DR DETECTOR

Developer Manual

Architecture · API Reference · ML Pipeline · Deployment

Version	1.0
Stack	React Native · Expo · TFLite · ONNX Runtime · SQLite
Audience	Software engineers and ML practitioners
Date	June 2026

Author Dr Neal Aggarwal, Version 1.2, May 2026

Contents

1. Architecture Overview

DR Detector is a fully offline clinical screening application. The architecture is split into two independent codebases that share the same domain logic and UI concepts but use platform-native inference and storage backends.

Layer	Mobile (React Native + Expo)	Desktop (Electron)
UI Framework	React Native 0.74 + Expo SDK 51	React (same components, web target)
ML Inference	react-native-fast-tflite (TFLite)	onnxruntime-node (ONNX Runtime)
Accelerator	CoreML (iOS) / NNAPI (Android)	DirectML (Win) / CoreML (mac) / CPU
Image Preprocessing	expo-image-manipulator + JS CLAHE	sharp + JS CLAHE
Database	expo-sqlite (SQLite WAL)	better-sqlite3 (SQLite WAL)
PDF Generation	expo-print (HTML → PDF)	Electron printToPDF
Storage	App sandbox (iOS) / App data (Android)	userData directory
Navigation	React Navigation v6 (Stack)	HTML router (single-page)
Network	None required	None required

Data flow

All data remains on the device. The flow for a single screening is linear and synchronous from the user's perspective:

- Camera frame → expo-camera CameraView (mobile) or file dialog (desktop)
- Image saved to local filesystem (JPEG, ~2–5 MB)
- expo-image-manipulator resizes to 224×224 JPEG
- JS CLAHE preprocessor equalises contrast channel-by-channel
- Float32Array [224×224×3] passed to TFLite / ONNX session
- Softmax output [4] → argmax severity + confidence
- Result + image path inserted into SQLite screenings table
- PDF generated from HTML template with embedded image (base64)

2. Project Structure

```
retina-dr-detector/
■■■ mobile/                          # React Native + Expo
■   ■■■ App.tsx                      # Navigation root, startup init
■   ■■■ app.json                     # Expo config (permissions, plugins)
■   ■■■ package.json
■   ■■■ babel.config.js
■   ■■■ tsconfig.json
■   ■■■ assets/
■   ■   ■■■ model/
■   ■       ■■■ dr_detector.tflite # TFLite model (place here)
■   ■■■ src/
■       ■■■ constants/index.ts     # SEVERITY map, COLORS, MODEL_CONFIG
■       ■■■ screens/
■       ■   ■■■ CaptureScreen.tsx  # Camera + patient ID
■       ■   ■■■ AnalysisScreen.tsx # 2-phase: spinner then results
■       ■   ■■■ HistoryScreen.tsx  # Patient screening timeline
■       ■■■ components/
■       ■   ■■■ SeverityBadge.tsx  # Colour-coded grade badge
■       ■■■ ml/
■       ■   ■■■ ModelManager.ts    # TFLite loader + runInference()
■       ■   ■■■ Preprocessor.ts    # Resize + CLAHE + normalise
■       ■■■ database/
■       ■   ■■■ Database.ts        # expo-sqlite CRUD + schema
■       ■■■ services/
■           ■■■ PDFExporter.ts     # HTML template + expo-print
■■■ desktop/                         # Electron
■   ■■■ main.js                      # Main process: ONNX, SQLite, IPC
■   ■■■ preload.js                   # contextBridge IPC bindings
■   ■■■ package.json
■   ■■■ assets/model/
■       ■■■ dr_detector.onnx       # ONNX model (place here)
■■■ train_model.py                   # PyTorch EfficientNet-B0 training
```

3. Technology Stack

Package	Version	Purpose
expo	~51.0.0	Managed workflow, asset bundling, native modules
expo-camera	~15.0.0	CameraView component, takePictureAsync()
expo-image-manipulator	~12.0.0	Resize to 224×224, JPEG encode
expo-sqlite	~14.0.0	Local SQLite database with WAL mode
expo-print	~12.8.0	HTML → PDF rendering
expo-sharing	~11.10.0	Native share sheet for PDF export
expo-file-system	~17.0.0	Local file I/O (image copy, base64 read)
expo-asset	~10.0.0	Load bundled .tflite asset by URI
expo-keep-awake	~13.0.0	Prevent screen timeout during clinic session
react-native-fast-tflite	^1.3.0	TFLite inference with CoreML/NNAPI delegates
@react-navigation/native	^6.1.18	Navigation container
@react-navigation/stack	^6.4.1	Stack navigator (Capture→Analysis→History)
react-native-reanimated	~3.10.1	Animation (required by react-navigation)
@expo/vector-icons	^14.0.2	Ionicons icon set

Desktop additional dependencies

Package	Version	Purpose
electron	^30.0.0	Desktop application shell
onnxruntime-node	^1.18.0	ONNX model inference (CPU/DirectML/CoreML)
better-sqlite3	^9.4.3	Synchronous SQLite with WAL
sharp	^0.33.3	Fast image resize and raw pixel decode
electron-builder	^24.13.3	Cross-platform installer packaging

4. Mobile App — React Native / Expo

4.1 Navigation and App Entry Point

App.tsx is the root component. It runs two async initialisations on mount — Database.initialise() and ModelManager.loadModel() — before rendering the navigator. Both are fire-and-forget; the Capture screen renders immediately and the Analysis screen awaits model readiness before running inference.

```
// App.tsx (simplified)
export type RootStackParamList = {
  Capture:  { patientId?: string };
  Analysis: { imageUri: string; patientId: string; isNewPatient: boolean };
  History:  { patientId: string };
};

useEffect(() => {
  Database.initialise();       // Creates tables; idempotent
  ModelManager.loadModel();    // Loads .tflite into TFLite runtime
}, []);
```

4.2 CaptureScreen

Manages camera permission state, patient ID input, and image capture. Uses expo-camera's CameraView with a dashed circular overlay rendered as a View with borderRadius:110 and borderStyle:'dashed'.

Key implementation details:

- Camera permission requested via useCameraPermissions() hook (expo-camera)
- Patient created in DB via Database.upsertPatient() before capture to ensure FK constraint on screenings table
- Photo copied from temporary camera URI to documentDirectory/screenings/.jpg for persistence across sessions
- useKeepAwake() prevents screen timeout during multi-patient clinic sessions
- The CAPTURE button is disabled and shows a spinner during the copy + navigate operation

4.3 AnalysisScreen

Two-phase component: 'analysing' and 'results'. Phase transitions driven by a single useEffect that fires on mount.

```
// AnalysisScreen.tsx — inference useEffect
useEffect(() => {
  let cancelled = false;
  (async () => {
    setProgressStep(0);
    await ModelManager.loadModel();           // no-op if already loaded
    setProgressStep(1);
    const input = await Preprocessor.prepareImage(imageUri);
    setProgressStep(2);
    const result = await ModelManager.runInference(input);
    setProgressStep(3);
    await new Promise(r => setTimeout(r, 300)); // brief UX pause
    if (!cancelled) { setResult(result); setPhase('results'); }
  })();
  return () => { cancelled = true; };
}, [imageUri]);
```

The cancelled flag prevents state updates on unmounted components if the user navigates back during inference — a React Native requirement.

4.4 HistoryScreen

Fetches all screenings for a patient on mount via Database.getScreeningsForPatient() and renders them in a FlatList. Records are immutable — no delete functionality is exposed to protect audit integrity.

5. ML Inference Pipeline

5.1 ModelManager

Singleton pattern — a module-level _model variable persists the TFLite session across React renders and navigations.

```
// ModelManager.ts
let _model: TensorflowModel | null = null;
let _loading: boolean = false;
let _useMock: boolean = false;   // true when .tflite file absent

export const ModelManager = {
  async loadModel(): Promise<void> {
    if (_model || _loading) return;   // idempotent
    _loading = true;
    const [asset] = await Asset.loadAsync(
      require('../../assets/model/dr_detector.tflite')
    );
    _model = await loadTensorflowModel(
      { url: asset.localUri! },
      'core-ml'            // falls back to CPU on unsupported devices
    );
  },

  async runInference(input: Float32Array): Promise<InferenceResult> {
    const outputs = await _model!.run([input]);
    const probs   = Array.from(outputs[0] as Float32Array);
    const severity = probs.indexOf(Math.max(...probs));
    return { severity, confidence: probs[severity], probabilities: probs, ... };
  }
};
```

The delegate parameter 'core-ml' enables the CoreML delegate on iOS (A-series chips) and the NNAPI delegate on Android 8.1+ automatically. Both fall back to CPU if unavailable. Typical inference times: 300–800 ms on CPU, 80–200 ms with hardware acceleration.

5.2 Preprocessor — CLAHE Pipeline

CLAHE (Contrast Limited Adaptive Histogram Equalization) is applied before normalisation. This is critical for fundus images: it enhances microaneurysms and haemorrhages that are invisible in raw images, particularly in non-dilated patients.

Full preprocessing pipeline:

- expo-image-manipulator resizes the image to 224×224 and encodes as JPEG base64
- base64ToUint8Array() decodes the base64 string to raw JPEG bytes
- decodeJpegToRGB() extracts raw RGB pixels (production: use fast-tflite native image API)
- applyCLAHE() performs bilinear-interpolated tile-based histogram equalisation (16×16 tiles, clip limit 3.0)
- Float32Array normalisation: each channel value divided by 255.0

```
// CLAHE parameters
const tileSize  = 16;   // 16×16 pixel tiles
const clipLimit = 3.0;  // contrast amplification limit
const APPLY_CLAHE = true;  // set false if model was trained without CLAHE
```

```
// Production fast path (react-native-fast-tflite native image API):
// const result = await model.runSync([{ image: { uri }, width: 224, height: 224 }]);
// This skips JS preprocessing entirely — model handles decode + resize natively.
```

5.3 Model Architecture (EfficientNet-B0)

Property	Value
Architecture	EfficientNet-B0 (timm implementation)
Parameters	~5.3 million
Input shape	[1, 224, 224, 3] float32 channels-last (HWC)
Input range	[0.0, 1.0] (after /255 normalisation)
Input normalisation	ImageNet mean [0.485, 0.456, 0.406] / std [0.229, 0.224, 0.225] (training only)
Output shape	[1, 4] float32 softmax probabilities
Output classes	0=No DR, 1=Mild NPDR, 2=Moderate NPDR, 3=Severe NPDR/PDR
TFLite model size	~30 MB (float32) / ~8 MB (INT8 quantised)
ONNX model size	~25 MB (float32)
Training dataset	EyePACS (88k images, Kaggle)
CLAHE in training	Yes — clip_limit=3.0, tile_grid_size=(16,16) via Albumentations
Class weighting	sklearn compute_class_weight('balanced') — addresses severe class imbalance

6. Database Layer

expo-sqlite (mobile) and better-sqlite3 (desktop) provide the storage layer. Both use SQLite WAL (Write-Ahead Logging) mode for crash-safe writes. The schema is identical across platforms.

```
-- Schema (SQLite)
CREATE TABLE patients (
  id         TEXT PRIMARY KEY NOT NULL,
  created_at INTEGER NOT NULL,        -- Unix milliseconds
  notes      TEXT DEFAULT ''
);

CREATE TABLE screenings (
  id             TEXT PRIMARY KEY NOT NULL,
  patient_id     TEXT NOT NULL REFERENCES patients(id),
  image_path     TEXT NOT NULL,       -- absolute path to JPEG on device
  severity       INTEGER NOT NULL,    -- 0|1|2|3
  confidence     REAL NOT NULL,       -- 0.0–1.0
  probabilities  TEXT NOT NULL,       -- JSON array, e.g. [0.82, 0.12, 0.04, 0.02]
  inference_ms   INTEGER NOT NULL,
  screened_at    INTEGER NOT NULL,    -- Unix milliseconds
  exported       INTEGER NOT NULL DEFAULT 0   -- 0=no, 1=yes
);

CREATE INDEX idx_screenings_patient
  ON screenings(patient_id, screened_at DESC);
```

Database API (Database.ts)

Method	Returns	Description
Database.initialise()	Promise<void>	Opens DB, runs CREATE TABLE IF NOT EXISTS, sets WAL mode
Database.upsertPatient(id, notes?)	Promise<Patient>	INSERT OR IGNORE — safe to call repeatedly
Database.getPatient(id)	Promise<Patient|null>	Look up patient by ID
Database.saveScreening(s)	Promise<void>	INSERT new screening record
Database.getScreeningsForPatient(id)	Promise<Screening[]>	All screenings for patient, DESC by time
Database.getLatestScreening(id)	Promise<Screening|null>	Most recent screening for patient
Database.markExported(screeningId)	Promise<void>	Set exported=1
Database.getStats()	Promise<{total, today}>	Counts for dashboard display
generateId()	string	Timestamp+random base36 ID (no dependencies)

7. PDF Export Service

PDFExporter.exportScreening() generates a formatted A4 report using expo-print (which converts HTML to PDF via WKWebView on iOS / WebView on Android) and opens the system share sheet via expo-sharing.

Implementation flow:

- Read the image file as base64 via FileSystem.readAsStringAsync(..., { encoding: Base64 })
- Construct a self-contained HTML string with inline CSS and the base64 image embedded as data URI
- Call Print.printToFileAsync({ html }) — returns a URI to a temporary PDF file
- Call Sharing.shareAsync(uri) to open the native share sheet
- Call Database.markExported(screeningId) to flag the record

The HTML template renders identically across iOS and Android because expo-print uses a native WebView renderer. All CSS is inline for maximum compatibility. The PDF is A4 format with severity-colour-coded borders matching the app UI.

```
// PDFExporter.ts — key call sequence
const imgData = await FileSystem.readAsStringAsync(imageUri, {
  encoding: FileSystem.EncodingType.Base64
});
const html = buildHTML({ patientId, result, imgData, ... });
const { uri } = await Print.printToFileAsync({ html, base64: false });
await Sharing.shareAsync(uri, { mimeType: 'application/pdf' });
await Database.markExported(screeningId);
```

8. Desktop App — Electron

8.1 Main Process (ONNX Runtime)

The Electron main process handles all privileged operations: ONNX inference, SQLite access, and filesystem dialogs. The renderer process (React) communicates exclusively via IPC using the contextBridge pattern.

```
// main.js — ONNX model loading at startup
onnxSession = await ort.InferenceSession.create(MODEL_PATH, {
  executionProviders: ['cpu'],   // also: 'dml' (Win), 'coreml' (mac)
  graphOptimizationLevel: 'all',
  enableCpuMemArena: true,
});

// Preprocessing with sharp (much faster than JS canvas)
const { data } = await sharp(imagePath)
  .resize(224, 224)
  .removeAlpha()
  .raw()
  .toBuffer({ resolveWithObject: true });

const float32 = new Float32Array(224 * 224 * 3);
for (let i = 0; i < data.length; i++) float32[i] = data[i] / 255.0;

const tensor  = new ort.Tensor('float32', float32, [1, 224, 224, 3]);
const outputs = await onnxSession.run({ [inputName]: tensor });
```

8.2 IPC API Reference

All main-process capabilities are exposed to the renderer via window.drDetector.* as defined in preload.js. All handlers are registered with ipcMain.handle() and called from the renderer with ipcRenderer.invoke().

IPC Channel	Arguments	Returns	Description
run-inference	imagePath: string	InferenceResult	ONNX inference + preprocessing via sharp
db-upsert-patient	id, notes?	Patient	INSERT OR IGNORE into patients
db-save-screening	Screening object	void	INSERT into screenings
db-get-patient-history	patientId: string	Screening[]	All screenings DESC
db-mark-exported	screeningId: string	void	SET exported=1
open-image-dialog	—	string|null	Native file open dialog (image filter)
save-pdf	htmlContent: string	string|null	printToPDF + save dialog

9. Training the Model

9.1 Dataset and Preprocessing

The model is trained on the EyePACS dataset from the Kaggle Diabetic Retinopathy Detection competition (2015). The dataset contains 88,702 training images graded 0–4 by retinal specialists.

EyePACS Grade	Label	Count (approx.)	Our Class
0 — No DR	No retinopathy	~25,000 (73%)	0
1 — Mild	Mild NPDR	~2,400 (7%)	1
2 — Moderate	Moderate NPDR	~5,300 (15%)	2
3 — Severe	Severe NPDR	~900 (2.6%)	3
4 — Proliferative	PDR	~700 (2%)	3 (merged with Severe)

Grades 3 and 4 are merged into class 3 because the clinical action for both is urgent referral, and the combined sample size improves model performance on this rare class. Class imbalance is addressed by sklearn compute_class_weight('balanced') passed as the weight argument to nn.CrossEntropyLoss.

Augmentation pipeline (Albumentations)

- Resize(224, 224) — mandatory
- CLAHE(clip_limit=3.0, tile_grid_size=(16,16), p=1.0) — mandatory, always applied
- RandomBrightnessContrast(0.2, 0.2, p=0.5)
- HueSaturationValue(10, 20, 10, p=0.3)
- HorizontalFlip(p=0.5), VerticalFlip(p=0.5)
- Rotate(limit=30, p=0.7)
- GaussNoise(var_limit=(10, 50), p=0.3)
- Normalize(ImageNet mean/std) + ToTensorV2()

9.2 Training Script

```
# train_model.py — key configuration
DEVICE      = 'cuda' | 'mps' | 'cpu'   # auto-detected
IMG_SIZE    = 224
NUM_CLASSES = 4
GRADE_MAP   = {0:0, 1:1, 2:2, 3:3, 4:3}   # merge grade 4 → class 3

# Model
model = timm.create_model('efficientnet_b0', pretrained=True, num_classes=4)

# Optimiser
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-2)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

# Loss (class-weighted)
weights   = compute_class_weight('balanced', classes=[0,1,2,3], y=train_labels)
criterion = nn.CrossEntropyLoss(weight=torch.tensor(weights).to(DEVICE))

# Train
python train_model.py --data_dir /path/to/eyepacs --epochs 20 --batch_size 32
```

Expected results:

Metric	EyePACS Test Set	Notes
4-class accuracy	83–86%	Varies by augmentation and epochs
Sensitivity (referable DR, Grade 2+)	~91%	Primary clinical metric
Specificity (referable DR)	~87%	
Cohen's Kappa vs. specialist	~0.78	Acceptable for AI screening tool
Training time (A100 GPU)	~4 hours (20 epochs)	~14 hours on RTX 3080

9.3 Exporting to TFLite and ONNX

```
# ONNX export (for desktop Electron app)
torch.onnx.export(model, dummy_input, 'dr_detector.onnx',
    input_names=['input'], output_names=['output'],
    dynamic_axes={'input':{0:'batch'}, 'output':{0:'batch'}},
    opset_version=17)

# TFLite export via ai-edge-torch (Google)
# pip install ai-edge-torch
import ai_edge_torch
edge_model = ai_edge_torch.convert(model, (dummy_input,))
edge_model.export('dr_detector.tflite')

# Alternative: onnx2tf
# pip install onnx2tf tensorflow
# onnx2tf -i dr_detector.onnx -o tflite_dir
```

9.4 INT8 Quantisation

Post-training INT8 quantisation reduces the TFLite model from ~30 MB to ~8 MB with less than 1% accuracy loss on the EyePACS test set. Recommended for all production mobile deployments.

```
# quantise_tflite() in train_model.py
converter = tf.lite.TFLiteConverter.from_saved_model(tflite_path)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset_gen  # 200 samples
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type  = tf.uint8
converter.inference_output_type = tf.uint8
tflite_quant = converter.convert()
```

10. Building and Deploying

Mobile — EAS Build (Expo Application Services)

```
# 1. Install EAS CLI
npm install -g eas-cli

# 2. Configure project (run once)
eas build:configure

# 3. Development build (includes native TFLite plugin)
eas build --platform android --profile development
eas build --platform ios     --profile development

# 4. Production builds
eas build --platform android --profile production   # .aab for Play Store
eas build --platform ios     --profile production   # .ipa for App Store

# 5. Over-the-air update (JS bundle only, no native rebuild needed)
eas update --branch production --message "Update severity thresholds"
```

Desktop — electron-builder

```
cd desktop
npm install

# Development
npm start

# Production installers
npm run build:win    # NSIS installer (.exe)
npm run build:mac    # DMG (.dmg)
npm run build:linux  # AppImage (.AppImage)

# Output in desktop/dist/
# Model file must be in desktop/assets/model/dr_detector.onnx before build
```

Model file placement

Platform	File	Destination
iOS / Android	dr_detector.tflite	mobile/assets/model/dr_detector.tflite
iOS / Android	dr_detector_int8.tflite (preferred)	mobile/assets/model/dr_detector.tflite
Windows / macOS / Linux	dr_detector.onnx	desktop/assets/model/dr_detector.onnx

Important: The model file must be present before running eas build. EAS bundles assets at build time — updating the model requires a new build, not an OTA update, because model weights are native assets.

11. Testing

Unit testing

- Database.ts: jest + expo-sqlite mock — test schema creation, upsert idempotency, FK constraint, getStats() counts
- ModelManager.ts: mock the TFLite module, verify argmax logic, confidence extraction, mock fallback path
- Preprocessor.ts: test with known pixel arrays, verify CLAHE does not change mean luminance by more than 15%
- PDFExporter.ts: mock expo-print and expo-sharing, assert HTML template contains patient ID and grade label

Integration testing

- Capture → Analysis → Save: Detox end-to-end test (mock camera, assert SQLite record created)
- Model inference test: load INT8 model with known EyePACS test image, assert correct grade
- PDF export: assert share sheet called with correct MIME type

Clinical validation

Before production deployment, validate on a local holdout set of at least 200 images graded by a local ophthalmologist. Target metrics:

Metric	Target	Rationale
Sensitivity (referable DR = Grade 2+)	≥ 90%	Avoids missing patients who need referral
Specificity (referable DR)	≥ 85%	Avoids overwhelming limited referral capacity
Cohen's Kappa	≥ 0.75	'Substantial agreement' threshold for clinical tools
Inference time (P95)	< 5 seconds	User experience requirement
Low confidence rate	< 15%	Image quality protocol adequacy check

12. Adding New Features

Adding a new screen

- Add the route name and params type to RootStackParamList in App.tsx
- Create src/screens/NewScreen.tsx following the AnalysisScreen pattern
- Add in App.tsx
- Navigate via navigation.navigate('NewScreen', { ...params })

Updating the ML model

- Retrain with train_model.py using updated dataset or hyperparameters
- Export to TFLite and ONNX (Section 9.3)
- INT8 quantise the TFLite model (Section 9.4)
- Replace mobile/assets/model/dr_detector.tflite and desktop/assets/model/dr_detector.onnx
- If input shape or class count changes, update MODEL_CONFIG in src/constants/index.ts
- Run eas build for a new mobile build (model is a native asset — OTA updates cannot change it)

Adding a new severity grade

- Update SEVERITY in src/constants/index.ts — add a new key with color, label, action
- Update SeverityLevel type to include the new index
- Update MODEL_CONFIG.numClasses
- Retrain with num_classes updated in train_model.py
- Update the referral guidance table in the user-facing documentation

Adding a language localisation

- Install expo-localization and i18n-js
- Extract all UI strings into src/i18n/en.json (English baseline)
- Create src/i18n/sw.json (Swahili), src/i18n/fr.json etc.
- Wrap string access with t('key') throughout screens
- SEVERITY labels and action strings in constants/index.ts also need localisation

13. Security and Compliance

Data protection

Control	iOS	Android	Desktop
Storage isolation	App sandbox (cannot read by other apps)	App data dir (/data/data/com....)	userData dir (user-account scoped)
Encryption at rest	iOS: Data Protection (hardware AES-256, default for app files)	Same	OS-level encryption (BitLocker / FileVault)
Network transmission	None	None	None
Biometric authentication	Optionally add LocalAuthentication (expo-local-authentication)	Same	OS login screen
Data deletion	No in-app delete (audit integrity)	Same	Same — manual file deletion only

Regulatory considerations

- DR Detector as shipped is a general-purpose screening aid — not a classified medical device
- If deployed at scale in a formal health system, assess against FDA Software as a Medical Device (SaMD) guidance and IEC 62304
- GDPR Article 9 applies to retinal images as biometric data — ensure DPA is in place before clinic deployment in EU
- HIPAA: retinal images + patient ID constitute PHI — US deployments require BAA with any cloud service used (none required for offline-only deployment)
- CDSCO (India), PMDA (Japan), NMPA (China): check local medical device software regulations before deploying in those jurisdictions
- Audit log: the screenings table with timestamps and patient IDs provides a minimal audit trail — consider adding a clinician_id field for multi-user deployments

Known security considerations

Model file integrity: The TFLite / ONNX model files are not cryptographically signed. A malicious actor with physical device access could replace the model file. For high-security deployments, verify a SHA-256 hash of the model file at startup and refuse to run if the hash does not match the expected value.