System Design Report: Scalable Hybrid AI Verification Architecture

Target Scale: 1 Million+ Users (10k-20k Concurrent)
Primary Goal: Reduce Latency, Eliminate Server Crashes, Minimize Cost
Strategy: Hybrid Client-Side "Gatekeeping" + Stateless Backend "Truth"

1. Executive Summary

2The current system relies on a monolithic, disk-heavy architecture that will fail under high concurrency due to I/O bottlenecks and synchronous blocking. To scale to 1M+ users, we must shift compute to the client (Edge) and make the backend purely stateless and asynchronous.

The Solution: A **Hybrid Architecture**.

1. **Client-Side (Mobile/Browser):** Acts as a strict "Gatekeeper" using lightweight AI (TFJS) to ensure image quality *before* upload.
2. **Server-Side (Cloud):** Acts as the "Final Truth" and security layer, processing single images in memory without disk writes.

2. Architecture Comparison

Current Bottlenecks (The "Crash" Scenario)

- **Disk I/O:** Every request writes to public/uploads or temp/. High concurrency = Inode exhaustion & slow response.
- **Bandwidth Waste:** Users upload blurry/empty images, wasting server ingress/egress.
- **Synchronous Blocking:** The main event loop blocks during model inference.

Proposed Architecture (The "Scale" Scenario)

- **Zero Disk Writes:** Images flow from Network $\rightarrow$ RAM $\rightarrow$ GPU $\rightarrow$ S3 (Async).
- **Client Filter:** 90% of "bad" requests are blocked on the phone. The server only sees high-quality candidates.
- **Async Archival:** S3 uploads happen *after* the user gets a success response.

3. Detailed Data Flow

Phase 1: Client-Side "Gatekeeping" (The Mobile Flow)

Objective: Prevent bad data from ever reaching the server.

1. **Initialization:**
    - User opens Web App (WebView).
    - Browser downloads quantized model (TFJS/MediaPipe) ~5MB (Cached via Service

Worker).

2. **Live Viewfinder (Manual Capture with Guardrails):**
    - User sees camera feed.
    - User clicks **"Capture"**.
    - **Action:** App freezes the video frame.
3. **Local Inference (0ms Latency):**
    - **Check 1 (Object Detection):** Is class aadhar_front present with >0.7 confidence?
    - **Check 2 (Quality):** Is variance (sharpness) $> Threshold$?
4. **Instant Feedback:**
    - **Pass:** Show Green Checkmark $\rightarrow$ Proceed to Upload.
    - **Fail:** Show Red Warning ("Blurry" or "Card Missing") $\rightarrow$ Unfreeze Camera $\rightarrow$ User Retries.

Phase 2: Server-Side "Verification" (The API Flow)

Objective: Security check and final validation.

1. **Ingestion:**
    - Endpoint receives **ONE** high-quality image (Base64/Binary).
    - **Crucial:** Load image bytes directly into RAM (using io.BytesIO). **DO NOT SAVE TO DISK.**
2. **Security Inference (YOLO):**
    - Run high-accuracy YOLO model on the in-memory image.
    - Check for Fraud: "Print Aadhaar", "Screen Replay" (Moiré patterns).
3. **Response:**
    - Return 200 OK { valid: true } immediately (Target < 200ms).
4. **Async Job (Fire & Forget):**
    - Push image to Message Queue (RabbitMQ/SQS).
    - **Worker:** Uploads to S3 bucket artifacts/{appId}/users/{userId}/....
    - **Worker:** Runs OCR extraction and updates DB.

4. Handling Edge Cases (The "No-Fail" Strategy)

We cannot trust AI 100%. We must account for valid users who fail detection due to lighting or hardware issues.

A. The "Three-Strike" Rule (Frontend)

- **Attempt 1 & 2:** Strict Mode. If Client AI fails, block the user.
- **Attempt 3:** Fallback Mode.
    - User fails for the 3rd time.
    - UI displays button: **"My card is clear, Upload Anyway."**
    - **Action:** Bypass client-side checks and force upload to backend.

B. The "Review Queue" (Backend)

- **Scenario:** A "Force Upload" image arrives, but Server YOLO confidence is low (e.g., 0.10).
- **Action:** Do **NOT** auto-reject.
  - Flag transaction as status: "pending_review".
  - Send response to Frontend: { success: true, status: "under_review" }.
  - **Frontend UX:** Show "Verifying Document..." screen (polling).
  - **Ops:** Image routed to a "Manual Review" queue/dashboard.

5. Technical Implementation Guide

Frontend Engineering (Next.js + TFJS)

- **Dependencies:** @tensorflow/tfjs, @tensorflow-models/coco-ssd (or custom YOLO export).
- **Model Export:** Convert PyTorch model to TFJS:
  yolo export model=best4.pt format=tfjs

- **Logic:** Implement CameraCapture.tsx to handle the "Capture -> Predict -> Feedback" loop.

Backend Engineering (Python FastAPI)

- **Statelessness:** Refactor main.py.
  - Remove shutil.copy, os.mkdir, tempfile.
  - Use files: UploadFile and read .read() directly.
  - Pass bytes to cv2.imdecode.
- **Concurrency:**
  - Use asyncio.to_thread or loop.run_in_executor for the model() inference call to prevent blocking the event loop.
- **Infrastructure:**
  - **Docker:** Containerize the app (ensure libgl1 is installed).
  - **Orchestration:** Deploy on K8s with HPA (Horizontal Pod Autoscaler) triggered by CPU usage.
  - **Redis:** Implement rate limiting (e.g., 10 req/min per IP).

6. Migration Checklist

1. [ ] **Model Export:** Export best4.pt to TFJS format (int8 quantized).
2. [ ] **Frontend Update:** Integrate TFJS into CameraCapture.tsx for client-side gating.
3. [ ] **Backend Refactor:** Rewrite detect_aadhaar_cards to accept bytes and skip disk saves.
4. [ ] **Async Worker:** Set up a simple Celery/RQ worker for S3 uploads.
5. [ ] **Load Test:** Use locust to simulate 500 concurrent users hitting the API to verify memory usage.

7. Cost & Performance Projection

Metric	Old Architecture	New Hybrid Architecture	Impact
Server Load	100% of frames	~5-10% (Only good frames)	**90% Cost Reduction**
Latency	3-5s (Wait for upload)	0.1s (Instant Feedback)	**10x Faster UX**
Bandwidth	Video/Multiple Uploads	Single JPEG Image	**Low Data Cost**
Reliability	Prone to I/O Crashes	Stateless & Auto-scaling	**99.9% Uptime**