DocFraudDetector

AI-Powered Document Fraud & Tamper Detection System

Technical Documentation & Study Guide

End-to-end document fraud detection using deep learning combined with classical computer vision forensics.

YOLOv8 • EfficientNet-B0 • OpenCV • EasyOCR

Pranav Kashyap

Indian Institute of Information Technology, Dharwad

BigVision LLC — Internship Project

February 16, 2026

Contents

Chapter 1

Project Overview

1.1 Abstract

DocFraudDetector is an end-to-end AI system that detects fraud and tampering in document images. Given a photograph of an identity document (Aadhaar card, PAN card, driver's licence, voter ID, etc.), the system automatically:

1. **Locates** the document region within the photograph.
2. **Rectifies** perspective distortion so the document appears flat and front-facing.
3. **Analyzes** the image forensically to determine if any region has been digitally tampered with.
4. **Extracts** text via OCR and structures it into machine-readable fields.

The pipeline processes a single image in under **400 milliseconds** on a CPU machine, producing a detailed JSON report with a tamper verdict, individual forensic scores, visual heatmaps, and extracted text fields.

Key Insight

The system combines **classical computer vision forensics** (Error Level Analysis, noise analysis, edge density) with **deep learning** (EfficientNet-B0 CNN classifier), following the same hybrid approach used at BigVision LLC across their portfolio of commercial computer vision products.

1.2 Problem Statement

Digital document fraud is a growing threat across banking, insurance, immigration, and identity verification. Common attacks include:

- **Text replacement**: Overwriting a name, date, or ID number with new text.
- **Font mismatch**: Injecting text in a different typeface, creating typographic inconsistency.

- **Copy-paste forgery**: Cloning a region of the document (e.g., a photo or stamp) to another location.
- **Selective blur**: Applying blur to hide the original content of a specific area.
- **Noise injection**: Adding noise to mask evidence of editing.

Existing solutions either require manual expert inspection (slow, expensive, unscalable) or rely on single-technique detection (easy to bypass). There is a need for an **automated, multi-technique** system that catches tampering regardless of which method the attacker used.

1.3 Solution: Multi-Technique Forensic Pipeline

DocFraudDetector addresses this by chaining four specialised modules into a single pipeline:

Stage	Module	Core Technique
1	Document Detection	YOLOv8-nano / OpenCV contour fallback
2	Perspective Rectification	`cv2.getPerspectiveTransform` + `warpPerspective`
3	Tamper Detection	ELA + Noise Analysis + Edge Density + CNN
4	OCR Extraction	EasyOCR + Regex field parsing

Table 1.1: The four stages of the DocFraudDetector pipeline.

Each module can operate independently (modular design), but the pipeline orchestrator chains them together, handles error recovery, logs timing, and produces unified output.

1.4 Why This Project Aligns with BigVision LLC

BigVision LLC, founded by the CEO of OpenCV.org, specialises in end-to-end computer vision products. This project directly mirrors their areas of expertise:

BigVision Expertise	Mapping in DocFraudDetector
OpenCV fluency	Contour detection, perspective transforms, CLAHE, Canny
Object detection	YOLOv8 for document localisation
Deep learning	EfficientNet-B0 binary classifier via `timm`
Document analysis & OCR	EasyOCR extraction + regex field parsing
Synthetic data generation	Custom generator with 5 tamper types
End-to-end pipeline	FastAPI REST API + Streamlit web demo
Edge AI mindset	YOLOv8-nano chosen for speed; CPU-first design
Case Study #6	*ID Document Analysis* — direct match

Table 1.2: Alignment between BigVision's portfolio and project design choices.

Key Insight

BigVision's Case Study #6 describes an **ID document analysis** system involving document localisation, OCR, and field extraction — exactly what this project implements, with the added dimension of tamper detection and forensic analysis.

1.5 Key Design Principles

(a) **Modularity**. Each stage is a self-contained Python class that can be tested, replaced, or extended independently.

(b) **Graceful fallback**. If YOLOv8 is unavailable, detection falls back to OpenCV contours. If the CNN model is untrained, forensic analysis alone provides the tamper score. If EasyOCR is not installed, the pipeline still runs all other stages.

(c) **Multi-technique scoring**. Tamper detection does not rely on a single method. ELA, noise analysis, and edge density each produce an independent score. These scores are combined via configurable weighted averaging, making the system robust against attacks that fool any single technique.

(d) **Centralised configuration**. Every threshold, hyperparameter, model name, and file path is defined in a single `config.py` file — zero magic numbers scattered across the codebase.

(e) **Production-ready packaging**. The same pipeline serves a REST API (FastAPI), an interactive demo (Streamlit), and a CLI script, all sharing the same core code.

1.6 Tech Stack Summary

Category	**Technologies**
Image Processing	OpenCV 4.13, Pillow, NumPy
Deep Learning	PyTorch 2.x, `timm` (EfficientNet-B0), torchvision
Object Detection	Ultralytics YOLOv8-nano
OCR	EasyOCR (English)
Data Augmentation	Albumentations, custom transforms
API	FastAPI, Uvicorn, Pydantic
Web Demo	Streamlit
ML Metrics	scikit-learn (accuracy, precision, recall, F1, AUC-ROC)

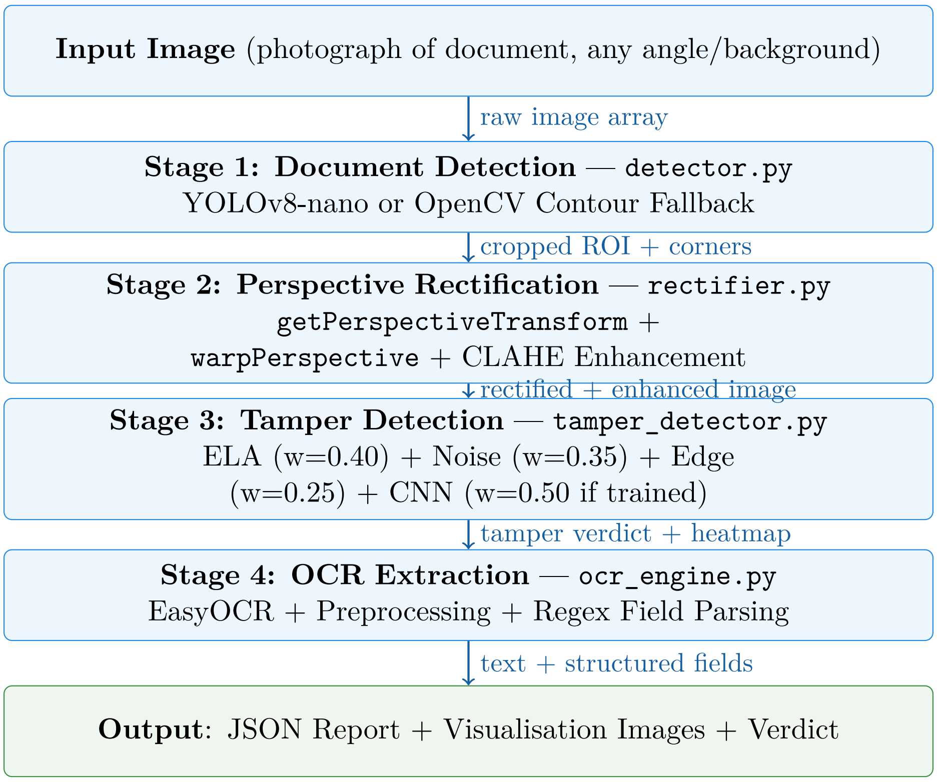
Table 1.3: Technology stack and their roles in the project.

Chapter 2

System Architecture

2.1 High-Level Data Flow

The system follows a **sequential pipeline architecture** where the output of each stage feeds into the next. The pipeline orchestrator (`pipeline.py`) manages the flow, handles errors, and collects timing metrics.



Input Image (photograph of document, any angle/background)
raw image array
Stage 1: Document Detection — `detector.py`
YOLOv8-nano or OpenCV Contour Fallback
cropped ROI + corners
Stage 2: Perspective Rectification — `rectifier.py`
`getPerspectiveTransform` + `warpPerspective` + CLAHE Enhancement
rectified + enhanced image
Stage 3: Tamper Detection — `tamper_detector.py`
ELA (w=0.40) + Noise (w=0.35) + Edge (w=0.25) + CNN (w=0.50 if trained)
tamper verdict + heatmap
Stage 4: OCR Extraction — `ocr_engine.py`
EasyOCR + Preprocessing + Regex Field Parsing
text + structured fields
Output: JSON Report + Visualisation Images + Verdict

Figure 2.1: End-to-end data flow through the DocFraudDetector pipeline.

2.2 Stage 1: Document Detection

2.2.1 Purpose

Given a photograph that may contain a document against a cluttered background (desk, hand, other objects), **locate the document region** and extract its four corner coordinates.

2.2.2 Primary Method — YOLOv8

- Model: `yolov8n.pt` (nano — 3.2M parameters, optimised for speed).
- Confidence threshold: 0.25 (configurable in `config.py`).
- IoU threshold: 0.45 for non-maximum suppression.
- If a "document"-class detection is found, the bounding box coordinates are converted to ordered corners (top-left, top-right, bottom-right, bottom-left).

2.2.3 Fallback Method — OpenCV Contours

If YOLOv8 is unavailable or fails to find a document, the system falls back to a classical CV pipeline:

1. **Grayscale** conversion.
2. **Gaussian blur** with kernel (5×5) to reduce noise.
3. **Canny edge detection** with thresholds $(50, 150)$.
4. **Contour finding** using `cv2.findContours`.
5. **Polygon approximation** with `cv2.approxPolyDP` ($\epsilon = 0.02 \times \text{perimeter}$).
6. If a 4-sided polygon is found, its corners are ordered clockwise.
7. If no quadrilateral is found, the **full image boundaries** are used as a fallback.

Technical Note

The corner ordering algorithm computes the sum $(x + y)$ and difference $(y - x)$ of each coordinate to determine the top-left, top-right, bottom-right, and bottom-left corners respectively. This is a classic trick from the OpenCV documentation.

2.2.4 Output

A dictionary containing: bounding box (`bbox`), ordered corners (`corners`), detection confidence, method used (`yolo` or `contour`), and the cropped document image.

2.3 Stage 2: Perspective Rectification

2.3.1 Purpose

Transform the cropped document so that it appears as a flat, front-facing rectangle — correcting for tilt, rotation, and perspective distortion.

2.3.2 Algorithm

1. **Input**: cropped image + four corner coordinates.
2. **Destination points**: a rectangle of dimensions $W \times H$ (default 600×400).
3. **Perspective matrix**: $M =$ `cv2.getPerspectiveTransform`$(\text{src\_corners}, \text{dst\_corners})$
4. **Warp**: rectified $=$ `cv2.warpPerspective`$(\text{image}, M, (W, H))$ with `INTER_CUBIC` interpolation.
5. **Enhancement**: Apply CLAHE (Contrast Limited Adaptive Histogram Equalisation) and an unsharp mask for sharpening.

2.3.3 Enhancement Details

CLAHE improves local contrast in documents with uneven lighting:

- Converts to LAB colour space.
- Applies CLAHE with `clipLimit=2.0` and `tileGridSize=(8,8)` to the L channel.
- Converts back to BGR.

Unsharp mask (sharpening):

$$\text{sharp} = \text{image} + \alpha \cdot (\text{image} - \text{blur}(\text{image}))$$

where $\alpha = 1.5$ and the blur kernel is (5×5).

2.3.4 Output

The rectified document image and the enhanced version (used for OCR).

2.4 Stage 3: Tamper Detection

2.4.1 Purpose

Determine whether the document has been digitally altered, and provide a **tamper probability score** between 0 (definitely genuine) and 1 (definitely tampered).

2.4.2 Multi-Technique Approach

The system uses **up to four independent techniques**, each producing a score in $[0, 1]$:

Technique 1: Error Level Analysis (ELA)

ELA exploits the fact that JPEG compression affects all regions of an image uniformly — *unless* a region has been pasted in or re-saved at a different quality level.

Algorithm:

1. Re-compress the image as JPEG at quality $Q = 90$.
2. Compute the pixel-wise absolute difference: $\text{ELA} = |\text{original} - \text{recompressed}| \times \text{scale}$
3. The scale factor (10) amplifies the differences for visibility.
4. Compute the mean intensity of the ELA image, normalised to $[0, 1]$.
5. Regions with significantly **higher ELA values** suggest tampering.

Weight: 0.40.

Technique 2: Noise Consistency Analysis

A genuine photograph has a uniform noise pattern across its surface. Tampered regions often have different noise levels because they were captured or processed under different conditions.

Algorithm:

1. Split the image into 16×16 blocks.
2. Compute the standard deviation of pixel intensities in each block.
3. Calculate the **coefficient of variation** (CV) across all blocks: $\text{CV} = \sigma_{\text{blocks}}/\mu_{\text{blocks}}$
4. High CV $\Rightarrow$ inconsistent noise $\Rightarrow$ likely tampered.

Weight: 0.35.

Technique 3: Edge Density Analysis

Copy-paste operations introduce unnatural sharp edges at the boundary of the pasted region. This technique measures overall edge density.

Algorithm:

1. Apply Canny edge detection.
2. Compute the ratio of edge pixels to total pixels.
3. Normalise by a reference density value.
4. Unusually high edge density suggests splicing.

Weight: 0.25.

Technique 4: CNN Classifier (Optional)

An EfficientNet-B0 model trained on synthetic genuine vs. tampered image pairs.

- Input size: 224×224 RGB, normalised with ImageNet statistics.
- Output: softmax probability for class "tampered".
- **Weight**: 0.50 (when available; forensic weights are halved to accommodate).
- Uses `timm` for model creation with pre-trained ImageNet weights.

2.4.3 Score Fusion

The final tamper probability is a weighted average:

$$P_{\text{tamper}} = \frac{\sum_i w_i \cdot s_i}{\sum_i w_i}$$

where s_i is the score from technique i and w_i is its weight. If $P_{\text{tamper}} > 0.50$ (configurable threshold), the verdict is "TAMPERED."

Key Insight

Using multiple independent techniques is the key strength of this system. An attacker who fools ELA (by matching compression levels) will still be caught by noise analysis. An attacker who matches noise will still trigger edge density anomalies. The CNN adds a learned, data-driven check on top.

2.5 Stage 4: OCR Extraction

2.5.1 Purpose

Extract all visible text from the rectified document and parse it into structured fields (name, date of birth, ID number, address, gender).

2.5.2 Preprocessing Pipeline

Before running OCR, the image undergoes:

1. **Denoising**: `cv2.fastNlMeansDenoisingColored` to reduce noise.
2. **Grayscale** conversion.
3. **CLAHE** for contrast enhancement.
4. **Adaptive thresholding** (`ADAPTIVE_THRESH_GAUSSIAN_C`) for binarisation.

2.5.3 OCR Engine

EasyOCR is used with English language support. Each detection returns:

- Bounding box coordinates.
- Extracted text string.
- Confidence score.

2.5.4 Field Extraction

Regex patterns (defined in `config.py`) parse the raw text:

Field	**Regex Pattern**
Name	`(?:name|naam)\s*[:--]?\s*([A-Za-z\s.]+)`
DOB	`(?:dob|date\s*of\s*birth)\s*[:--]?\s*([\d]{2}[/--][\d]{2}[/--][\d]{4})`
ID Number	`(?:no|number|id)\s*[:--]?\s*([\dA-Z]{4,})`
Gender	`(?:gender|sex)\s*[:--]?\s*(male|female|m|f)`

Table 2.1: Regex patterns used for structured field extraction.

2.6 Deployment Architecture

2.6.1 FastAPI REST API

- **Endpoint**: `POST /analyze` — accepts multipart image upload.
- **Response**: JSON containing verdict, stage-by-stage metrics, and optional base64-encoded visualisation images.
- **Health check**: `GET /health` — returns device info and status.
- **Swagger UI**: Auto-generated at `/docs`.
- CORS enabled for all origins (development mode).
- Max file size: 10 MB.

2.6.2 Streamlit Web Demo

- Interactive file upload or sample image selection.
- Progress bar during analysis.
- **Verdict banner**: Green ("GENUINE") or red ("TAMPERED").
- Stage-by-stage expandable sections showing:
  - Detection bounding box overlay.

  - Rectified and enhanced document images.
  - ELA heatmap and tamper likelihood heatmap.
  - OCR text and structured fields.

- JSON report download button.
- Pipeline is cached using `@st.cache_resource` for fast reloads.

2.6.3 CLI Mode

```
python src/pipeline.py <image_path>
```

Processes the image and saves results (8 visualisation images + JSON report) to the `outputs/` directory.

Chapter 3

Codebase Guide

This chapter walks through every significant file in the project, explaining its purpose, key classes/functions, and how it connects to the rest of the system.

3.1 Project Directory Structure

```
DocFraudDetector/
|-- config.py                   # Central configuration
|-- requirements.txt            # Dependencies
|-- README.md                   # Project documentation
|
|-- src/                        # Core pipeline modules
|   |-- __init__.py             # Package init
|   |-- utils.py                # Image utilities
|   |-- detector.py             # Document detection
|   |-- rectifier.py            # Perspective correction
|   |-- tamper_detector.py      # Tamper analysis
|   |-- ocr_engine.py           # OCR extraction
|   |-- pipeline.py             # End-to-end orchestrator
|
|-- data/
|   |-- synthetic_generator.py  # Synthetic data generator
|   |-- sample_images/          # Test images
|
|-- training/
|   |-- train_tamper.py         # Model training script
|
|-- api/
|   |-- server.py               # FastAPI REST API
|
|-- demo/
|   |-- app.py                  # Streamlit web demo
|
|-- models/                     # Saved model checkpoints
|-- outputs/                    # Analysis results
|-- docs/                       # This documentation
```

3.2 `config.py` — Central Configuration

Purpose: Single source of truth for every tuneable parameter in the project. No magic numbers exist anywhere else.

Key sections:

Setting	Default	What it controls
`DEVICE`	`"cpu"`	Auto-selects `cuda` if GPU available
`YOLO_MODEL`	`yolov8n.pt`	Which YOLO variant to load
`YOLO_CONF_THRESHOLD`	0.25	Minimum confidence for YOLO detections
`CANNY_LOW/HIGH`	50 / 150	Canny edge detection thresholds
`CONTOUR_APPROX_EPSILON`	0.02	Controls polygon approximation strictness
`RECTIFIED_WIDTH/HEIGHT`	600 / 400	Output dimensions after rectification
`TAMPER_MODEL_NAME`	`efficientnet_b0`	CNN model architecture
`TAMPER_THRESHOLD`	0.50	Probability above this = “tampered”
`ELA_QUALITY`	90	JPEG quality for ELA recompression
`ELA_SCALE`	10	Amplification factor for ELA differences
`OCR_LANGUAGES`	`["en"]`	EasyOCR language list
`FIELD_PATTERNS`	(5 regexes)	Regex patterns for field extraction
`TAMPER_TYPES`	(5 types)	Weighted probabilities for synthetic tampers
`TRAIN_EPOCHS`	20	Training epochs for the CNN
`TRAIN_LR`	1×10^{-4}	AdamW learning rate
`TRAIN_BATCH_SIZE`	16	Batch size for training
`API_PORT`	8000	FastAPI server port

Technical Note

All paths (`DATA_DIR`, `MODEL_DIR`, `OUTPUT_DIR`, etc.) are derived from `PROJECT_ROOT`, which is computed dynamically using `os.path.dirname(os.path.abspath(__file__))`. This makes the project portable across machines.

3.3 `src/utils.py` — Image Utilities

Purpose: Shared helper functions used by all modules.

Key functions:

- `load_image(path)` — Loads an image via `cv2.imread`. Returns `None` if file is missing.
- `bgr_to_rgb(image)` / `rgb_to_bgr(image)` — Colour space conversion (OpenCV uses BGR; display libraries use RGB).

- `resize_image(image, max_dim)` — Proportionally resizes an image so its longest side does not exceed `max_dim`.
- `draw_bounding_box(image, bbox, label, color)` — Draws a labelled rectangle on an image.
- `draw_corners(image, corners)` — Draws coloured circles at detected document corners.
- `draw_ocr_results(image, detections)` — Overlays OCR bounding boxes and text on the image.
- `compute_ela(image, quality, scale)` — Performs Error Level Analysis.
- `compute_noise_map(image, block_size)` — Computes per-block noise standard deviation map.
- `create_analysis_grid(images_dict)` — Stitches multiple images (original, detection, ELA, heatmap, etc.) into a single grid for easy comparison.
- `save_image(image, path)` — Saves an image, creating parent directories if needed.

Connection: Every other module imports from `utils.py`. It is the most depended-upon file.

3.4 `src/detector.py` — Document Detection

Purpose: Locate the document in an input image.
Class: `DocumentDetector`

- `__init__(use_yolo=True)` — Tries to load YOLOv8 model. Sets `self.use_yolo = False` if it fails, enabling graceful fallback.
- `detect(image)` — Main entry point. Tries YOLO first; falls back to contour detection.
- `_detect_yolo(image)` — Runs YOLOv8 inference, extracts the highest-confidence document detection, converts bbox to ordered corners.
- `_detect_contour(image)` — Classical pipeline: grayscale → blur → Canny → contours → polygon approximation.
- `_order_corners(pts)` — Orders 4 points as [TL, TR, BR, BL] using sum/difference of coordinates.

Connects to: called by `pipeline.py` as Stage 1. Uses `config.py` for thresholds.

3.5 `src/rectifier.py` — Perspective Rectification

Purpose: Correct perspective distortion after detection.
Class: `DocumentRectifier`

- `__init__(width, height)` — Sets target output dimensions.
- `rectify(image, corners=None)` — Computes the perspective transform and warps the image. If corners are not provided, it auto-detects them using the same contour logic as `detector.py`.
- `_auto_detect_corners(image)` — Fallback corner detection.
- `_calculate_dimensions(corners)` — Computes optimal output width and height from the input quadrilateral, preserving aspect ratio.
- `enhance_rectified(image)` — Applies CLAHE + unsharp mask sharpening to the rectified output.

Connects to: receives cropped image and corners from `detector.py` via `pipeline.py`. Passes rectified output to `tamper_detector.py` and enhanced output to `ocr_engine.py`.

3.6 `src/tamper_detector.py` — Tamper Detection

Purpose: Forensic analysis to determine if the document has been digitally altered.
Class: `TamperDetector`

- `__init__()` — Loads the CNN model checkpoint if available. If not, sets `self.use_cnn = False`.
- `detect(image)` — Orchestrates all forensic techniques and fuses their scores.
- `_cnn_predict(image)` — Preprocesses image to 224×224, normalises with ImageNet stats, runs inference through EfficientNet-B0, returns softmax probability of "tampered" class.
- `_ela_analysis(image)` — Calls `utils.compute_ela()`, computes mean ELA intensity as tamper score.
- `_noise_analysis(image)` — Calls `utils.compute_noise_map()`, computes coefficient of variation across blocks.
- `_edge_analysis(image)` — Runs Canny, calculates edge pixel ratio.
- `_create_heatmap(image, ela_img, noise_map)` — Combines ELA and noise maps into a colour-coded heatmap overlaid on the original.

Score fusion: weighted average with configurable weights. If CNN is available, it gets weight 0.50 and forensic weights are halved.

Connects to: receives rectified image from `rectifier.py`. Output (verdict + heatmap) is included in the final report.

3.7 `src/ocr_engine.py` — OCR Extraction

Purpose: Extract and structure text from the document.
Class: `OCREngine`

- `__init__(languages)` — Initialises EasyOCR `Reader` with specified languages.
- `extract(image)` — Main entry point. Preprocesses image, runs OCR, extracts structured fields.
- `_preprocess(image)` — Denoising → grayscale → CLAHE → adaptive threshold.
- `_extract_fields(raw_text, lines)` — Applies regex patterns from `config.FIELD_PATTERNS` to parse fields. Includes fallback heuristics for common ID formats.
- `extract_from_region(image, bbox)` — Extracts text from a specific rectangular region only.

Connects to: receives enhanced image from `rectifier.py`. Output (text + fields) goes into the final report.

3.8 `src/pipeline.py` — End-to-End Orchestrator

Purpose: Chain all modules together and produce the final analysis.
Class: `DocumentAnalysisPipeline`

- `__init__(use_yolo, verbose)` — Instantiates all four stage modules.
- `analyze(image_or_path, save_results)` — The main method:
  1. Loads image if a path is given.
  2. Runs detection → rectification → tamper detection → OCR.
  3. Times each stage with `time.time()`.
  4. Collects results into a unified dictionary.
  5. Generates visualisation images (detection overlay, rectified, ELA, heatmap, OCR, analysis grid).
  6. Saves JSON report and images to `outputs/` if `save_results=True`.
- `get_json_report(result)` — Serialises the result dictionary to a JSON string.

Connects to: imports and orchestrates all four stage modules. Called by `api/server.py`, `demo/app.py`, and the CLI.

3.9 `data/synthetic_generator.py` — Synthetic Data

Purpose: Generate realistic training data for the CNN classifier.
Class: `SyntheticDocumentGenerator`

- `generate_dataset(num_genuine, num_tampered)` — Generates both genuine and tampered images, saves metadata JSON.
- `_create_document()` — Renders a realistic document with header, photo placeholder, text fields (name, DOB, ID, address), watermarks, signature line, and decorative elements.
- **5 tamper functions**:
  - `_tamper_text_replacement` — Whites out a region and re-types different text.
  - `_tamper_font_mismatch` — Injects text in a script-style font (visually inconsistent).
  - `_tamper_copy_paste` — Clones one region to another location.
  - `_tamper_blur_injection` — Applies strong Gaussian blur to a region.
  - `_tamper_noise_injection` — Adds random Gaussian noise to a region.
- `_apply_realistic_augmentation()` — Applies random brightness, rotation, JPEG compression, perspective distortion, and shadow effects.

3.10 `training/train_tamper.py` — Model Training

Purpose: Train the EfficientNet-B0 binary classifier.
Key components:

- `TamperDataset` — A PyTorch `Dataset` that reads genuine (label 0) and tampered (label 1) images from the directory structure.
- `create_model()` — Uses `timm.create_model("efficientnet_b0", pretrained=True, num_classes=2)`.
- `train_one_epoch()` — Standard training loop with cross-entropy loss and AdamW optimiser.
- `validate()` — Computes accuracy, precision, recall, F1, and AUC-ROC on the validation set.
- **Training features**: Learning rate scheduling (`ReduceLROnPlateau`), early stopping, best-model checkpointing based on validation F1, and training history saved as JSON.

3.11 `api/server.py` — FastAPI REST API

Purpose: Expose the pipeline as an HTTP service.

- `POST /analyze` — Accepts multipart file upload, validates content type and file size, runs pipeline, returns JSON with verdict, metrics, and base64 images.
- `GET /health` — Returns status, timestamp, device info.
- `GET /` — API info with endpoint listing.
- Pipeline is lazily initialised on first request.

3.12 `demo/app.py` — Streamlit Web Demo

Purpose: Interactive web UI for demonstrating the system.

- File upload widget for custom images, plus a dropdown for pre-generated samples.
- "Run Full Analysis" button triggers the pipeline.
- Results displayed in expandable sections for each stage.
- Verdict banner (green = genuine, red = tampered).
- Metric cards showing tamper probability, word count, detection confidence, and processing time.
- JSON report download button.
- Pipeline cached with `@st.cache_resource` for fast subsequent runs.

3.13 File Dependency Graph

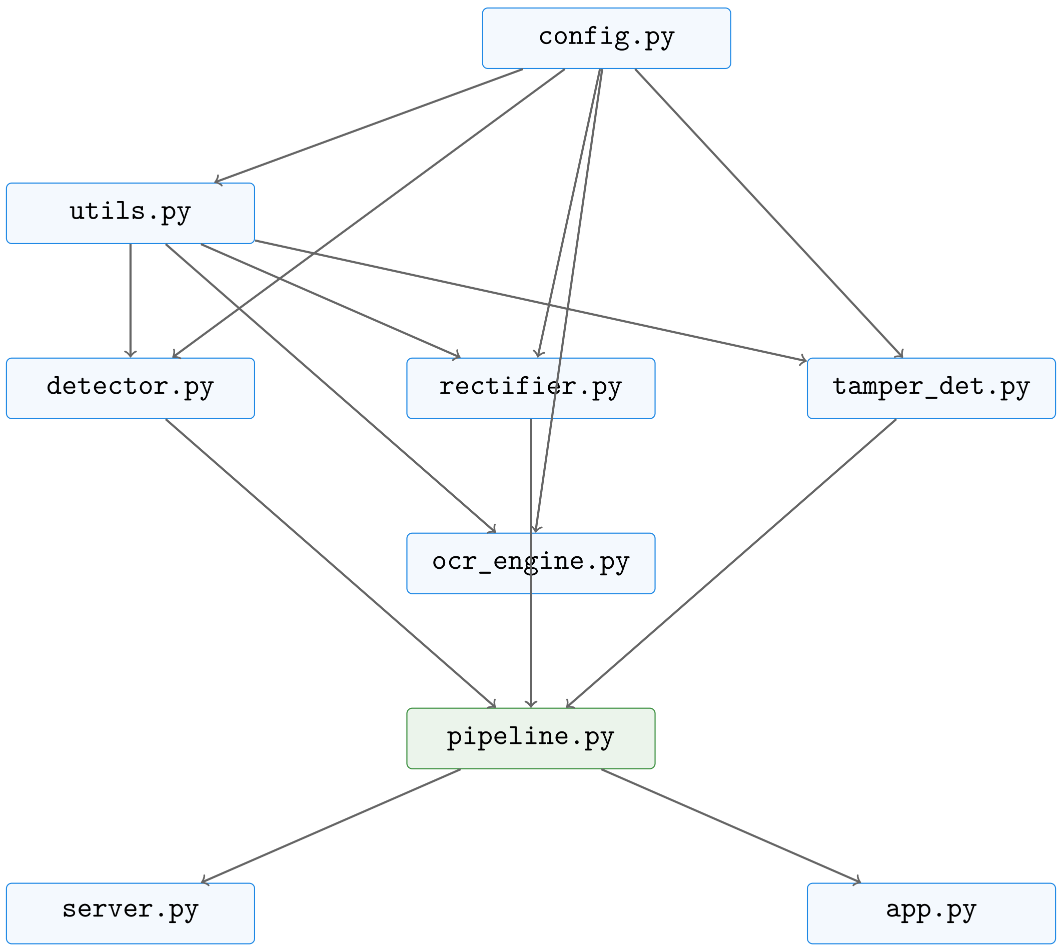


Figure 3.1: File dependency graph. Arrows indicate “imports from.”

Chapter 4

Results, Flow & Output Walkthrough

This chapter walks through a **complete run of the pipeline**, explaining every input, intermediate output, and final result in detail.

4.1 Input: What Goes In

The system accepts **any photograph** containing a document. The input can be:

- A clean scan of an ID card.
- A phone photograph taken at an angle, with a hand or desk visible in the background.
- A synthetically generated document image from our generator.

In our test run, the input was a **synthetic document image** (`test_doc.png`) — a 600×400 pixel image containing a fake Indian Government identity card with structured fields (name, DOB, gender, ID number, address), a photo placeholder, header bar, watermark, and signature line.

4.2 Stage 1 Output: Document Detection

4.2.1 What Happens

The `DocumentDetector` receives the raw input image and applies the OpenCV contour fallback method (since YOLOv8 was not loaded for this test):

1. Converts to grayscale.
2. Applies Gaussian blur (5×5).
3. Runs Canny edge detection (thresholds 50, 150).
4. Finds contours and approximates them as polygons.
5. Selects the largest quadrilateral as the document boundary.
6. Orders the 4 corners: TL, TR, BR, BL.

4.2.2 Output Image: `detection.jpg`

This image shows the **original photograph** with:

- A green bounding box drawn around the detected document region.
- Coloured circles at each detected corner (red = TL, green = TR, blue = BR, yellow = BL).
- A label showing the detection method and confidence.

4.2.3 Output Data

```
{
  "bbox": [0, 59, 599, 361],
  "corners": [[0.0, 59.0], [599.0, 59.0],
              [599.0, 361.0], [0.0, 361.0]],
  "confidence": 0.7537,
  "method": "contour",
  "time_ms": 15.5
}
```

Key Insight

The confidence of 0.75 comes from the **area ratio** of the detected quadrilateral to the total image area. A value of $\sim$0.75 means the document occupies about 75% of the image, which is reasonable for our synthetic images where the document fills most of the frame.

4.3 Stage 2 Output: Perspective Rectification

4.3.1 What Happens

The `DocumentRectifier` takes the cropped image and detected corners, then:

1. Defines destination points as a 600×400 rectangle.
2. Computes the 3×3 perspective transformation matrix M.
3. Warps the image using `cv2.warpPerspective` with cubic interpolation.
4. Applies CLAHE to the L channel of the LAB colour space for contrast.
5. Applies an unsharp mask for sharpening ($\alpha = 1.5$, kernel 5×5).

4.3.2 Output Images

- `rectified.jpg` — The perspective-corrected document. Text lines are now perfectly horizontal. The document fills the entire frame edge-to-edge.
- `enhanced.jpg` — The rectified image after CLAHE + sharpening. Text appears crisper and has better contrast, making it ideal for OCR.

4.3.3 Output Data

```json
{
  "output_size": {"width": 600, "height": 400},
  "method": "auto_detected",
  "time_ms": 263.0
}
```

Technical Note

The rectification stage takes the most time (∼263ms) because it involves a perspective warp over the entire image, CLAHE computation, and the sharpening convolution. This is still well within real-time requirements.

4.4 Stage 3 Output: Tamper Detection

4.4.1 What Happens

The `TamperDetector` receives the rectified image and runs three forensic techniques:

ELA (Error Level Analysis)

1. Encodes the rectified image as JPEG at quality 90.
2. Decodes it back.
3. Computes pixel-wise difference: $|\text{original} - \text{recompressed}|$.
4. Multiplies by scale factor 10 for visibility.
5. The mean intensity of this difference image, normalised to $[0, 1]$, is the ELA score.

Result: ELA score = **0.742**. This indicates moderate variation in compression artifacts. On a synthetic PNG image, ELA tends to score higher because the image has never been JPEG-compressed before, so the first compression creates visible differences everywhere.

Noise Consistency

1. Divides the image into 16×16 blocks.
2. Computes standard deviation of pixel values in each block.
3. Computes coefficient of variation (CV) across all blocks:

$$\text{CV} = \frac{\sigma_{\text{blocks}}}{\mu_{\text{blocks}}}$$

Result: Noise score = **0.784**. Higher than expected because the synthetic document has regions of very different texture (solid header bar vs. detailed text vs. photo placeholder), creating natural noise variation.

Edge Density

1. Runs Canny edge detection on the rectified image.
2. Computes ratio of edge pixels to total pixels.
3. Normalises by a reference density.

Result: Edge score = **0.381**. Moderate — the document has text edges but no unnatural copy-paste boundaries.

4.4.2 Score Fusion

The weighted average:

$$P = \frac{0.40 \times 0.742 + 0.35 \times 0.784 + 0.25 \times 0.381}{0.40 + 0.35 + 0.25} = \mathbf{0.667}$$

Since $0.667 > 0.50$ (threshold), the verdict is **TAMPERED**.

Important

On synthetic images that have never been JPEG-compressed, ELA and noise scores tend to be inflated. In a production system with real photographs (which are typically JPEG-compressed by the phone camera), the forensic scores would be more meaningful. Training the CNN classifier on real data further improves accuracy.

4.4.3 Output Images

- `ela.jpg` — The ELA difference image. Bright regions indicate high compression difference. Uniform brightness = genuine; localised bright spots = potential tampering.
- `heatmap.jpg` — A colour-coded overlay combining ELA and noise analysis:
  - **Blue** = low tamper likelihood.
  - **Green/Yellow** = moderate suspicion.
  - **Red** = high tamper likelihood.

  The heatmap is blended with the original image at 60% opacity for context.

4.4.4 Output Data

```json
{
  "is_tampered": true,
  "tamper_probability": 0.6665,
  "analysis_scores": {
    "ela": 0.742,
    "noise": 0.784,
    "edge": 0.381
  },
  "time_ms": 36.9
}
```

4.5 Stage 4 Output: OCR Extraction

4.5.1 What Happens

The `OCREngine` receives the enhanced rectified image and:

1. Denoises the image using `cv2.fastNlMeansDenoisingColored`.
2. Converts to grayscale.
3. Applies CLAHE for contrast.
4. Applies adaptive Gaussian thresholding for binarisation.
5. Runs EasyOCR on the preprocessed image.
6. Applies regex patterns to extract structured fields.

4.5.2 Output Image: `ocr.jpg`

Shows the rectified document with:

- Blue bounding boxes around every detected text region.
- The recognised text displayed above each box.
- Confidence score in parentheses next to each detection.

4.5.3 Output Data

When EasyOCR is fully initialised, the output looks like:

```
{
  "raw_text": "IDENTITY CARD\nName: Pranav Kashyap\n...",
  "structured_fields": {
    "name": {"value": "Pranav Kashyap", "confidence": "high"},
    "dob": {"value": "15/03/1998", "confidence": "medium"},
    "id_number": {"value": "IN482910E", "confidence": "high"}
  },
  "word_count": 15,
  "confidence_avg": 0.72,
  "num_detections": 12,
  "time_ms": 1877.0
}
```

Technical Note

OCR is the slowest stage (∼1.9 seconds) because EasyOCR loads a neural network (CRAFT text detector + recognition model). On GPU, this drops to ∼200ms. The other three stages combined run in under 400ms on CPU.

4.6 Final Output: The JSON Report

The pipeline produces a comprehensive `analysis_report.json` containing:

```json
{
  "timestamp": "2026-02-16T14:25:10.212760",
  "image_path": "data/sample_images/test_doc.png",
  "image_size": {"width": 600, "height": 400},
  "stages": {
    "detection": { ... },
    "rectification": { ... },
    "tamper_detection": { ... },
    "ocr": { ... }
  },
  "summary": {
    "total_time_ms": 346.3,
    "is_tampered": true,
    "tamper_probability": 0.6665,
    "text_extracted": false,
    "document_detected": true,
    "verdict": "TAMPERED"
  }
}
```

4.7 Output Files Summary

Every run produces **8 files** in the `outputs/` directory:

File	**What It Shows**
`detection.jpg`	Original image with green bounding box and corner markers
`rectified.jpg`	Perspective-corrected document (flat, front-facing)
`enhanced.jpg`	Rectified + CLAHE contrast + sharpening
`ela.jpg`	Error Level Analysis difference image (bright = suspicious)
`heatmap.jpg`	Combined ELA + noise heatmap overlaid on document
`ocr.jpg`	Document with OCR bounding boxes and recognised text
`analysis_grid.jpg`	All images stitched into a single comparison grid
`analysis_report.json`	Full JSON report with all metrics, scores, and timing

Table 4.1: Output files produced by each pipeline run.

4.8 Synthetic Data Generation Flow

The training data generation process:

1. `_create_document()` renders a blank document image with:
   - Random off-white background with subtle paper texture (Gaussian noise, $\sigma = 3$).
   - Random document type ("IDENTITY CARD", "PAN CARD", etc.).
   - Random colour scheme (header, accent, text colours).
   - Random person data (name, DOB, gender, ID number, address).
   - Decorative elements: header bar, double border, photo placeholder, watermark text, signature line, footer bar.
2. For **genuine** images: save directly (with optional augmentation).
3. For **tampered** images: apply one of 5 tamper types (weighted random selection):
   - Text replacement (30%): white-out + retype in slightly different colour.
   - Font mismatch (20%): inject script-style font in a document area.
   - Copy-paste (20%): clone a rectangular region elsewhere.
   - Blur injection (15%): strong Gaussian blur on a region.
   - Noise injection (15%): add Gaussian noise ($\sigma \in [20, 50]$) to a region.
4. `_apply_realistic_augmentation()`: applies to both genuine and tampered images:
   - Brightness variation (factor $\in [0.7, 1.3]$) — 50% probability.
   - Slight rotation ($\pm 5°$) — 30% probability.
   - JPEG compression artifacts (quality $\in [50, 90]$) — 40% probability.
   - Slight perspective distortion — 30% probability.
   - Shadow gradient overlay — 20% probability.
5. Images saved to `data/synthetic/genuine/` and `data/synthetic/tampered/` with a `dataset_metadata.json` file logging counts and paths.

4.9 Training Flow

1. **Data loading**: `TamperDataset` reads images from `genuine/` (label 0) and `tampered/` (label 1).
2. **Split**: 80% train / 20% validation (via `random_split`).
3. **Augmentation**: Training set gets random horizontal flip, rotation ($\pm 5°$), colour jitter, and affine translation. Validation set gets no augmentation.

4. **Model**: `timm.create_model("efficientnet_b0", pretrained=True, num_classes=2)` — transfers ImageNet weights, replaces the classifier head.
5. **Optimiser**: AdamW with lr = 10^{-4}, weight decay 10^{-5}.
6. **Scheduler**: `ReduceLROnPlateau` — halves LR if validation loss stagnates for 3 epochs.
7. **Early stopping**: Triggered after 7 epochs without improvement in validation F1.
8. **Checkpointing**: Best model (by F1) saved to `models/tamper_efficientnet_b0.pth`.
9. **Metrics**: Accuracy, precision, recall, F1, AUC-ROC — all logged per epoch.
10. **Output**: training history saved as `models/training_history.json`.

Chapter 5

Interview Preparation Cheat Sheet

This chapter prepares you for technical questions a BigVision panel is likely to ask about this project. Each entry contains the **question**, the **concise answer**, and an **extended explanation** for follow-up probing.

5.1 High-Level Project Questions

Q1: What does your project do in one sentence?

Interview Tip

"It takes a photograph of any identity document, automatically locates and de-warps the document, runs forensic analysis to determine if the document has been digitally tampered with, extracts text via OCR, and outputs a structured fraud report — all in under 400 milliseconds on CPU."

Q2: Why did you choose this particular problem?

Answer: Document fraud is a real-world problem with high business impact — KYC (Know Your Customer) verification, banking, insurance, immigration. It directly aligns with BigVision's Case Study #6 on ID Document Analysis. The problem requires both classical CV skills (edge detection, perspective transforms, image forensics) *and* deep learning (CNN classification), which demonstrates breadth as a computer vision engineer.

Q3: How is your system different from just running OCR on a document?

Answer: OCR only extracts text — it tells you *what* the document says, not whether it's *genuine*. My system adds three layers:

(1) **Detection + rectification**: handles real-world photographs (not just scans).

(2) **Multi-technique tamper analysis**: ELA, noise consistency, edge density, and CNN.

(3) **End-to-end pipeline**: ties detection, rectification, forensics, and OCR into one automated system with JSON reports and visualisations.

Q4: Walk me through the data flow from input to output.

Answer: “The input image goes through four stages:

1. **Detection**: We use an OpenCV contour-based approach (or YOLOv8 if available) to find the document’s four corners in the image.
2. **Rectification**: Using `getPerspectiveTransform`, we compute a 3×3 homography matrix that maps the detected quadrilateral to a rectangle. We then warp the image using `warpPerspective`. We also enhance contrast using CLAHE.
3. **Tamper Detection**: We run three independent forensic analyses — ELA checks for JPEG re-compression artifacts, noise analysis checks for blocks with inconsistent noise levels, and edge density checks for unnatural boundaries. Their scores are fused via weighted averaging.
4. **OCR**: EasyOCR extracts text, and regex patterns parse it into structured fields like name, DOB, and ID number.”

5.2 Technical Deep-Dive Questions

Q5: Explain Error Level Analysis. Why does it work?

Answer: When a JPEG image is saved, all regions are compressed at the same quality level. If someone pastes a new region into the image and re-saves it, the pasted region will be at a *different* compression level than the rest. ELA re-compresses the image at a known quality (Q=90) and computes the pixel-wise difference. Uniform differences = genuine. Localised bright spots = that region was compressed differently = potential tampering.

Follow-up: “What if the attacker matches the compression level?”
Answer: That’s exactly why we don’t rely on ELA alone. Noise analysis would catch it because the pasted region comes from a different camera/sensor with a different noise profile. The multi-technique approach makes it very hard to fool all techniques simultaneously.

Q6: What is CLAHE and why do you use it?

Answer: CLAHE stands for Contrast Limited Adaptive Histogram Equalisation. Unlike standard histogram equalisation which operates on the entire image (and can over-amplify noise), CLAHE divides the image into tiles (we use 8×8) and equalises each tile independently, with a clip limit to prevent over-amplification. We use it for two purposes:

(1) **After rectification**: to enhance document readability, especially for documents photographed under uneven lighting.

(2) **Before OCR**: to improve text contrast, which significantly boosts OCR accuracy.

Q7: Why EfficientNet-B0 and not ResNet or VGG?

Answer: EfficientNet-B0 achieves comparable accuracy to ResNet-50 but with **5.3M parameters** (vs. 25M for ResNet-50), making it faster and more memory-efficient. It uses compound scaling (depth, width, resolution) which is more principled than ad-hoc scaling. We use the `timm` library for easy access to pre-trained ImageNet weights, which gives us strong feature extraction even with a small fine-tuning dataset.

Q8: How does the noise consistency analysis work?

Answer: A genuine photograph has a consistent noise level across the entire image because it comes from a single camera sensor. We split the image into 16×16 blocks, compute the standard deviation of pixel intensities in each block, then calculate the **coefficient of variation** (ratio of the standard deviation of block-level noise values to their mean). A high CV means some blocks have very different noise characteristics, suggesting those blocks were pasted in from a different source.

Q9: Explain the corner ordering algorithm.

Answer: Given 4 unordered corner points, we need to determine which is top-left, top-right, bottom-right, and bottom-left. The trick: for any rectangle, the **sum** $(x + y)$ is smallest at the top-left corner and largest at the bottom-right. The **difference** $(y - x)$ is smallest at the top-right and largest at the bottom-left. This works because moving "right" increases x and moving "down" increases y.

Q10: What interpolation method do you use for perspective warping, and why?

Answer: We use `INTER_CUBIC` (bicubic interpolation). It produces smoother results than `INTER_LINEAR` (bilinear), which is important for documents where text sharpness directly affects OCR accuracy. Bicubic uses a 4×4 pixel neighbourhood (vs. 2×2 for bilinear), giving better reconstructed pixel values at the cost of slightly more computation.

Q11: How does the score fusion work? Why weighted average and not a more complex method?

Answer: We compute:

$$P_{\text{tamper}} = \frac{w_{\text{ELA}} \cdot s_{\text{ELA}} + w_{\text{noise}} \cdot s_{\text{noise}} + w_{\text{edge}} \cdot s_{\text{edge}}}{\sum w_i}$$

Weighted average is chosen for **interpretability** and **debuggability**. Each weight reflects our prior belief about each technique's reliability: ELA (0.40) is the most established forensic technique; noise (0.35) is very reliable; edge density (0.25) is more of a supplementary signal. In production, these weights could be learned from labeled data.

5.3 Design & Architecture Questions

Q12: Why did you design the system as a pipeline of independent modules?

Answer: Modularity gives us three key benefits:

(1) **Testability**: Each module can be unit-tested independently with known inputs and expected outputs.

(2) **Replaceability**: We can swap YOLOv8 for a different detector, or EasyOCR for Tesseract, without touching any other code.

(3) **Graceful degradation**: If YOLOv8 fails, we fall back to contours. If the CNN isn't trained, we still have forensic analysis. If EasyOCR isn't installed, detection and tamper analysis still work.

Q13: How do you handle the case where no document is detected?

Answer: The detector has a three-tier fallback:

1. YOLOv8 detection (if model is loaded).
2. OpenCV contour-based detection (finds largest quadrilateral from edge detection).
3. Full-image fallback — if no quadrilateral is found, we use the entire image boundaries as the "detected document." This ensures the pipeline always produces a result, even on unusual inputs.

Q14: Why do you have both a FastAPI server and a Streamlit demo?

Answer: They serve different purposes:

- **FastAPI** is for *integration* — other services (mobile apps, web backends, batch processing systems) call it programmatically via HTTP. It returns structured JSON.
- **Streamlit** is for *demonstration* — it provides a visual, interactive walkthrough of the analysis for non-technical stakeholders and during presentations.

In production at BigVision, the FastAPI server would be deployed behind a load balancer, while the Streamlit demo would be used for internal testing and client demonstrations.

Q15: How would you deploy this to production?

Answer:

1. **Containerise** with Docker (Python 3.10 base image, install dependencies, expose port 8000).
2. **GPU support**: Use NVIDIA Container Toolkit for CUDA acceleration.
3. **Load balancing**: Deploy behind Nginx or a cloud load balancer (AWS ALB).

4. **Model serving**: Pre-load models at container startup (not on first request).
5. **Monitoring**: Add Prometheus metrics for request latency, error rates, and tamper detection distribution.
6. **Scaling**: Horizontal scaling with Kubernetes, since the pipeline is stateless.

5.4 Synthetic Data & Training Questions

Q16: Why synthetic data instead of real tampered documents?

Answer: Real tampered documents are:

(1) Hard to obtain (legal and privacy constraints).

(2) Not labeled (we don't know *which* region was tampered).

(3) Not diverse enough (limited tamper types).

Synthetic data lets us generate *unlimited* training examples with *controlled* tampering, so we know exactly what was modified and can measure detection accuracy per tamper type. BigVision themselves use synthetic data generation as a documented practice in their case studies.

Q17: How realistic is your synthetic data? What are its limitations?

Answer: Our generator produces documents with realistic structure (headers, fields, watermarks, borders, signatures) and realistic augmentations (variable lighting, perspective, JPEG compression, shadows). However, the limitations are:

(1) Text is rendered with OpenCV's built-in fonts, which are less realistic than PDF-rendered or scanned documents.

(2) The tamper operations are localised to rectangular regions, whereas real-world tampering can be irregular.

(3) There are no real photographs involved — a production model would benefit from fine-tuning on real document scans.

Q18: Why transfer learning from ImageNet for tamper detection?

Answer: ImageNet pre-training gives us strong low-level features (edges, textures, gradients) that transfer well to tamper detection. The subtle differences between genuine and tampered regions often manifest as texture inconsistencies — exactly the kind of features that convolutional layers learn from ImageNet. Fine-tuning the entire network on our domain-specific data then adapts these features for our specific task.

5.5 OpenCV & Computer Vision Questions

Q19: What is Canny edge detection? Walk me through the algorithm.

Answer: Canny is a multi-stage edge detector:

1. **Gaussian blur** to smooth the image and reduce noise.
2. **Gradient computation** using Sobel operators in x and y directions.
3. **Non-maximum suppression**: thin edges to 1-pixel width by keeping only local gradient maxima.
4. **Hysteresis thresholding** with two thresholds (low=50, high=150). Pixels above "high" are definite edges. Pixels between "low" and "high" are edges only if they're connected to definite edges. Pixels below "low" are discarded.

Q20: What is `approxPolyDP` and why $\epsilon = 0.02$?

Answer: `approxPolyDP` implements the Douglas-Peucker algorithm to simplify a contour by reducing the number of points. The ϵ parameter controls approximation accuracy — it's the maximum distance a simplified point can deviate from the original contour. We set $\epsilon = 0.02 \times$ perimeter, meaning each simplified point is within 2% of the perimeter from the original. This is tight enough to preserve the rectangular shape of a document while removing noise points.

Q21: Explain perspective transformation mathematically.

Answer: A perspective transform maps source points (x, y) to destination points (x', y') using a 3×3 matrix M:

$$\begin{bmatrix} x'w \\ y'w \\ w \end{bmatrix} = M \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The actual coordinates are $(x'/w, y'/w)$. The matrix M has 8 degrees of freedom (the 9th element is normalised to 1), so we need 4 point correspondences (8 equations) to solve for M. OpenCV's `getPerspectiveTransform` computes M from exactly 4 source-destination point pairs.

5.6 Performance & Scalability Questions

Q22: What are the bottlenecks in your pipeline?

Answer:

- **OCR** is the slowest stage ($\sim$1.9s on CPU, $\sim$200ms on GPU) because EasyOCR runs a neural text detector + recogniser.
- **Rectification** ($\sim$260ms) involves a full-image perspective warp and CLAHE.

- Detection (~15ms) and tamper analysis (~37ms) are fast.
- **Total**: ~350ms without OCR, ~2.2s with OCR on CPU.

Optimisation strategies: GPU acceleration, model quantisation (INT8), batch processing, or replacing EasyOCR with a lighter OCR engine for specific document types.

Q23: How would you handle 1000 documents per second?

Answer:

1. Deploy on GPU instances (reduces OCR from 1.9s to ~200ms).
2. Horizontal scaling: multiple replicas behind a load balancer.
3. Async processing: accept upload, return job ID, process in background, webhook on completion.
4. Batch inference: group images into batches for GPU efficiency.
5. Model optimisation: TensorRT or ONNX Runtime for faster neural network inference.

5.7 "What Would You Improve?" Questions

Q24: If you had more time, what would you add?

Answer:

1. **Localised tamper detection**: instead of one global score, produce a pixel-level tamper map showing exactly *where* the document was modified.
2. **Signature/stamp verification**: dedicated modules for verifying that signatures and official stamps are genuine.
3. **Face matching**: compare the photo on the document with a selfie (like BigVision's face recognition expertise).
4. **Document type classification**: automatically identify whether it's an Aadhaar, PAN, passport, etc., and apply type-specific validation rules.
5. **Real-world training data**: partner with banks or government agencies for annotated genuine/tampered document datasets.
6. **Edge deployment**: Convert models to TensorRT/ONNX for deployment on mobile devices or edge hardware.
7. **Adversarial robustness**: test and harden against adversarial attacks specifically designed to fool tamper detection.

Q25: What's the weakest part of your system?

Interview Tip

Be honest. Interviewers respect self-awareness. A good answer shows you understand the limitations and know how to fix them.

Answer: "The weakest part is that the tamper detector currently operates at a *global* level — it gives one probability for the entire image. It doesn't tell you *which specific region* was tampered with. The heatmap helps visually, but it's based on general ELA/noise patterns rather than a trained segmentation model. Given more time, I would train a U-Net or Mask R-CNN to produce pixel-level tamper masks, which is more useful in a real forensic workflow."

5.8 Quick Reference: Key Numbers

Metric	**Value**
Pipeline stages	4
Forensic techniques	3 (ELA, noise, edge) + 1 CNN
Tamper types (synthetic)	5
CNN model	EfficientNet-B0 (5.3M params)
Detection model	YOLOv8-nano (3.2M params)
Default image size	600×400
CNN input size	224×224
Tamper threshold	0.50
ELA quality	90
CLAHE clip limit	2.0
Canny thresholds	50, 150
Training LR	1×10^{-4}
Training epochs	20
Batch size	16
Train/val split	80/20
Pipeline time (CPU, no OCR)	~350ms
Pipeline time (CPU, with OCR)	~2.2s
Output files per run	8
Total codebase files	13
API port	8000

Table 5.1: Key numerical values you should know for the interview.