ChatGPT

Designing a Doorbell-Triggered Face Recognition Home Security System

Planning the Project

Define Objectives: The goal is to build a low-cost home security system that uses a smart doorbell camera for facial recognition. When someone presses the doorbell, the system should capture their image, recognize if they are a known (authorized) person or not, and then respond accordingly. For unfamiliar faces, it should alert the homeowner and even cross-check against a database of wanted criminals (e.g. FBI's most-wanted list). Key requirements include:

- **Face Recognition on Doorbell Press:** The doorbell button triggers a camera to capture footage of the visitor. The system detects faces in the image and compares them against a **whitelist** of known individuals (e.g. family, friends) and a **blacklist** of flagged individuals (e.g. known threats or wanted persons) [1] .
- **Unknown Person Handling:** If the visitor's face is not recognized (not in whitelist), the system should treat them as unknown. This might include sending a real-time alert with the visitor's photo to the homeowner's phone, so they can decide how to respond [2] . It should also log the event for later review.
- **"Most Wanted" Cross-Check:** If the visitor's face happens to match someone in a wanted criminals database (our blacklist), the system should raise an alarm or a high-priority alert. For example, it could display a red warning light on the device and send an urgent notification [2] . (In practice, false matches are possible, so this would be a cautious alert for the homeowner's awareness.)
- **No Press, No Capture (to save resources):** The camera will activate **only when the doorbell is pressed**, rather than continuously monitoring, to conserve power and processing. (Optionally, a motion sensor could be added to capture intruders who don't ring the bell, but that increases complexity.)
- **Local Processing & Privacy:** Plan to perform face recognition **locally** on-device for privacy and cost control. This avoids sending video to cloud services continuously. A local system ensures visitor images remain private and the system can work even without internet [3] . Cloud checks (like querying the FBI database) can be limited to text/image data as needed.

Feasibility and Constraints: We target a **low-cost implementation** using readily available hardware (e.g. a Raspberry Pi with camera) and open-source software. The system should run 24/7, so it needs to be power-efficient and reliable. We must consider the processing capability needed for face recognition – Raspberry Pi's have limited CPU/GPU, so we'll choose algorithms and optimizations accordingly. The plan also has to account for environment conditions (lighting at the door, weather-proofing the camera, etc.). During planning, we identify potential challenges: ensuring face recognition accuracy in varied lighting, minimizing false positives/negatives, and maintaining a frequently updated criminal database. These will guide our design choices and testing focus.

Outline of Solution Approach:

- Use a **Raspberry Pi 4** as the core of the system (acts as an edge device). RPi4 strikes a good balance between cost and performance. Earlier Pi models can work but are slower for image processing (Pi 3 can do basic face rec but struggles <2 FPS, whereas Pi 4 can achieve ~5–8 FPS with optimizations [4]). A Pi 4 (2GB or 4GB RAM) is recommended for smooth performance. Alternatives like NVIDIA Jetson Nano offer more AI performance but cost more (~2–3× the price) [5], so we'll stick with Raspberry Pi for cost-efficiency.
- Attach a **camera** to capture visitor images. The Raspberry Pi Camera Module (v2 or v3) is ideal – it plugs into the Pi's CSI port and provides decent resolution. (Camera Module v1 is cheaper, but the v2 offers better image quality and longer support [6], worth the slight extra cost.) A standard USB webcam is an alternative if you have one handy, but Pi's own camera tends to integrate more reliably and can use the Pi's GPU for capture.
- Connect a **doorbell button** to the Pi's GPIO. If replacing or augmenting an existing doorbell, you can wire the Pi to detect the existing doorbell circuit (using a relay or optocoupler for isolation). Simpler, you can use a new low-voltage button (or even an arcade button as in some DIY projects) wired directly to a GPIO pin on the Pi [7]. When the button is pressed, the Pi will register a falling/rising signal and trigger the camera capture routine. (We will use the Pi's internal pull-up/down resistors to stabilize the input and avoid false triggers.) The existing chime can remain in parallel so the bell still rings as usual, if desired.
- Maintain two lists of faces: a **Whitelist (authorized)** and a **Blacklist (wanted/suspicious)**. The whitelist will be populated with the homeowners' and frequent visitors' faces. The blacklist will include known threat faces – we will leverage publicly available images of criminals (for example, the FBI's "Most Wanted" list) to populate this. The FBI even provides an API to fetch data on wanted individuals [8], which we can use to periodically update our blacklist dataset. This way, the system stays up-to-date with any new high-profile criminals.
- Set up a **notification method** for alerts. For low-cost and ease, using a Telegram bot is an excellent choice. Telegram allows sending messages and photos via a simple API, and it's free. We'll have the Pi send a Telegram message with the visitor's snapshot whenever someone unknown or blacklisted appears. (The DIY doorbell project by Erientes uses `python-telegram-bot` to notify a caregiver with a photo in real-time [2] [9], which aligns with our needs.) Alternatives like email (slower) or SMS (cost per message) were considered, but Telegram strikes a good balance of immediacy and zero recurring cost. The homeowner just needs the Telegram app on their phone.

Technology Considerations: During planning, we compare a **cloud-based** recognition approach vs. an **edge-based** approach. A cloud service (like AWS Rekognition, Azure Face API, etc.) could handle the face recognition and even criminal matching for us, but it comes with latency, privacy concerns, and potential costs per image. For example, cloud APIs are billed per request (on the order of \$1 per 1000 images, i.e. ~$0.001 each) [10] – seemingly cheap, but if our camera triggers often, costs add up over time. Moreover, continuous video would consume bandwidth (HD video streaming can be gigabytes per day, and cloud providers charge for data transfer) [11]. Edge processing, on the other hand, has a higher upfront cost (buying the hardware) but virtually no per-use cost and minimal latency (decisions can be made on-site in milliseconds to a couple seconds) [12] [13]. Given we want *low ongoing cost and independence*, we choose an edge solution: the Raspberry Pi will do the heavy lifting locally. This also keeps sensitive images within our home network [3]. The cloud will be used only for optional updates (like fetching FBI data or sending notifications), not for processing video.

With the objectives clear and initial decisions made, we can proceed to detailed design, knowing our plan is feasible with off-the-shelf components and open-source software. Next, we'll design the system architecture and choose the specific tech stack for each component, always weighing pros and cons to ensure we meet the low-cost and reliability goals.

Designing the System Architecture

Overall Architecture: The system can be visualized in two parts – **hardware setup** and **software workflow** – working together. When the doorbell is pressed, the hardware triggers the software pipeline: capture image -> detect/recognize face -> decide response -> act (notify/log/etc.). Below we break down the design for each aspect:

Hardware Design Choices

- **Computing Unit:** We select the **Raspberry Pi 4** as the brain of the system. It's a single-board computer that is low-cost and capable of running Linux and computer vision libraries. A Pi 4 (4GB RAM) costs roughly \$50 and provides significantly better performance than its predecessors for AI tasks [4]. In fact, tests show a Pi 3 can struggle to process even 2 frames per second for face recognition, whereas a Pi 4 can reach ~5–8 FPS with proper optimizations [4]. This difference is crucial for reducing the delay between a doorbell press and identification. An alternative was the **NVIDIA Jetson Nano**, which has a GPU for AI and would speed up deep learning-based recognition. The Jetson would likely handle face recognition faster (perhaps ~1–2 FPS vs ~0.5 FPS for Pi when using deep learning) and is recommended if high throughput is needed [5]. However, the Jetson's cost (~\$100) and power consumption are higher, and it complicates the setup. Since our use-case (doorbell) is intermittent – not a constant video stream – the Raspberry Pi's performance is acceptable. It might take ~1-2 seconds to identify a face, which is reasonable for someone waiting at the door. Thus, for cost-efficiency and simplicity, **Raspberry Pi 4** is our choice. (If one already has an old laptop or mini-PC, that could also be used as the server, but then power usage might increase. The Pi is designed for 24/7 operation at low power.)

- **Camera Module:** The camera is critical for capturing clear images of visitors. We will use the **Raspberry Pi Camera Module v2** (8MP sensor) or v3 (12MP with HDR, if available) for this project. The v2 module offers a good balance of price (~\$25) and quality. It connects directly to the Pi's CSI interface, which has the advantage of fast data transfer and low CPU overhead (the Pi's GPU can handle camera data). The camera can be configured for 1080p or 720p images; we might capture at 720p to reduce processing load while still getting sufficient detail for recognition. As noted, the original 5MP camera (v1) is slightly cheaper but has lower resolution and is end-of-life [6], so v2 is preferred for longevity. We also consider low-light conditions: if the doorway is dim or used at night, we may add an **IR illumination** or use the NoIR (infrared-sensitive) version of the Pi camera plus IR LEDs for night vision. Cons: the Pi Camera requires proper mounting and focusing (it's a bare module), whereas a USB webcam often comes in an enclosure. But the benefit of the Pi cam is proven software support and full control of settings. Alternative: a USB webcam could be used (some are as cheap as \$10–15). The trade-off is that OpenCV can interface with a USB camera easily, but you lose some of the Pi-specific optimizations. Also, cheap webcams might have poorer image quality or low FPS. Overall, the **official Pi camera** is the robust choice here for clear images and integration.

- **Doorbell Button & Wiring:** We need a mechanism to know when the doorbell is pressed. In a DIY setup, a simple **momentary push-button** can serve as the doorbell. This button will be connected to a GPIO pin on the Pi and to ground, configured with an internal pull-up resistor so that the Pi can detect a clean high/low signal on press. When designing the circuit, we'll **debounce** the input in software (or hardware with a capacitor) to avoid false multiple triggers. If the home already has a wired doorbell (which typically runs on a low voltage AC transformer, e.g. 12VAC), we have two sub-options:

    - **Integrate with existing bell:** Use a relay or optocoupler to detect the voltage when the bell is pressed. The Pi can read the relay's output (acting like a button). This way, the old chime still rings and the Pi also knows about the press. The downside is minor electrical work and ensuring the Pi's ground is properly referenced.
    - **Replace with smart button:** Use a custom button connected only to the Pi (and perhaps use a buzzer or speaker for a chime sound). This is simpler electronically, though you lose the redundancy of the analog chime.
      In our design, for simplicity and cost, we can use a large weatherproof button by the door wired to the Pi. In Erientes' project, they used arcade-style LED buttons and even built a traffic-light style indicator panel for feedback [1], but those are optional. We will plan a basic LED indicator (like a bi-color LED) to show status (e.g. green for recognized, red for alert) which costs pennies and can help the user (or visitor) get feedback. However, to keep costs minimal, this indicator isn't strictly required; the primary feedback/alert will be on the homeowner's phone.

- **Power Supply:** The Raspberry Pi 4 needs a stable 5V ~3A supply. We'll use a standard Raspberry Pi power adapter (around \$10) to ensure sufficient current, especially if the Pi will also drive an IR LED or other peripherals. Low-cost tip: If integrating with a traditional doorbell's power, you might use a DC-DC converter from the doorbell transformer to 5V, but it's often easier to just use the proper adapter. Since the system runs continuously, a small UPS (uninterruptible power supply) HAT or a battery backup could be considered to handle power outages – but that adds cost, so perhaps not in the initial build. Instead, we'll at least handle sudden power loss by using a robust file system or periodically syncing critical data to avoid corruption (discussed in Maintenance).

- **Networking:** The system needs network access (for Telegram alerts and FBI data updates). The Pi has Wi-Fi and Ethernet. Wi-Fi is convenient if the device is near the door (ensure the signal is strong at that location). If the Pi is indoors and just the camera is at the door (with a long ribbon cable or using a Pi Zero with camera as a satellite), Wi-Fi will be fine. We should secure the network connection (use WPA2, possibly put the Pi on a separate IoT VLAN if available, etc.). No additional cost here as Pi's built-in networking will suffice.

Hardware Summary: Raspberry Pi 4 + Camera + Button + Power. This can all be achieved for on the order of \$70 or less in parts. We intentionally avoid expensive extras. The design leverages the Pi's capabilities to handle both the sensing (camera input, button GPIO) and processing (face recognition) on one device. This edge device approach ensures minimal latency (no waiting for cloud) and zero cloud computing costs aside from using the internet for notifications [14] [10]. Now that hardware choices are made, we move to the software design.

Software Design and Tech Stack

- **Operating System & Environment:** The Raspberry Pi will run **Raspberry Pi OS (Raspbian)**, a Debian-based Linux optimized for the Pi. It's free and comes with all tools needed. We'll use Python 3 as the main programming language, since Python has excellent libraries for computer vision and hardware interfacing on the Pi. The Pi OS already includes Python and can easily install libraries via pip or apt. Python is chosen for quick development and ample community examples (many Pi facial recognition projects are Python-based [15]). An alternative could be C++ (using OpenCV's C++ API) for slightly better performance, but development would be slower and more complex. Given our performance requirements are moderate (infrequent events, a second or two of processing), Python is more than sufficient and hugely speeds up development with its rich ecosystem.

- **Face Detection & Recognition Algorithm:** This is the core of the system. We have a few approaches to choose from:

- **OpenCV's Haar Cascade + LBPH Recognizer:** OpenCV comes with a pre-trained Haar Cascade classifier for face detection (fast on CPU) and provides a local binary pattern histogram (LBPH) based face recognition algorithm. This classic approach involves first detecting the face region, then computing a histogram-based "fingerprint" of the face and comparing it to a trained database of known faces. Pros: it's lightweight and works on Pi; LBPH models are very small (often <1 MB per person trained) [16]; training can be done on the Pi itself. Cons: accuracy is lower than modern deep learning methods – in controlled conditions you might get 85–95% accuracy [17], but under varying lighting/angles it can drop (one study reported ~75% accuracy with some errors) [18]. Also, maintaining the model means whenever you add a new person, you retrain or update the model. OpenCV's face recognizer (Eigenfaces/Fisherfaces/LBPH) was commonly used in earlier Raspberry Pi projects [19]. For example, TheEngineeringProjects tutorial trains an OpenCV LBPH model (outputting a `face-trainner.yml` file) and then uses it to identify faces and even "utter" the person's name via speaker [19]. This shows it's feasible to use LBPH for a Pi-based system, including real-time feedback. However, given the availability of more accurate methods now, LBPH might serve as a backup plan if resource usage became an issue.
- **Dlib Deep Learning Face Recognition (via `face_recognition` library):** This method uses a pre-trained deep convolutional neural network to generate a 128-dimensional embedding for each face, then compares these embeddings. The Python `face_recognition` library (by Adam Geitgey) wraps dlib's state-of-the-art face recognition model, which achieves ~99.38% accuracy on the Labeled Faces in the Wild benchmark [20] – essentially as good as commercial systems. Pros: extremely high accuracy and robustness to variations in lighting and pose; no explicit "training" needed per person – you just supply one or a few images of a person, compute their encodings, and store them. Recognition is then just a matter of computing an embedding for a new face and checking distance against known embeddings. It's very convenient and proven (many hobby projects use this on Pi). Cons: it is heavier computationally. The face detection step can use either Histogram of Oriented Gradients (HOG) or a CNN; on CPU, HOG is faster and will be our choice. Even then, processing a single frame with detection+embedding might take on the order of 1-2 seconds on a Pi 4 (approx 0.5–1 frame per second) [21]. This is acceptable for a doorbell snapshot scenario. Installation of dlib can be tricky (needs compilation), but there are guides and even pre-built Docker images for Pi to simplify this [15] [22]. The instructables project used this `face_recognition` library on a Raspberry Pi 3 and managed the performance by capturing only until a face is found (not continuous video) [23]. Given the accuracy benefit (99% vs ~85% for LBPH) and the intermittent nature of use, **we**

choose the dlib-based `face_recognition` **approach**. We will optimize as needed (e.g., tune image resolution or use HOG detector) to ensure the Pi can handle it. It provides us with confidence that if the person is indeed someone in our database, it will recognize them, and false matches are very rare when using an appropriate distance threshold.

- **Cloud-Based Recognition:** e.g. using **AWS Rekognition** or **Google Vision API** to identify faces. While we decided on local processing, let's briefly consider this alternative in design. Amazon Rekognition can compare faces against a collection stored in AWS. We could, for instance, store our known faces and FBI most-wanted faces in the cloud collection and on each doorbell press, send the photo to AWS for analysis. The advantages are that AWS has highly optimized models and can even return attributes (smile, eyes open, etc.) and likely very accurate matching. However, the **cost** and **dependency** are downsides. AWS Rekognition charges after a generous free tier – for example, beyond the first 1,000 images per month (free), it's ~$0.001 per image for face detection/analysis, and additional for face search in a collection [24]. If your doorbell triggers 10 times a day (~300/month), that's negligible cost, but if you start doing continuous monitoring or get false triggers, it could grow. More importantly, every image of your visitors would be sent to AWS, which may be a privacy trade-off some homeowners don't want. And if your internet is out or AWS has an outage, your system fails to recognize faces. **Considering our low-cost, self-sufficient goal, we opt not to rely on cloud recognition.** We note it as an alternative if one needed a quick solution without worrying about Pi's performance (just pay per use), but in the long run local processing is more cost-effective and private for a home system [13] [3].

In summary, **our chosen stack** for vision is: OpenCV (for image handling) + Dlib via `face_recognition` (for face detect & embed) + possibly OpenCV's own detection as backup. The Python packages we'll need include: `face_recognition`, `dlib` (if not bundled), `opencv-python` (cv2), `numpy`, and `Pillow` (for image formats) [15]. The instructable confirms these packages are used together on Pi. Additionally, we will use `RPi.GPIO` to interface with the button hardware and `python-telegram-bot` for notifications [25]. This tech stack is completely free and open-source. The only complexity is installing dlib on the Pi, but we have guides (the instructable even provided a Docker container to avoid local install issues [22]).

- **Recognition Database Design:** We'll manage two sets of known face data:
- **Whitelist (Authorized Faces):** This could be as simple as a folder of images (one folder per person or filenames labeled with person's name). During initialization, the software will load each known image, detect the face, compute the 128D encoding, and store it in an array along with a label (the person's name or ID). Alternatively, we can store known encodings in a file (e.g., a Python pickle or JSON) for faster startup. For simplicity, we might start with loading from images (so updating the whitelist is as easy as dropping a new photo into a directory and restarting or running an update script). The number of known individuals for a home is typically small (maybe 5–10 people), so performance is not an issue – comparing an unknown face to 10 known encodings is instantaneous. The system can even speak out the recognized name or light a green LED when a known person is recognized, to give feedback. (In one tutorial, they had the Pi say the name using text-to-speech [26], which is a fun extension we might consider later.)
- **Blacklist (Watch Faces):** This is the collection of suspect or wanted faces. We'll populate this initially with a set of mugshots or photos of criminals – for example, the FBI's **Ten Most Wanted Fugitives**. The FBI Wanted API provides JSON data including image URLs for wanted persons [8], which we can use to automatically fetch the latest images. We'll download a selection (perhaps top 10 or top 100 wanted persons) and compute encodings for each. These encodings are stored similarly in a list with labels (maybe the person's name or an ID). The number here could be larger, but still on the order of tens or hundreds – which is manageable. Face matching in our system involves computing distances

between the visitor's encoding and each encoding in the list; a few hundred comparisons of 128-D vectors is trivial for the Pi. For manageability, we might store blacklist encodings in a separate CSV file (`blacklist.csv`) as done in the reference project [27]. The system will treat any match to this list as a **blacklisted person detected**. (One caveat: false positives can occur if someone happens to look similar to a wanted person. The threshold for matching can be set stricter for blacklist to reduce this risk. For instance, the default `face_recognition` tolerance is 0.6 – we might use 0.5 for blacklist matches to be safe.) We'll also log these events extensively in case law enforcement needs records, though our system is not an official security device, just a helpful tool.

- **Storage and Updates:** Both lists (whitelist and blacklist) can be updated over time. The design should include an **easy way to add a new known person**. One user-friendly method (demonstrated by the instructable) is to have a special mode: press a certain button (like a learn button) and the system will capture a face and add to whitelist on the fly [28]. For now, we might choose a simpler approach: a script or small web interface where the owner can upload a new face photo and label to the Pi. Given our focus is on design rather than UI, we'll assume the developer (user) can add files via SSH or a network share. For blacklist updates, we plan to periodically pull new wanted-person photos. The instructable's design automates this by scanning a folder `img/blacklist` every hour for any new images added and then encoding them [29] – we can implement a similar scheduled task. Or more directly, a cron job could run a Python script to call the FBI API weekly and update images/encodings. This keeps maintenance low-effort (more on maintenance later).

- **Workflow Logic:** Now we detail how the software will operate step by step when the doorbell is pressed:

- **Idle Monitoring:** The main program runs a loop (or uses an interrupt) waiting for the doorbell GPIO signal. It also could handle other periodic tasks (like timed blacklist updates) in the background. In idle state, perhaps an LED glows steady to show the system is on and ready.

- **Doorbell Press Detected:** When the button press is detected, the system **activates the camera**. We will capture images in a quick loop. For example, take a photo, check for a face; if no face is found (maybe the person stood too far or wasn't looking yet), take another after a fraction of a second, up to a few attempts. This is exactly what the instructable system does: it captures images until it "shoots a photo in which a face is detected" [23]. This ensures we actually have the visitor's face before proceeding. We likely won't need more than 1–3 frames in most cases. The moment a face is detected in the frame:
    - We can optionally stop capturing further (to save processing). We'll use the first clear shot with a face.
    - Face detection will likely be done via `face_recognition.face_locations(image)` which uses HOG by default on CPU. This returns coordinates of any faces.
- **Face Recognition:** For each face detected (usually one face at a door, but handle the case of multiple people), compute the facial embedding using `face_recognition.face_encodings(image, known_face_locations)`. This yields the 128-D vector for the face [30]. Now compare this against our stored data:
    - Check against **Whitelist encodings** to see if this face is one of the known individuals. We can use `face_recognition.compare_faces(known_encodings, visitor_encoding)` which returns a list of True/False for matches within a certain tolerance. Or use `face_recognition.face_distance` to get confidence levels. Suppose it matches a

known person (say it recognizes “Alice” at the door). In that case, we classify this face as **Known**. If multiple known faces are present, we’d mark each accordingly. (If at least one known family member is there, likely no alert is needed, though logging is fine.)

- Check against **Blacklist encodings** similarly. If the face encoding matches one in the blacklist (e.g., it’s very close to “John Doe – FBI Wanted”), mark it as **Blacklisted**. In a rare scenario, a face could be both (e.g., someone you know turned criminal and you happened to blacklist them too) – the instructable actually accounted for a person being in both lists [31]. In such a case they treated it as a special condition with a distinct alert (they flashed a yellow light and still alerted via Telegram) [32]. We can design a priority: blacklist status will override whitelist for security (i.e., if a known person is also blacklisted, treat them as blacklisted/suspicious).
- If a face is neither in whitelist nor blacklist, it’s classified as **Unknown**.
- If no face was detected at all (e.g., maybe the person walked away or covered the camera), the system might try a few seconds then time out. In that case, we can still send an “unknown visitor” alert with whatever image we got (or no image) to notify the owner that someone rang but no face was seen.
- In case multiple faces: we will likely handle each face and combine results. For example, if one face is recognized (say a family member) and another is unknown (they brought a friend), the system should probably treat it as a semi-unknown situation. Perhaps still notify the homeowner that an extra unknown person is there with a known person. To keep it simple, we could decide: **if any face is blacklisted -> treat as blacklist alert (highest urgency)**; else if any face is unknown (and none blacklisted) -> treat as unknown alert (because at least someone is unrecognized); else if all faces are known -> no alert needed (just log event, maybe greet). This logic is similar to the traffic-light system in the reference: green if all known, yellow if any unknown, red if any blacklisted [31].

• **Response/Actions:** Based on the classification from above, the system takes appropriate action:

- **Known Person (Whitelist):** In this case, we assume it’s a friendly visitor or family member. We can light a **green LED** on the doorbell device to reassure the visitor (as done in the traffic-light feedback) [33]. We might not send a phone notification because the homeowner likely expects them or they themselves might be home. Although, a customizable option could be to still send a notification like “Your spouse has arrived at the door” for information. To keep things simple, we might skip notifying on known, or perhaps log “Alice was recognized at 5:30 PM”. Another possible action: if we integrate a smart lock, we could automatically unlock the door for known persons. That would involve additional hardware (servo or smart lock integration) which is beyond the basic scope, but design-wise it’s feasible since once you trust the recognition, you could trigger a relay to open an electronic lock. For now, we’ll focus on the surveillance aspect rather than entry control.
- **Unknown Person:** This is our main use-case for alerts. When an unknown face is seen, the system will immediately send a **Telegram alert** to the homeowner’s phone. The message could be: “ *[Home Security] Unknown person at the front door!”* and include the captured photo of the visitor (the face or full frame). Using Telegram’s API, we can send the photo with a caption via our bot. The instructable did this using a Telegram bot token – their bot sent photos with an orange alert icon for unknowns [32]. We’ll do similarly. The homeowner can then decide to respond (e.g., use an intercom or just be alerted). In addition, the system will **log the event** locally: save the image to an “unknowns” folder with timestamp, and perhaps store the encoding. If later the homeowner identifies this person (say it was a new mailman

or a neighbor), they could move that image to the whitelist folder to improve the system (essentially learning that face).

- **Blacklisted Person:** In this alarming scenario, the system will treat it as a security threat. Actions can include: flashing a **red LED** on the device (to signal danger) [34], emitting a loud sound (siren or prerecorded message) to possibly startle the person (though this could also scare harmless visitors if it was a false match, so that might be optional/manual). Definitely, a **priority alert** will be sent to the homeowner via Telegram: e.g. "*[Home Security] ALERT: Person matching* John Doe (Wanted) *is at your door. (See photo)*" along with the photo. The owner can then call authorities if appropriate. Since we plan on using FBI data, we could even include a snippet of the wanted person's description in the alert (the FBI API returns details that we could parse). However, caution is needed – the system is not 100% foolproof, so we might label it as "possible match to XYZ". Logging is critical here: we save the image and perhaps tag it with which blacklist ID it matched and the similarity score, for future evidence. We will also design the system to not automatically do anything drastic like contacting police, because false positives or errors in DIY security could lead to trouble. It's ultimately an assistive tool for the homeowner, not a replacement for official security systems.
- **Continuous Monitoring After Alert:** One consideration – once an unknown or blacklisted alert is sent, should the system continue capturing images (say, to track if the person stays or leaves)? For simplicity, we will not implement full CCTV recording. But one could incorporate a short video clip recording after the doorbell press. In our design, the doorbell press is a one-time trigger that leads to a snapshot and classification. If the scenario requires more, the homeowner can check a live feed (if we set one up) or come to the door. To keep it low-cost, we won't require additional cloud storage for video or such. The images will suffice for our needs.

- **Notification System (Telegram) Setup:** We choose Telegram as the notification channel due to its cost-free API and ability to send multimedia. Design-wise:

- We will create a Telegram bot (through BotFather) which gives us a **bot token**. We also obtain our chat ID (the user's ID) by talking to the bot once. These credentials (token and chat_id) will be stored in a config file on the Pi (for security, not hard-coded in code). The instructable outlines this process: you create a new bot and save the token in a `credentials_telegram.py` file [35].
- Using the `python-telegram-bot` library (or even simpler, the `requests` library to hit Telegram's sendMessage API endpoint), the Pi will send messages. We'll have functions like `send_alert(photo_path, text, alert_type)` that handles sending. For unknown or blacklist, we attach the photo. We can also differentiate the alerts by adding an emoji or label (as they did with colored icons) [32]. Possibly, we could create two bots or two chat threads if we wanted to separate critical alerts, but one bot sending all with clear text is fine.
- Pros of Telegram: Instant delivery, images are viewable inline, and we can even receive responses (if we want to extend the system so the user can send commands back, e.g., "who is this?" to trigger face re-scan or "open door" command if we had a lock – but these are future expansions).

- Alternatives considered: **Email** (free but slower, images might not show immediately in notifications); **SMS/MMS** (would need a Twilio API or similar, involves per-message cost and MMS might not reliably deliver images to all phones); **Mobile App with Push** (requires developing a custom app or using a service like Firebase Cloud Messaging – too heavy for this project scope). So Telegram hits the sweet spot for a DIY solution.

- **User Interface:** Aside from Telegram notifications, the primary "UI" for the system is minimal. We might implement:

- **LED indicators** on the device as mentioned (green/yellow/red) to give immediate feedback at the door. This is helpful for occupants (e.g., an elderly person inside can glance at the LED panel to know if it's a friend (green) or unknown (yellow/red) before opening) [2]. If cost is a concern, even a single dual-color LED can provide basic signals. These LEDs will be driven by GPIO outputs in our software whenever a state is determined.

- **Logging interface:** At minimum, we'll keep logs in a file (with timestamps, results, etc.). Optionally, we could set up a simple web server on the Pi (maybe a lightweight Flask app) to display recent visitors, allow adding a face to whitelist by uploading a photo, etc. This would make maintenance easier for a non-developer user. However, setting up a web UI means additional packages and some security considerations (we'd need to restrict it to LAN or protect it with a password). To stick to our low-cost, minimal-frills mandate, we might not include a web dashboard initially. The user (developer) can access the Pi via SSH or VNC to manage it, or simply use the file shares. In future, this could be an area of enhancement.

- **Safety & Security Design:** We should ensure the system itself is secure since it's part of home security:

- **Pi Security:** Change default passwords, possibly disable SSH password auth in favor of keys, keep the system updated. The Pi is inside the network, but if someone compromised it, they might access camera or images. We will treat the Pi like any IoT device – secure the ports and only expose what's needed (no unnecessary open ports; if a web interface is running, secure it).
- **Data encryption:** Not strictly necessary on the Pi, but for example, storing the Telegram token in plaintext is sensitive. We might secure it by file permissions. Communications over Telegram API are encrypted via HTTPS. If we did any cloud queries (like to FBI API), those would also be HTTPS.
- **Privacy considerations:** We should inform any users/visitors that a camera is in use (e.g., a small sticker "CCTV in use" or so) for legal compliance, depending on jurisdiction. Our design keeps data local, which is a plus for privacy – images of neighbors or visitors aren't being uploaded to third-party servers (except as needed for alerts to the homeowner). If this system were to be used in, say, an apartment complex, one must consider regulations around facial recognition. Since it's a personal home project, it's under the homeowner's discretion.

With the design choices above, we have a clear blueprint of how each component will function and interact. We prioritized choices that meet the *low-cost* and *effective* criteria: Raspberry Pi over expensive edge devices, open-source libraries over paid services, local processing over cloud dependence, and simple interfaces like Telegram over complex apps. We also compared alternatives at each step (for instance, cloud vs local, LBPH vs Dlib, etc.) and reasoned out the final choice with pros/cons supported by sources. This design should provide a solid foundation to proceed into the building phase.

Building the System

With planning and design settled, building the system involves assembling hardware, configuring the software environment, and writing the code for each functionality. Let's break it down into actionable steps:

Hardware Assembly

1. **Mount the Raspberry Pi and Camera:** Place the Raspberry Pi in a case (if available) to protect it. Connect the Camera Module to the Pi's CSI camera port (lift the connector latch, insert ribbon, lock it). Mount the camera at your door location. You might use a case or bracket for the camera – many 3D-printable or off-the-shelf camera mounts exist [36]. Ensure the camera has a clear view of a visitor's face at the doorstep (typical eye level ~5.5 feet high). If you want to keep the Pi indoors for security, you could extend the camera via a longer ribbon cable through the door or wall (camera outside, Pi inside). Keep the ribbon safe from damage. Alternatively, house the Pi and camera together near the door (but then consider enclosure and weatherproofing).
2. **Wire the Doorbell Button:** Connect the chosen doorbell button to a GPIO pin on the Pi (e.g., GPIO 18) and to a ground pin. If the button has polarity (LED inside etc.), follow its specs. Use a resistor for hardware debounce if needed (not usually necessary if using Pi's internal pull-ups). In software, we will enable an internal pull-up or pull-down on that GPIO so it reads a stable HIGH when not pressed and LOW when pressed (or vice versa). For testing, a simple LED or buzzer can be attached to another GPIO to act as an output indicator or chime. Make sure to use a proper resistor for any LED. If interfacing with an existing doorbell circuit, use a relay module: the relay coil connects to the doorbell wiring, and the relay's NO/NC contacts to the Pi GPIO (with appropriate voltage conversion, maybe via an optocoupler or using the relay as an isolator and a separate GPIO input that the relay shorts to ground). This part can get technical; if unsure, the safer route is using a separate button for the Pi.
3. **Provide Power and Network:** Plug in the 5V power supply to the Pi. Connect an Ethernet cable to your router (if using Ethernet) or be prepared to set up Wi-Fi credentials. On first boot, you might connect a keyboard/monitor to do initial config, or use headless setup with SSH enabled. Ensure the Pi's power LED stays solid (voltage is stable) and it boots properly.

At this stage, hardware should be connected. Before coding the full application, it's wise to **test each hardware component**: - Boot the Pi and enable the camera in OS settings (`raspi-config` -> Interface Options -> Enable Camera) [37]. Reboot after enabling. Then test the camera with `libcamera-still -o test.jpg` or older `raspistill` if using legacy, to ensure it captures an image. We can also use OpenCV/Python to grab a frame to verify (more on that in testing). - Test the doorbell GPIO: in Python, set up a simple script to print when the button is pressed. Ensure that pressing the physical button yields the expected signal (may need to invert logic depending on wiring). Adjust pull-up/down as needed. - If using an LED for feedback, test turning it on/off via GPIO as well.

Software Setup

1. **Install OS and Dependencies:** Install Raspberry Pi OS (Bullseye or latest) on a microSD card (at least 16GB, as recommended; 32GB if storing many images) [38]. Once the Pi is up:
2. Update the system: `sudo apt update && sudo apt full-upgrade` [39] to get latest packages and firmware (including camera drivers).
3. Install Python libraries and tools. We will likely need:
    - **face_recognition library**: This can be installed via pip: `pip3 install face_recognition`. This will try to install `dlib` as a dependency. On Pi, this can take a long time to compile. We have alternatives: use `apt` to install `python3-dlib` (which might be an older version but pre-compiled), or use `pip3 install dlib` first if a pre-built wheel is available for your Pi architecture. There are also unofficial wheels or scripts for Pi (for example, the gist by ageitgey on installing dlib on Pi) [9]. If compilation is an issue, one trick

is to install via Docker as provided by the instructable author (they provided a Docker image `erientes/doorbell` that has everything pre-installed, and they run the code inside that container [22]). Using Docker is an option: it simplifies installation (no need to compile locally) and isolates the environment. However, running Docker on Pi adds overhead and complexity for newcomers. We can attempt direct installation: often the community provides pre-compiled wheels for dlib (e.g., `dlib-19.22` for armv7). Assuming we manage to install `face_recognition` successfully, it will bring in `dlib`.

- **OpenCV**: We want OpenCV mainly for possible camera interfacing and image handling. We can try `pip3 install opencv-python`. On Raspberry Pi OS, a lighter alternative is `sudo apt install python3-opencv` which installs OpenCV from apt repository (usually an older version but sufficient). OpenCV is optional if we use PiCamera library to capture images and `face_recognition` to do detection (since face_recognition/dlib can operate on NumPy arrays directly). Nonetheless, OpenCV is handy for any image preprocessing (resizing, conversions) and debugging (we can use it to display image windows if we connect a screen).
- **PiCamera or libcamera**: There's a Python library `picamera` (for the legacy camera stack) used in many tutorials [15]. However, newer Raspberry Pi OS uses `libcamera`. If using Raspberry Pi OS Bullseye or newer, the `picamera` (v1) library might not work unless legacy support is enabled. Instead, one can use `picamera2` (the new library) or just capture with OpenCV's VideoCapture. To keep it simple, we might enable legacy camera support (in raspi-config) and use `picamera` library, as it's quite straightforward in Python. So: `pip3 install "picamera[array]"` (the array part allows directly obtaining images as NumPy arrays for OpenCV/dlib). If that fails, Plan B is using OpenCV's `cv2.VideoCapture(0)` which, with the proper backend, can fetch frames from the camera.
- **RPi.GPIO**: This usually comes with RPi OS, but if not: `pip3 install RPi.GPIO`. This allows us to listen for button presses and control LEDs.
- **python-telegram-bot**: `pip3 install python-telegram-bot` (v13 or v20 depending on Python version). This library simplifies sending messages via the bot. Alternatively, we could use `requests` to hit Telegram APIs, but the library handles a lot for us, including formatting the photo send. The instructable lists it as a requirement [25], meaning they used it to communicate with Telegram easily.
- **Other**: numpy will be installed as a dependency of face_recognition or OpenCV. Pillow is also often a dependency (for image file I/O). Ensure they are installed/updated (`pip3 install --upgrade pillow numpy`).

4. In summary, a one-liner (if we trust pip on Pi for these heavy libs) might be: `pip3 install face_recognition opencv-python RPi.GPIO python-telegram-bot picamera` – but be prepared to wait or troubleshoot if any build errors occur. It's often useful to increase swap memory on Pi when building dlib to avoid memory errors. Given this is a deep dive, we mention these potential hiccups and assume we solve them (or use Docker as a fallback environment which the instructable explicitly recommended for ease [40]).

5. **Prepare Known/Blacklist Data:** Before coding the main logic, gather the initial data:

6. In a directory (say `/home/pi/security_system/known_faces/`), place images of the family members or frequent visitors. Ideally, use clear, front-facing photos (passport-style) for the initial dataset – this will yield good encodings. We can name files like `alice.jpg`, `bob.jpg` or organize

subfolders per person. For simplicity, one image per person is enough to start (dlib's embeddings work well with just one image per person, unlike LBPH which benefits from multiple). We'll also create an empty list or folder for new captures to be added if needed.

7. Prepare the blacklist images. Using the FBI Wanted API, we could write a small script (later in maintenance we might automate this). For now, say we manually download a few images of well-known targets (this is just for demonstration – hopefully none show up at our door!). Save them in `blacklist_faces/` with identifiable names (e.g., `wanted_johndoe.jpg`). The instructable suggests simply dropping images into `img/blacklist` and the software will encode and move them [29]. Our implementation can do something similar on startup: read all images in `blacklist_faces/`, encode them, then (optionally) move them to a subfolder after encoding (or just keep them).

8. It's important to label these as well. We might maintain a Python dictionary for names: e.g., `known_names = ["Alice", "Bob"]` corresponding to the known encodings list, and `blacklist_names = ["John Doe (FBI#123)", ...]` for blacklisted. The labels help in logging and notifications.

9. **Coding the Application:** Now, build the Python application (let's call it `doorbell_security.py`). Key components in code:

10. **Initialization:** Set up GPIO pins (e.g., configure the doorbell pin as input with pull-up, LED pin as output), initialize camera, load the face data. For loading face data:

```
import face_recognition
known_encodings = []
known_names = []
for file in os.listdir("known_faces"):
    img = face_recognition.load_image_file("known_faces/"+file)
    enc = face_recognition.face_encodings(img)
    if enc:
        known_encodings.append(enc[0])
        name = os.path.splitext(file)[0]
        known_names.append(name)
# Similarly for blacklist
blacklist_encodings = []
blacklist_names = []
for file in os.listdir("blacklist_faces"):
    img = face_recognition.load_image_file("blacklist_faces/"+file)
    enc = face_recognition.face_encodings(img)
    if enc:
        blacklist_encodings.append(enc[0])
        name = os.path.splitext(file)[0]
        blacklist_names.append(name)
```

We might wrap that in try/except in case of any corrupted image, etc. After this, we'll have our face databases in memory. If performance or memory were a concern with many faces, we could store a KD-tree or something for encodings, but with tens of faces, that's overkill.

11. **Camera capture function:** Depending on whether using PiCamera library or OpenCV: *If using PiCamera:* We can use the `picamera` library to capture an image to an array. For example:

```
import picamera
import numpy as np
camera = picamera.PiCamera()
camera.resolution = (1280, 720)  # or 640x480 for faster
output = np.empty((720, 1280, 3), dtype=np.uint8)
camera.capture(output, 'rgb')
# 'output' now contains the image data
```

This captures a frame in RGB format directly into a NumPy array, which we can feed to face_recognition (which expects RGB np.array). We might want to adjust camera settings (brightness, contrast, maybe rotate if the camera is mounted upside down, etc.) in initialization. Also possibly use camera's LED or IR if available. *If using OpenCV:* Alternatively:

```
import cv2
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)
ret, frame = cap.read()
rgb_frame = frame[:, :, ::-1]  # convert BGR (OpenCV default) to RGB
```

The above should also get an image. We have to ensure the camera is accessible; sometimes on Pi, you need to enable V4L2 drivers or use `cv2.CAP_ANY` backend. This is more fiddling, so picamera is often simpler for still images. In either case, we will integrate this capture logic in the doorbell handling.

12. **Main Loop:** We use `RPi.GPIO` to wait for the button. We could do:

```
import RPi.GPIO as GPIO
BUTTON_PIN = 18
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Then use an event detection:

```
def doorbell_pressed(channel):
    handle_doorbell_event()
GPIO.add_event_detect(BUTTON_PIN, GPIO.FALLING, callback=doorbell_pressed,
bouncetime=300)
```

This will call our `handle_doorbell_event()` whenever the button is pressed (falling edge if using PUD_UP). The bouncetime debounces it for 300ms. Alternatively, a simple loop polling `GPIO.input(BUTTON_PIN)` with `time.sleep()` can also work, but event/callback is cleaner.

13. **Handle Doorbell Event:** In `handle_doorbell_event()`, implement the capture and recognition workflow:

    1. Perhaps turn on an indicator LED or log “Doorbell pressed!”.
    2. Capture images until a face is found or up to N tries. For example:

    ```
    face_locations = []
    for attempt in range(3):
        image = capture_image()  # using one of the methods above
        face_locations = face_recognition.face_locations(image)
        if face_locations:
            break
        time.sleep(0.5)
    ```

    If `face_locations` remains empty after 3 attempts, proceed with the last image anyway.
    3. If face(s) found, for each face compute encoding:

    ```
    encodings = face_recognition.face_encodings(image,
    known_face_locations=face_locations)
    ```

    Typically, we expect one face, but loop just in case.
    4. Initialize flags: `person_known = False`, `person_name = None`, `person_blacklisted = False`, `blacklist_name = None`. For each encoding `enc` in `encodings`:
        - Compare with known:

        ```
        matches = face_recognition.compare_faces(known_encodings, enc,
        tolerance=0.6)
        ```

        If any True, we have a known person. We could take the first match (or the best match by comparing distances using `face_distance`). Identify the name index:

        ```
        if True in matches:
            idx = matches.index(True)
            person_known = True
            person_name = known_names[idx]
        ```

        (Note: If multiple faces, we might only care if at least one is known; but we might refine logic later.)
        - Compare with blacklist:

        ```
        matches_bl = face_recognition.compare_faces(blacklist_encodings,
        enc, tolerance=0.5)
        if True in matches_bl:
        ```

```
    idx2 = matches_bl.index(True)
    person_blacklisted = True
    blacklist_name = blacklist_names[idx2]
```

We use a slightly stricter tolerance 0.5 for blacklist to reduce false alarms.

- We could also gather distance scores to possibly include in logs.

5. Determine outcome for this event:
   - If `person_blacklisted` is True (regardless of known), we classify as **Blacklist Alert**. (If the person was also known, that's an odd scenario but it means a known person is in blacklist; we still treat it as threat as per design.)
   - Else if `person_known` is True (and not blacklisted), classify as **Known**.
   - Else (not known, not blacklisted) classify as **Unknown**.
   - If multiple faces, we could refine: for instance, if any blacklist present -> blacklist alert (highest priority). If none blacklisted but at least one unknown -> unknown alert. Only if *all* faces were recognized as known do we consider it a known scenario. This matches the logic table from earlier [31] . We can implement that by aggregating results across faces.
6. **Take Actions:**
   - If Known: log "Known person {name} at {time}" to a file. Light green LED blink maybe. We might not send a Telegram message in this case to avoid spamming the owner for routine family arrivals. (However, this can be a user preference – maybe they *do* want a notification "Your child arrived home from school". For now, we'll assume not.)
   - If Unknown: log "Unknown person at {time}". Save the image to `captures/unknown_2025-09-02_17-30-00.jpg` (timestamped). Then send a Telegram message: e.g.

     ```
     bot.send_photo(chat_id=TELEGRAM_CHAT_ID,
     photo=open(capture_path, 'rb'),
     caption="  Unknown person at the door")
     ```

     If not using the bot library, use requests to POST to `https://api.telegram.org/bot<token>/sendPhoto`. But we have the library, which makes it one-liner. We might reuse the same photo for alerts that we saved. Possibly, also turn on a yellow LED for a few seconds to indicate an unknown (as feedback to anyone monitoring inside).
   - If Blacklisted: similar steps but more urgent. Log "ALERT: Blacklisted person (possibly {blacklist_name}) at {time}". Save image to `captures/alert_...jpg`. Send Telegram alert with a distinct message, maybe with emoji. If we have the name from blacklist, include it: " *Alert: Person matching {blacklist_name} detected at your door!*" [34] . We can also differentiate in the Telegram message (maybe use a different chat or sticker), but text is enough. Turn on a red LED or flash lights. We might even make a buzzer sound via GPIO if an alarm is connected. Those are optional; design-wise, a sound could deter the person or alert anyone in the house.
7. Reset/cleanup: After handling, we could turn off any LEDs after a delay, close camera if needed (though keeping it open is fine for next use), etc. The system then goes back to idle waiting for the next press.

14. **Error Handling:** The code should handle exceptions gracefully so that the program doesn't crash on a single error (we want high uptime).

    - Camera errors: If capture fails (camera busy or error), log it and maybe try re-initializing camera.
    - face_recognition errors: If no faces and functions return empty, handle that case (we did).
    - Telegram errors: if message fails to send (e.g., no internet at that moment), catch and perhaps retry or at least log the failure. The system should not hang on notification – we can spawn the notification send in a separate thread or just move on if it fails.
    - GPIO events: If multiple rapid presses happen, ensure our code can handle concurrent triggers (using event_detect callback might spawn multiple calls). We might add a software lock that if one is in process, ignore additional until done (or queue them).
    - We can also implement a cooldown: e.g., if doorbell is pressed multiple times in a short interval, maybe only process once to avoid duplicate alerts. But this may not be too important.

15. **Testing as We Build:** It's recommended to test each piece of the code as we write it:

16. Test the camera capture function alone (e.g., call capture_image, then use OpenCV to show or just save the image to disk to verify it looks correct).
17. Test face recognition on static images offline. For instance, take an image containing a known person's face and an unknown, run the encoding comparison to see if it correctly identifies. We might use some sample images first (even the ones we placed in known_faces and blacklist for a dry run in code). Ensuring the tolerance and matching logic works as expected is crucial (we can print out the distance values for some test cases).
18. Test Telegram notification by calling the send function with a known image: see if the phone receives it. This requires the bot token and chat ID to be set correctly. (During development, we often start with a test chat with ourselves.)
19. Test GPIO trigger: simulate a button press (or just call the handler manually in code) to see the whole pipeline triggers.

We can incorporate some unit tests or at least logging extensively to verify each step. For example, log how many faces found, the names matched or distances. This helps in debugging issues like "face not detected due to lighting" or "wrong person matched".

By the end of the build phase, we should have a working prototype: pressing the doorbell yields console logs and possibly Telegram messages, etc. Next, we focus on verifying its behavior thoroughly through structured testing.

Testing the System

Testing is vital for a security system to ensure it performs reliably under various conditions and that our design choices indeed yield the intended results. We will use a mix of **functional testing**, **performance testing**, and **edge-case testing**:

- **Functional Tests (Features):** Verify each requirement end-to-end:

- *Face Recognition Accuracy:* Collect a set of test images of people who are in the whitelist, and some who are not, plus one or two who are in the blacklist. In a controlled environment, feed these to the system (we can simulate by calling the face recognition function with these images). Check that known people are correctly identified as known (green path), unknown people trigger the unknown path, and a blacklisted face triggers the alert path. For example, show the camera a picture of a family member – does the system recognize them and refrain from alerting? Then show a picture of a stranger – does it send an unknown alert? For a blacklist test, you could use an image of a "wanted" person from your dataset; the system should send the red alert. We expect near-perfect outcomes for these tests given the high accuracy of the chosen algorithm [20], but we might find issues if the images are low quality or if lighting differs. If any misidentification occurs (e.g., a known face not recognized due to angle), that indicates we may need more reference images or to tweak the tolerance. The thinkrobotics guide notes that lighting can drop accuracy from 90% to 60% [41], which we should test: try the system in low light vs good light. Possibly add an IR illuminator or increase camera ISO for low light if needed.
- *Doorbell Trigger Flow:* Physically press the button on the assembled device and observe: does the camera LED turn on (if using one), does the system capture and then send the appropriate notification? We might simulate an actual scenario – a team member stands in front of camera and presses button. Time how long until the Telegram alert arrives. The target is ideally <5 seconds for an unknown (the instructable's system likely did it within a couple seconds on Pi 3). On Pi 4 we expect faster. If it's too slow, check if any part (face detection or sending) is the bottleneck. (We can optimize by capturing at lower resolution if needed to speed up face finding.)
- *Multi-face scenario:* If two people come to the door together (say two family members, or a family member plus an unknown friend), what does the system do? Our logic would currently mark it as unknown if any unknown is present. Test this: have one known and one unknown face in view. Ideally, the system should treat it as unknown (with maybe a message like "Unknown person with [Known Name] at door" if we programmed such detail). The instructable's traffic light would show yellow (unknown) in this case even if one known [32], which seems reasonable. We confirm that it doesn't mistakenly ignore the unknown because a known was present.
- *No Face scenario:* Test ringing the bell with nobody (or covering the camera). The system will try a few frames and likely send an unknown alert with no face. We should see how it handles "face_locations" being empty. We coded it to break after a few attempts – ensure it does eventually send something or at least resets. Perhaps in this case, sending an alert like "Doorbell pressed, but no face detected" could be useful. We can adjust based on this test.
- *Blacklist false positive:* This one is tricky to simulate intentionally, but we can check how close a non-blacklisted face's encoding might be to a blacklist encoding by printing distances. If by chance a family member's encoding is within tolerance to some criminal's encoding (extremely unlikely with a good threshold), we'd get a false alarm. We set a strict threshold for blacklist (0.5). We can test on a variety of non-matching faces to ensure none triggers a match. If one does, we might lower false alarm risk further (e.g., require two images match or raise threshold).

- *Button Debounce and Multiple Presses:* Press the doorbell rapidly or twice in a row. The system should ideally not start two parallel processes. With our event detection and a slight bouncetime, it should handle a double-press as one event. We verify that a second press during an ongoing processing is either queued or ignored. (We can enforce ignoring subsequent presses until done by disabling event detect until finished, if needed.)

- **Performance and Load Testing:** Although this is a small-scale system, we test a few performance aspects:

- *Latency:* Measure time from button press to classification. We can instrument the code to timestamp at button press and after face recognition, and after sending notification. Ensure the longest step is reasonable (likely the face_recognition step). On Pi 4, if processing a 720p image takes ~1 second for detection+encoding, plus maybe 0.5s to send Telegram, we expect ~1.5s total. If it's significantly more, consider optimizations like resizing the image before encoding (e.g., scale down to 75% if face is still clearly visible). The thinkrobotics FAQ suggests Pi 4 can do multiple faces at 5–8 FPS [4], which is encouraging for single face ~0.2s if optimized; however, our use of Python and possibly larger resolution might be slower. We adjust as needed (maybe set camera to 640x480 which still can recognize a face in close range and will be much faster to process).
- *Resource usage:* Use `htop` or similar to see memory and CPU usage during run. Dlib might use a lot of RAM to load its models (150MB perhaps) and CPU will spike to 100% on one core during processing. That's fine as long as it's brief. Ensure no memory leaks (run it many times, see if memory usage grows).

- *Robustness:* Let the system run for hours and ensure it doesn't crash. We might simulate periodic activity (if nobody rings for hours, system just idles which should be fine). One way to test continuous operation is to create a script to simulate doorbell events every few minutes (calling the handler with a sample image) and see if it stays stable.

- **Usability Testing:** From an end-user perspective:

- Is the Telegram alert message clear and helpful? Perhaps test it with someone who didn't build the system – do they understand the alert? Maybe we should include in the message the snapshot and a line like "Reply to this bot or call home for further action." If we wanted, we could allow the user to send a command back to the bot to add the person to whitelist if they trust them, etc., but that's extra functionality.
- The physical placement: ensure the camera captures faces well (not too high/low). We might adjust the camera angle after initial tests when someone of different heights stands there. Using a wide-angle camera lens could be beneficial if the person stands off-center. There are camera modules with wider FOV if needed.

- Does the doorbell still function as a doorbell? If we replaced the chime, make sure the homeowner is notified one way or another (maybe the phone alert is enough, or maybe a simple buzzer on Pi to act as doorbell sound inside). We can test that the homeowner hears/sees the alerts promptly.

- **Edge Cases:**

- If multiple unknown people come (say two delivery personnel), the system would send one alert (we might only send one combined alert per ring, even if multiple faces). That should be fine. It would show both in the photo anyway.
- If the face is partially visible (person turning head): can test if recognition still works. Dlib is quite robust, but extreme angles might fail detection. We note any such cases. Possibly instruct the camera to take a couple of pictures a second apart – maybe the person moves a bit and one of them catches a full face.
- If a known person wears a hat or mask (like current scenarios with masks) – face_recognition may fail or misidentify. We can test with a mask on a known person; likely it will be unknown. This is acceptable since security-wise it's better to fail to recognize (treat as unknown) than falsely recognize. We just need to be aware and maybe the homeowner will then manually open (since they

see it's probably their family with a mask on via the photo). Advanced approach could incorporate mask recognition, but out of scope.

Any issues found during testing will lead to iterative improvements: - For example, if false unknowns happen for known people in poor lighting, we can add more images of that person in various lighting to the whitelist (improving the model's robustness) [42] [41] . Or adjust camera settings (enable night mode). - If performance is lagging, consider downsizing images or using a smaller model. (There are alternatives like MobileFaceNet or even using an **ESP32-CAM with built-in face recognition** for very low cost, as some projects do [43] – but those are less accurate and more of a novelty. Our Pi approach is fine, but just noting that exists as extreme low-cost: an ESP32-CAM is $10 and can do basic face recognition on-board albeit with limited accuracy.) - We should also test the FBI matching specifically: perhaps take one of the FBI images we loaded and show it to the camera (or directly feed it). It should match with itself obviously. Maybe test a lookalike to see if it triggers incorrectly.

Once testing validates that each part works and the system behaves as expected in realistic scenarios, we'll be ready to deploy it for real use.

Deployment

Deployment involves installing the system in its real environment and setting it up for continuous operation with minimal manual intervention. Here are the steps and considerations for deployment:

- **Physical Installation:** Mount the Raspberry Pi (in its case) near the door. If it's indoors, ensure the camera's ribbon cable is threaded through to the outside (perhaps through the door frame or a small hole in a wall). Secure the camera in a weather-resistant way – e.g., under an eave or inside a doorbell unit with just the lens peeking out. Many DIYers repurpose old doorbell enclosures or 3D print custom ones; even a small plastic project box can house the camera module (just cut a hole for the lens and cover with clear plastic). The doorbell button should be easily accessible to visitors; if it's separate from the camera, mount it nearby and label it if needed. We also mount any status LED (maybe on the doorbell housing). If we built a traffic-light style indicator (like the instructable, which had a box with green/yellow/red LEDs) [2] , place it where the resident can see it (maybe inside the house near the door or on a desk). If not, we at least have the Telegram alerts as feedback.
- **Power and Connectivity:** Plug the Pi into a permanent power outlet (avoid using a power bank unless it's a UPS scenario, because you don't want it dying). If using Wi-Fi, configure the Pi to connect to home Wi-Fi on boot (we likely did this already in setup). Optionally, reserve a static IP or DHCP reservation for the Pi in your router, so you can reliably access it on the network for maintenance. Ensure the Wi-Fi signal at the install spot is strong; if not, consider an Ethernet cable or a Wi-Fi extender. We can also disable power management on Wi-Fi (sometimes Pi's Wi-Fi sleep could delay Telegram sends, though usually that's not an issue when actively sending).
- **Autostarting the Software:** We want the security software to run automatically on boot and run continuously. There are a few ways:
- The modern approach: create a **systemd service**. For example, create `/etc/systemd/system/doorbell.service` with content:

```
[Unit]
Description=Doorbell Face Recognition Service
After=network.target
```

```
[Service]
ExecStart=/usr/bin/python3 /home/pi/doorbell_security.py
WorkingDirectory=/home/pi
StandardOutput=append:/home/pi/doorbell.log
StandardError=append:/home/pi/doorbell.err
Restart=always

[Install]
WantedBy=multi-user.target
```

Then run `sudo systemctl enable doorbell.service` to start at boot. This will ensure the script starts on boot and restarts if it crashes [44] (the above config uses `Restart=always`). We direct output to log files for debugging.

- Simpler but less robust: add a line to `crontab -e` like `@reboot python3 /home/pi/doorbell_security.py &`. This works but doesn't restart on crash and logs need manual handling.
- Or add to `/etc/rc.local`. But systemd is the recommended method nowadays.
- If we used Docker (like the instructable's docker image and aliases approach) [45], we'd ensure Docker runs on boot and maybe use a cron or systemd to execute the `doorbell_run main.py` command on startup. However, since we went with direct installation, we stick to running the Python script directly.
- **Environment Configuration:** Make sure the `credentials_telegram.py` (with bot token/chat id) is in place if our code expects it (the instructable had that separate config file loaded) [46]. Alternatively, store these in environment variables or a config JSON that our script reads. Ensure file permissions are such that no one else on the system can read the token (for security).
- **Final Checks:** Boot the system fresh and verify that everything launches correctly:
- The service starts, no errors (check `doorbell.err` or systemctl status for issues).
- Press the doorbell, ensure the whole chain works now that it's fully autonomous. This is essentially a live test of the integrated system.
- Walk through a few scenarios in the actual installed location (because lighting/background might differ from test environment). Maybe come at night to test night mode, etc.
- Monitor the logs for any repeated errors (for example, if every hour our FBI update script runs, see that it's succeeding). If we implemented a scheduled update (say via cron job that runs a script to fetch new FBI entries), ensure that's running. The instructable's approach was to run a check every hour within the program itself (could do with a threading Timer or simply in the main loop check time) [29]. We can incorporate that: e.g., have a thread that every 24 hours runs an update routine (downloads new blacklist images and encodes them).
- **User Training:** If this is for yourself, you know how to use it. If deploying for someone else (say an elderly parent as in the original motivation [47]), you should provide a simple guideline:
- Have the Telegram app running to receive alerts. If something happens, here's what the light colors mean (if we have the LED panel).
- How to add a new known person: (if we have a process like pressing a learn button or dropping an image file). Possibly, we can hide the complexity by having a mode where if you press a special button combination, the next face seen is added to whitelist. The instructable did exactly this: pressing a yellow button put it in "learning mode" to add face to whitelist [28]. We might not have that hardware interface; alternatively, just SSH in and add an image. But for user-friendliness, one

might implement a Telegram command like "/add Alice" that triggers the Pi to take a picture of the face at the door and add it as Alice. This requires two people (one at door, one issuing command). This is an area to develop if needed, but not critical initially.

- What to do on an alert: e.g., if you get an unknown alert, you might check the camera (maybe we have a separate way to access a live feed? We didn't explicitly add that, but one could use something like RTSP or a simple MJPEG server on Pi to view live video when needed – however, that adds overhead. Alternatively, you could manually start a camera feed from phone via something like Home Assistant or UV4L if set up. Given cost considerations, we skip a dedicated monitor feed). At minimum, the photo in Telegram gives the info. If a blacklist alert comes, the homeowner should verify (the photo vs known mugshot) and call authorities if it's truly that person – though realistically the chance of a FBI fugitive at your door is extremely low; it's more of a cool feature than an expected event.

- **Scaling and Interoperability:** If this system is successful and one wants to deploy multiple cameras (say front door, back door), one could replicate this setup. Possibly have them report to the same Telegram bot but with different labels ("Front door unknown vs Back door unknown"). Our design can scale to that, but it would be separate hardware per door unless using one Pi with multiple cameras (Pi 4 can handle 2 cameras max with an adapter, or multiple USB cams, but that complicates things) [48] . For now, focus on the single door scenario.

- **Costs Recap:** Deployment phase is a good time to reflect that we stayed within a low budget:

- Raspberry Pi 4 (~\$50)
- Camera Module (~\$25)
- SD Card (~\$10)
- Misc (button, wires, LEDs) (~\$5–10)
- Power supply (~\$10)
  In total, roughly \$100 or less. There are **no ongoing subscription costs**. Telegram and FBI data are free services [49] . The only recurring cost might be a negligible increase in electricity usage (the Pi 4 uses maybe 3-5W idle, so a few dollars per year). This is much cheaper over time than a cloud-based camera which might require a subscription for face recognition or video storage (like some commercial doorbell cams).

With the system deployed and running, we then move into the **maintenance** phase, to ensure it continues to operate smoothly and remains up-to-date.

Maintaining the System

Maintenance is important to keep the home security system effective over the long term, especially as conditions change (new family members, evolving threat databases, software updates, etc.). Here's how we plan to maintain and possibly improve the system over time, while keeping costs low:

- **Routine Updates (Software):**
- The Raspberry Pi OS should be kept updated for security. Every few months, run `apt update && apt upgrade`. However, be cautious with major upgrades – for instance, a new OS release might change the camera stack or library versions which could break our setup. It's wise to **backup the SD**

card (clone it) before major changes. Since this is a security device, we want it stable; we might choose to only apply critical security patches and not always the bleeding-edge updates to avoid downtime.

- The Python libraries (face_recognition, OpenCV, etc.) rarely need updating unless a bug is found that affects us. Dlib's model is static and well-performing; newer versions might not significantly change face recognition results. If everything works, we might *not* upgrade those often. "If it ain't broke, don't fix it" applies to an extent, because an update could require re-compilation. Only if we encounter a bug or need a new feature would we update those.

- Telegram bot API may update but the library usually remains backward compatible. Just monitor deprecation warnings in logs.

- **Hardware Maintenance:**

- The camera lens might get dirty or spiderwebs, etc. Clean it periodically to ensure clear images.
- If the camera or Pi is exposed to weather, check seals and covers for wear. Water or moisture ingress can be disastrous for electronics, so ensure the enclosure remains waterproof especially before rainy seasons.
- The SD card can be a point of failure if there are too many writes (they wear out). Our system doesn't write a ton (some log entries, occasional images). Using a quality SD card (or even an external USB SSD if needed) can prolong life. It might be wise to replace the SD card every couple of years as a preventative measure (and again, keep backups of the data/config).

- Ensure the Pi's power supply continues to deliver stable voltage. Sometimes cheap adapters degrade; if you see low-voltage warnings in Pi logs or flashing power LED, consider replacing the adapter or cable.

- **Face Database Maintenance (Whitelist):**

- **Adding New Faces:** Whenever a new person needs access (e.g., you hire a new babysitter, or a relative starts visiting often), you should add them to the whitelist so the system doesn't flag them as unknown. This can be done by capturing their face. If our system includes a learning mode (e.g., press a special button to add next face), use it. Otherwise, take a clear photo of them and place it in the `known_faces` folder and update the known encodings. We might have to restart the service for it to load the new data (unless we built a dynamic update feature). To improve the system, add at least 2-3 photos of each person from slightly different angles or with different lighting. The face_recognition model can usually recognize from one photo, but multiple can help if, say, they sometimes wear glasses versus not. We can occasionally retrain LBPH (if we had that) or re-generate encodings for new images.
- **Removing Faces:** If someone is no longer authorized (say a roommate moves out), remove their images from the whitelist and also consider adding them to a watch list if needed (though hopefully not). Also, if a family member's appearance changes drastically (major haircut, facial hair changes), update their photos in the database to avoid non-recognition. Our system can handle moderate changes, but very big changes (e.g., long beard vs clean-shaven might drop confidence). Updating images will maintain accuracy.

- It's a good practice to review the **unknown visitor log** periodically. Perhaps every few weeks, look at the saved snapshots of unknown visitors. They might include the mailman, delivery folks, neighbors,

etc. If you see a recurring friendly person, you might decide to add them to known (to reduce future alerts). Conversely, if you see any suspicious individuals lurking (unknowns at odd hours), you can be aware or share info with neighbors/police if something happened in the area.

- **Blacklist Updates:** This is where automation helps:

- The FBI "Most Wanted" list can change. Our system should update the blacklist encodings at some regular interval. We can schedule a cron job or incorporate into our code a periodic refresh. For example, a small script using the FBI API: `requests.get('https://api.fbi.gov/wanted/v1/list')` will return JSON of current wanted persons [8]. We parse it for new entries or images. We could download those images and compare with what we have. In the instructable, the user would manually add images of notorious people to the folder and the program scanned the folder hourly [29]. We can similarly automate: perhaps weekly, fetch the top 10 wanted and ensure our `blacklist_faces` folder has those (update any changed). Then our service, if written to periodically check the folder, will auto-encode new ones. Alternatively, stop the service, regenerate blacklist encodings from updated images, and restart. This upkeep ensures even newly added criminals are in our system. The cost of doing this is just some time and bandwidth, negligible in cost.
- Also consider local crime information: One might manually add a local suspicious person (if you have a photo from police bulletins, etc.) to the blacklist. The system is flexible – just drop the photo in and it's now watching for them. Always ensure the photo is of sufficient quality (face_recognition does well with clear frontal or slightly angled faces; very low-res CCTV stills might not encode well).

- If the blacklist grows large (say hundreds of faces), monitor performance. A larger list means slightly more comparison work. A few hundred 128-D comparisons is trivial (0.001s), but if someone went overboard with thousands, one might then consider using more efficient search (like KD-tree). Likely unnecessary for our scope.

- **Monitoring and Logging:**

- Keep an eye on the `doorbell.log` (or whatever log we set). We should see regular "idle" or heartbeat messages perhaps, and entries whenever an event happens. If the system suddenly stops logging or you stop receiving Telegram alerts, that's a red flag it went down. That's why we set it to auto-restart on crash. But what if something non-critical fails (like Telegram send fails due to network outage)? We might implement a watchdog: e.g., a daily test message or a simple LED blink that indicates the loop is running. Another idea: have the system send a "I'm alive" message once a day (maybe to a private log or even Telegram at a set time). This could be annoying, so perhaps not to Telegram, but maybe an LED that blinks every minute as a heartbeat. The instructable remarks that operating via logging into the Pi gives more info, but the basic status is shown by the traffic light LEDs [50]. We can mimic that: e.g., when idle, maybe a slow green blink indicates okay.
- If the Pi or service does crash for some reason, the systemd will try to restart it. We should investigate crashes by reading error logs (`doorbell.err`). Memory leaks or unhandled exceptions could cause repeated crashes. Through testing we hope to eliminate those. Setting `Restart=always` ensures even if it crashes at 2am, it comes back immediately.

- We should also ensure the system's **time is correct**, as it affects logging timestamps and possibly the scheduling of updates or any time-based rule. If the Pi lacks a Real-Time Clock module, it relies on internet time sync (NTP). So if it boots without internet, time could be wrong until it connects.

Consider adding a cheap RTC module if the Pi will often be offline. Otherwise, ensure NTP is working (it is by default).

- **Future Improvements (optional, if cost allows later):**

- **Integrate with Smart Lock:** If at some point you install a smart lock or an electronic strike on the door, the system could automatically unlock when a trusted face is recognized (and maybe when you explicitly allow via phone). This adds convenience but also some risk (face spoofing attacks, etc., though dlib's model is not easily fooled by photos – it looks at 3D embeddings. But still, one might want to require a second factor, like also recognizing the voice or requiring a phone confirmation).
- **Multi-Factor Alerts:** For blacklisted matches, one might integrate a service to cross-verify. For example, use a second algorithm or send the image to a cloud face recognition as double-check. This reduces false alarms but increases cost (API calls) and latency. Probably not needed given our low threshold approach.
- **Better UI:** Perhaps create a simple web interface to review logs and captured images, and to manage the whitelist/blacklist. This could be done with a Flask app that displays images from the unknown folder and has buttons like "Add to Known (name)" or "Mark as false alarm". It would make maintaining the face lists easier for a non-technical user. We didn't include this to keep things focused, but it's a logical extension once the core works.
- **Mobile App Integration:** Instead of just Telegram, some might integrate with existing home automation apps or services. For example, Home Assistant can take in camera feeds and do face recognition (there's a component for it, using local or Facebox) [51] . Our system could output events to Home Assistant (via MQTT or API) if one uses that platform. However, we aimed for a standalone system, not requiring these external systems, to keep it simple and cost-free aside from what we built.
- **Scale & Cloud Backup:** If the system accumulates many images (especially unknowns), storage could slowly fill. 32GB is plenty for thousands of JPEGs, but if it ever becomes an issue, archiving older images to a NAS or cloud storage might be considered. Alternatively, automatically delete images older than X days (especially unknowns that were deemed not important). This is maintenance housekeeping. It can be done via a cron script or a small addition in code (e.g., at startup, purge anything older than 90 days in logs and images).

- **Security Hardening:** If this project becomes mission-critical, one might isolate the Pi on the network, use firewalls, etc., especially since it's sending data externally (to Telegram). One might worry about someone hacking the Pi and seeing camera feed. Using strong passwords, not exposing ports, and keeping it updated is usually enough for a personal setup.

- **Costs of Maintenance:** Most maintenance tasks (updating software, adding faces) cost only some time. The system doesn't inherently require additional funds. Possibly if new hardware is needed (replacing a camera or SD card after years) that's a small cost. The design avoids any subscription or cloud costs, so you're mainly maintaining hardware and the data. If something breaks, you can likely troubleshoot or replace cheap components. The largest investment is the initial one; after that, costs to maintain are minimal – mostly ensuring everything stays current and functional.

In conclusion, maintaining this home security system is quite manageable. We planned for automated updates (blacklist via FBI API) [8] and ease of adding new data. The heavy lifting (face recognition model) is stable and shouldn't require frequent tuning. As long as we keep the system running and adjust to any

changes in our environment (new faces, changed looks, etc.), the system will continue to provide a smart security watch at our door for years at a very low ongoing cost.

Conclusion

We have conducted a deep-dive analysis of building a doorbell-triggered face recognition home security system, covering all phases from initial planning to deployment and maintenance. At each step, we compared alternatives and justified our choices:

- **Planning:** We defined the system's goals (face recognition on doorbell press, unknown alerts, FBI wanted checks) and prioritized a local, low-cost solution over cloud-based ones for privacy and cost reasons [3] [13].
- **Design:** We chose hardware (Raspberry Pi 4 and camera) for edge processing power at low cost, and decided on an accurate deep-learning face recognition approach (dlib via `face_recognition`) for reliability [20] while discussing simpler alternatives (OpenCV LBPH) [19]. We designed how whitelist/blacklist face databases would work, and opted for Telegram messaging for free, instant notifications [25] [32]. We also planned the system architecture to be modular and secure.
- **Building:** We outlined assembling the hardware (mounting camera, wiring the button) and setting up the software environment (Python, needed libraries, etc.). We detailed how to implement the code – capturing images, detecting and recognizing faces, and handling each case with appropriate actions (green light for known [33], Telegram alerts for unknown or blacklisted with different urgency [32]). We also considered the pros/cons of various coding strategies (like using Docker vs direct install, or picamera vs OpenCV for capture) and chose what best fits a low-cost Pi setup.
- **Testing:** We emphasized testing each component and scenario – verifying recognition accuracy with different people and conditions, measuring system response time, and checking the robustness (e.g., multiple quick presses, no-face scenarios). We noted adjustments if tests reveal issues (such as adding training data if lighting causes misses [41] or tightening thresholds to avoid false alarms).
- **Deployment:** We covered how to install the system in a real home environment, ensuring it autostarts on boot and stays running. We talked about physical considerations (camera placement, weatherproofing, network setup) and making the system user-friendly for the homeowner. We also kept in mind the cost: after deployment, there are basically no recurring fees, which is a big advantage over many commercial systems.
- **Maintenance:** We planned for longevity – updating the software and face databases, periodically pulling FBI wanted list updates [8], and handling changes (new family members, appearance changes). We also discussed monitoring the system's health and expanding or improving it in the future (without incurring big costs).

Throughout this analysis, we leveraged multiple sources to validate our decisions and to cite performance and accuracy figures. For instance, we cited real project examples (like the Instructables doorbell system) to ensure our design is grounded in proven concepts [1] [27]. We also used technical references (thinkrobotics guide, etc.) to back up statements about Raspberry Pi capabilities and model accuracy [4] [20]. By examining alternatives at each juncture (hardware options, face recognition methods, notification channels, etc.), we ensured that the final choices are well-justified for a low-cost, DIY scenario.

In conclusion, building this home security system from scratch is entirely feasible with modest resources and programming effort, thanks to readily available technology and open-source software. The end result is a smart doorbell that not only rings, but "knows" who's at the door – greeting familiar faces and warning

you of strangers or potential threats. And importantly, it achieves this with one-time hardware purchases and free software, aligning perfectly with the low-cost requirement. With careful planning, solid design choices, and thorough testing as outlined above, one can successfully implement the described system and enjoy an enhanced level of security and convenience at their doorstep.

Sources:

- Erientes, *Doorbell With Face Recognition* – Instructables project (used for inspiration on system workflow and hardware setup) [1] [27].
- ThinkRobotics, *OpenCV Face Recognition Raspberry Pi: Complete Setup Guide 2025* – (provided insights on Pi 4 performance and accuracy expectations) [4] [17].
- Ahmed Yasin, *Smart Security System using Facial Recognition with Raspberry Pi 4* – TheEngineeringProjects (demonstrated an OpenCV/LBPH approach as an alternative) [19].
- FBI Wanted API Documentation – (for integrating an updated most-wanted persons database into the system) [8].
- API4AI Medium Blog, *Edge AI Cameras vs Cloud* – (helped compare edge vs cloud cost and latency considerations) [52] [14].
- Raspberry Pi Documentation and Forums – (for best practices on camera setup, autostart, and general Pi usage).
- Project-specific code references from face_recognition and telegram bot libraries (as linked in the Instructables) [20] [35].

These sources and references guided the design decisions and corroborated the feasibility and advantages of our chosen approach. By following this comprehensive plan and utilizing the cited resources, one should be able to implement the entire project successfully from scratch. Enjoy building your smart face-recognizing doorbell security system, and stay safe! [27] [3]

---

[1] [2] [5] [6] [7] [9] [15] [20] [22] [23] [25] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [44] [45] [46] [47] [50] Doorbell With Face Recognition : 7 Steps (with Pictures) - Instructables
https://www.instructables.com/Doorbell-With-Face-Recognition/

[3] [4] [16] [17] [41] [42] [48] OpenCV Face Recognition Raspberry Pi: Complete Setup Guide 2025 – ThinkRobotics.com
https://thinkrobotics.com/blogs/learn/opencv-face-recognition-raspberry-pi-complete-setup-guide-2025?srsltid=AfmBOooqXF3COGjQ5HUz7G6t3ADTxHkQrwZFSQb6Hc0HGg-BWMuXX2nA-

[8] [49] Wanted API — FBI
https://www.fbi.gov/wanted/api

[10] [11] [12] [13] [14] [52] Edge AI vs Cloud: What Leaders Must Know | API4AI | Medium
https://medium.com/@API4AI/edge-ai-cameras-vs-cloud-balancing-latency-cost-reach-7e660131977f

[18] Face Recognition Using Local Binary Patterns Histogram Method ...
https://www.researchgate.net/publication/377088835_Face_Recognition_Using_Local_Binary_Patterns_Histogram_Method_Using_Raspberry_PI

[19] [26] Smart Security System using Facial Recognition with Raspberry Pi 4 - The Engineering Projects
https://www.theengineeringprojects.com/2022/05/smart-security-system-using-facial-recognition-with-raspberry-pi-4.html

21 face_recognition performance on Raspberry PI and Jetson Nano
https://github.com/ageitgey/face_recognition/issues/1379

24 Amazon Rekognition: Image and Video Analysis with AI - DataCamp
https://www.datacamp.com/tutorial/amazon-rekognition

43 ESP32-CAM Face Recognition Door Lock System - Circuit Digest
https://circuitdigest.com/microcontroller-projects/esp32-camface-recognition-door-lock-system

51 Ring Door Bell Snapshot for Facial recognition - Configuration
https://community.home-assistant.io/t/ring-door-bell-snapshot-for-facial-recognition/135777