Nepal ID Verification System

Operations Architecture Manual

Manjil Budhathoki

January 1, 2026

Contents

1 Introduction

This manual documents the **Nepal ID Verification System**, a production-grade machine learning pipeline designed to extract and verify data from Nepali Citizenship Cards.

The system uses a microservices architecture, containerized with Docker, featuring high-performance OCR (PaddleOCR), Object Detection (YOLOv8), and a PostgreSQL database for persistent audit logging.

2 System Architecture Directory Structure

The codebase follows a **Clean Architecture** pattern, separating the API layer, business logic, machine learning inference, and database access.

2.1 Directory Tree

```
/project-root
├── app/ .....................Main Application Package
│   ├── api/ .....................Interface Layer (FastAPI Routes)
│   ├── services/ ............Business Logic (Orchestrator, Auditor)
│   ├── ml/ .....................Machine Learning Layer
│   │   ├── detection/ ......YOLO Logic (Smart Split, Cropping)
│   │   └── ocr/ .................PaddleOCR Pipeline & Parsers
│   ├── db/ .....................Database Layer (Models, Session)
│   ├── schemas/ ............Pydantic Data Contracts
│   ├── utils/ .................Text Normalization & Date Utilities
│   └── main.py ..............Application Entry Point
├── models/ .....................ML Weights (YOLO .pt files)
├── debug_crops/ ............Intermediate images for debugging
├── Dockerfile ..............Container build instructions
├── docker-compose.yml ...Orchestration config
└── requirements.txt ......Python Dependencies
```

3 Technology Stack & Packages

3.1 Machine Learning Layer

- **ultralytics (YOLOv8):** Performs object detection to locate the ID card boundaries, face, fingerprint, and text regions. It handles the initial cropping.
- **paddlepaddle & paddleocr:** The core OCR engine. We utilize the V5 models ('en$_PP-OCRv5$'and'devanagari$_PP-OCRv5$')$for state-of-the-art accuracy on both English and Nepali text.$
- **opencv-python-headless:** Used for image pre-processing (grayscale, thresholding) and resizing. The "headless" version is strictly required for Docker to reduce image size and avoid GUI dependency errors.
- **numpy:** Handles high-performance array manipulations for image data.

3.2 Backend API Layer

- **fastapi:** A modern, high-performance web framework for building APIs with Python 3.10+. It handles request validation and async processing.

- **uvicorn:** An ASGI web server implementation used to run FastAPI.
- **gunicorn:** A production-grade process manager that manages Uvicorn workers, ensuring stability and handling restarts.
- **pydantic-settings:** Manages configuration via Environment Variables (e.g., DB credentials, Model paths) securely.

3.3 Infrastructure & Database

- **Docker:** Wraps the entire application and its OS dependencies (like 'libGL') into a portable container.
- **PostgreSQL:** A robust relational database used for storing verification history.
- **SQLAlchemy:** The ORM (Object Relational Mapper) that translates Python classes into SQL queries, making the code database-agnostic.
- **psycopg2-binary:** The PostgreSQL adapter for Python.

4 Installation & Local Operation

4.1 Prerequisites

Ensure the following are installed on your machine:

- Docker Desktop (or Docker Engine on Linux)
- Git

4.2 Step 1: Setup

Clone the repository and prepare the model files.

```
# 1. Clone the repository
git clone https://github.com/your-username/nepal-id-verifier.git
cd nepal-id-verifier

# 2. Ensure your YOLO model is in the correct place
# The Docker container expects the model at: models/yolo/best.pt
mkdir -p models/yolo
cp /path/to/your/best.pt models/yolo/
```

4.3 Step 2: Build and Run with Docker

We use `docker-compose` to spin up both the API and the Database simultaneously.

```
# Build the images and start the containers
docker compose up --build

# To run in the background (detached mode)
docker compose up -d
```

Note

The first run will take a few minutes as it downloads the PostgreSQL image and installs Python dependencies. Wait until you see: `[INFO] Application startup complete.`

4.4 Step 3: Accessing the System

Once the system is running:

- **API Endpoint:** `http://localhost:8000`
- **Interactive Documentation (Swagger UI):** `http://localhost:8000/docs`
- **Health Check:** `http://localhost:8000/health`

5 Operational Maintenance

5.1 Viewing Logs

To debug issues or view OCR output logs:

```
# Follow logs in real-time
docker compose logs -f
```

5.2 Benchmarking Performance

The system includes a bulk benchmarking script to test latency and throughput.

```
# Ensure you have images in the 'test_dataset' folder
python benchmark_bulk.py
```

5.3 Stopping the System

To stop containers and remove network bridges:

```
docker compose down
```

6 Troubleshooting

6.1 Port Conflicts

Error: `Bind for 0.0.0.0:8000 failed: port is already allocated.`
Solution: Stop any other service running on port 8000, or edit `docker-compose.yml` to map to a different port (e.g., `"8080:8000"`).

6.2 Database Connection Errors

Error: `Connection refused` for PostgreSQL.
Solution: Ensure the `depends_on` section in `docker-compose.yml` is present. Docker sometimes starts the API before the Database is fully ready. The application is configured to auto-retry, so wait 10 seconds.

6.3 Memory Issues

The OCR models are memory-intensive. Ensure your Docker Desktop settings allow for at least **4GB of RAM**.