PORTAL

A Cryptographically Gated Peer-to-Peer Video Communication System with Biometric Identity Verification and On-Chain Access Control

Max Tindall

June 2026 | v0.1.0

ABSTRACT

Portal is a real-time, browser-based peer-to-peer video communication platform that enforces access control through cryptographically signed portal keys, optionally anchored to the Solana blockchain via a Rust smart contract compiled with the Anchor framework. Access to a portal session is gated by key ownership: only the verified keyholder may initiate a room, while guests join freely by knowing the shared code. The system augments access control with continuous biometric face verification, performed by a dedicated Python microservice implementing a YuNet deep neural network face detector and a MobileFaceNet ONNX recognition model. Verified identities are spatially annotated in real time through a WebRTC data channel, surfacing matched portal keys and guest designations directly on the video canvas. The architecture operates across two independently deployed cloud services, communicating through a Node.js signalling and proxy layer, with face enrollment data persisted on a dedicated block-storage volume. This paper provides a complete technical account of Portal's architecture, technology stack, data flows, security model, and deployment infrastructure.

1. INTRODUCTION

Conventional video conferencing platforms authenticate participants through account credentials managed by a centralised identity provider. Portal takes a different approach: access is governed entirely by possession of a portal key — an eight-character alphanumeric token that may be issued by an administrator off-chain or purchased on-chain through a Solana smart contract. No accounts, passwords, or third-party OAuth flows are required. The initiator of a session must present a valid key; their counterpart joins as a guest by knowledge of the shared code alone.

Beyond key-based gate-keeping, Portal continuously verifies that the face visible on the initiator's camera matches the face enrolled against their key at registration time. This biometric layer is implemented without any client-side machine-learning library: all inference is performed server-side by a Python FastAPI service, preserving low browser overhead and keeping model weights off the

client entirely. Verified identities are communicated back to the browser and rendered as annotated bounding boxes on the live video feed via an HTML5 Canvas element overlaid on the WebRTC video stream.

The following sections describe each architectural layer in full technical detail, from the WebRTC signalling protocol and Socket.IO room management, through the face recognition pipeline, to the Solana program design and Render cloud deployment configuration.

2. SYSTEM ARCHITECTURE

Portal is composed of three logical tiers, each independently deployable:

- The signalling and proxy server — a Node.js process exposing HTTP REST endpoints and a Socket.IO namespace, responsible for room lifecycle management, key validation, and reverse-proxying requests to the face service.
- The face recognition microservice — a Python FastAPI application that loads ONNX model weights at startup and services image-bearing HTTP POST requests for face detection, enrollment, verification, and 1:N identification.
- The Solana program — an on-chain Rust program compiled with Anchor that records key purchases as on-chain account state, enabling trustless verification that a key was legitimately purchased rather than administratively issued.

The browser client communicates with the signalling server over WebSockets (Socket.IO) for room coordination and over HTTP for face service requests, which are proxied through the Node.js layer. Peer-to-peer audio and video are transmitted directly between browsers using WebRTC, negotiated through the ICE/STUN/TURN infrastructure configured on the signalling server. An auxiliary data channel, established alongside the media channel, carries face bounding box metadata and UI state synchronisation messages between peers without routing through the server.

2.1 Communication Flow

The connection lifecycle proceeds as follows. A prospective initiator enters a portal key in the browser and invokes the join procedure. The browser captures a frame from the local camera, submits it to /face/identify via an HTTP POST, and simultaneously emits a join event to the Socket.IO namespace. The server validates the key — first against an in-memory Set of administrative keys loaded from the face service's persistent disk at startup, and subsequently against on-chain KeyRecord accounts if a PROGRAM_ID environment variable is configured. If the key is invalid, an invalid_key event is emitted, the status element turns red, and the page reloads after 2.5 seconds.

If valid and the room has capacity, the server assigns a user number and notifies both parties to begin WebRTC negotiation. The second participant (the guest) requires no key validation — the room's existence, indexed by the shared code, is sufficient proof of the initiator's prior authorisation.

Once the WebRTC peer connection is established, both parties exchange SDP offers and ICE candidates through the Socket.IO relay. A data channel is opened on the connection and used to transmit serialised JSON messages of the following types: face_box (bounding box coordinates and matched identity), ui_state (camera toggle, video minimise, chat maximise), chat (plaintext messages), and remote_claim (remote control assignment).

3. SIGNALLING AND PROXY SERVER

3.1 Runtime and Framework

The signalling server is implemented in Node.js (runtime version determined by Render's free-tier environment) using Express 4.21.2 for HTTP routing and Socket.IO 4.8.3 for WebSocket management. The server listens on a single port, serving static assets from the public/ directory alongside its API surface. The entry point is server.js, which conditionally loads the portal-keys.js module when the PORTAL_KEYS_ENABLED environment variable is set to true.

3.2 Key Validation Module (portal-keys.js)

portal-keys.js encapsulates all key-related logic. On module load, it establishes a Solana RPC connection using @solana/web3.js 1.91.1 and instantiates an Anchor 0.29.0 provider against the configured program address. An in-memory Set named adminKeys is populated from three sources in order of precedence: (1) the ADMIN_KEYS environment variable, a comma-separated list of permanent keys that survive all redeploys; (2) the admin-keys.json file persisted on the face service's block-storage volume, fetched via HTTP from the face service at startup; and (3) keys generated at runtime via the /admin/generate-key endpoint, which are immediately synced to the face service for durable storage.

The validateKey function first performs an O(1) Set membership test. If the key is absent, it issues a fresh HTTP GET to the face service's /admin-keys endpoint, repopulates the Set, and retests — accommodating the race condition where the signalling server starts before the face service is ready. If PROGRAM_ID is configured and the key is still unmatched, it fetches all KeyRecord program-derived accounts from the Solana cluster and scans their account data for a matching portal key string.

Portal keys follow a strict format: eight characters drawn from the unambiguous charset ABCDEFGHJKLMNPQRSTUVWXYZ23456789 (omitting I, O, 0, and 1 to prevent visual confusion). Guest face enrollments are stored under a composite key of the form BASEKEY:gN, where N is an auto-incremented integer. The base key is stripped before returning match results so that both the keyholder and any enrolled guests are associated with the same portal key in the UI.

3.3 HTTP API Surface

The server exposes the following REST endpoints:

POST /admin/auth Validates the x-admin-key header against the ADMIN_KEY environment variable. Returns 200 on success, 401 on failure. Used by the admin panel to gate the unlock flow.

POST /admin/generate-key Generates a new eight-character portal key, adds it to the in-memory Set, and syncs it to the face service's persistent admin-keys.json. Accepts either x-admin-key or x-admin-secret authentication headers.

GET /admin/enrollments Fetches the enrolled key list from the face service and cross-references each key against the adminKeys Set and the on-chain program (when PROGRAM_ID is set) to determine whether each key is private or on-chain, returning this classification to the admin panel.

DELETE /admin/enrollments/:key Removes a face enrollment from the face service. Restricted to private (administrative) keys; on-chain keys cannot be deleted through this endpoint.

POST /face/enroll Admin-authenticated proxy to the face service /enroll endpoint.

POST /face/enroll-guest Unauthenticated proxy to the face service /enroll-guest endpoint, used during live calls.

POST /face/identify Proxy to the face service /identify endpoint. Accepts a base64-encoded JPEG and returns face bounding boxes with matched portal keys.

POST /face/verify Proxy to /verify; performs 1:1 matching against a specific portal key.

POST /face/detect Proxy to /detect; returns bounding boxes without identity resolution.

GET /face/health Proxy to the face service /health endpoint; used for pre-warming the cold-start container.

GET /ice-config Returns the ICE server configuration as JSON, including STUN and optionally TURN server URLs and credentials drawn from environment variables.

3.4 Socket.IO Room Management

Rooms are maintained as an in-memory dictionary keyed by portal code. Each room holds an ordered array of up to two socket IDs. The first socket to join a room is the initiator (user number 1); the second is the guest (user number 2). When both parties are present, the server emits peer events with initiator flags to begin WebRTC negotiation. On disconnect, the peer is notified via a peer_left event that causes an immediate page reload on both sides. The server also relays face_warning and face_verified socket events between peers to synchronise the identity status display.

4. FACE RECOGNITION MICROSERVICE

4.1 Technology Selection

The face recognition service was developed through several iterations driven by the memory constraints of Render's Starter plan (512 MB RAM). The initial implementation using DeepFace and TensorFlow was abandoned because TensorFlow alone consumes approximately 500 MB at import time, leaving insufficient headroom for inference. A subsequent attempt using the face_recognition library (which wraps dlib) failed at build time: dlib is compiled from C++ source during pip install, exhausting the 8 GB build disk allotment. A third iteration using the insightface framework crashed at runtime due to its own memory overhead.

The final stack — onnxruntime with opencv-python-headless — resolves all three constraints. Both packages ship as pre-compiled wheels, incurring no build-time compilation. Runtime memory consumption peaks at approximately 150–200 MB, well within the 512 MB limit. All model weights are downloaded at build time via a dedicated download_models.py script and stored in the face_service/models/ directory, which is part of the project's rootDir and therefore persists from build container to runtime container on Render.

4.2 Model Architecture

Two ONNX models are used in tandem:

- YuNet (face_detection_yunet_2023mar.onnx, approximately 350 KB) — a lightweight convolutional neural network designed for real-time face detection. It is instantiated via OpenCV's cv2.FaceDetectorYN interface with a score threshold of 0.45 (lowered from the default 0.6 to improve detection under low-light conditions), an NMS threshold of 0.3, and a top-k limit of 5000 candidate anchors. The model processes each frame at the native resolution of the submitted image, returning bounding boxes with landmark coordinates that are subsequently clamped to image boundaries. YuNet was sourced from the OpenCV Zoo repository.
- MobileFaceNet (w600k_mbf.onnx, approximately 17 MB) — a compact face recognition network trained on the WebFace600K dataset, extracted from InsightFace's buffalo_s model pack. The model accepts a 112x112 BGR image normalised to the range [-1, 1] and outputs a 512-dimensional embedding vector. Embeddings are L2-normalised before storage and comparison. The model is loaded once at startup via onnxruntime.InferenceSession with the CPUExecutionProvider, eliminating any GPU dependency.

4.3 Enrollment and Matching

Face data is stored as a JSON object on the persistent disk at the path specified by the FACE_DATA_FILE environment variable (default: /data/encodings.json). Each entry maps a portal key string to its 512-

dimensional embedding, serialised as a JSON array of floats. Guest embeddings are stored under composite keys of the form BASEKEY:gN.

Identity matching is performed via cosine similarity: the dot product of the query embedding against each stored embedding (both L2-normalised, making the dot product equivalent to cosine similarity). The match with the highest similarity score is returned if it exceeds the MATCH_THRESHOLD of 0.40. The /identify endpoint performs this 1:N scan across all enrolled keys for every detected face in a submitted image, returning a list of face records each containing normalised bounding box coordinates (relative to image dimensions), the matched portal key (with :gN suffix stripped), an is_guest boolean, and the confidence score.

4.4 FastAPI Endpoints

POST /enroll Detects all faces in the submitted image, selects the largest by bounding box area, extracts its embedding, and stores it under the provided portal_key. Returns 422 if no face is detected.

POST /enroll-guest Identical to /enroll but automatically assigns the next available guest slot (BASEKEY:g1, BASEKEY:g2, ...) for the given portal key.

POST /identify Detects all faces and performs 1:N matching for each. Returns an empty faces list if image_b64 is absent, preventing 400 errors from pre-warm calls.

POST /verify Performs 1:1 matching for a specific portal_key. Returns match: true with reason: no_enrollment if the key has no stored embedding, permitting calls to proceed for unenrolled keys.

POST /detect Returns bounding boxes only, without identity resolution.

GET /enrolled Returns the count and list of all enrolled keys, including guest sub-keys.

DELETE /enrolled/{portal_key} Removes a specific enrollment by exact key match.

GET /admin-keys Returns the list of portal keys stored in admin-keys.json on the persistent disk.

POST /admin-keys Appends a key to admin-keys.json. Called by the signalling server each time a new key is generated.

DELETE /admin-keys/{key} Removes a key from admin-keys.json.

GET /health Returns detector and recognizer load status as booleans.

4.5 Python Dependencies

fastapi 0.111.0 ASGI web framework for the REST API surface.

uvicorn 0.30.1 ASGI server; started with --workers 1 to prevent multiple model loads.

onnxruntime Cross-platform ONNX inference engine (CPU provider only).

opencv-python-headless Computer vision library providing the YuNet FaceDetectorYN interface and image I/O.

Pillow >= 10.3.0 PIL-compatible image decoding for base64-encoded JPEG inputs.

numpy >= 1.26.4 Array operations for embedding arithmetic and cosine similarity computation.

pydantic >= 2.7.1 Request schema validation for FastAPI endpoints.

5. BROWSER CLIENT

5.1 Architecture

The browser client is a single-page application served as static HTML from public/index.html. It has no build step, no frontend framework, and no client-side dependencies beyond the Socket.IO client library. All interactivity is implemented in approximately 1,500 lines of vanilla JavaScript embedded directly in the HTML file. This approach eliminates bundle tooling complexity and keeps the client footprint minimal.

5.2 WebRTC Implementation

Peer connections are established using the browser's native RTCPeerConnection API. ICE server configuration is fetched from /ice-config at connection time, supporting both STUN and TURN servers. The initiating peer creates an offer, and both peers exchange SDP and ICE candidates through the Socket.IO relay. A separate RTCDataChannel named portal-data is opened on the connection and used to transmit JSON messages without server involvement, reducing latency and eliminating server load for in-call communication.

Local camera and microphone are acquired via navigator.mediaDevices.getUserMedia. Camera frames are captured at a fixed resolution of 480x360 pixels into a persistent off-screen canvas element for face analysis. The analysis loop is adaptive: each iteration is scheduled with setTimeout after the previous HTTP round-trip completes, preventing request pile-up when the face service is slow.

5.3 Face Verification Loop

Every five seconds (or after the previous response returns, whichever is later), the client captures a JPEG frame from the local video element at 0.7 quality and POSTs it to /face/identify. The response contains an array of face records. For the local user, the largest detected face is selected and its bounding box and matched identity are sent to the peer via the data channel as a face_box message. The peer renders this on an HTML5 Canvas overlay sized and positioned to match the remote video element.

The canvas overlay renders each face box using the colour #004225 (dark green) for both the bounding rectangle and the label background. The label displays two lines: the matched portal key on the first line, and either a check mark with 'identified' or '· guest' on the second line depending on the is_guest flag. When no face is matched, the current portal code is displayed with a sequential peer number.

Identity status is reflected in the connection status element beneath the peer counter. When the face service returns a verified match, the element displays 'connected' with the portal key rendered in #00ff88 (green). A mismatch renders the key in #ff4444 (red) and emits a

face_warning socket event that causes the same red display to appear on the peer's screen. The socket._faceResult variable persists the current verification state across rescan cycles.

5.4 Keyboard Shortcuts

Space Toggle microphone mute / unmute.

Cmd+D Toggle local camera on / off.

Cmd+V Minimise or restore the local video tile.

Cmd+C Maximise or restore the chat panel.

Cmd+S Toggle remote control mode.

Cmd+I Force an immediate identity rescan (calls /face/verify with current camera frame).

Cmd+G Context-sensitive: for the initiator (user 1), triggers identity re-validation (rescueIdentity); for the guest (user 2), enrolls the guest’s face from their own local camera under the current portal key.

Cmd+R Disconnect and reload.

5.5 Guest Enrollment Flow

When the guest (user 2) presses Cmd+G, the enrollGuest function captures a frame from the guest’s own local video element and POSTs it to /face/enroll-guest with the current portal code as the portal_key parameter. A 10-second AbortController timeout guards against cold-start hangs on the face service; if exceeded, the UI displays “ENROLLMENT TIMEOUT — retry”. The face service detects the largest face in the frame, extracts its MobileFaceNet embedding, and stores it under the next available guest slot (e.g., ABCD1234:g1). On success, the connection status element turns green, socket._faceResult is set to ‘verified’, and a local face rescan is triggered after 400 ms to confirm identity immediately. If myUserNumber has not yet been assigned via the Socket.IO assigned event, the keydown handler exits early to prevent a race-condition mismatch.

5.6 Role Labelling

Both video corner labels and in-call chat messages display role identifiers derived from a single roleLabel helper function. The initiator is labelled #PORTALKEY (e.g., #ABCD1234) and the guest is labelled #PORTALKEY-G (e.g., #ABCD1234-G). These labels appear on the video overlay positioned at the bottom corners of each video element, and as a subdued header above each chat message. Role assignment is determined by the userNumber value received in the Socket.IO assigned event.

6. ADMINISTRATION PANEL

The administration panel is served from public/admin.html as a separate static page. It is protected by an auth gate that validates the entered key against the server's /admin/auth endpoint before revealing the management interface. The ADMIN_KEY environment

variable, auto-generated by Render at first deployment, serves as the panel password and is distinct from portal keys used for room access.

Upon successful authentication, the panel presents two sections. The Face Service section fetches the enrolled key list from /admin/enrollments and renders each entry with its chain classification (on-chain or private), a copy-to-clipboard button, and — for private keys — a delete button. Guest enrollments are indented beneath their parent key, labelled as 'guest'. The enrollment count is displayed in the section header.

The Portal Key section allows an administrator to generate a new portal key by clicking the GENERATE button, which POSTs to /admin/generate-key with the x-admin-key header. The generated key is displayed and held in a currentPortalKey variable. The ENROLL button becomes active only when both a key has been generated and a face image has been captured or uploaded. Face images may be provided via live camera capture or drag-and-drop file upload. On enrollment, the key and its face embedding are stored on the face service's persistent disk and the enrollments list refreshes automatically.

7. SOLANA SMART CONTRACT

7.1 Program Design

The on-chain access control layer is implemented as a Solana program written in Rust using the Anchor 0.29.0 framework. The program defines a single account type, KeyRecord, which is a Program-Derived Account (PDA) seeded with the bytes 'key_record' concatenated with the buyer's public key. This seed scheme ensures that each wallet address can hold at most one KeyRecord at a time, preventing duplicate purchases.

The KeyRecord struct contains four fields: owner (PublicKey, 32 bytes), portalKey (String, variable length), purchasedAt (i64, Unix timestamp), and used (bool). The total serialised size is 93 bytes, a constant used as a filter when scanning program accounts.

7.2 Instructions

The program exposes two instructions. purchase_key accepts a transfer of exactly 100,000,000 lamports (0.1 SOL) from the buyer to the treasury wallet and initialises the KeyRecord PDA with a randomly generated eight-character key string using the same character set as administrative keys. mark_used is an authority-gated instruction that sets the used flag to true on a KeyRecord, intended for single-use key revocation workflows.

7.3 Client-Side Integration

The buy.html page in the public directory provides a wallet-connected purchase flow. It uses @solana/web3.js and the Phantom wallet adapter

to construct and sign the purchase_key transaction client-side. The serialised transaction is submitted to the Solana RPC endpoint and confirmed before the server's listenForKey polling function detects the new KeyRecord account and returns the portal key to the buyer via a session-keyed polling endpoint (/key/:sessionId).

7.4 Server-Side Validation

When PROGRAM_ID is configured, the signalling server validates on-chain keys by calling connection.getProgramAccounts with a dataSize filter of 93 bytes and scanning each account's binary data at fixed offsets: the portal key string at byte 44 (preceded by a four-byte little-endian length prefix at byte 40) and the used flag at byte 44 + strLen + 8. A key is considered valid if it matches the submitted code and the used flag is false.

8. DEPLOYMENT INFRASTRUCTURE

8.1 Render Cloud Services

Both services are deployed on Render and defined in a single render.yaml Infrastructure-as-Code file. The Node.js signalling server runs on Render's free tier with npm install as its build command and node server.js as its start command. The Python face service runs on Render's Starter plan (512 MB RAM, 0.5 CPU) with a composite build command:

```
pip install -r requirements.txt && python download_models.py
```

and the start command:

```
uvicorn main:app --host 0.0.0.0 --port $PORT --workers 1
```

The --workers 1 flag is critical: multiple workers would each load the ONNX models independently, causing RAM exhaustion. The workers parameter in uvicorn.run() within the __main__ block is insufficient to prevent this — only the CLI flag reliably enforces a single worker process.

8.2 Persistent Disk

A 1 GB persistent disk is attached to the portal-face service and mounted at /data. This volume survives service redeploys, preserving both face enrollment data (encodings.json) and the administrative key registry (admin-keys.json). The FACE_DATA_FILE environment variable points to /data/encodings.json. Importantly, the disk is attached exclusively to the face service; the signalling server has no direct disk access and reads admin keys via HTTP at startup.

8.3 Environment Variables

PORTAL_KEYS_ENABLED Boolean flag; when true, loads portal-keys.js and enforces key validation on join.

ADMIN_KEY Auto-generated by Render; used to authenticate admin panel requests.

ADMIN_SECRET Auto-generated; legacy secret for key generation endpoint.

ADMIN_KEYS Comma-separated list of permanent portal keys that survive all redeploys.

FACE_SERVICE_URL Base URL of the deployed face service, used for all proxy and admin-key sync calls.

SOLANA_RPC_URL Helius or mainnet RPC endpoint for on-chain key validation and purchase handling.

PROGRAM_ID Deployed Solana program address; when absent, only administrative keys are accepted.

TREASURY_PUBKEY Wallet address that receives SOL from key purchases.

TURN_URL_UDP / TURN_URL_TCP TURN server URLs for NAT traversal in restrictive network environments.

TURN_USERNAME / TURN_CREDENTIAL TURN server authentication credentials.

FACE_DATA_FILE Path to encodings.json on the persistent disk (default: /data/encodings.json).

8.4 Model Distribution

download_models.py is executed at build time, downloading the YuNet ONNX model from the OpenCV Zoo GitHub repository and the MobileFaceNet ONNX model by fetching the buffalo_s.zip archive from the InsightFace GitHub releases and extracting the single w600k_mbf.onnx member. Both files are written to face_service/models/, which is within the project rootDir and therefore present in the runtime container. Download is skipped if the files already exist, allowing Render's build cache to avoid redundant downloads on subsequent deployments.

9. SECURITY CONSIDERATIONS

Portal's security model rests on two complementary mechanisms. Key-based access control ensures that only holders of a valid portal key may initiate a session. Keys may be administratively issued (private) or on-chain verified, the latter providing a trustless proof of purchase that does not require server-side record-keeping. The distinction between private and on-chain keys is surfaced in the admin panel but does not affect the real-time validation path.

Biometric verification provides a second layer: the face enrolled against a key at registration time must match the face presented on the live camera. This guards against key sharing — a valid key used by an unrecognised face will trigger a mismatch warning visible to both participants. The 0.40 cosine similarity threshold was selected to balance false positive rejection (too strict) against false acceptance (too permissive) for the MobileFaceNet embedding space.

The admin panel is protected by a server-side secret (ADMIN_KEY) that is never exposed to the client beyond the duration of the authenticated session. Face service endpoints are proxied through the Node.js server, ensuring that the face service URL is never disclosed to browser clients. The face service itself performs no

authentication, relying on network isolation — direct access is expected to be unavailable from the public internet in production configurations.

Known limitations include the in-memory nature of the adminKeys Set on the signalling server (mitigated by face-service-backed persistence), the absence of end-to-end encryption on the Socket.IO relay (mitigated by TLS at the transport layer), and the susceptibility of cosine-similarity face matching to adversarial photographs presented to the camera (a limitation shared with all appearance-based biometric systems operating without liveness detection).

10. COMPLETE TECHNOLOGY STACK

Frontend

- HTML5 / CSS3 / Vanilla JavaScript (ES2022)
- WebRTC (RTCPeerConnection, RTCDataChannel, getUserMedia)
- Socket.IO 4.8.3 (client)
- HTML5 Canvas API (face overlay rendering)
- Web Clipboard API (admin panel copy function)

Signalling Server

- Node.js (LTS)
- Express 4.21.2
- Socket.IO 4.8.3 (server)
- @solana/web3.js 1.91.1
- @coral-xyz/anchor 0.29.0
- Node.js built-ins: crypto, fs, path, http

Face Recognition Service

- Python 3.10
- FastAPI 0.111.0
- Uvicorn 0.30.1
- ONNX Runtime (CPU)
- OpenCV (opencv-python-headless) — YuNet FaceDetectorYN
- NumPy >= 1.26.4
- Pillow >= 10.3.0
- Pydantic >= 2.7.1

Machine Learning Models

- YuNet (face_detection_yunet_2023mar.onnx) — OpenCV Zoo, ~350 KB
- MobileFaceNet (w600k_mbf.onnx) — InsightFace buffalo_s, ~17 MB
- 512-dimensional L2-normalised face embeddings
- Cosine similarity matching, threshold 0.40

Blockchain

- Solana (mainnet-beta / devnet)

Rust (Anchor 0.29.0 framework)
Program-Derived Accounts (PDA) for key storage
Helius RPC provider

Infrastructure

Render (cloud PaaS) — free tier (Node.js) + Starter (Python)
Render persistent disk — 1 GB block storage at /data
STUN: stun.l.google.com:19302
TURN: operator-configured (UDP + TCP endpoints)
GitHub (source control and CI/CD trigger)
render.yaml (Infrastructure-as-Code)

Languages

JavaScript (Node.js server, browser client)
Python (face recognition microservice)
Rust (Solana smart contract)
HTML / CSS (client UI and admin panel)
YAML (deployment configuration)

11. CONCLUSION

Portal demonstrates that a cryptographically gated, biometrically verified peer-to-peer video communication system can be constructed from commodity open-source components and deployed within the constraints of low-cost cloud infrastructure. The separation of concerns across a JavaScript signalling layer, a Python inference service, and an on-chain Rust program allows each component to be developed, deployed, and scaled independently. The use of pre-compiled ONNX model weights eliminates build-time compilation requirements and keeps runtime memory within the 512 MB boundary of the Render Starter plan.

The guest enrollment mechanism extends the keyholder model to support recognisable recurring participants without requiring each guest to purchase or receive their own portal key, preserving the asymmetric access structure that defines the system's trust model. Future work may include liveness detection to mitigate photograph-based spoofing, end-to-end encrypted signalling, multi-face enrollment per key, and expansion of the on-chain program to support transferable or time-bounded key validity.