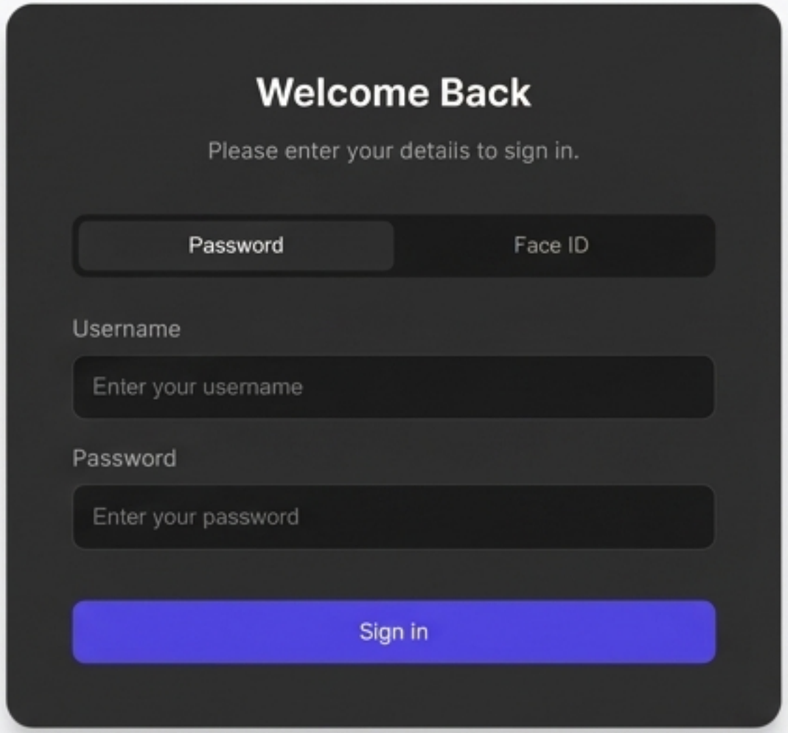


How Does This Work?

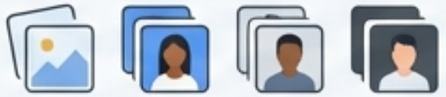
We’re going to build the facial recognition system behind this screen, step-by-step.

This presentation deconstructs a complete Face Security System, teaching you how each component is built and how they collaborate to create a seamless user experience.

The System Blueprint: From Data to Detection

Phase 1: Offline Training (The 'Learning' Phase)

Image Files (`/FaceData`)



`detector.py`
MTCNN & InceptionResnetV1

Training Script (`train_model.py`)

Face Model (`face_model.pth`)
Database of known face fingerprints.

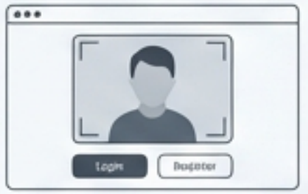
Phase 2: Online Recognition (The 'Working' Phase)

Webcam (`camera.py`)

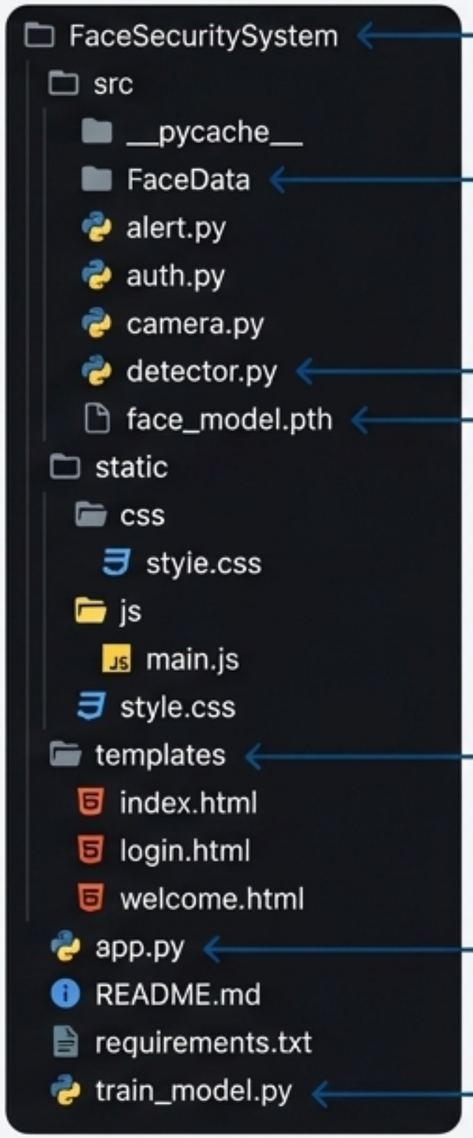
`detector.py`

Web Application (`app.py`)

User Interface



Organizing Our Codebase



Our **starting point.** Reads image data and creates the recognition model. (Phase 1)

The **raw material.** Contains labeled images of known individuals.

The core intelligence. Contains the logic for finding faces and generating/comparing their unique “fingerprints”.

The **output of our training.** A single file containing all known face data.

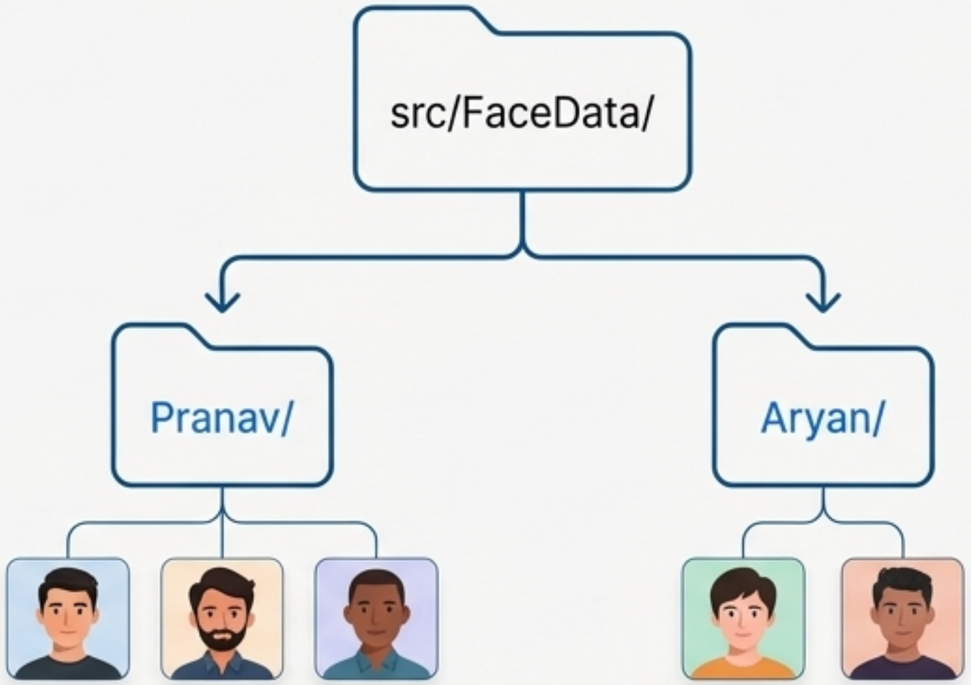
The system’s “eyes.” Captures the video feed and performs real-time detection.

The central nervous system. A Flask web server that handles user requests, streams video, and manages authentication.

The user interface (HTML, CSS, JS).

Step 1: Gathering the Raw Data

The foundation of any recognition system is its dataset. Our system learns from a simple folder structure.



Location
`src/FaceData/`

Logic
The system walks through this directory. The name of each sub-folder is used as the label for all images within it.

Example

```
src/FaceData/Pranav/
    - 001.jpg
    - 002.png
src/FaceData/Aryan/
    - image_1.jpg
```

All images inside the `Pranav` folder will be learned and labeled as 'Pranav'. This simple convention is the basis for our training.

Step 2: Running the Training Script

The `train_model.py` script is a simple but powerful utility. Its sole purpose is to convert the images in `/FaceData` into a compact, efficient model file that our live system can use.

Process Overview

1. Initialize

It creates an instance of our `FaceDetector`.

2. Load & Process

It calls `detector.load_known_faces()` to scan the `/FaceData` directory, find all faces, and calculate their embeddings.

3. Save

It calls `detector.save_model('src/face_model.pth')` to bundle all the processed face data into a single file.

Code Snippet (`train_model.py`):

```
# Initialize detector
detector = FaceDetector()

# Load faces from the raw images directory
detector.load_known_faces()

# Save the model
model_path = 'src/face_model.pth'
detector.save_model(model_path)
```

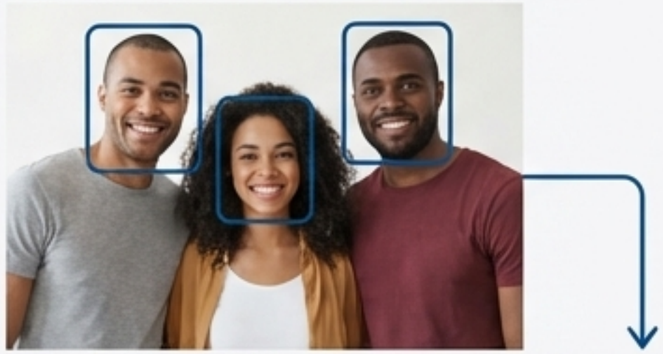
The AI Core: Two Models, Two Jobs

Inside `detector.py`, we use two pre-trained models to do the heavy lifting. We are not training these models from scratch; we are using their power for our specific tasks.

1. MTCNN (Multi-task Cascaded Convolutional Networks)

Job: Face Detection

Scans an image and returns the precise coordinates (bounding boxes) of any faces it finds. It is our "face finder."

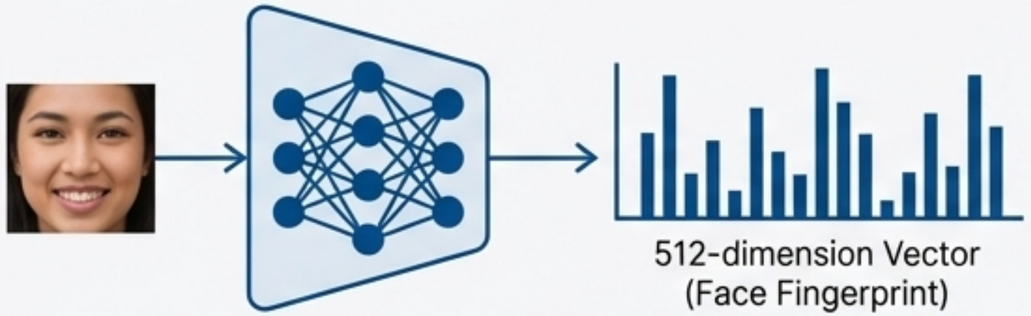


```
self.mtcnn = MTCNN(keep_all=True, device=self.device)
```

2. InceptionResnetV1 (trained on VGGFace2)

Job: Feature Extraction

Takes a cropped image of a single face (from MTCNN) and converts it into a 512-dimension vector, also known as an "embedding." This vector is a unique mathematical signature—a "face fingerprint."

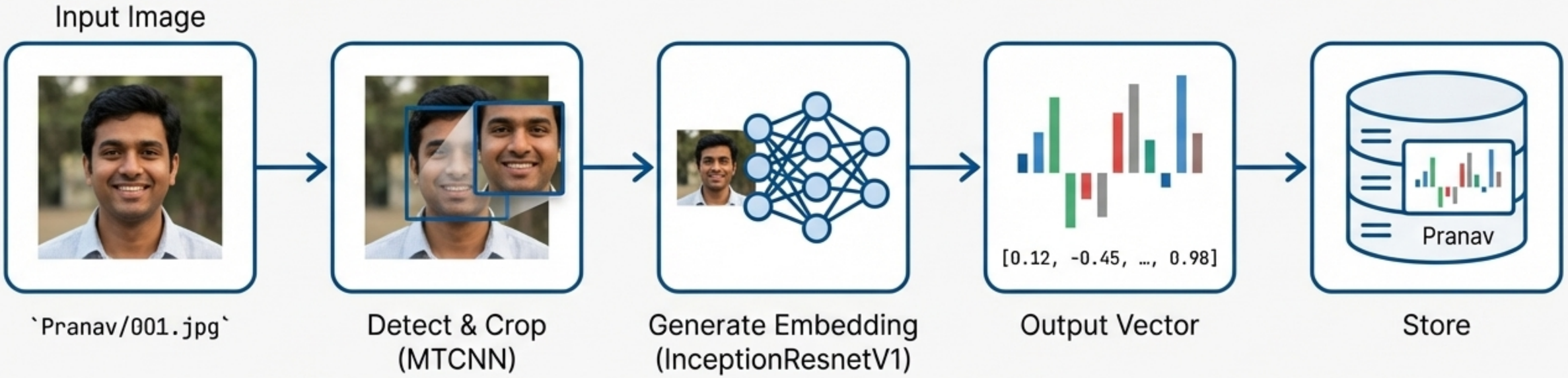



```
self.resnet = InceptionResnetV1(pretrained='vggface2').
eval().to(self.device)
```

How We Create a "Face Fingerprint"

The `load_known_faces()` method executes a clear sequence for every image in our dataset. This process converts a picture into a storable, comparable mathematical representation.

The Embedding Pipeline



This process is repeated for every image, creating a comprehensive database of known face embeddings.

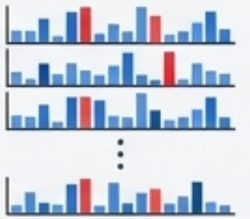
The Result of Training: `face_model.pth`

After the training script finishes, we are left with a single file: **`face_model.pth`**. This file is the entire "memory" of our system.

What's inside?

It's not a neural network. It's a simple dictionary saved using PyTorch, containing two tensors:

- **'embeddings'**: A stacked tensor containing all the 512-dimension face vectors from our known individuals.



- **'names'**: A list of names, where each name corresponds to an embedding at the same index.

```
['Pranav',
 'Aryan',
 'Pranav',
...]
```

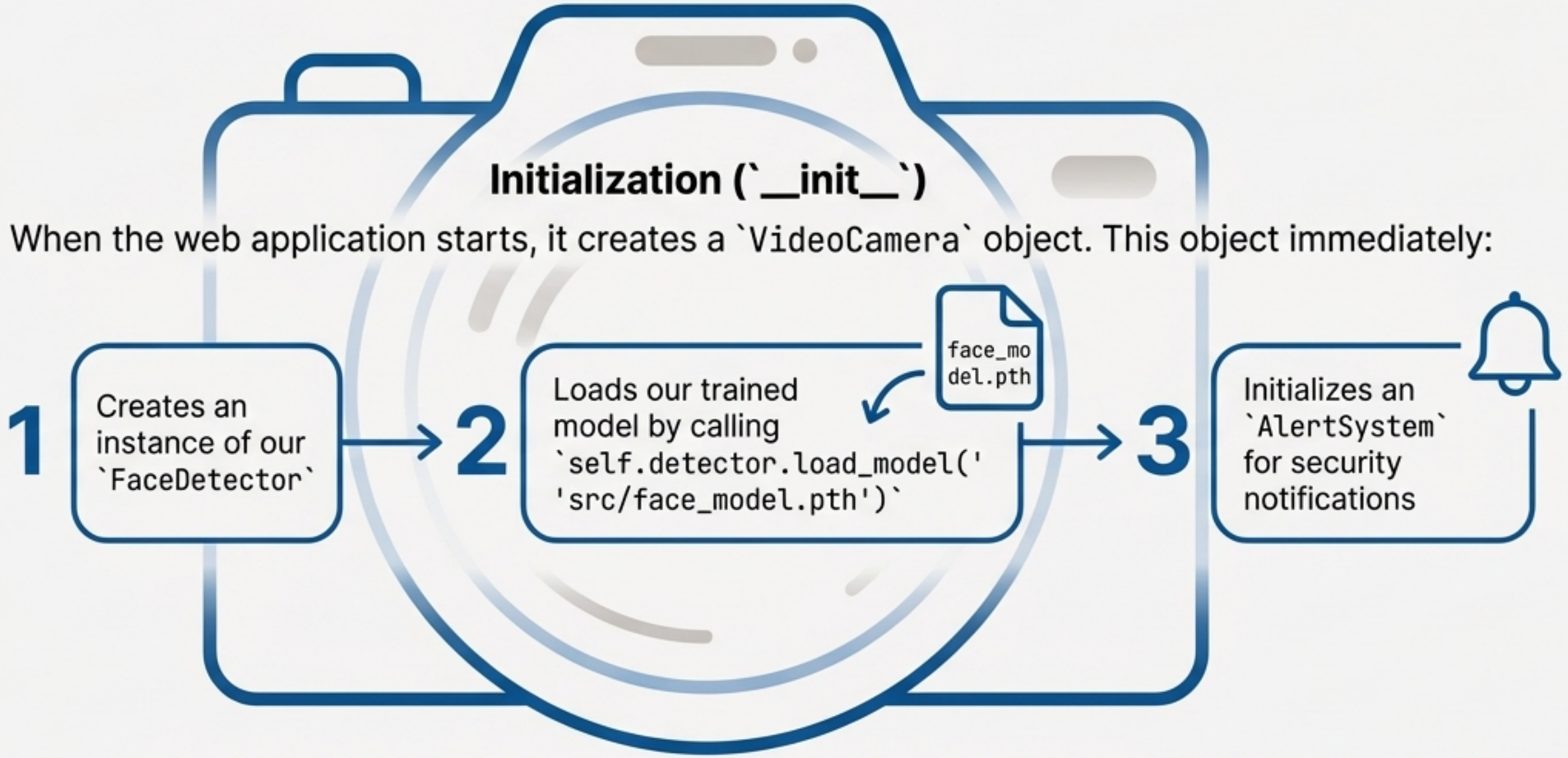
Code Snippet (`detector.py`)

```
# The data structure we save
torch.save({
    'embeddings': self.known_embeddings,
    'names': self.known_names
}, model_path)
```

With this file created, our system is now 'trained' and ready for live recognition.

Act II: The System's 'Eyes'

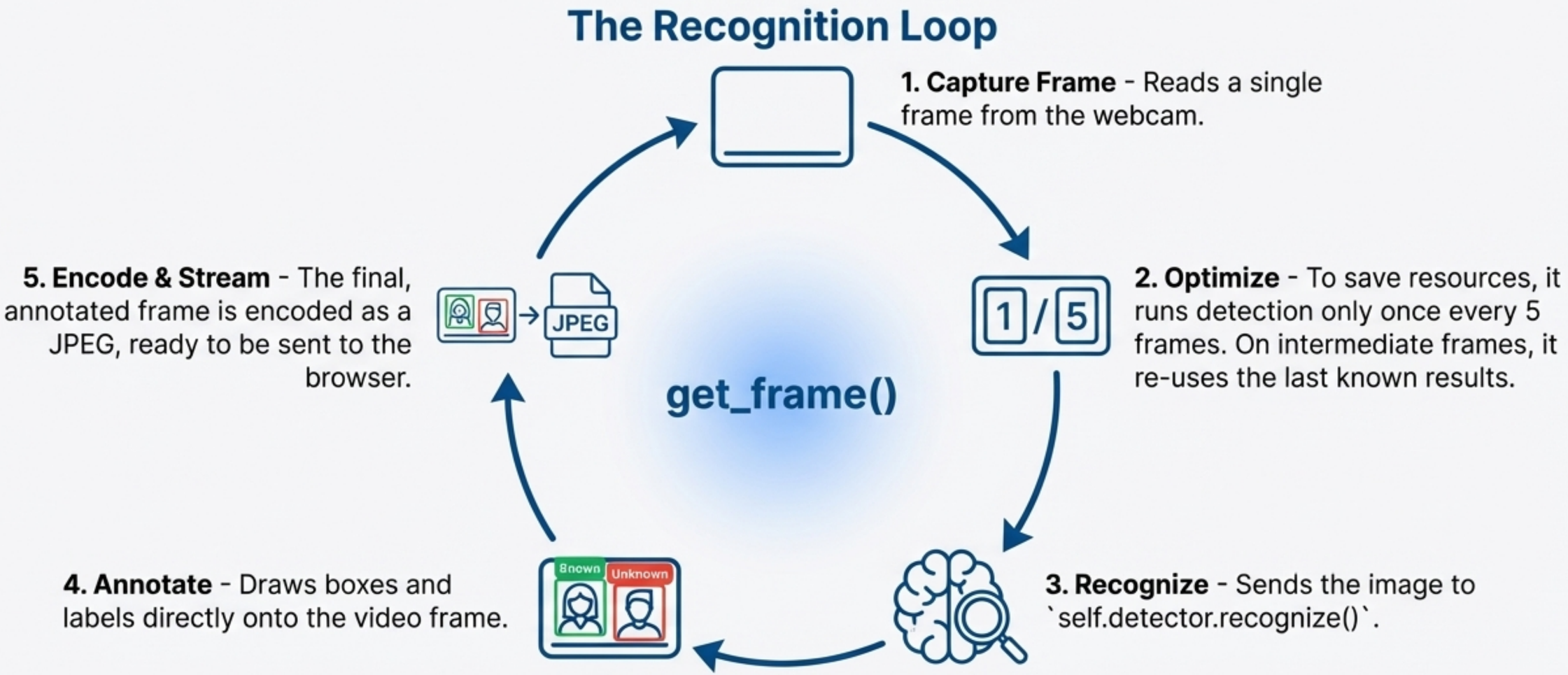
With our model trained, we now build the live system. The `camera.py` script and its `VideoCamera` class are the bridge between our AI and the real world.



This means that before the camera even turns on, our system has loaded its entire memory of known faces and is ready to perform recognition.

Processing the Live Feed, Frame by Frame

The `get_frame()` method in `VideoCamera` is the heart of the live system. It runs in a continuous loop to capture and process video.

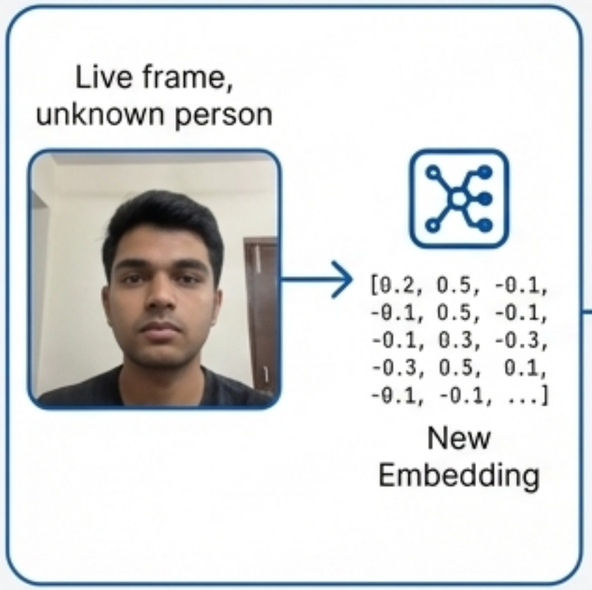


The Moment of Recognition: How a Match is Made

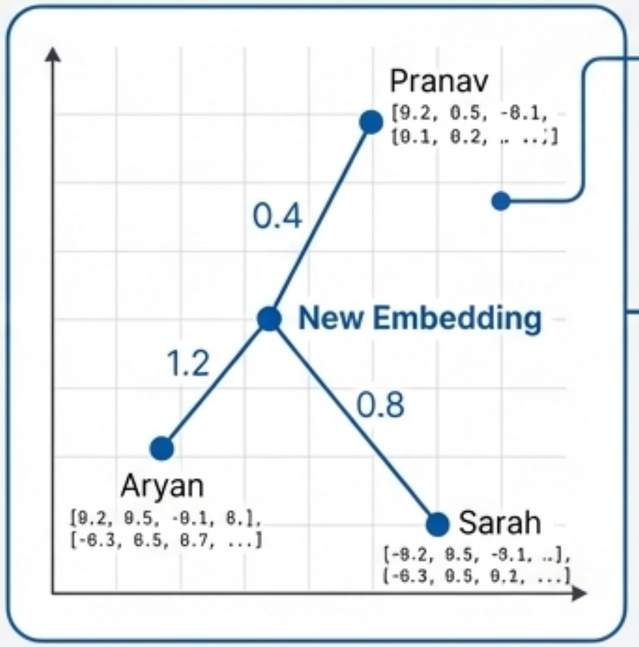
The `recognize()` method in `detector.py` performs the critical comparison logic. Here's how it identifies a person in a live frame.

Step-by-Step Comparison

1. Detect & Embed

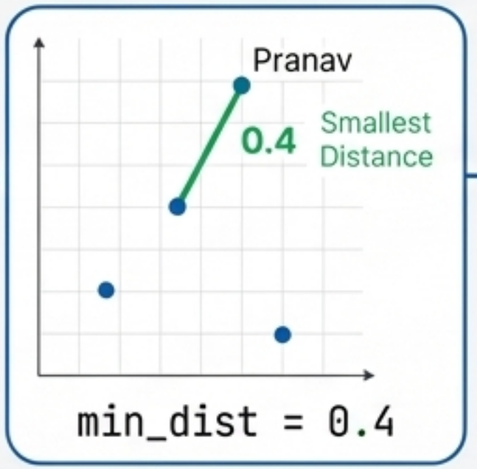


2. Calculate Distance



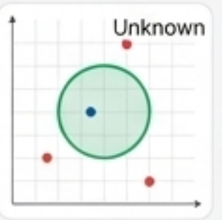
3. Find Closest Match

```
# Calculate Euclidean (L2) distance
distances = (embedding - self.known_embeddings).norm(p=2, dim=1)
```



4. Apply Threshold

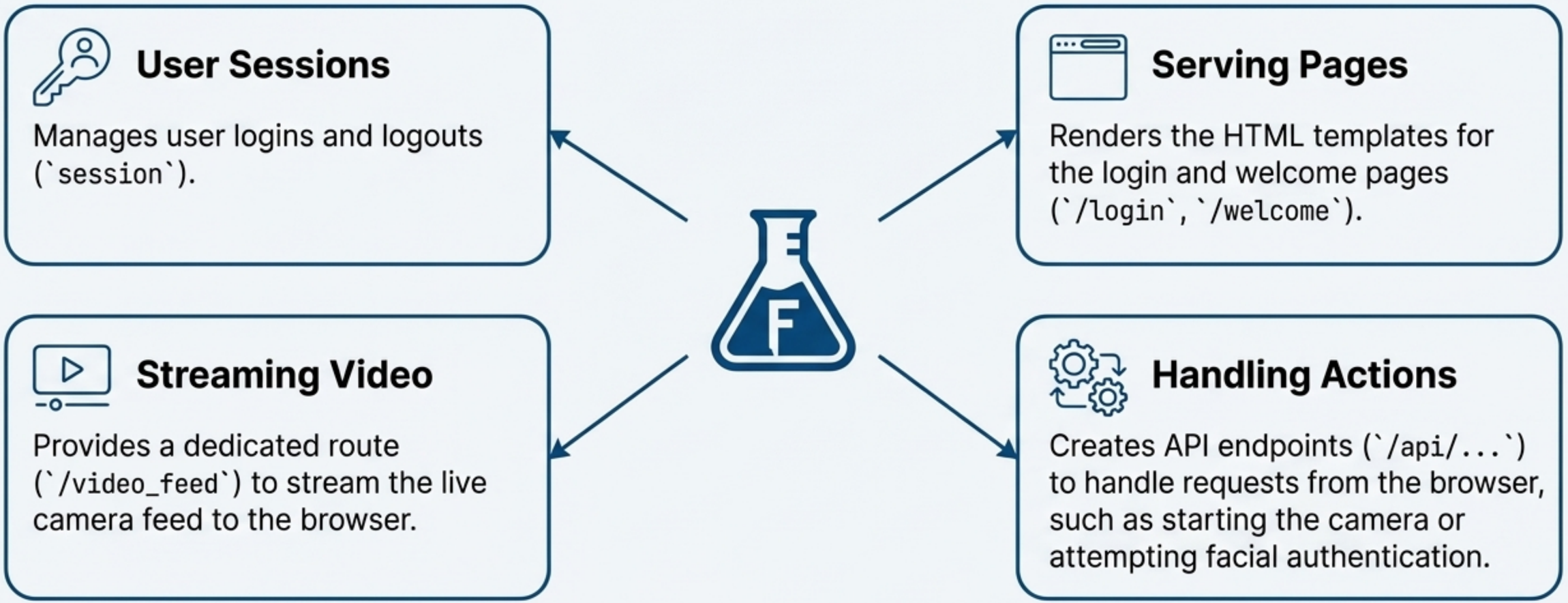
If `min_dist` < `0.6`, it's a match. The name is 'Pranav'.



No match found, Unknown

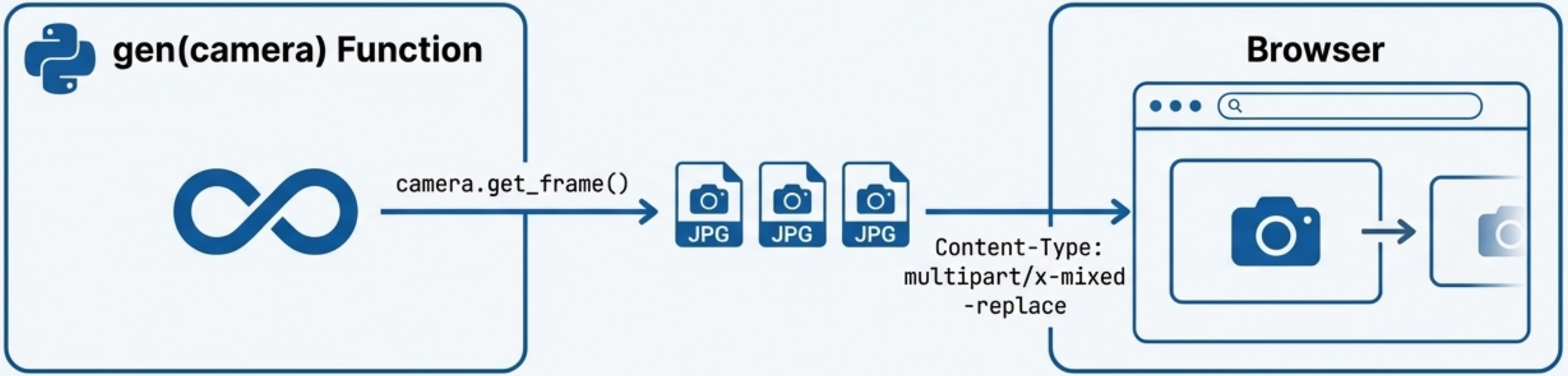
The Control Center: The Flask Web Application

`app.py` is the web server that ties everything together. It uses the Flask framework to create a user interface and API endpoints for our system to interact with.



How We Stream Live Video to a Web Page

How does a **Python script show a live video in a browser?** We use a technique called '**Motion JPEG**' over HTTP.

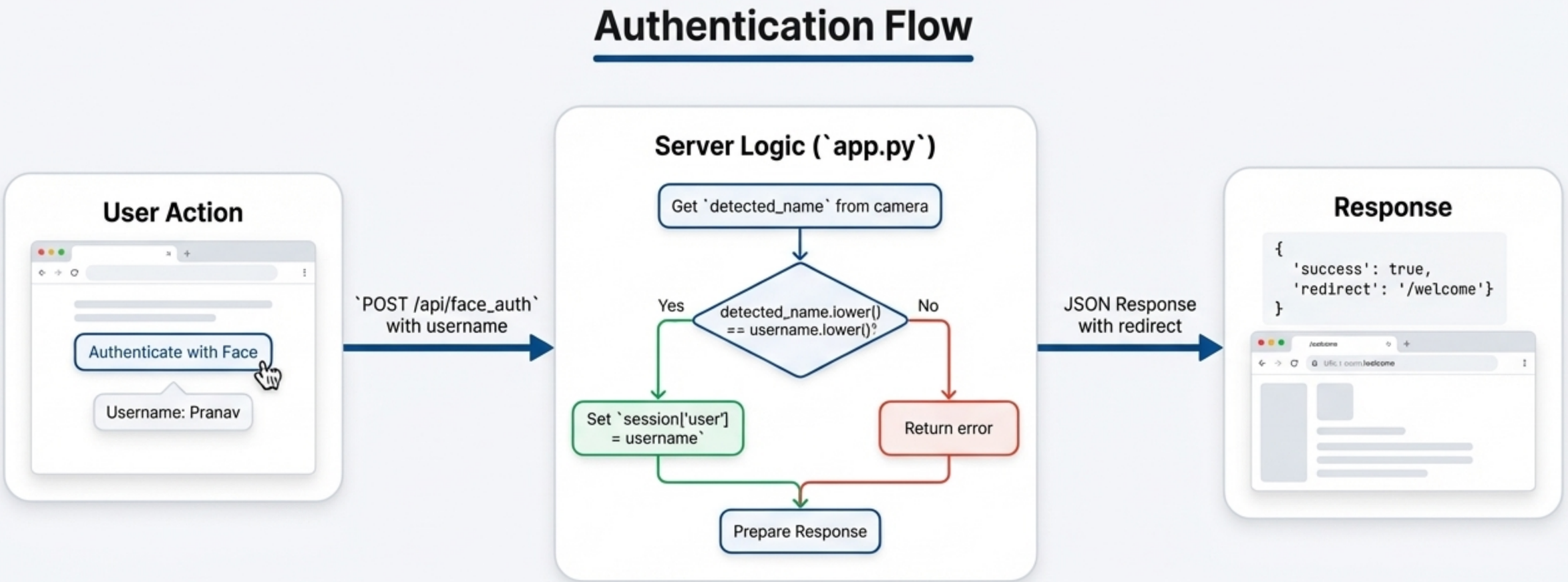


This **Python function** runs in an **infinite** loop. In each loop, it yields the latest processed frame as a block of bytes, which Flask sends to the browser.

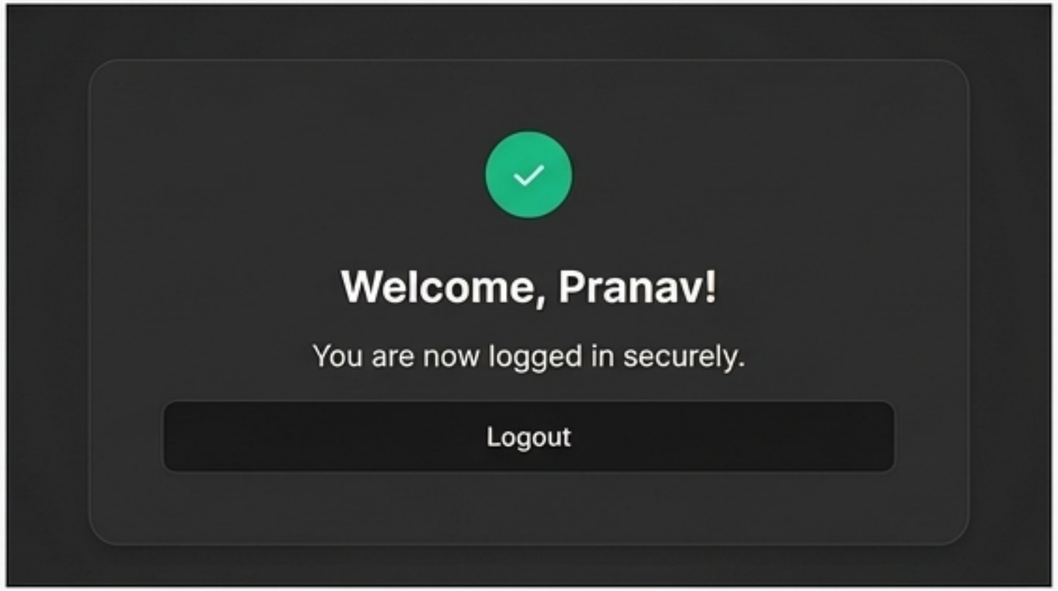
The browser is told: "I'm going to send you a **series of images**. **Replace** the old one **with** the new one as soon as it arrives." The result is a smooth, **live video feed**.

The Final Step: Authenticating with Face ID

This is where the user experience and backend logic meet. The `/api/face_auth` endpoint handles the final verification.



From Data to Detection: Success



After the server successfully verifies the user's face, it redirects them to the welcome screen. The entire system—from data preparation and model training to real-time detection and web integration—has worked together to achieve this simple, secure result.

You have now seen every major component and logical step required to build a complete facial recognition security application.