CMPE 492
Senior Project 2

Coffee Machine Assistance for the Visually Impaired: A Computer Vision-Based Mobile Application

Low-Level Design Report

Project Supervisor
Prof. Dr. Hakkı Gökhan İlk

Project Team Members
Murad Huseynov
Bedir Esen
Hazem Moustafa
Alperen Dalgıç

Table of Contents

1. Introduction

This document presents the Low-Level Design (LLD) for the **"Coffee Machine Assistance for the Visually Impaired"** mobile application. It expands upon the previously submitted High-Level Design (HLD) by translating the major system components into concrete, implementation-ready specifications. The LLD provides a detailed breakdown of subsystems into their respective classes, data structures, algorithmic workflows, and module-to-module interactions.

The scope of this document includes the internal architecture of the four primary modules—**User Interface (UI), Input, Computer Vision (CV), and Voice Guidance**—as well as the class interfaces, state machine behavior, and persistent data model required to support reliable real-time assistance. This refined Version 2 release delivers a clearer, more complete, and implementation-oriented description of the system.

1.1 Object Design Trade-offs

Several important design decisions and trade-offs were made to satisfy the project's functional requirements, performance expectations, and accessibility goals:

- **Real-time Performance vs. Detection Accuracy**
  The system must generate guidance quickly while maintaining reliable object detection.
  Decision: The EfficientDet model was selected for its optimal balance of accuracy and computational efficiency. When converted to TensorFlow Lite, it provides real-time performance on mobile hardware without significant loss of precision.

- **Resource Consumption vs. Responsiveness**
  Processing every camera frame would increase latency and drain device resources.
  Decision: The ImageCaptureService uses interval-based frame extraction (e.g., every 1–2 seconds), supplying high-quality snapshots to the CV module while limiting CPU/GPU usage.

- **Instruction Clarity vs. User Pace**
  Users must receive helpful guidance without being overwhelmed.
  Decision: A centralized **state machine (StateController)** manages instruction flow. Only one context-aware instruction is delivered at a time, and transitions occur only after the current step is verified.

- **Cross-Platform Development vs. Native Output Quality**
  The app must run consistently on Android and iOS but still provide high-quality speech output.
  Decision: The application is implemented in **React Native** for a unified codebase, with

native bridges to Android TextToSpeech and iOS AVSpeechSynthesizer for optimal responsiveness and natural-sounding audio.

1.2 Interface Documentation Guidelines

All class interfaces in this LLD follow the notation used in the system's UML diagrams:

vis attribute: type

vis operation(arg list): return type

Where:

- **vis**: Visibility symbol
    - + public
    - # protected
    - - private
- **attribute**: A data field or stored value
- **operation**: A method or function
- **Underlining** indicates static attributes or static operations

This standard ensures consistency and clarity across all diagrams and technical specifications.

1.3 Engineering Standards

To ensure accessibility, security, maintainability, and cross-platform consistency, the project follows the engineering standards below:

1. **Accessibility: WCAG 2.1 (AA Level)**
   This is the project's primary compliance standard. The UI and interaction model are designed according to POUR principles—Perceivable, Operable, Understandable, and Robust. The design ensures compatibility with TalkBack (Android) and VoiceOver (iOS), proper focus order, and accessible touch targets.

2. **Privacy & Security: Privacy by Design**
   No Personally Identifiable Information (PII) is collected. All computer vision and text-

to-speech processing is executed locally on the device. No visual data or usage information is transmitted to external servers.

1.2 Cross-Platform Development Requirements

The application supports **Android 7.0+** and **iOS 13+**, consistent with the project's platform guidelines.

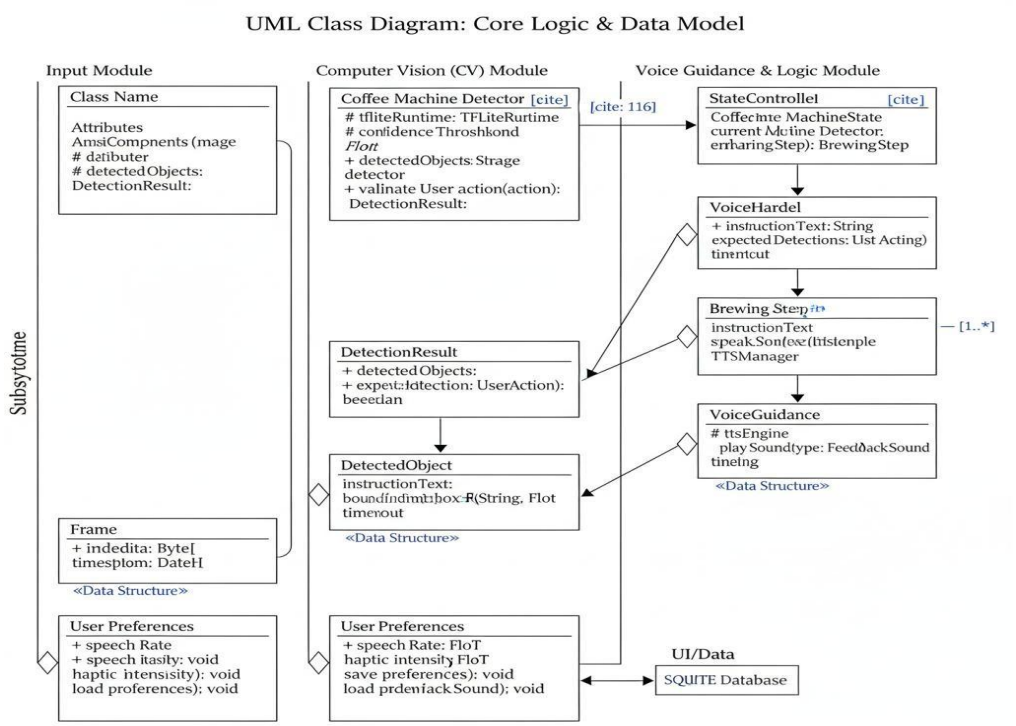
1.3 Code Architecture: Modular Design

The system follows modular principles to isolate UI, CV, and logic layers. This improves scalability, simplifies testing, and supports future extension—such as adding additional machine models or new guidance workflows.

1.4 Glossary

- **CV (Computer Vision):** AI techniques enabling machines to interpret and understand visual input.
- **EfficientDet:** A scalable, high-efficiency object detection model family.
- **TFLite (TensorFlow Lite):** A lightweight machine learning framework for mobile inference.
- **TTS (Text-to-Speech):** Converts text into synthesized speech.
- **React Native:** A cross-platform framework for building mobile applications using a shared codebase.
- **State Machine:** A computational model controlling application logic through well-defined states and transitions.
- **Bounding Box:** A rectangular coordinate region that identifies the location of a detected object within an image.

2. Subsystem Decomposition



Model (UML diagram)

input Module

- **Responsibility:** Manages camera permissions and hardware, with the sole responsibility of capturing and providing **image frames** for processing.
- **Key Data Structure:** The **Frame** data structure holds the raw image data and a timestamp.

Computer Vision (CV) Module

- **Responsibility:** Receives an image frame, preprocesses it, and runs the **EfficientDet** TFLite model for object detection to detect and identify coffee machine components.
- **Key Data Structures: DetectionResult** (a list of objects found in the frame) and **DetectedObject** (information for a single component, including a **Bounding Box**).

Voice Guidance & Logic Module

- **Responsibility:** The "brain" of the application. It contains the **StateController** (a state machine) that requests detections, interprets results, and generates appropriate voice instructions via native **TTS** engines. It ensures the user receives one clear, context-aware instruction at a time.
- **Key Classes: StateController**, **VoiceGuidance**, **ErrorHandler**, and **BrewingStep**.

User Interface (UI) Module

- **Responsibility:** The front-end **React Native** layer. It renders the camera view, provides accessible touch targets, and speaks the text provided by the Voice Guidance Module via the screen reader (compatible with **TalkBack** and **VoiceOver**).
- **Accessibility Standard:** Adheres to **WCAG 2.1 (AA Level)**.

3. Class Interfaces

This section provides the detailed class-level design for each subsystem, based on the Object and Class Model from the Analysis Report.

3.1 User Interface & Preferences Classes

Class	Key Attributes	Key Methods	Description
UserInterfaceModule	# camera: CameraManager , # guidanceText: String	+ displayGuidance(text: String): void , + triggerHapticFeedback(intensity: Float): void , + getSettings(): UserPreferences [8], + setSettings(preferences: UserPreferences): void [9]	The main React Native component that renders the camera view and accessibility overlays, and updates the text announced by the screen reader.
UserPreferences	+ speechRate: Float , + hapticIntensity: Float	+ savePreferences(): void , + loadPreferences(): void	A data-handling class that serializes and deserializes user settings from the SQLite database.

These classes manage the front-end view and store user settings.

UserInterfaceModule: The main React Native component. It renders the camera view and accessibility overlays. displayGuidance updates the text that the screen reader will announce.

UserInterfaceModule
camera: CameraManager

guidanceText: String
+ displayGuidance(text: String): void
+ triggerHapticFeedback(intensity: Float): void
+ getSettings(): UserPreferences
+ setSettings(preferences: UserPreferences): void

UserPreferences: A data-handling class that serializes and deserializes user settings from the SQLite database.

UserPreferences
+ speechRate: Float
+ hapticIntensity: Float
+ savePreferences(): void
+ loadPreferences(): void

3.2 Input Module Classes

Class	Key Attributes	Key Methods	Description
CameraManager	- cameraRef: Object permissionsGranted: boolean	+ requestPermissions(): boolean , + getFrame(): Frame	A wrapper (native bridge) for the device's camera API. getFrame is called to capture a single frame[22].
Frame	+ imageData: Byte[] , + timestamp: DateTime	N/A	A data structure holding the raw image data and a timestamp.

These classes are responsible for accessing the camera and providing image data.

CameraManager: A wrapper (native bridge) for the device's camera API. getFrame is called by the StateController (via the ImageCaptureService) to capture a single frame for processing.

CameraManager
- cameraRef: Object
- permissionsGranted: boolean
+ requestPermissions(): boolean
+ getFrame(): Frame

Frame: A data structure holding the raw image data and a timestamp.

Frame
+ imageData: Byte[]
+ timestamp: DateTime

3.3 Computer Vision (CV) Module Classes

These classes handle all object detection logic[26].

Class	Key Attributes	Key Methods	Description
CoffeeMachineDetector	- tfliteRuntime: TFLiteRuntime , - model: String , - confidenceThreshold: Float	+ detectComponents(image: Frame): DetectionResult , + updateState(newState: MachineState): void	The core CV class that orchestrates the detection pipeline, runs inference via TFLite, and returns a DetectionResult.
DetectionResult	+ detectedObjects: List<DetectedObject>	N/A	A data structure containing a list of objects found in the frame.
DetectedObject	+ label: String , + boundingBox: Rectangle [36], + confidence: Float	N/A	A data structure holding information for a single detected component.

These classes handle all object detection logic.

CoffeeMachineDetector: The core CV class. Its detectComponents method orchestrates the entire detection pipeline: it takes a Frame, sends it for preprocessing, runs inference via tfliteRuntime, filters the results by confidenceThreshold, and returns a DetectionResult.

CoffeeMachineDetector
- tfliteRuntime: TFLiteRuntime
- model: String
- confidenceThreshold: Float
+ detectComponents(image: Frame): DetectionResult
+ updateState(newState: MachineState): void

DetectionResult: A data structure that contains a list of objects found in the frame.

DetectionResult
+ detectedObjects: List<DetectedObject>

DetectedObject: A data structure holding the information for a single detected component.

DetectedObject
+ label: String
+ boundingBox: Rectangle
+ confidence: Float

3.4 Voice Guidance & Logic Module Classes

his is the most critical subsystem, managing the application's state, logic, and user guidance[39].

Class	Key Attributes	Key Methods	Description
StateController	- currentState: MachineState , - currentStep: BrewingStep , - detector: CoffeeMachineDetector , - errorHandler: ErrorHandler	+ validateUserAction(action: UserAction): boolean , + getNextStep(): BrewingStep , + logError(error: String): void	The central **state machine**; it coordinates all modules, validates detections against the currentStep, and triggers instructions or error handling.
VoiceGuidance	- ttsEngine: TTSManager	+ speakInstruction(text: String): void , + playSound(type: FeedbackSound): void , + adjustSpeed(rate: Float): void	A wrapper for the native TTS engines (Android TTS and iOS VoiceOver) that receives instructions and manages speech synthesis.
ErrorHandler	N/A	+ handleError(error: String, state: MachineState): void , + generateRecoveryInstruction(error: String, state: MachineState): String	Dedicated to managing off-nominal states by generating user-friendly recovery instructions.
BrewingStep	+ instructionText: String , + expectedDetections: List<String> , + timeout: Integer	N/A	A data structure that defines a single sequential step of the coffee-making process, including the instruction and the required detections for validation.
MachineState (Enum)	N/A	N/A	Defines the application's possible states: **IDLE, DETECTING, GUIDING, BREWING, ERROR, COMPLETE**.

This is the largest and most critical subsystem, managing the application's state, logic, and all user guidance.

StateController: The central state machine. It holds the currentState and the currentStep (which defines what needs to be detected). It coordinates all other modules: it requests detections from the detector, compares the result against the currentStep, and, on success, tells the voice module to speak the next instruction. If validation fails, it triggers the errorHandler.

StateController

- currentState: MachineState
- currentStep: BrewingStep
- history: List<BrewingStep>
- detector: CoffeeMachineDetector
- voice: VoiceGuidance
- errorHandler: ErrorHandler
+ validateUserAction(action: UserAction): boolean
+ getNextStep(): BrewingStep
+ logError(error: String): void

VoiceGuidance: A wrapper for the native TTS engines. It receives simple text instructions from the StateController and manages the speech synthesis.

VoiceGuidance
- ttsEngine: TTSManager
+ speakInstruction(text: String): void
+ playSound(type: FeedbackSound): void
+ adjustSpeed(rate: Float): void

ErrorHandler: A class dedicated to managing off-nominal states. If StateController reports an error (e.g., wrong button detected, or machine lost), this class generates a user-friendly recovery instruction (e.g., "Wrong button. Please aim at the button at 3 o'clock.").

ErrorHandler
+ handleError(error: String, state: MachineState): void
+ logError(error: String): void
+ generateRecoveryInstruction(error: String, state: MachineState): String

MachineState: A simple enumeration defining the application's possible states, managed by the StateController.

MachineState (Enum)
IDLE
DETECTING
GUIDING
BREWING
ERROR
COMPLETE

BrewingStep: A data structure that defines a single step of the coffee-making process. For example, a step might be:

- instructionText: "Please press the power button."
- expectedDetections: ["power_button", "power_button_on"]
- timeout: 10000 (ms)

BrewingStep
+ instructionText: String
+ expectedDetections: List<String>
+ timeout: Integer

4. Core Algorithm: Detection-to-Guidance Loop

The application's behavior is orchestrated by a state machine centered around the `StateController` class. The core algorithm implements a **detection-to-guidance loop** that continuously cycles between: (i) issuing an instruction, (ii) capturing a frame, (iii) running component detection, and (iv) validating the result against the current brewing step.

The algorithm is designed to:

- Provide **deterministic, step-wise guidance** to visually impaired users.
- Ensure **real-time responsiveness** under mobile hardware constraints.
- Be **robust to transient failures**, such as missed detections or incorrect user actions.

4.1 State Machine Overview

The `MachineState` enumeration defines the high-level states of the application: `IDLE`, `GUIDING`, `DETECTING`, `BREWING`, `ERROR`, and `COMPLETE`. The `StateController` maintains two key variables:

- `currentState: MachineState` – current high-level mode of the app.
- `currentStep: BrewingStep` – the step being executed in the brewing sequence.

In addition, the controller preserves:

- `history: List<BrewingStep>` – a log of successfully completed steps.
- `detector: CoffeeMachineDetector` – a reference to the CV module.
- `voice: VoiceGuidance` – a reference to the TTS-based feedback module.
- `errorHandler: ErrorHandler` – a strategy object for off-nominal scenarios.

The state transitions can be summarized as:

- IDLE → GUIDING when the user initiates the process.
- GUIDING → DETECTING after an instruction is spoken.
- DETECTING → GUIDING on successful validation of a step.
- DETECTING → ERROR on inconsistent or incorrect detections.
- ERROR → GUIDING once a recovery instruction is generated.
- GUIDING → COMPLETE once all steps in the brewing sequence have been completed.

4.2 Detailed Step-by-Step Flow

The core loop from the `StateController` perspective is as follows:

1. **Initialization (IDLE State)**
    - The app starts in `IDLE`.
    - On user action (e.g., tapping a large "Start Guidance" button on the UI), the `StateController` loads the predefined **brewing sequence** (a list of `BrewingStep` objects) from configuration.
    - `currentStep` is set to the first step and `currentState` transitions to `GUIDING`.
2. **Guidance Phase (GUIDING State)**
    - The `StateController` reads `currentStep.instructionText`, which describes the next required action, such as:

      "Please locate the power button on the machine."

    - The text is passed to `VoiceGuidance.speakInstruction(text)` for synthesis via the native TTS engine.
    - Optionally, `UserInterfaceModule.triggerHapticFeedback()` is invoked to reinforce the transition between steps.
    - Once speech output is scheduled (or completed, depending on implementation), the controller switches `currentState` to `DETECTING`.
3. **Detection Phase (DETECTING State)**

- The `StateController` requests a frame from the input layer via `CameraManager.getFrame()` (usually mediated by an `ImageCaptureService` abstraction).
- The returned `Frame` is passed to `CoffeeMachineDetector.detectComponents(frame)`.
- The detector runs the TFLite EfficientDet model and returns a `DetectionResult`, encapsulating a list of `DetectedObject` instances.

4. **Validation Phase (within DETECTING)**
   - The `StateController` evaluates whether the detection matches the expectation for `currentStep`.
   - The validation logic can be expressed as:
     - If there exists a `DetectedObject.label ∈ currentStep.expectedDetections` and its confidence >= `confidenceThreshold`, then the step is considered **successfully completed**.
     - If no relevant object is detected (empty or non-matching list), the system may either:
       - **Retry detection** until `currentStep.timeout` is exceeded, or
       - Immediately prompt the user to slightly adjust the camera orientation via an informative guidance message.
     - If an object is detected that corresponds to a known but **incorrect** control (e.g., “steam_knob” when expecting “power_button”), the system treats this as a **wrong action** and transitions to `ERROR`.
5. **Step Advancement on Success**
   - On success:
     - `currentStep` is appended to `history`.
     - `VoiceGuidance.playSound(FeedbackSound.SUCCESS)` may be invoked to provide immediate non-verbal confirmation.
     - The next `BrewingStep` in the sequence is loaded via `getNextStep()`.
     - If a next step exists, `currentState` moves back to `GUIDING` and the loop continues.

        - If no further steps exist, the state transitions to `COMPLETE`, and a final message such as “Coffee is ready. Enjoy safely.” is spoken.

6. **Error Handling (ERROR State)**
    - In case of wrong detections or repeated timeouts, the controller calls:
        - `ErrorHandler.generateRecoveryInstruction(error, currentState)` to obtain a context-aware message, e.g.:

          “Incorrect button. Move your phone slightly to the right and try again.”

    - The error string is logged via `ErrorHandler.logError(error)`.
    - The system then transitions back to `GUIDING` with the recovery instruction, followed by renewed detection attempts.
7. **Completion (COMPLETE State)**
    - Once all steps have been executed, the `StateController` announces completion and can either:
        - Reset to `IDLE` after a brief delay; or
        - Wait for an explicit user action (e.g., “Start again”) before resetting.

4.4 Performance and Robustness Considerations

- **Frame Sampling Interval**: Rather than processing every camera frame, the system uses an interval-based strategy (e.g., every 1–2 seconds) to reduce CPU/GPU load and power consumption while preserving responsiveness.
- **Confidence Threshold Tuning**: The `confidenceThreshold` parameter in `CoffeeMachineDetector` is configurable and can be adapted per coffee machine model to optimize the trade-off between false positives and false negatives.
- **Transient Failures**: Short-term occlusions or blurring are handled by repeated detection attempts within the step’s timeout window. Only prolonged failures transition to `ERROR`.
- **Scalability**: New `BrewingStep` sequences (for different machine models or brewing workflows) can be added without modifying the core algorithm, as long as they follow the same `BrewingStep` data contract.

5. Persistent Data Management

Persistent data management in the application is designed to be **fully local**, **privacy-preserving**, and **lightweight**, in line with the project's requirements. All long-term data is stored on the device using:

1. A **local SQLite database** for structured user preferences and status flags.

2. **Internal file storage** for static assets required by the CV pipeline (model and labels) and configuration.

No data is transmitted to external servers, and no personally identifiable information (PII) is collected at any stage of the detection-to-guidance loop.

Low_Level_Desgin_Report[1]

5.1 Database Schema

The application employs a single local SQLite database, with an initial minimalistic schema that can be extended as needed. The primary table is `UserPreferences`, used to store user-specific configuration values that affect the behavior of the Voice Guidance and UI modules.

`loadPreferences()` executes a `SELECT` query (e.g., `SELECT * FROM UserPreferences WHERE id = 1`) on app startup. If no row is found, it creates one with default values.

`savePreferences()` issues an `UPDATE` statement (or `INSERT` on first run) whenever the user modifies settings in the accessible settings screen.

5.1.1 Data Lifecycle

1. **Application First Launch**

    - The database helper checks whether the `UserPreferences` table exists; if not, it creates the table and inserts a default row.

2. **Settings Modification**

    - When the user changes speech rate, volume, or haptic intensity via the UI, the updated values are immediately written to the database using `savePreferences()`.

3. **Application Resume**
    - On subsequent launches, `loadPreferences()` is called to restore the last known configuration, ensuring a consistent user experience across sessions.

5.1.2 Future Extensions

The schema has been intentionally kept small, but can be extended with additional fields such as:

- `preferredLanguage` (TEXT) – for multi-lingual support.
- `lastMachineProfile` (TEXT) – to remember the last used coffee machine model.
- `analyticsOptIn` (BOOLEAN) – should any optional, local-only analytics be added in the future.

These extensions would involve a standard SQLite migration procedure (e.g., `ALTER TABLE`) controlled by a database versioning mechanism in the application's data access layer.

5.2 Local File Storage

Static assets required for the CV pipeline and application configuration are stored in the app's **internal, read-only file storage**. These assets are bundled with the application during the build process and are not modified at runtime.

The core assets are:

- `model.tflite`
    - The quantized and mobile-optimized EfficientDet model used by `CoffeeMachineDetector`.
    - Loaded once at application startup (or lazily on first use) into `tfliteRuntime`.
    - Resides in the platform-specific assets directory (e.g., `android/app/src/main/assets` for Android, app bundle resources on iOS).

- `brewing_steps.json`
    - Defines the sequence of `BrewingStep` objects (instruction texts, expected detections, timeouts) for each supported machine profile.
    - Enables non-developers (e.g., accessibility experts) to tweak guidance content without modifying the code.

- `sounds/` directory

- Contains short audio clips for confirmation or error tones, used by `VoiceGuidance.playSound()`.

5.2.1 Access Patterns

- On startup, the `CoffeeMachineDetector` reads `model.tflite` and `labels.txt` into memory.
- The `StateController` or an associated configuration manager loads the `brewing_steps.json` file and instantiates the sequence of `BrewingStep` objects for the selected coffee machine profile.
- No write operations occur to these assets at runtime, ensuring immutability and simplifying integrity validation.

5.3 Data Access Layer Design

To prevent the rest of the application from depending directly on SQLite queries or file paths, a **Data Access Layer (DAL)** abstraction is recommended:

- `PreferencesRepository`
  - Provides high-level methods such as `getUserPreferences()` and `updateUserPreferences(prefs: UserPreferences)`.
  - Internally uses platform-specific storage APIs (React Native bridges to SQLite) but exposes a platform-agnostic interface to the UI and Voice Guidance modules.
- `ConfigurationRepository`
  - Provides methods like `getBrewingSequence(machineId: String): List<BrewingStep>`.
  - Handles parsing `brewing_steps.json` and mapping JSON structures to strongly typed `BrewingStep` instances.

This separation ensures:

- **Testability**: DAL classes can be unit-tested independently with mock storage backends.
- **Maintainability**: Storage details can change (e.g., migration from SQLite to another local store) without affecting business logic.

5.4 Privacy and Security Considerations

The persistent data design is aligned with the **Privacy by Design** principle described in the introduction:

- No PII or biometric data is persisted.
- Camera frames are processed in memory and are not written to disk.
- All CV inference and TTS operations are executed locally, preventing leakage of user behavior to remote servers.
- Database and configuration files are stored in app-specific internal storage spaces that are sandboxed per OS security guarantees (Android, iOS).

If required in the future, additional security measures (such as encrypting the SQLite database or obfuscating configuration files) can be introduced by extending the DAL implementation, without altering the public interfaces consumed by the rest of the system.

6. References

[1] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020. arXiv:1911.09070.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv:1704.04861, 2017.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., "TensorFlow: A System for Large-Scale Machine Learning," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265-283.

[4] OpenCV team, "OpenCV (Open Source Computer Vision Library)". [Online]. Available: https://opencv.org/. [Accessed: October 28, 2025].

[5] Android Developers, "TextToSpeech | Android Developers". [Online]. Available: https://developer.android.com/reference/android/speech/tts/TextToSpeech. [Accessed: October 29, 2025].

[6] Apple Inc., "VoiceOver | Apple Developer Documentation," [Online]. Available: https://developer.apple.com/documentation/accessibility/voiceover. [Accessed: October 29, 2025].

[7] TensorFlow, "TensorFlow Lite Guide". [Online]. Available: https://www.tensorflow.org/lite/guide. [Accessed: October 30, 2025].

[8] World Wide Web Consortium, "Web Content Accessibility Guidelines (WCAG) 2.1," W3C Recommendation, 2018. [Online]. Available: https://www.w3.org/TR/WCAG21/. [Accessed: October 31, 2025].