Architecting and Evaluating an InsightFace-Based Face Recognition System

1. Introduction

This report outlines the design, evaluation, and deployment considerations for a face recognition system leveraging pre-trained models from the InsightFace repository. The primary objective is to provide a comprehensive review of the proposed system architecture, suggest improvements for modularity and clarity, and recommend best practices for implementation, evaluation, and deployment.
The system is envisioned to utilize pre-trained InsightFace models, specifically buffalo_l and buffalo_s, for face feature extraction. Its performance will be benchmarked on three standard academic datasets: Labeled Faces in the Wild (LFW), Cross-Age LFW (CALFW), and Cross-Pose LFW (CPLFW). For each dataset, the evaluation will focus on computing Accuracy, False Non-Match Rate (FNMR), and False Match Rate (FMR) across 6000 face pairs. The deployment model is a server-client architecture, where the server hosts the face recognition model and performs inference, while clients send API requests with image pairs and receive similarity scores or match/non-match determinations.
A robust design, characterized by clarity, modularity, and adherence to correct evaluation protocols, is paramount for any machine learning system intended for reliable operation. This ensures maintainability, scalability, and trustworthiness of the results, particularly in sensitive applications like face recognition.

2. High-Level System Design Review

The proposed system architecture comprises several key components working in concert to deliver face recognition capabilities.
Core Components:

- **Face Recognition Engine (InsightFace):** This is the heart of the system, responsible for taking input images, detecting faces, and extracting discriminative feature embeddings using pre-trained InsightFace models (e.g., buffalo_l, buffalo_s).
- **Evaluation Module:** This component is dedicated to assessing the performance of the Face Recognition Engine. It will manage the loading of evaluation datasets (LFW, CALFW, CPLFW), process the defined image pairs through the engine, and compute the specified performance metrics (Accuracy, FNMR, FMR).
- **API Server:** This server-side application will expose the face recognition functionality via a defined API. It will load the InsightFace model(s) upon startup and handle incoming requests from clients. FastAPI or Flask are considered for this role.
- **Client:** A client application (or script) will interact with the API Server. It will be

responsible for sending pairs of images (either as raw data or paths accessible by the server) and receiving the comparison results, such as a similarity score or a binary match/non-match decision.

Data Flow:

The operational data flow can be described as follows:

1. A client application initiates a request by sending a pair of images (or references to them) to a designated API endpoint on the server.
2. The API Server receives this request, validates it, and forwards the image data to the Face Recognition Engine.
3. The Face Recognition Engine processes each image:
    - It first detects faces within the image.
    - For each detected face, it extracts a feature embedding (a numerical vector) using the loaded pre-trained InsightFace model.
4. Once embeddings are extracted for both images in the pair, the engine computes a similarity score between them (typically using cosine similarity for normalized embeddings).
5. The API Server then takes this similarity score. It may return the raw score directly or, based on a predefined threshold, convert it into a match/non-match decision. This result is then sent back to the client.

For the evaluation process, the flow is slightly different:

1. The Evaluation Module loads the image pairs and their corresponding ground truth labels (match or non-match) from one of the specified datasets (LFW, CALFW, or CPLFW).
2. For each pair, it utilizes the Face Recognition Engine to obtain feature embeddings and calculate a similarity score.
3. These scores, along with the ground truth labels, are then used by the Evaluation Module to compute Accuracy, FNMR, and FMR, typically across a range of decision thresholds.

Initial Assessment:

The described high-level design represents a standard and logical approach for developing and deploying a face recognition service. The clear separation of concerns—model inference within the engine, API handling by the server, and performance assessment by the evaluation module—is a positive attribute that promotes modularity. Key areas that will require careful attention during development include efficient model management (loading and sharing), robust handling of API requests, and the implementation of a rigorous and correct evaluation pipeline that adheres to established dataset protocols.

3. Suggested Improvements and Best Practices

To enhance the system's architecture, modularity, clarity, and readiness for deployment, several improvements and best practices are recommended.

3.1. Architecture and Modularity

A well-architected system is easier to develop, test, maintain, and scale. The following architectural considerations are crucial:

- **Decouple Core Logic:** The core face analysis functionality—which includes face detection, alignment, and embedding extraction—should be encapsulated within a distinct module or class. This "face processing" unit can then be instantiated and utilized by both the API server for real-time requests and the evaluation scripts for batch processing. This decoupling ensures that the fundamental face recognition capabilities are independent of the web framework (e.g., FastAPI or Flask) or any specific evaluation harness. Such separation simplifies unit testing of the core logic and allows for its potential reuse in other applications or workflows without modification.
- **Configuration Management:** Avoid hardcoding parameters such as model paths, API server settings (host, port), similarity thresholds, or dataset locations. Instead, these should be externalized into configuration files (e.g., YAML format) or managed through environment variables. This practice facilitates seamless transitions between different environments (development, testing, staging, production) without requiring code changes. It also enhances security by preventing sensitive information or critical parameters from being embedded directly within the source code.
- **Asynchronous Operations (API):** For the API server, especially when using a framework like FastAPI that has strong asynchronous support, consider designing request handlers to be asynchronous. Image uploading and potentially the model inference step (if it involves I/O or can be offloaded) are often I/O-bound. Asynchronous processing allows the server to handle multiple client requests concurrently without being blocked by a single long-running operation, thereby improving overall throughput and responsiveness of the API. This is particularly important for a system that is expected to serve multiple users simultaneously.

3.2. Using InsightFace Cleanly and Correctly

Properly integrating and utilizing the InsightFace library is key to achieving reliable performance.

- **Model Loading and Management:**
  - **Model Zoo and Sources:** InsightFace provides a variety of pre-trained models, including buffalo_l and buffalo_s, through its official GitHub repository and associated model zoo.[1] The buffalo_l model typically combines a RetinaFace-10GF detector with a ResNet50-based recognition network trained on WebFace600K, while buffalo_s uses a lighter RetinaFace-500MF detector and an MBF (MobileFaceNet-based) recognition network, also trained on WebFace600K.[1] Downloadable model packages (e.g., buffalo_l.zip at 275 MB, buffalo_s.zip at 122 MB) are available from the releases section of the InsightFace GitHub.[2] An ONNX version of buffalo_l's recognition model (1k3d68.onnx) is approximately 144 MB.[3]
  - **Python Package Usage:** The most straightforward method for using these models is via the insightface Python package. This package offers a high-level API that simplifies model downloading, setup, and inference. For instance, loading a

model pack can be as simple as app = FaceAnalysis(name='buffalo_l').[1] The package internally manages the download of necessary model files and their initialization.
    - **Licensing Considerations:** This is a critical aspect. The pre-trained models provided by InsightFace, including those derived from datasets like MS1M, are generally licensed for **non-commercial research purposes only**.[1] Any deployment, particularly in a commercial context, must strictly adhere to these licensing terms. If commercial use is intended, it may necessitate specific licensing agreements with InsightFace or the use of models trained on datasets with more permissive licenses.
    - **Model Caching/Singleton Pattern:** Deep learning models are resource-intensive to load. To avoid significant performance degradation, the InsightFace model should be loaded only once when the API server starts. This single model instance should then be reused for all subsequent inference requests. This can be implemented using a global variable, a class designed as a singleton, or managed by the application framework's lifecycle (e.g., FastAPI's startup events or dependency injection). Repeatedly loading the model per request would render the system impractical for real-world use.
- **Inference:**
    - **Input Preprocessing:** Ensure that input images are preprocessed according to the requirements of the specific InsightFace model being used. This typically involves decoding the image, converting it to BGR color format, and potentially normalization. When using the FaceAnalysis class from the insightface package, much of this preprocessing is handled internally.
    - **Batching:** If the API is designed to process multiple faces simultaneously or if the underlying detection or recognition models support batch inference, leveraging this capability can significantly improve throughput. Some SCRFD detection models and ArcFace recognition models in certain InsightFace implementations support batching.[6] This is particularly beneficial for the evaluation phase where many image pairs are processed.
    - **Error Handling:** Implement robust error handling for various scenarios that can occur during inference. These include cases where no face is detected in an image, multiple faces are detected when only one is expected (requiring a strategy to select one or reject), or when the input image data is corrupted or in an unsupported format. The insightface-just-works repository, for example, was created to provide more explicit error handling (throwing exceptions or returning None) compared to potential segmentation faults in earlier versions.[7]

3.3. Structuring Code

A logical code structure enhances readability, maintainability, and testability.

- Model Loading and Core Logic:
  It is highly recommended to encapsulate all InsightFace model interactions (loading,

preprocessing, inference, postprocessing) within a dedicated class, for instance, FaceEmbedder or FaceAnalyzer, located in a core module (e.g., src/core/face_analyzer.py). This class would initialize the insightface.app.FaceAnalysis object during its instantiation, potentially taking the model name and context (CPU/GPU) as parameters.
A simplified structure for such a class could be:
Python
```
# src/core/face_analyzer.py
from insightface.app import FaceAnalysis
import numpy as np
import cv2 # OpenCV for image handling

class FaceEmbedder:
   def __init__(self, model_pack_name='buffalo_l', ctx_id=0, det_thresh=0.5,
det_size=(640, 640)):
      """
      Initializes the FaceAnalysis model.
      :param model_pack_name: Name of the InsightFace model pack (e.g., 'buffalo_l',
'buffalo_s').
      :param ctx_id: Context ID for inference (0 for GPU, -1 for CPU).
      :param det_thresh: Detection threshold.
      :param det_size: Input size for the detection model.
      """
      self.app = FaceAnalysis(name=model_pack_name,
                        allowed_modules=['detection', 'recognition'])
      self.app.prepare(ctx_id=ctx_id, det_thresh=det_thresh, det_size=det_size)

   def get_embedding(self, image_bgr: np.ndarray) -> np.ndarray | None:
      """
      Detects faces and returns the embedding of the largest detected face.
      :param image_bgr: Input image in BGR format (NumPy array).
      :return: Normalized face embedding (NumPy array) or None if no face is detected.
      """
      faces = self.app.get(image_bgr)
      if not faces:
         return None
      # Strategy: Use the largest detected face if multiple are present
      # faces are typically sorted by detection score or size by default
      # For simplicity, taking the first one, assuming it's the most prominent.
      # A more robust strategy might involve selecting based on face size or centrality.
      return faces.normed_embedding

   @staticmethod
```

```
  def calculate_similarity(emb1: np.ndarray, emb2: np.ndarray) -> float:
    """
    Calculates cosine similarity between two normalized embeddings.
    :param emb1: First normalized embedding.
    :param emb2: Second normalized embedding.
    :return: Cosine similarity score.
    """
    if emb1 is None or emb2 is None:
      return 0.0
    # Cosine similarity for L2 normalized embeddings is their dot product
    similarity = np.dot(emb1, emb2)
    return float(similarity)

# Helper function for loading image, could be in utils.py
def load_image_from_bytes(image_bytes: bytes) -> np.ndarray | None:
  try:
    nparr = np.frombuffer(image_bytes, np.uint8)
    img_bgr = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
    return img_bgr
  except Exception:
    return None
```

- Request Handling (API Layer):
  Within the API module (e.g., src/api/endpoints.py), define the API routes. These routes will accept image data, typically as file uploads. Pydantic models, defined in src/api/schemas.py, should be used for validating request payloads and structuring responses. The endpoint handlers will then invoke methods from the FaceEmbedder instance to process the images and obtain results.
- Evaluation Logic:
  The evaluation component should be structured into several files:
  - src/evaluation/datasets.py: This file will contain functions responsible for loading and parsing the LFW, CALFW, and CPLFW datasets. These datasets typically provide predefined lists of image pairs and their corresponding labels (match/non-match).[8] The LFW protocol specifies 6000 pairs, evenly split between 3000 positive (same identity) and 3000 negative (different identities) pairs, often divided into 10 cross-validation folds.[8] CALFW and CPLFW adapt this protocol, also using 6000 pairs in total, with their 3000 positive pairs specifically chosen to highlight challenges in cross-age and cross-pose scenarios, respectively.[9]
  - src/evaluation/metrics.py: This module will implement functions for calculating the required performance metrics:
    - **Accuracy**: Defined as (TP+TN)/(TP+TN+FP+FN), representing the overall correctness.[13]

        - **FNMR (False Non-Match Rate)**: Calculated as FN/(TP+FN). This is the proportion of genuine pairs (same person) that the system incorrectly identifies as non-matches.[14]
        - **FMR (False Match Rate)**: Calculated as FP/(FP+TN). This is the proportion of imposter pairs (different persons) that the system incorrectly identifies as matches.[14] These metrics are critically dependent on the similarity threshold used to decide a match.
    - src/evaluation/evaluate.py: This will be the main script to orchestrate the entire evaluation process. Its workflow will involve:
        1. Loading the image pairs and labels for a chosen dataset.
        2. Instantiating the FaceEmbedder.
        3. For each image pair:
            - Loading the images.
            - Extracting their embeddings using the FaceEmbedder.
            - Calculating the similarity score between the embeddings.
        4. Once all similarity scores are computed, the script will iterate through a range of possible similarity thresholds. For each threshold, it will determine the counts of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
        5. Using these counts, it will calculate and report Accuracy, FNMR, and FMR. Often, FNMR is reported at specific FMR values (e.g., FNMR @ FMR = 10−3), and vice-versa.[14] This typically involves generating data for ROC (Receiver Operating Characteristic) or DET (Detection Error Tradeoff) curves.

3.4. Building a Minimal but Robust API Interface (FastAPI Recommended)

FastAPI is recommended over Flask for new API development due to its modern features, high performance (built on Starlette and Pydantic), automatic data validation, serialization, and interactive API documentation generation (Swagger UI and ReDoc).

- **Key Endpoints:**
    - /health: A simple GET endpoint that returns a success status (e.g., {"status": "healthy"}) to indicate the API server is operational. This is useful for load balancers and monitoring systems.
    - /compare (or /recognize): A POST endpoint that accepts two images.
        - **Input:** Typically, two image files sent as multipart/form-data.
        - **Output:** A JSON response containing the similarity_score (a float between -1.0 and 1.0 for cosine similarity). Optionally, it can include an is_match boolean field (determined by applying a configurable default threshold to the similarity score) and an error field for any issues encountered.
        - **Request/Response Schemas (using Pydantic in src/api/schemas.py):**
          Python
          ```
          from pydantic import BaseModel
          ```

```
from typing import Optional

class ComparisonInput(BaseModel):
    # If accepting image paths or URLs, define here.
    # For file uploads, FastAPI handles them directly in the path operation
function.
    pass

class ComparisonResponse(BaseModel):
    similarity_score: float
    is_match: Optional[bool] = None # Based on a server-side threshold
    message: Optional[str] = None   # For errors or additional info
```

- **Error Handling:** Utilize FastAPI's built-in HTTPException for returning appropriate HTTP status codes and error details to the client in a structured JSON format. This ensures graceful failure modes.
- **Logging:** Implement comprehensive logging throughout the API. Log incoming requests, key processing steps, inference times, computed similarity scores, and any errors encountered. This is invaluable for debugging, monitoring, and auditing.
- **Dependency Injection for Model:** FastAPI's dependency injection system is an elegant way to manage the lifecycle of the FaceEmbedder instance. The model can be loaded during application startup and injected into the path operation functions that need it, ensuring it's loaded once and shared efficiently.
- **Example API Structure (src/api/main.py using FastAPI):**
  Python
  ```
  from fastapi import FastAPI, File, UploadFile, HTTPException, Depends
  from typing import Annotated # For Depends in newer FastAPI versions
  import numpy as np
  import logging

  from src.core.face_analyzer import FaceEmbedder, load_image_from_bytes # Assuming
  these are in your core module
  from src.api.schemas import ComparisonResponse # Your Pydantic response model

  # Configure logging
  logging.basicConfig(level=logging.INFO)
  logger = logging.getLogger(__name__)

  app = FastAPI(title="Face Recognition API", version="1.0.0")

  # --- Model Loading and Dependency Injection ---
  # This is a simplified way to load the model.
  # For production, consider more robust configuration management for model_name,
  ```

```
ctx_id etc.
# e.g., from environment variables or a config file.
face_embedder_instance = FaceEmbedder(model_pack_name='buffalo_l', ctx_id=0) #
Use GPU 0

def get_face_embedder():
    # This function could implement more complex logic for providing the embedder,
    # e.g., connection pooling if the embedder was a remote service.
    return face_embedder_instance

# --- API Endpoints ---
@app.post("/compare", response_model=ComparisonResponse)
async def compare_faces_endpoint(
    image1: UploadFile = File(...),
    image2: UploadFile = File(...),
    embedder: FaceEmbedder = Depends(get_face_embedder) # Dependency injection
):
    """
    Compares two uploaded images and returns their face similarity score.
    """
    try:
        logger.info(f"Received comparison request for {image1.filename} and
{image2.filename}")

        contents1 = await image1.read()
        contents2 = await image2.read()

        img1_bgr = load_image_from_bytes(contents1)
        img2_bgr = load_image_from_bytes(contents2)

        if img1_bgr is None or img2_bgr is None:
            logger.error("Invalid image format or failed to decode one or both images.")
            raise HTTPException(status_code=400, detail="Invalid image format provided for
one or both images.")

        emb1 = embedder.get_embedding(img1_bgr)
        emb2 = embedder.get_embedding(img2_bgr)

        if emb1 is None:
            logger.warning(f"No face detected in image: {image1.filename}")
            return ComparisonResponse(similarity_score=0.0, message=f"No face detected
in {image1.filename}.")
        if emb2 is None:
            logger.warning(f"No face detected in image: {image2.filename}")
            return ComparisonResponse(similarity_score=0.0, message=f"No face detected
in {image2.filename}.")

        similarity = embedder.calculate_similarity(emb1, emb2)
        logger.info(f"Comparison successful. Similarity: {similarity:.4f}")

        # Optionally, determine is_match based on a server-side threshold
        # default_threshold = 0.5 # Example threshold, should be configurable
        # is_match = similarity >= default_threshold

        return ComparisonResponse(similarity_score=similarity) #, is_match=is_match)

    except HTTPException as http_exc:
        # Re-raise HTTPException to let FastAPI handle it
        raise http_exc
    except Exception as e:
        logger.exception("An unexpected error occurred during face comparison.") # Logs
stack trace
        raise HTTPException(status_code=500, detail=f"Internal server error: {str(e)}")

@app.get("/health")
async def health_check():
    """
    Health check endpoint.
    """
    logger.info("Health check successful.")
    return {"status": "healthy", "model_loaded": True if face_embedder_instance else
False}

# To run this (example): uvicorn src.api.main:app --reload
```

This structure promotes clarity by separating concerns: model logic in core, API definitions in api, and data structures in schemas. Using dependency injection for the model loader ensures it's initialized efficiently.

4. Dataset Evaluation Details

Rigorous evaluation on standard datasets is crucial for understanding the performance characteristics of the chosen face recognition models. The LFW, CALFW, and CPLFW datasets are specifically designed to test different aspects of unconstrained face verification.

4.1. LFW (Labeled Faces in the Wild)

- **Description:** LFW is a widely adopted benchmark for unconstrained face verification. It comprises 13,233 images of 5,749 distinct individuals, collected from the web.[20] A subset of 1,680 individuals have two or more images in the dataset.[20]
- **Protocol:** The standard evaluation protocol for LFW, known as the "View 2" protocol, involves 6000 face pairs. These pairs are divided into 10 disjoint folds, with each fold containing 600 pairs: 300 positive pairs (images of the same person) and 300 negative pairs (images of different persons).[8] Performance is typically averaged across these 10 folds.
- **Metrics:** The primary metric reported for LFW is accuracy. However, FNMR at specific FMR values (e.g., FNMR @ FMR=10−3) is also a common metric in more detailed benchmarks.[19]
- **Expected Performance:** High-performing models like InsightFace's buffalo_l are reported to achieve LFW accuracy of 99.83%, while buffalo_s achieves 99.70%.[1] These figures serve as a baseline for the evaluation.

4.2. CALFW (Cross-Age LFW)

- **Description:** CALFW is an extension of LFW designed to specifically evaluate the robustness of face verification algorithms to age variations.[9] It was created because performance on LFW had approached saturation, potentially masking challenges like aging.[9] CALFW achieves this by deliberately selecting 3,000 positive pairs from LFW identities that exhibit significant age differences.[9] To maintain challenge, negative pairs are selected to be of the same gender and race.[9]
- **Protocol:** CALFW maintains the overall data size and the "same/different" verification protocol of LFW.[9] This implies a total of 6000 pairs for evaluation (3000 positive cross-age pairs and 3000 challenging negative pairs), also likely divided into 10 folds similar to LFW. Performance on CALFW is typically lower than on LFW due to the added age complexity; for example, ArcFace's accuracy was reported to drop from 99.82% on LFW to 95.87% on CALFW.[9]
- **Metrics:** Similar to LFW, accuracy is a key metric, along with FNMR and FMR.
- **Expected Performance:** While direct CALFW scores for buffalo_l and buffalo_s are not listed in the provided snippet [1], their performance on AgeDB-30 (another age-focused dataset) can be indicative. buffalo_l achieves 98.23% on AgeDB-30, and buffalo_s achieves 96.58%.[1] The evaluation will establish their specific CALFW performance.

4.3. CPLFW (Cross-Pose LFW)

- **Description:** CPLFW is another LFW variant, constructed to evaluate face verification performance under significant pose variations.[10] Similar to CALFW, it addresses the limitations of the original LFW benchmark where pose differences might not be sufficiently challenging. CPLFW includes 3,000 positive pairs specifically selected for their cross-pose nature.[10] Negative pairs are also carefully chosen to be of the same gender and race to increase difficulty.[10]

- **Protocol:** CPLFW adheres to LFW's data size and verification protocol structure, implying 6000 total pairs (3000 positive cross-pose pairs and 3000 challenging negative pairs), likely organized in 10 folds.[10] Performance on CPLFW is generally lower than on LFW due to pose variations. For instance, ArcFace's accuracy reportedly dropped from 99.82% (LFW) to 92.08% (CPLFW).[10]
- **Metrics:** Accuracy, FNMR, and FMR are the target metrics.
- **Expected Performance:** Specific CPLFW scores for buffalo_l and buffalo_s are not readily available in the provided materials.[1] The evaluation phase of this project will determine these values.

4.4. Metric Calculation and Reporting

The calculation of Accuracy, FNMR, and FMR from the 6000 pairs in each dataset requires a clear understanding of how similarity scores are converted into match/non-match decisions.

- **Thresholding:** The face recognition model outputs a similarity score for each pair of faces (typically cosine similarity, ranging from -1 to 1, or 0 to 1 if scaled). A decision threshold (τ) is applied to this score:
    - If similarity score > τ, the pair is classified as a "match."
    - If similarity score $\leq\tau$, the pair is classified as a "non-match."
- **Determining TP, TN, FP, FN:** For each pair, based on the ground truth and the model's decision at a given threshold:
    - **True Positive (TP):** Ground truth is "match" (same person), and model decides "match."
    - **False Negative (FN):** Ground truth is "match," but model decides "non-match."
    - **True Negative (TN):** Ground truth is "non-match" (different persons), and model decides "non-match."
    - **False Positive (FP):** Ground truth is "non-match," but model decides "match."
- **Metric Formulas:**
    - **Accuracy:** Accuracy=(TP+TN)/(TP+TN+FP+FN) [13]
    - **False Non-Match Rate (FNMR):** FNMR=FN/(TP+FN) (The proportion of actual matches that were missed) [15]
    - **False Match Rate (FMR):** FMR=FP/(FP+TN) (The proportion of actual non-matches that were incorrectly declared as matches).[15] Note: (FP+TN) represents the total number of actual negative pairs.
- **Reporting Standards:**
    - **Accuracy:** Often reported at the threshold that maximizes this value. This "optimal" threshold can be found by sweeping through a range of possible threshold values and calculating accuracy at each step.
    - **FNMR at specific FMR values (and vice-versa):** This is a more comprehensive way to report performance, as it shows the trade-off between security (low FMR) and convenience (low FNMR). For example, FNMR might be reported at FMR = 10−2, 10−3, 10−4, etc..[14] This requires generating data points for an ROC (Receiver Operating Characteristic) curve or a DET (Detection Error Tradeoff) curve by

  varying the decision threshold across its entire range and calculating FNMR and FMR at each point.
  - The evaluation across all 6000 pairs for each dataset is a standard derived from the LFW protocol.[8]

The evaluation process should systematically vary the similarity threshold to observe its impact on TP, TN, FP, and FN counts, thereby allowing for the calculation of metrics across a spectrum of operating points. This provides a much richer understanding of model performance than a single accuracy value at an arbitrary threshold.

5. Proposed Folder Layout

A well-organized project structure is essential for clarity, maintainability, and collaboration. The following folder layout is proposed, incorporating best practices for Python projects and machine learning systems:

```
project_root/
├──.env              # Optional: For environment variables (API keys, non-default model paths)
├──.gitignore         # Standard Python.gitignore to exclude virtualenvs, pycache, etc.
├── Dockerfile         # For containerizing the API server for deployment
├── requirements.txt    # List of Python dependencies for pip
├── config/           # Directory for configuration files
│   └── model_config.yaml # Example: Model names, paths, default thresholds, API settings
├── data/             # Directory for storing raw and processed evaluation datasets
│   ├── lfw/          # LFW images and pair lists
│   ├── calfw/         # CALFW images and pair lists
│   └── cplfw/         # CPLFW images and pair lists
├── models/           # Directory for storing downloaded pre-trained models
│   └── insightface/    # Subdirectory for InsightFace models
│       ├── buffalo_l/  # Files for buffalo_l model pack
│       └── buffalo_s/  # Files for buffalo_s model pack
├── notebooks/         # Jupyter notebooks for experimentation, data exploration, quick tests
├── src/              # Main source code for the application
│   ├── __init__.py     # Makes 'src' a Python package
│   ├── api/          # Code related to the FastAPI/Flask application
│   │   ├── __init__.py
│   │   ├── main.py     # FastAPI app initialization, root path operations, startup/shutdown
events
│   │   ├── endpoints.py# API route definitions (e.g., /compare, /health)
│   │   └── schemas.py  # Pydantic schemas for request/response validation and serialization
│   ├── core/          # Core face recognition logic, independent of API or evaluation
│   │   ├── __init__.py
│   │   ├── face_analyzer.py # Class wrapping InsightFace model loading and inference (e.g.,
FaceEmbedder)
│   │   └── utils.py       # Utility functions (image loading/processing, common helpers)
│   └── evaluation/     # Scripts and modules for model evaluation
│       ├── __init__.py
│       ├── datasets.py # Functions for loading and parsing LFW, CALFW, CPLFW pairs
│       ├── metrics.py  # Functions for calculating Accuracy, FNMR, FMR, etc.
│       └── evaluate.py # Main script to run evaluations on specified datasets and models
├── scripts/           # Utility scripts (e.g., download_models.sh, download_datasets.sh, data
preprocessing)
├── tests/             # Unit and integration tests for the system
│   ├── __init__.py
│   ├── api/           # Tests for the API endpoints and logic
│   └── core/          # Tests for the core face recognition logic
└── README.md          # Project overview, setup instructions, usage guidelines, and
evaluation results summary
```

This proposed folder structure promotes a clear separation of concerns. The src directory houses all primary application code, further divided into api, core, and evaluation submodules. This distinction is fundamental:

- The core module contains the central face processing intelligence, reusable across different parts of the system.
- The api module focuses solely on exposing this intelligence via a web interface.
- The evaluation module provides the tools to benchmark the core components. Such organization directly supports the goals of clarity and readiness for evaluation and deployment by making the codebase easier to navigate, test, and manage. For instance, the API can be deployed independently by containerizing the relevant parts of src/api and src/core, along with the models and configurations. Similarly, evaluations can be run using src/evaluation and src/core without needing the API components. This modularity is a cornerstone of robust software engineering.

6. Conclusion and Next Steps

This report has reviewed the proposed design for an InsightFace-based face recognition system, offering suggestions for architectural improvements, best practices for implementation, and a detailed plan for evaluation. The goal is to achieve a system that is clear, modular, and ready for both rigorous evaluation and potential deployment.

Recap of Key Recommendations:

- **Licensing Adherence:** The most critical non-technical consideration is the licensing of pre-trained InsightFace models. These are typically available for **non-commercial research purposes only**.[1] Any deviation, especially towards commercial application, requires careful review and potentially direct engagement with InsightFace for appropriate licensing. This constraint must be respected throughout the project lifecycle.

- **Modular Design:** A strong emphasis should be placed on decoupling the core face recognition logic from the API server and the evaluation scripts. This enhances reusability, testability, and maintainability.
- **API Framework Choice:** FastAPI is recommended for the API server due to its modern features, high performance, native asynchronous support, and built-in data validation and documentation capabilities, which align well with the need for a robust interface.
- **Rigorous Evaluation Protocol:** Adherence to the standard evaluation protocols for LFW, CALFW, and CPLFW (6000 pairs each, with specific positive/negative pair selection criteria) is essential. Metric calculation must be comprehensive, including Accuracy, and FNMR/FMR reported at various operating thresholds (e.g., FNMR @ specific FMR values), ideally visualized with ROC/DET curves.
- **Structured Code and Configuration:** Employing the suggested folder layout and externalizing configurations will contribute significantly to the project's clarity and ease of management.

Suggestions for Phased Implementation:

A phased approach can help manage complexity and ensure each component is developed and tested systematically:

1. **Phase 1: Setup and Basic Inference:**
    - Install the insightface Python package and its dependencies.
    - Develop the initial FaceEmbedder class in src/core/ to load the buffalo_l and buffalo_s model packs.
    - Implement basic inference functionality to extract embeddings from sample images and calculate similarity. Test this with a few known image pairs.
2. **Phase 2: Evaluation Pipeline Development:**
    - Implement the dataset loading functions in src/evaluation/datasets.py for LFW first.
    - Develop the metric calculation functions in src/evaluation/metrics.py.
    - Create the main src/evaluation/evaluate.py script to process the LFW dataset, perform threshold sweeping, and generate Accuracy, FNMR, and FMR results.
    - Validate results against known benchmarks for the chosen models if available.
    - Extend the evaluation pipeline to support CALFW and CPLFW.
3. **Phase 3: API Development:**
    - Set up the FastAPI application structure in src/api/.
    - Implement the /health and /compare endpoints, using Pydantic schemas for request/response handling.
    - Integrate the FaceEmbedder from src/core/ using FastAPI's dependency injection.
    - Implement basic error handling and logging.
4. **Phase 4: Integration, Refinement, and Testing:**
    - Thoroughly test the API endpoints with various valid and invalid inputs.
    - Implement comprehensive unit tests for the core logic and api endpoints.
    - Refine error handling, logging, and configuration management based on testing.
    - Ensure all components work together seamlessly.

5. **Phase 5: Deployment Preparation:**
   - Create a Dockerfile to containerize the FastAPI server.
   - Test the Dockerized application.
   - If performance is a critical bottleneck for the target deployment environment, investigate advanced optimization techniques such as model conversion to TensorRT (the InsightFace-REST repository offers examples and claims significant speedups [6]).

Further Considerations (Beyond Initial Scope but Relevant for Maturation):

- **Fine-tuning:** If the pre-trained models exhibit suboptimal performance on a specific, in-domain dataset that is critical for the application, fine-tuning might be considered. However, this is a complex task requiring a substantial and diverse training dataset, and it is often challenging to surpass the generalization of models trained on massive datasets like WebFace600K.[23] Licensing of the base model and the fine-tuning dataset would also need careful consideration.
- **Bias and Fairness Assessment:** Face recognition systems can exhibit performance disparities across different demographic groups (e.g., based on race, gender, age). It is crucial to be aware of these potential biases and, if possible, evaluate the system's performance on diverse demographic subgroups to ensure fairness and equity.[18]
- **Anti-Spoofing (Liveness Detection):** For any real-world deployment where security is a concern, integrating robust anti-spoofing or presentation attack detection (PAD) mechanisms is essential. This prevents the system from being fooled by photographs, videos, or masks.[25] InsightFace has research and challenges related to anti-spoofing.[5]
- **Alternative Deployment Backends:** For large-scale, high-throughput deployments, exploring specialized inference servers like NVIDIA Triton Inference Server could be beneficial. The InsightFace-REST project mentions Triton as a potential future backend [6], indicating its relevance for robust deployment scenarios.

By following these recommendations and adopting a structured development process, the project can deliver a well-architected and thoroughly evaluated face recognition system that meets the specified requirements for clarity, simplicity, and readiness for evaluation and deployment.

Works cited

1. insightface/model_zoo/README.md at master - GitHub, accessed May 31, 2025, https://github.com/deepinsight/insightface/blob/master/model_zoo/README.md
2. Releases · deepinsight/insightface - GitHub, accessed May 31, 2025, https://github.com/deepinsight/insightface/releases
3. models/buffalo_l/1k3d68.onnx · lithiumice/insightface at 1141cd22e2bff0d4036d10ba4151903605a8902d - Hugging Face, accessed May 31, 2025, https://huggingface.co/lithiumice/insightface/blob/1141cd22e2bff0d4036d10ba4151903605a8902d/models/buffalo_l/1k3d68.onnx
4. swoook/insightface: Cloned from deepinsight/insightface

(https://github.com/deepinsight/insightface) - GitHub, accessed May 31, 2025, https://github.com/swoook/insightface
5. deepinsight/insightface: State-of-the-art 2D and 3D Face ... - GitHub, accessed May 31, 2025, https://github.com/deepinsight/insightface
6. InsightFace REST API for easy deployment of face recognition services with TensorRT in Docker. - GitHub, accessed May 31, 2025, https://github.com/SthPhoenix/InsightFace-REST
7. Insightface face detection and recognition model that just works out of the box. - GitHub, accessed May 31, 2025, https://github.com/kiselev1189/insightface-just-works
8. paperswithcode.com, accessed May 31, 2025, https://paperswithcode.com/sota/face-recognition-on-lfw#:~:text=The%20LFW%20dataset%20contains%2013%2C233,reported%20on%206000%20face%20pairs.
9. Cross-Age LFW (CALFW) Database - Weihong Deng, accessed May 31, 2025, http://whdeng.cn/CALFW/index.html
10. Cross-Pose LFW (CPLFW) Database - Weihong Deng, accessed May 31, 2025, http://www.whdeng.cn/CPLFW/index.html
11. stefhoer/PartialLFW: Official repository for Attention-based Partial Face Recognition - GitHub, accessed May 31, 2025, https://github.com/stefhoer/PartialLFW
12. Cross-Age LFW (CALFW) Database - Weihong Deng, accessed May 31, 2025, http://www.whdeng.cn/CALFW/index.html
13. Classification: Accuracy, recall, precision, and related metrics ..., accessed May 31, 2025, https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall
14. Face Recognition Technology Evaluation (FRTE) 1:1 Verification - NIST Pages, accessed May 31, 2025, https://pages.nist.gov/frvt/html/frvt11.html
15. Metrics for evaluating an identity verification solution | AWS Machine ..., accessed May 31, 2025, https://aws.amazon.com/blogs/machine-learning/metrics-for-evaluating-an-identity-verification-solution/
16. What is False Non-Match Rate (FNMR)? - Facia.ai, accessed May 31, 2025, https://facia.ai/knowledgebase/what-is-false-non-match-rate-fnmr/
17. The Secret to Better Face Recognition Accuracy: Thresholds - Kairos, accessed May 31, 2025, https://face.kairos.com/blog/the-secret-to-better-face-recognition-accuracy-thresholds
18. Evaluating Proposed Fairness Models for Face Recognition Algorithms - Maryland Test Facility, accessed May 31, 2025, https://mdtf.org/publications/ICPR2022-Fairness.pdf
19. LFW Benchmark (Face Recognition) | Papers With Code, accessed May 31, 2025, https://paperswithcode.com/sota/face-recognition-on-lfw
20. Labelled Faces in the Wild (LFW) Dataset - GTS.AI, accessed May 31, 2025,

https://gts.ai/dataset-download/labelled-faces-in-the-wild-lfw-dataset/
21. lfw | TensorFlow Datasets, accessed May 31, 2025, https://www.tensorflow.org/datasets/catalog/lfw
22. CPLFW(Cross-Pose LFW) - OpenDataLab, accessed May 31, 2025, https://opendatalab.com/OpenDataLab/CPLFW/download
23. Transfer learning insightface on my own dataset - Stack Overflow, accessed May 31, 2025, https://stackoverflow.com/questions/79517641/transfer-learning-insightface-on-my-own-dataset
24. Examples of the validation datasets (LFW, CALFW, CPLFW, CFP-FP and... - ResearchGate, accessed May 31, 2025, https://www.researchgate.net/figure/Examples-of-the-validation-datasets-LFW-CALFW-CPLFW-CFP-FP-and-AgeDB-30-augmented_fig1_362704996
25. Real Time Face Recognition System using python Insightface library | Opencv and numpy, accessed May 31, 2025, https://www.youtube.com/watch?v=teMsJZyjiro