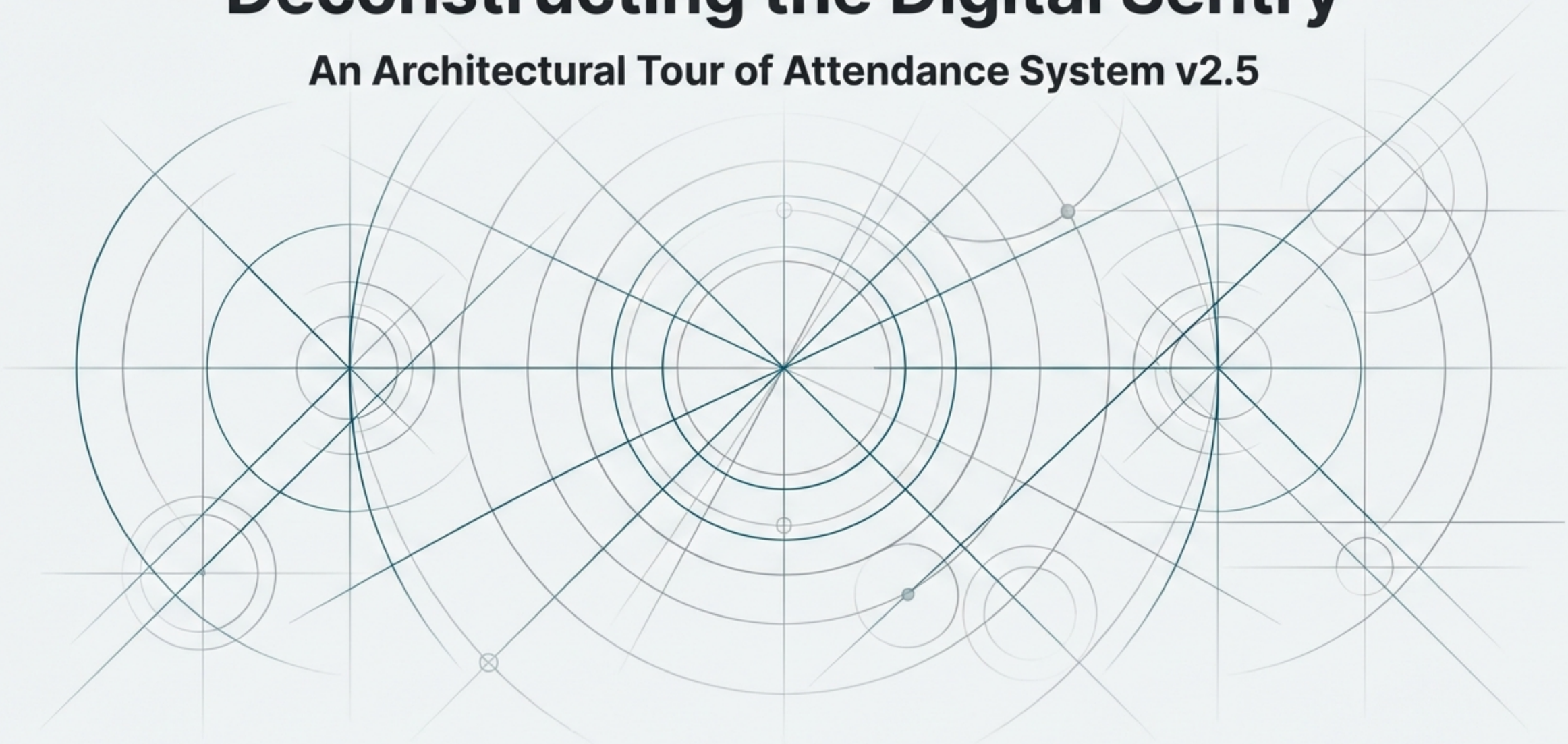
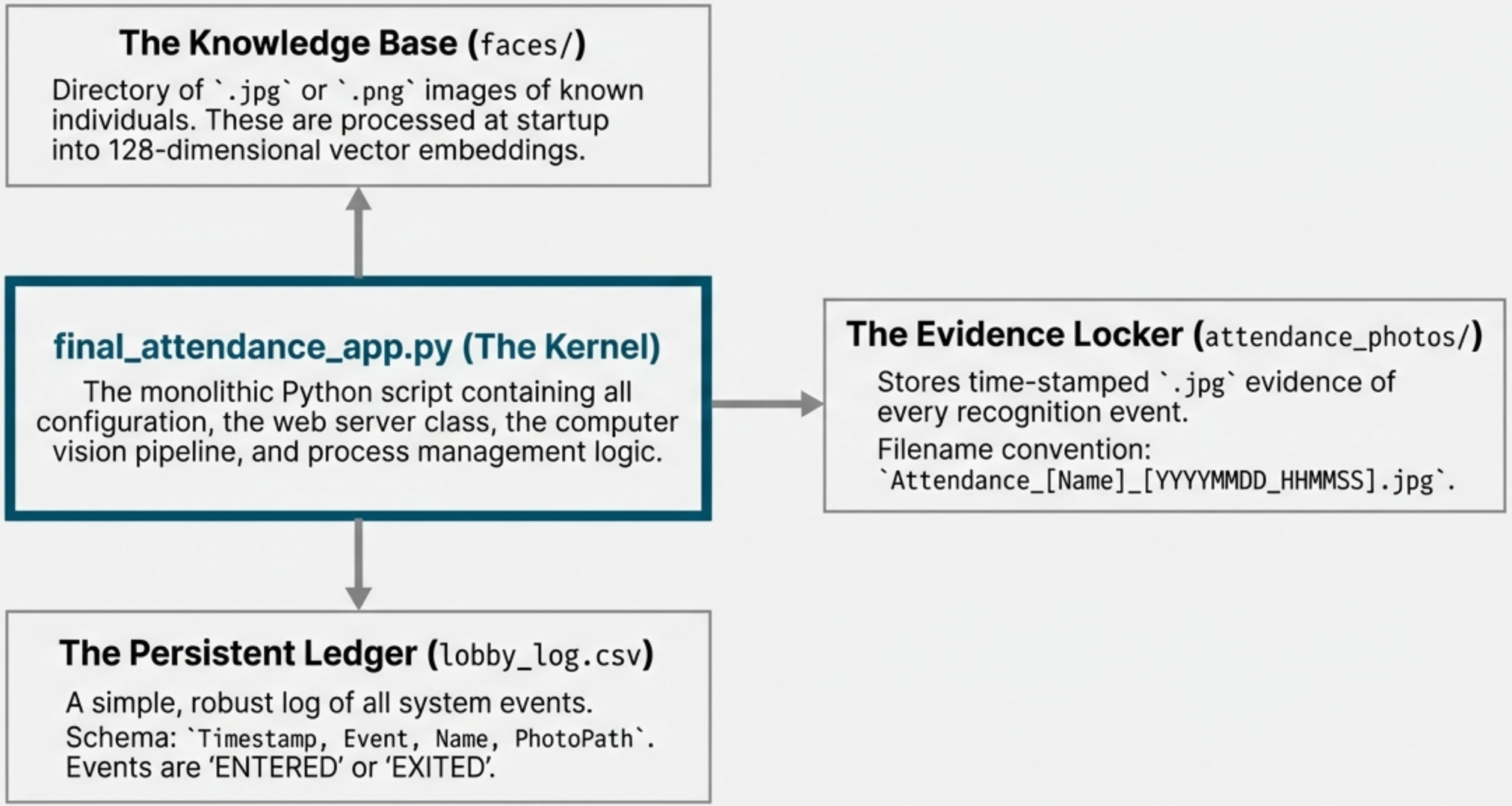
Deconstructing the Digital Sentry

An Architectural Tour of Attendance System v2.5



A “Teeth to Bone” analysis of the system’s design, logic, and core optimizations.

The System Blueprint: Anatomy of a Monolithic Kernel



The System's State: Core Variables in Memory

known_face_encodings (List of Lists of Floats)

The Identity Library

The mathematical representation of every known face. A list of 128-d vectors.

known_face_names (List of Strings)

The Name Roster

A parallel list mapping each vector to a human-readable name. `encodings[i]` corresponds to `names[i]`.

present_people (Dictionary)

The Live Tracker

The system's short-term memory. `Key`: Name, `Value`: Last Seen Timestamp. Prevents log duplication and detects exits.

process_this_frame (Integer)

The CPU Governor

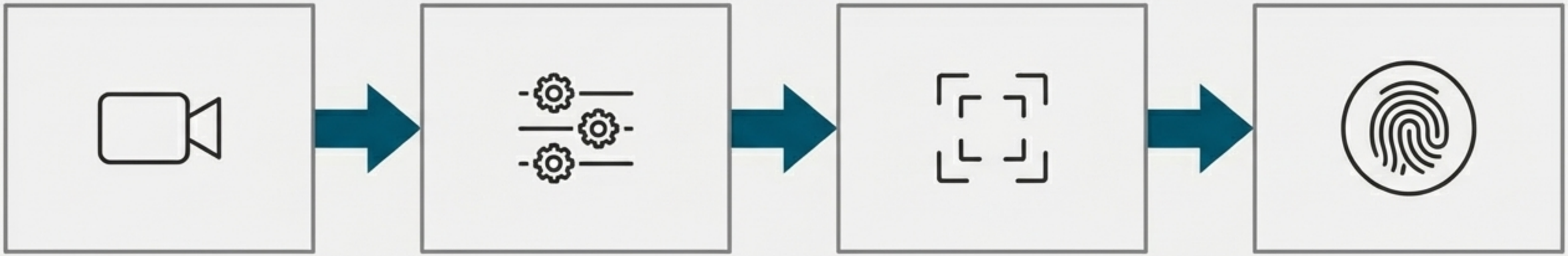
A simple counter used with modulo arithmetic to ensure the computationally expensive recognition logic only runs on select frames.

server_process (multiprocessing.Process)

The Process Handle

A handle to the child process running the Flask web server. Is `None` when the server is off, allowing for state tracking.

Inside the Brain: The Computer Vision Pipeline



Acquisition

Capture raw video frame from the source.

Pre-processing

Resize and convert the frame for optimal performance.

Detection

Locate all potential faces within the frame (‘Where’).

Recognition

Identify the located faces against the knowledge base (‘Who’).

This pipeline runs in a continuous `while True` loop, processing the video stream in real-time.

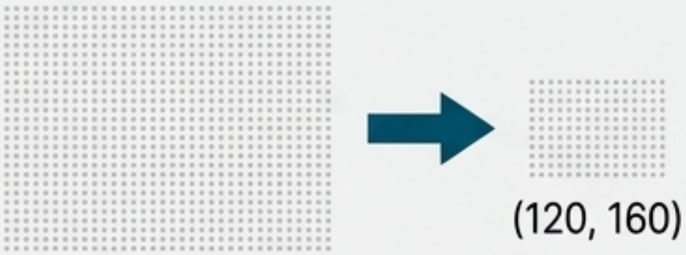
Step 1 & 2: Frame Acquisition and Strategic Optimization

Acquisition: The Raw Data

- Source: `cv2.VideoCapture(0)` (Default Webcam)
- Data Structure: A numpy array of shape `(480, 640, 3)`
- Total Pixels: 307,200

Pre-processing: The Rationale for Speed

Resizing



Code: `cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)`
"Input": (480, 640) -> **"Output"**: (120, 160)
"Pixel Reduction": 307,200 -> 19,200 (A 94% reduction)

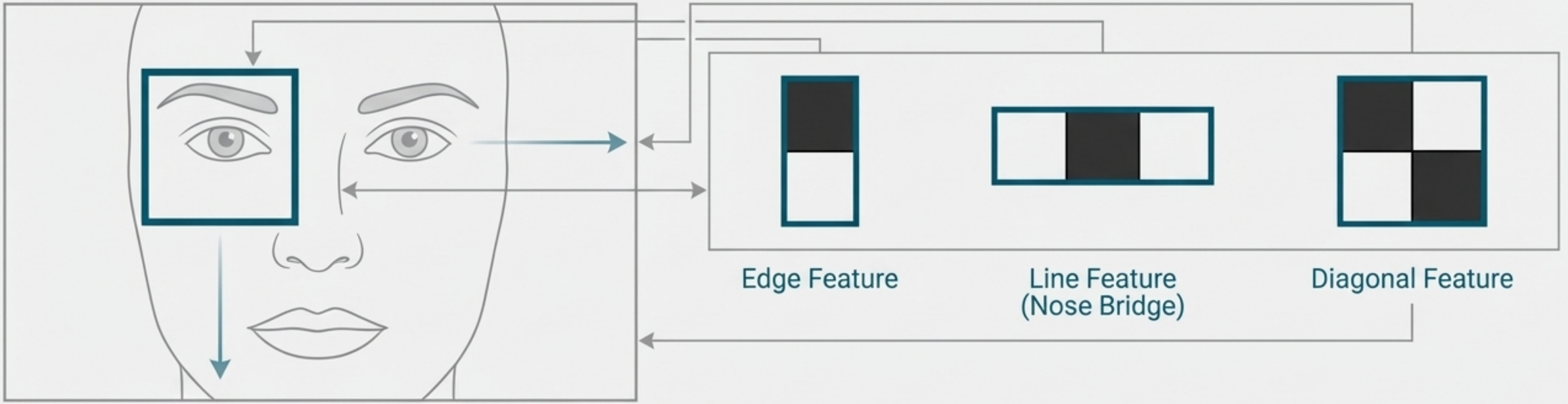
WHY: Reduces data dimensionality before it hits detection algorithms, which have $O(N^2)$ or $O(N \log N)$ complexity. This is the single most important performance tweak.

Color Conversion

The frame is converted to both Grayscale (for Haar detection) and RGB (for dlib recognition) to meet the specific requirements of each library.

Step 3: Detection - Answering “Where is the Face?”

The Viola-Jones Framework with Haar Cascades



Sliding Window at Multiple Scales

Description of Mechanism

1. The algorithm slides a window across the grayscale image at multiple scales.
2. At each position, it looks for simple, pre-defined "Haar-like features" (e.g., a light region surrounded by two dark regions, characteristic of eyes).
3. A cascade of checks rapidly discards non-face regions.

Key Settings: `scaleFactor=1.1`, `minNeighbors=5`

The Output: The result is not an identity, but simply a list of rectangles `[(x, y, w, h)]`, defining the coordinates of every face found in the frame.

Step 4: Recognition - Answering ‘Who is this Person?’

Deep Learning with dlib’s ResNet Model

The Gate

The system conserves resources by only running this heavy process on a subset of frames.

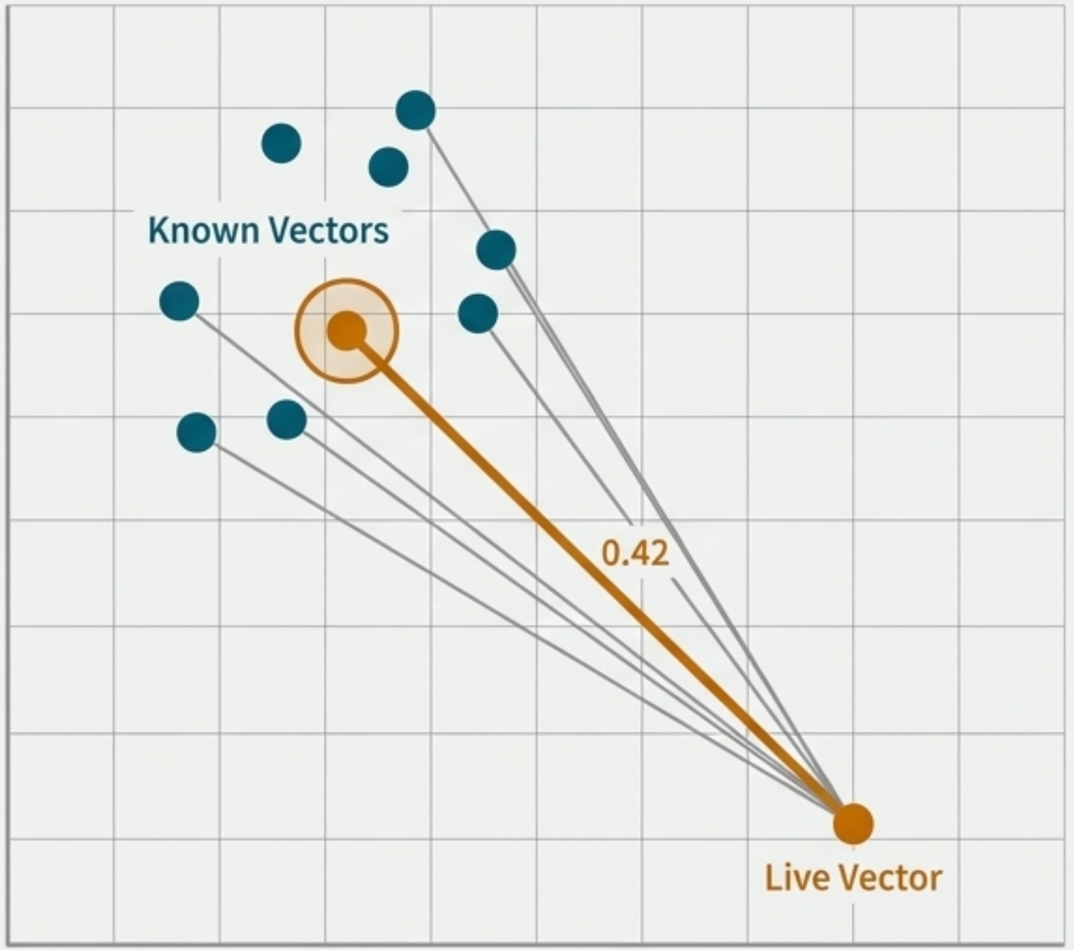
Condition: `process_this_frame % 5 == 0`

Encoding to a Vector

The `160x120` RGB image and face location rectangle are passed to dlib. A Deep Residual Network (ResNet) model processes the face and outputs a 128-dimensional vector (an ‘embedding’). This vector is a unique mathematical signature for that face.

Matching by Distance

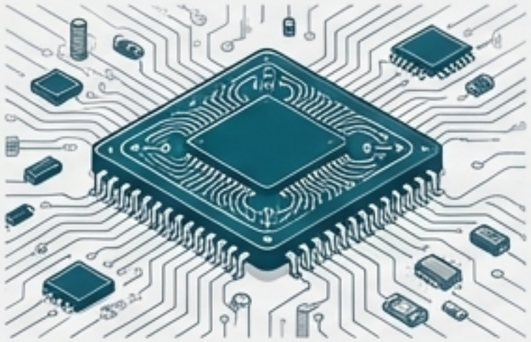
The new vector is compared against all vectors in `known_face_encodings` by calculating the Euclidean distance. The name associated with the vector having the smallest distance is chosen as the match, provided the distance is below the tolerance threshold of 0.6.



Distance < 0.6 indicates a positive match.

A Masterclass in Memory Architecture: Heavy vs. Light Processes

Main Process: The Vision Engine



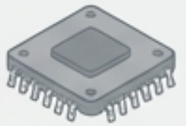
~600MB

Loads:

- `cv2` (OpenCV)
- `numpy`
- `face_recognition` (dlib wrapper)
- `dlib` (C++ Engine)

The dlib ResNet and landmark predictor models account for ~500MB of the total footprint.

Child Process: The Web Interface



~30MB

Loads:

- `flask`
- `socket`
- `threading`

The `import face_recognition` statement is scoped *inside* the main vision loop. Because the web server is spawned as a separate process from a different function, it *never* imports dlib or its models, resulting in a 20x smaller memory footprint.

The On-Demand Interface: Spawning and and Terminating the Web Server

Web Server
~30MB

running

Start

Event: User presses ‘s’ key

```
server_process =
multiprocessing.Process(target=run_flask_app)
server_process.start()
```

The OS clones the main interpreter’s memory space (without the dlib models loaded) and begins executing the `run_flask_app` function. The Flask server starts listening on Port 5000.

Stop

Event: User presses ‘s’ key again

```
server_process.terminate()
```

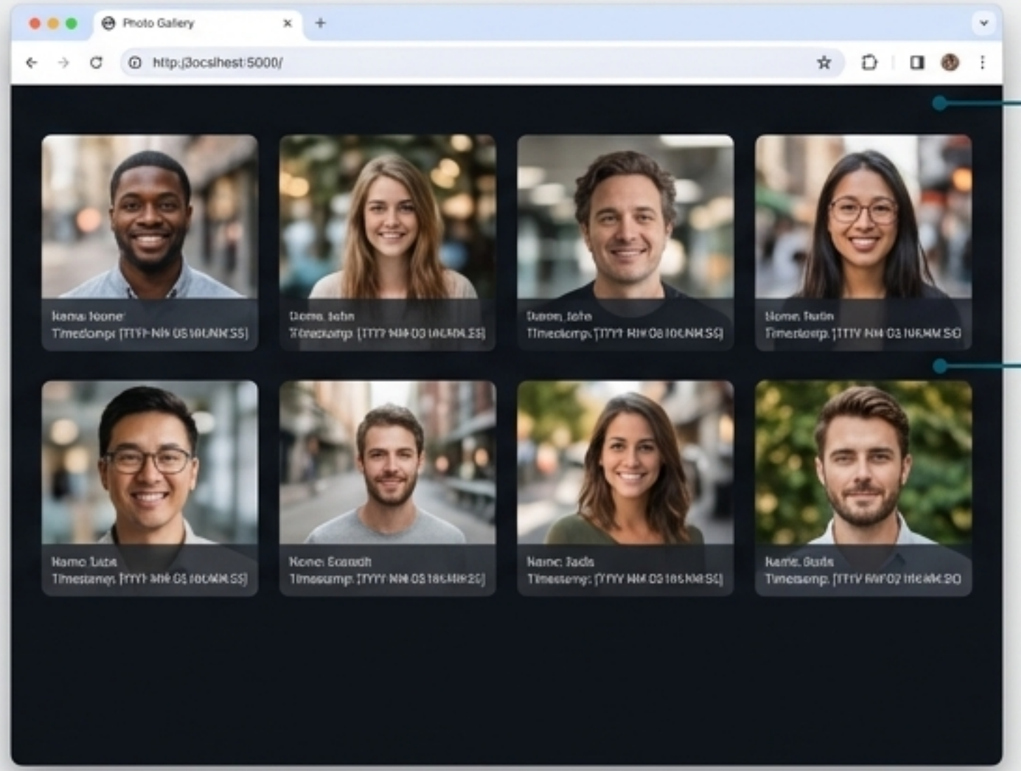
A `SIGTERM` signal is sent to the child process. The OS immediately reclaims all ~30MB of memory used by the server. The `server_process` handle is set back to `None`.

The Public Face: A Simple and Efficient Web Interface

Backend API (Flask)

- `GET /`:
  Serves the main `HTML_TEMPLATE`.
- `GET /api/photos`:
  Scans the `attendance_photos/` directory, sorts images by modification time, and returns a clean JSON array of photo metadata (URL, name, timestamp).
- `GET /photos/<filename>`:
  Acts as a static file server to deliver the actual image assets.

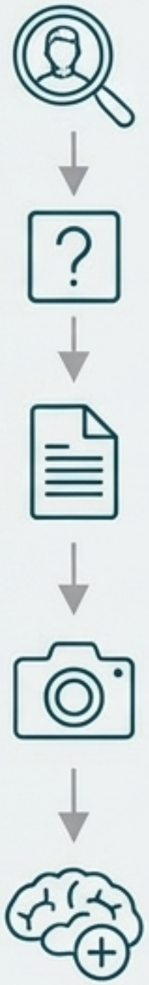
Frontend Logic (HTML/JS)



Uses modern CSS variables for a clean, dark-mode theme (`--bg-color: #0d1117`).

An `async` function, `fetchPhotos()`, polls the API. To prevent unnecessary DOM manipulation, a hash of current filenames is stored. The grid is only re-rendered if the hash changes, simulating a Virtual DOM concept for efficiency.

Logic in Motion: Tracing an "Entry" Event



1. Recognition
A new face is detected and recognized as ‘User X’.

2. State Check
The system checks `if "User X" in present_people`. → **False**

3. Action - Log
A new line is appended to `lobby_log.csv`.
`2023-10-27 10:00:00,ENTERED,User X,photos/Attendance_UserX_...jpg`

4. Action - Evidence
The current video frame is copied. The timestamp and name are burned onto the image using `cv2.putText`. The result is saved to `attendance_photos/` via `cv2.imwrite`.

5. State Update
The system updates its short-term memory.
`present_people["User X"] = <Current Unix Timestamp>`

Logic in Motion: Tracing an ‘Exit’ Event

1. Absence

For several consecutive frames, no faces are detected in the video stream.

2. State Iteration

The system loops through every name currently in the `present_people` dictionary.

3. Timestamp Check

For 'User X', it calculates the elapsed time.

`Current Time - present_people["User X"]`

Is the result **> 3.0** seconds? → **True**

4. Action - Log

A new line is appended to `lobby_log.csv`.

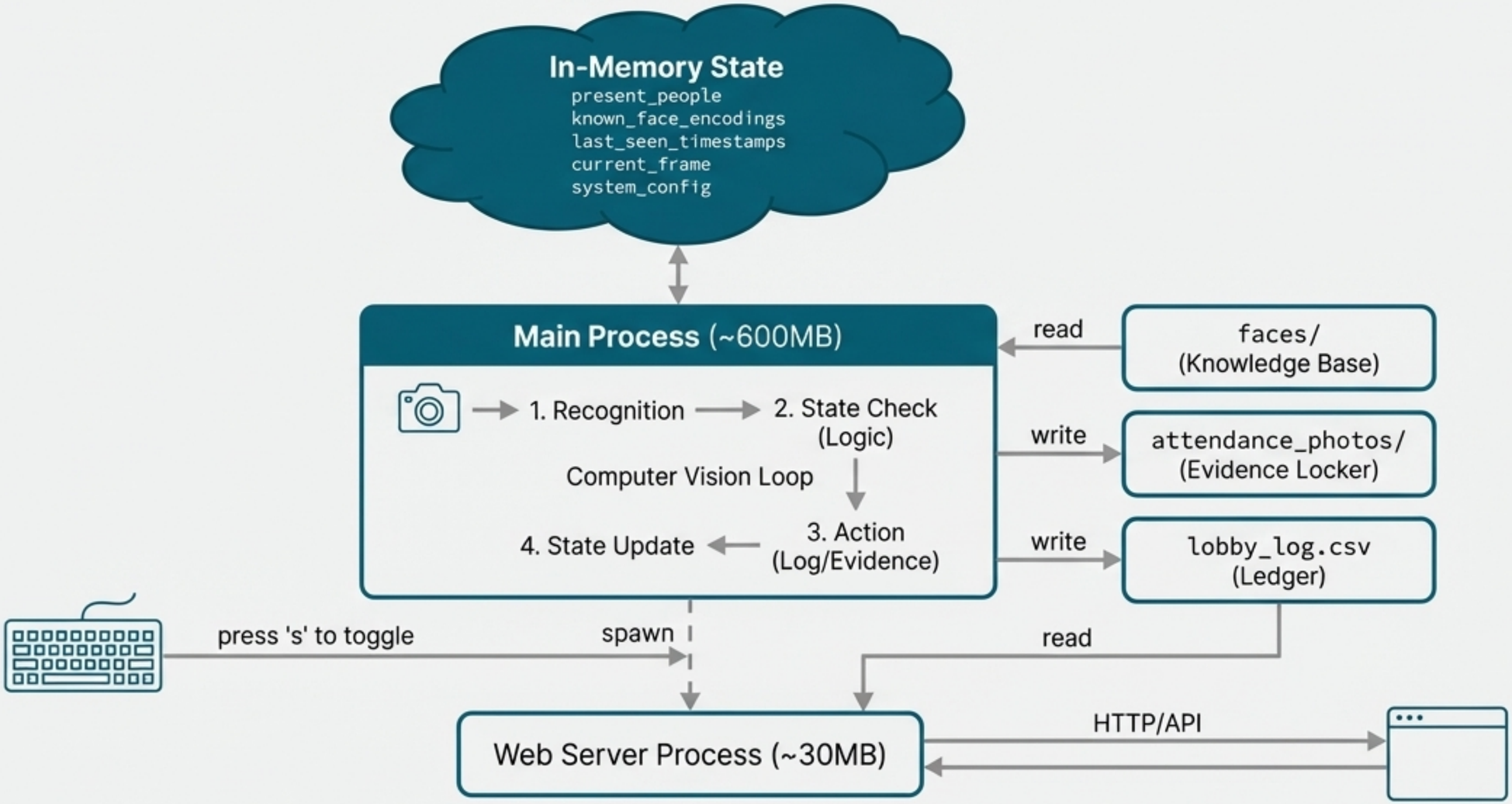
`2023-10-27 10:05:00,EXITED,User X`

5. State Update

The user is removed from short-term memory to allow for a new 'Entry' event later.

`del present_people["User X"]`

The Complete System Architecture



Attendance System v2.5 demonstrates a synthesis of efficient real-time processing, intelligent resource management, and stateful logic, creating a robust and performant monitoring tool from foundational components.