal-interaction interface** to ensure that the app remains practical and intuitive for its target users.

Why Lumen Stands Out

Lumen's offline-first approach is not just a feature; it's a deliberate engineering choice:

✔ Privacy & Trust

For visually impaired users, the app often captures personal objects, private documents, or sensitive living environments.
Sending this data to the cloud is a privacy red flag.

Lumen keeps **all computation on-device**, demonstrating a deep respect for user safety and ethical AI .

✔ Reliability & Real-World Accessibility

Visually impaired users may not always have stable data access.
Offline functionality ensures:

- The app works **anywhere** — indoors, rural areas, during network outages.
- The user is never stranded without assistance.

This highlights real-world viability and inclusiveness.

✔ Technical Difficulty (Arm Optimization)

Running multimodal AI workloads on-device requires:

- Quantization
- Delegate usage (such as NNAPI / GPU)
- Memory-efficient models
- Real-time processing pipelines

These optimizations have been applied in Lumen to ensure strong performance and efficiency on Arm-based devices, allowing the app to run smoothly across a wide range of mobile hardware.

✔ Superior User Experience

Since inference happens locally, users receive:

- **Ultra-low latency** for live object detection
- **Instant OCR** for short text
- **Faster scene description and document reading** compared to cloud approaches

The responsiveness dramatically improves usability for visually challenged users.

Functionality

Home Page

The home page acts as the **central entry point** into all four major modes of Lumen. Its purpose is to give users quick access to the core features, with each option opening a dedicated activity.

Core Functionality

The home screen presents four primary options, each linked to one of Lumen's **offline AI capabilities**:

1. **Object Recognition**
   Launches **MainActivity**, starting the **live camera feed** and loading the **on-device object detection model** for real-time identification and audio feedback.

2. **Live Text Reader**
   Opens **LiveTextActivity**, enabling **continuous OCR** through the camera with immediate spoken output.

3. **Document Assistant**
   Starts **DocumentActivity**, allowing users to **capture or upload documents**, **extract text**, **listen to readings**, and **ask questions** about the document content.

4. **Scene Description**
   Opens **SceneActivity**, where the app generates concise **natural-language descriptions** of the surroundings using an on-device vision model.

All of these actions are triggered using simple **onClick listeners** inside `HomeActivity.kt`, each launching the corresponding activity via an intent.

Supporting Structure

The layout defined in `activity_home.xml` arranges the four options in a clean vertical structure, keeping the interface straightforward and easy to navigate.

Purpose of This Page

Overall, the home page is designed to **streamline mode selection**. With a single tap, users can move directly into any of the main capabilities of Lumen, making the app **efficient**, **accessible**, and practical for visually challenged individuals who rely on fast, predictable navigation.

Object Recognition Mode

Object Recognition is Lumen's real-time, camera-based mode for identifying nearby objects and informing the user by voice. The implementation is designed around reliable on-device inference, low-latency updates, and concise spoken feedback so that the feature is practical for daily, real-world use.

High-level behavior

- When the user selects Object Recognition from the home screen, **MainActivity** starts the camera preview and the on-device detector.

- The app loads a compact EfficientDet-Lite0 model (packaged as model.tflite in the app assets) and runs detection locally — no network required. The detector is created and configured inside the **ObjectDetectorHelper**.

Core implementation points

- **Camera and analysis pipeline**
  - Camera preview and frame analysis are handled in **MainActivity** using CameraX Preview and ImageAnalysis. Frames are delivered to a background thread for processing.
  - The app converts CameraX ImageProxy frames into bitmaps via an extension function before passing them to the detector (`ImageProxy.toBitmap()` implementation). This conversion is implemented in **ImageExtensions.kt**.

- **On-device detection pipeline**
  - Detector initialization and options live in **ObjectDetectorHelper**. It configures the TFLite Task API ObjectDetector with a score threshold, maximum results, thread count, and crucially requests NNAPI usage to leverage Arm hardware acceleration when available (`useNnapi()` and `setNumThreads`).
  - Input images are rotated correctly using an ImageProcessor and a Rot90Op before inference so detection remains aligned with the camera orientation.

- **Overlay and visual feedback**
  - Detection results (bounding boxes, labels, scores) are converted into scaled Box objects and posted to an **OverlayView**, which draws rectangles and label panels over the preview. The overlay code and drawing logic are in **OverlayView.kt**.

Spoken feedback (user-facing behavior)

- Text-to-speech (TTS) is initialized in MainActivity and used to announce objects aloud. Announcements are **short and queued** so they do not interrupt each other.

- Announcement logic is tuned for usability:

  - The app attempts to **avoid repeating** the same label continuously by **tracking currently visible labels** and when each label was last spoken. This prevents **redundant announcements** while an object remains in view.
  - A **cooldown period (5 seconds)** prevents re-announcing a label too quickly after it was last spoken.
  - **Frame skipping / throttling** is applied so detection and announcements run at a practical rate (MainActivity enforces a minimum interval between processed frames, roughly targeting **~3 FPS**). This **reduces CPU usage** and stabilizes TTS pacing.
  - **Each utterance is kept short**: the implementation limits how many labels are joined in one sentence (configurable, set to 3), and builds **natural phrases** like “person and cup” instead of long lists.

Performance and robustness decisions

- Model choice and optimization: Using **EfficientDet-Lite0** (a compact detection model) keeps the memory and compute footprint low so the app runs on a wide range of Arm-based devices. The model is packaged as model.tflite for fully offline inference.

- Hardware acceleration: The helper requests **NNAPI** and allows multi-threading to take advantage of Arm NPUs/accelerated backends where available. This demonstrates practical Arm-focused optimization.

- Trade-offs to favor user experience: The app intentionally throttles processing rate and constrains spoken output to avoid overwhelming the user or saturating the device — a balance between **latency**, **accuracy**, and **battery/CPU usage**.

Live Text Reader

The Live Text Reader mode allows the user to point the camera at printed text and have it **recognized and spoken aloud in real time**. It is designed for quick reading of signs, labels, and short text in everyday environments.

Core Workflow

When this mode starts, **LiveTextActivity** sets up the **CameraX** pipeline and begins analyzing frames continuously. Each frame is converted into a bitmap and processed by the **on-device ML Kit text recognizer**, which works fully offline and maintains stable performance regardless of connectivity.

The layout in `activity_live_text.xml` provides a **full-screen preview** and a simple **back button**, keeping the interface focused on the reading experience without unnecessary elements.

Text Processing

To maintain clarity, the system applies a few filtering steps before speaking any detected text. These checks help remove noise that commonly appears when the camera is moving or when the recognizer captures background patterns. The logic works as follows:

- **Very short fragments**:
  Lines with **length < 3 characters** or containing mostly punctuation are ignored. This avoids announcing stray characters like "|", "—", "O", etc.

- **Repeated character patterns**:
  The system checks for sequences where **more than 70% of the characters are identical**, or where the text matches simple repetition patterns (for example "11111", "aaaaa", "-----"). Such lines are skipped to reduce false positives.

- **Overly long or noisy lines**:
  If a line exceeds a **reasonable character limit** (based on a threshold for single-line signs) or contains a high ratio of non-alphabetic symbols, it is considered noisy and removed. This prevents reading out distorted text from motion blur or decorative elements.

These brief checks ensure that only **meaningful, readable lines** move forward to the speech output stage, keeping the Live Text Reader reliable even when the camera feed is unstable.

Speech Output

Speech feedback is generated through **TextToSpeech**, with several adjustments to keep narration smooth:

- A **cooldown system** prevents repeating the same line while it stays in view.
- Only **newly appearing text** is spoken.
- Multiple short lines can be **combined into a single utterance** for more natural phrasing.

- Frame analysis is spaced slightly to maintain **stable performance** and avoid overwhelming the user.

These choices make the mode comfortable to use while scanning real-world text.

Overall Behavior

The Live Text Reader is built for situations where users need **instant understanding of visible text** without taking a photo. Because everything runs **entirely on-device**, recognition stays fast, predictable, and available anywhere. The combination of lightweight processing, continuous detection, and controlled speech output makes this mode practical for everyday use.

Document Assistant

The Document Assistant mode is built to handle full documents instead of short text fragments. Users can **capture a page**, **upload a file**, extract text, listen to it aloud, or ask questions about the content using an on-device MobileBERT QA model.

1. Document Capture, Upload, and OCR (DocumentActivity)

Core Workflow

When this mode opens, **DocumentActivity** sets up the camera preview and lets the user either capture a document or upload an existing file. After a bitmap is obtained, the text is extracted using **ML Kit's on-device OCR**, which keeps the process fast and independent of network conditions.

The layout in `activity_document.xml` displays the preview, capture/upload buttons, and a scrollable text area with two buttons above it (**Read aloud** & **Ask Queries**) once OCR is complete.

File Handling

The mode supports several document types:

- **Images** → Converted to an InputImage and processed directly
- **Text files** → Loaded and displayed immediately
- **PDFs** → The app reads the page count and lets the user choose which page to process

PDF pages are rendered using `PdfRenderer` before being passed to OCR.

Output

Once text extraction finishes, the user can:

- View the recognized text
- Use **Read Aloud** (TextToSpeech)
- Continue to **Ask Queries**, which opens the chatbot

Because everything runs locally, users can work with personal documents comfortably.

2. Question Answering Chatbot (DocChatActivity)

Model and Setup

DocChatActivity loads the extracted document text along with the **MobileBERT QA model** (`mobilebert_qa.tflite`). The model runs on-device and is initialized using TFLite's BertQuestionAnswerer API.

MobileBERT is lightweight enough to run efficiently on typical Arm-based mobile hardware.

Chat Interface

The interface (defined in `activity_doc_chat.xml`) includes:

- A chat history (RecyclerView)
- A text field
- A **Send** button
- A **Mic** button for voice questions

Messages are displayed through **ChatAdapter**, which separates **user**, **bot**, and **system** messages by checking the **message type** stored in each `ChatMessage` object. The adapter uses different item layouts and background colors depending on whether:

- the message comes from the **user**
- the message is generated by the **model**
- the message is a **system/guide message**

This allows each message category to appear visually distinct in the chat interface.

Answering Flow

1. The user asks a question (typed or spoken).

2. The assistant passes the question and document text to MobileBERT.

3. A ranked answer list is returned; the best answer is shown and spoken automatically.

If nothing relevant is found, the assistant returns a simple fallback message.

Voice Questions

Voice input is handled using Android's **SpeechRecognizer**, making it easier for visually challenged users to ask questions hands-free.

Overall Behavior

The Document Assistant provides a complete offline workflow: capture or upload a document, extract its text, read it aloud, or interact with it through natural-language questions.
Running both OCR and MobileBERT locally ensures that the feature stays **fast, private, and available anywhere**, without depending on network quality.

Scene Description

The Scene Description mode lets users capture or upload an image and receive a **natural-language caption** describing the overall scene. This feature is useful when users want a quick understanding of their surroundings rather than object-level details.

It works fully offline using a two-part captioning system made up of **InceptionV3** and an **LSTM decoder**, both running on-device.

Core Workflow

When opened, **SceneActivity** sets up a CameraX preview and provides two input paths:

- **Capture Scene** using the camera
- **Upload Image** from storage

Once an image is captured or selected, the activity sends the bitmap to the **Captioner** class for caption generation.

The layout for this mode (`activity_scene.xml`) includes the preview, capture buttons, a status indicator, and a scrollable caption display.

Captioning Pipeline

Model Structure

Inside **Captioner**, two TensorFlow Lite models are loaded:

- **inceptionv3_1.tflite** – extracts visual features from the image
- **lstm_2.tflite** – generates a word sequence based on the extracted features

InceptionV3 produces an initial feature vector, which is then fed into the LSTM decoder step-by-step to build a short sentence. This design keeps inference efficient on typical **Arm-based mobile CPUs**, since the CNN and LSTM run in compact, quantized TFLite form.

Image Processing

Before inference:

- The input image is resized to **346 × 346**
- Pixel values are normalized and stored in a 3-channel tensor

These values are passed to the InceptionV3 model to produce the **initial LSTM state**.

Caption Generation

The **LSTM decoder** generates a caption one token at a time. At each step:

1. The current token ID is fed into the LSTM.
2. The model produces a probability distribution over the vocabulary.
3. The most suitable token is selected using a temperature-controlled softmax.
4. The process repeats until an **END** token is reached or a fixed length is reached.

The built-in vocabulary loader maps these token IDs to English words, producing a concise caption.

Caption Refinement Logic

After the LSTM generates its token sequence, the Captioner applies a few targeted refinements to keep the final caption clear and natural:

1. **Word selection and token filtering**
   START and END tokens are skipped, and any **blank or invalid words** are ignored when converting token IDs to vocabulary terms.

2. **Low-confidence handling**
   The model applies **temperature scaling (TEMP = 1.2)** to smooth the probability distribution, then checks a **minimum confidence threshold of 0.05**.
   Any predicted word with confidence **below 0.05** is **skipped**, allowing the decoder to move on to a more reliable token.

3. **Trailing filler removal**
   Common filler words at the end (such as **"a", "an", "the", "of", "in", "and", "or"**) are trimmed using a loop that removes them until the caption ends on a meaningful word.

4. **Fallback behavior**
   If the caption becomes empty after refinement, the system returns a **fallback message**, ensuring the user always receives a useful description.

This results in short, coherent descriptions that remain useful in real-world scenes.

Output and Interaction

Once captioning completes:

- The caption appears in the **scrollable preview**
- The **Read Aloud** button becomes active
- The system optionally announces that the description is ready

Users can then press **Read Aloud** to hear the caption through TextToSpeech.
`SceneActivity` also saves caption text in an internal file for reference.

Overall Behavior

The Scene Description mode provides a quick, sentence-level summary of an image, using a combination of **on-device CNN + LSTM models**.
Running locally ensures that the feature stays responsive and private, even without internet access.
The pipeline is lightweight enough to perform reliably on typical Arm-powered Android devices while still producing meaningful, natural-language output.

Setup Instructions

A. Run Lumen on an Arm-based Android Device

1. **Install the APK**

   - **Download the APK**
     You can either:
     • Build the APK locally from the project, **or**
     • Download the prebuilt APK directly from the project's **GitHub Releases** page.

     If building locally:
     Open the Lumen Android Studio project, then select Build → Build APK(s) or Build → Generate APK.
     The APK will be created at:
     Lumen/app/build/outputs/apk/debug/app-debug.apk

   - **Enable USB debugging on the device**
     On the Android phone, open Developer Options and enable **USB debugging**.

   - **Install the APK using adb**
     Connect the phone to your computer.
     Navigate to the folder containing the APK and install it using the command:
     adb install -r app-debug.apk
     After installation completes, you can open Lumen from the app drawer and access all four modes.

   - **If adb installation is not successful**
     You can simply transfer the APK to your phone and install it directly from the file manager after enabling "Install unknown apps."

2. **Grant Permissions and Launch**

   - On first launch, approve **CAMERA** and **RECORD_AUDIO** (for voice queries).
   - You should land on the **Home Page** with the four main modes.

3. **Verify Core Features Quickly**

   - **Object Recognition**: point the camera at objects and confirm short spoken labels.
     (Detection uses a lightweight model with NNAPI requested when available.)

   - **Live Text Reader**: point at printed text; check overlay and voice output.
   - **Document Assistant**: capture or upload a page; verify OCR output, Read Aloud, and Ask Queries.
   - **Scene Description**: capture/upload an image; confirm the caption appears.

4. **Notes**

   - Ensure a system **Text-to-Speech** engine is installed.
   - If a model fails to load, check available storage or try a different Arm device with stable NNAPI support.

B. Build Lumen From Source

The GitHub repository already contains the necessary files. Below are the **main things to confirm** before building.

1. **Open and Sync in Android Studio**

   - Open the repo, let **Gradle sync**, and install any required SDK components.

2. **Check Required Assets**
   Ensure these files exist in **app/src/main/assets**, matching the names expected in the code:

   - EfficientDet model for Object Recognition.
   - **mobilebert_qa.tflite** for the Document Assistant chatbot.
   - **inceptionv3_1.tflite**, **lstm_2.tflite**, and the vocabulary file for Scene Description.

3. **Verify Dependencies Briefly**
   The project uses:

   - **CameraX** for preview and capture.
   - **ML Kit (on-device OCR)** for text extraction.
   - **TensorFlow Lite** for MobileBERT and the captioning models.

   No cloud services are required; everything runs locally.

4. **Build and Install**

   - In Android Studio: **Build > Build APK(s)**.
   - Install via adb:
     ```
     adb install -r app/build/outputs/apk/debug/app-debug.apk
     ```

5. **Runtime Validation**

    - Use **Logcat** to check tags like “Doc”, “LiveText”, “Scene”, “Captioner” for inference or model-loading issues.
    - If performance varies, adjust thread counts or test on another **Arm device**, since NNAPI behavior differs per chipset.

Final Summary

Lumen brings together multiple offline AI capabilities into one practical accessibility tool designed for visually challenged users. Each mode—Object Recognition, Live Text Reader, Document Assistant, and Scene Description—runs entirely on the device using lightweight models and efficient pipelines, making the app reliable on everyday Arm-based Android phones. The focus throughout the project has been on real usability: fast responses, clear spoken feedback, minimal navigation, and the ability to work anywhere without depending on a network connection.

Along with this write-up, a **demo video** has been submitted to showcase how each mode behaves in real scenarios and to provide a clear picture of the end-to-end user experience. Together, the implementation and the demonstration highlight the completeness, practicality, and readiness of the project.