LabGuard

Saturday, September 13, 2025 2:14 AM

Detailed system architecture — LAB-Guard

(Desktop Electron clients on lab PCs) — Local VM during dev → University on-prem VM for final deployment

Below is a complete, actionable architecture you can implement end-to-end: components, dataflow, storage, offline sync model, API shapes, security details, deployment + migration steps, and developer checklist for iteration 1 (due Oct 17). I'll give concrete examples (DB schema sketches, API endpoints, signing/encryption choices, docker-compose snippet) so you can start coding immediately.

1. Overall high-level architecture (text + ASCII diagram)

Clients (Electron, Windows lab PCs) ⇄ On-prem Server (Dev VM / Production university VM)

```
+----------------------+          HTTPS / TLS + Signed Bundles          +----------------------+
| Electron Client(s)   |  <==============================================> |  Backend Server      |
| (each lab PC)        |          Sync Agent / Upload & ACK protocol     | (VM: Ubuntu)         |
| - Local SQLite DB    |                                                  | - REST API (FastAPI)|
| - CV process (Py)    |                                                  | - Postgres DB        |
| - App monitor        |                                                  | - File store (FS /   |
| - Rule engine        |                                                  |   MinIO)             |
+----------------------+                                                  +----------------------+
         |                                                                          |
         | Local UI for teacher/student                                             | Teacher Portal (Web)
         | (start exam, consent)                                                    | Reports, Download evidence
         v                                                                          v
   Local evidence (AES-GCM encrypted)                                     Admin UI / Audit logs
```

Key points:

- Clients are **offline-first**: they store events locally and only sync when the server is reachable.
- Evidence (camera snapshots) is stored encrypted on the client and uploaded in encrypted bundles.
- Bundles are **signed** by the client (device private key) so server can detect tampering.
- Server stores metadata in Postgres and evidence blobs on filesystem/MinIO.

2. Components (what to build)

2.1 Client (Electron app on Windows lab PCs)

- UI (Electron + React)
    - Login (student/teacher)
    - Teacher exam creation/upload

    - Student start/stop exam, consent dialog
    - Dashboard: current suspicion score (for teacher view), sync status
- Local DB: **SQLite**
    - store events, device state, pending upload queue
- Monitoring modules:
    - App/process monitor (Node native or Windows APIs): active window title, process names, timestamps
    - Keystroke *metadata* collector: only timings / density (never raw keystrokes)
    - Camera module: runs as Python child process (OpenCV + MediaPipe) for:
        - face detection & verification at start
        - gaze/brief glance detection + flagging
        - capture evidence snapshot when rule triggers
- Rule Engine (Node):
    - consumes events, computes per-student suspicion score
    - creates flagged events with explanations
- Local storage of evidence:
    - images stored encrypted (AES-GCM) and referenced in SQLite by event row
- Sync Agent:
    - packages pending events & evidence into a signed, compressed bundle and sends to server
- Device provisioning:
    - during install, client receives/generates a device private key and registers with server

2.2 Server (Dev VM → University VM)

- REST API (FastAPI or Express)
    - authentication, upload endpoints, teacher portal endpoints
- DB: **Postgres**
    - users, devices, exams, events, suspicion reports, audit entries
- File store:
    - encrypted bundle storage on filesystem or MinIO (S3-compatible)
- Teacher web portal (React)
    - exam management, report viewer, evidence download
- Admin UI:
    - device revocation, audit trail, retention management
- Optional: background worker for post-processing (code plagiarism, report generation)

3. Data model (schemas sketches)

3.1 SQLite (client-side) — simplified

Tables:

- device { device_id TEXT PRIMARY KEY, public_key TEXT, registered_at TIMESTAMP }
- exam_state { exam_id TEXT, started_at, teacher_id, allowed_apps JSON, status }
- events { event_id TEXT PRIMARY KEY, exam_id TEXT, timestamp UTC, type TEXT, payload JSON, evidence_path TEXT, synced BOOLEAN DEFAULT 0 }
- bundles { bundle_id, created_at, signed_by_device boolean, upload_status }

Example events.payload types: {window: "Code.exe - main.py", process: "Code.exe"}, {keystroke_density: 0.23}, {face_verified: true}, {gaze_away_count: 4}

3.2 Postgres (server-side) — simplified

- users (user_id, name, role, email, hashed_password, created_at)
- devices (device_id, device_pubkey, owner_user_id, registered_at, revoked BOOLEAN)

- exams (exam_id, teacher_id, title, pdf_path, allowed_apps JSON, start_time, end_time)
- events (event_id, exam_id, student_id, device_id, timestamp, type, payload JSON, evidence_blob_id, received_at)
- evidence_blobs (blob_id, path, size, checksum, encrypted BOOLEAN, created_at)
- suspicion_reports (report_id, exam_id, student_id, score, explanation JSON, generated_at, teacher_ack BOOLEAN)
- audit_logs (audit_id, action, actor_user_id, details JSON, timestamp)

4. API design (example endpoints & payloads)

All endpoints over HTTPS. Use JWT for user sessions.

Auth

- POST /api/auth/login -> returns JWT
- POST /api/auth/register-device (device provisioning) -> Accepts device_pubkey and returns device_id and server nonce

Teacher / Exam

- POST /api/exams (teacher uploads metadata + PDF)
  body: {title, start_time, end_time, allowed_apps: ["Code.exe","calc.exe"], pdf_file}

Uploading bundles (client → server)

- POST /api/upload/bundle (multipart or binary)
  headers:
    - X-Device-ID: <device_id>
    - X-Device-Signature: <base64(sig)> — signature over bundle hash
      payload: encrypted_bundle.zip (contains events.json + evidence files + bundle_manifest.json)

Server validates:

1. Device is registered and not revoked.
2. Signature verifies against device public key.
3. Parses bundle_manifest for event IDs and checks for duplicates.

Server response example:

{ "status":"ok", "ack": [{"event_id":"e1","server_event_id":"s123"}, ...] }

Teacher portal

- GET /api/exams/{exam_id}/reports — list reports & evidence links
- GET /api/evidence/{blob_id} — download encrypted blob (teacher must have rights)

5. Encryption & signing (details)

5.1 Evidence encryption (client-side)

- Algorithm: **AES-256-GCM**
- Key management: on install, generate a random symmetric key (K_client), store in OS-protected store. For extra security, device private key can encrypt K_client for safe transfer. For FYP, storing a random key in local app data with file system permissions is acceptable if you document tradeoffs.
- Evidence file storage: file.enc = AES-GCM-Encrypt(K_client, file_bytes, associated_data=event_id)

5.2 Bundle signing

- Each device has an **RSA-2048** keypair generated at provisioning.
- Before upload:
  - Create events.json and include event IDs and checksums for evidence files.
  - Create bundle_hash = SHA256(events.json || concatenated evidence checksums)
  - Sign bundle_hash with device private key: signature = Sign_RSA(privkey, bundle_hash)
  - Upload bundle and include X-Device-Signature header.
- Server verifies signature with device public key.

5.3 Transport

- HTTPS with TLS. If on-prem, use Let's Encrypt or internal CA.

5.4 Tamper detection

- If server receives events that fail signature/checksum verification, mark as suspicious and do not accept them as authoritative.

6. Offline-first sync protocol (clients)

1. Client writes events to local SQLite immediately as they occur.
2. Evidence snapshots saved encrypted to local disk and referenced by event rows.
3. Sync policy:
   - periodic (e.g., every 60s) tries to upload pending events
   - also triggered when student finishes exam / server becomes reachable
4. Upload process:
   - batch N events (configurable; default N=100) into events.json
   - attach associated encrypted evidence blobs
   - compute bundle signature, upload
   - server returns ACK for each event id
   - client marks events as synced and may optionally delete local evidence older than retention period
5. Retries: exponential backoff on failures; keep bundles until ACK.
6. Idempotency: server uses client event IDs (UUIDv4) to avoid duplicates.

Edge cases:

- Partial upload/network failure — client retries; server deduplicates by event ID.
- Device clock skew — use UTC timestamps and server records received_at. For ordering, prefer server timestamps if needed.

7. Retention & privacy policy (recommended defaults to include in SRS)

- Evidence (camera snapshots): retain **30 days** by default; then delete from server unless teacher flags case for review (teacher can request longer retention).
- Event metadata: retain 90 days in DB. Keep minimal metadata (avoid raw keystrokes).
- Consent: prompt student at exam start; record consent event and store on server.
- Deletion: only admin can delete with audit trail (do not implement user-side deletion).

8. Deployment & migration path (Dev VM → University VM)

8.1 Development (local)

- Use Docker + docker-compose for reproducible environment:
  - backend (FastAPI/Express)
  - postgres
  - minio (optional)
  - nginx (reverse proxy for TLS / local testing)
- VM options: VirtualBox / VMware / WSL2. For multi-developer access, expose VM on local LAN (with proper firewall rules).
- Use environment variables for config (DB URL, JWT secret, device provisioning secret).
- Use self-signed certs locally; have config switch to production certs.

8.2 Staging / Final migration to University VM

- Prepare deploy.sh or docker-compose.prod.yml for university server.
- Deliver:
  - Docker images or code repo + deploy scripts
  - Database migration scripts (alembic / flyway)
  - Instructions for provisioning devices (how to inject server URL and register devices)
- Migration steps:
  1. Request VM from university (Ubuntu 22.04 LTS recommended) with network access from lab PCs.
  2. Install Docker and docker-compose.
  3. Transfer docker-compose.prod.yml and environment variable secrets (JWT_SECRET, DB password).
  4. Start containers, run DB migrations.
  5. Optionally, migrate dev DB: pg_dump -> pg_restore (scrub test data).
  6. Provision devices: update client config to point to university server and re-register (or use device re-registration flow).
  7. Issue TLS certs (Let's Encrypt) for the server host.

8.3 Device provisioning & revocation

- On first run: client generates RSA keypair and calls POST /api/auth/register-device with device_pubkey and admin_token (or teacher code during lab provisioning).
- Server creates device record and issues device_id.
- To revoke: admin hits POST /api/admin/revoke-device — server updates devices.revoked = true. Clients with revoked ID are rejected.

9. Technologies / libraries (practical picks)

Client:

- Electron (front-end)
- Node.js (backend of Electron)
- Python child process for CV: OpenCV + MediaPipe
- SQLite (better-sqlite3)
- crypto libs: node-forge or crypto builtin for RSA sign/verify; crypto for AES-GCM

Server:

- FastAPI (Python) or Express (Node)
- PostgreSQL
- MinIO (optional)

- Docker / docker-compose
- Nginx reverse proxy
- PyJWT / jsonwebtoken for JWT handling

10. Example Docker Compose (dev) — minimal

```
version: "3.8"
services:
 db:
  image: postgres:15
  environment:
   POSTGRES_DB: labguard
   POSTGRES_USER: labguard
   POSTGRES_PASSWORD: labguardpass
  volumes:
   - db_data:/var/lib/postgresql/data
 backend:
  build: ./backend
  environment:
   DATABASE_URL: postgres://labguard:labguardpass@db:5432/labguard
   JWT_SECRET: devsecret
  ports:
   - "8000:8000"
  depends_on:
   - db
 minio:
  image: minio/minio
  command: server /data
  environment:
   MINIO_ROOT_USER: minio
   MINIO_ROOT_PASSWORD: minio123
  ports:
   - "9000:9000"
volumes:
 db_data:
```

11. Minimal dev tasks for Iteration 1 (deliverables for Oct 17)

1. Electron client:
   - Login/consent screen
   - Local SQLite storage
   - Simple app-monitoring (active window titles logged)
   - Basic camera module: face detection + start-of-exam face verification (compare embedding to registered image file)
   - Rule engine creates flagged events and stores encrypted snapshots
2. Local dev VM:
   - Backend with /api/upload/bundle endpoint that accepts and verifies signature (can stub verification initially)

    - Postgres for user/exam metadata
    - Teacher can upload an exam PDF via teacher portal
3. Sync:
    - Client can upload a small bundle and server stores metadata & evidence.
4. Documentation:
    - README with provisioning, dev VM setup steps, deploy scripts
    - SRS partial for iteration 1

12. Security checklist (implement these early)

- Generate device keypair at install, never transmit private key
- Sign bundles and verify on server
- AES-GCM encrypt evidence files locally
- TLS for API (self-signed for dev)
- Do not log sensitive data (no raw keystrokes)
- Implement device revocation mechanism
- Record consent as an event

13. Testing plan & acceptance criteria (iteration 1)

- Functional:
    - Student can start exam, face verification succeeds when same person sits in front of camera.
    - App monitoring logs active window switches; rule engine flags using non-allowed app.
    - Client packages events and evidence into a bundle and uploads successfully to dev VM.
- Security:
    - Server rejects bundle if signature invalid.
    - Evidence files are present on server encrypted.
- Resilience:
    - Client can queue events when server unreachable and successfully sync after server becomes reachable.

Security

Sunday, September 28, 2025 6:41 AM

Security & integrity measures (what to implement)

Because you want local processing and tamper-resistance, build these into the design:

1. **Local encryption at rest:** encrypt evidence files (AES-GCM) and protect the key using OS-provided facilities or a per-install key derived from an install secret. For FYP, a locally stored key created at provisioning is fine; note this could be stolen if the attacker has admin access — document the tradeoff.
2. **Signed log bundles:** each time the client uploads a batch, sign it (HMAC with a per-device secret or RSA signature of a private key). Server verifies signature before accepting. This helps detect client-side tampering of logs.
3. **TLS everywhere:** use HTTPS with server certificate (Let's Encrypt if public, or internal CA for on-prem). Do not send sensitive data over plaintext.
4. **Immutable audit trail on server:** do not allow deletion of evidence unless through an audited admin process. Keep DB audit columns.
5. **Secure provisioning:** when installing on lab PCs, create and register a device identity with server (store per-device keys). Use this to revoke a device if compromised.
6. **Least-privilege monitoring:** store only metadata for keystrokes (timing and density), **never raw keystrokes**. Only store camera snapshots when a rule triggers; keep retention short (e.g., 30 days) unless required.
7. **Access controls & roles:** teacher vs admin vs student. Teachers should only see reports for their classes.
8. **Privacy notice & consent flow:** integrate a consent screen at first-run that logs acceptance; keep a copy of the consent on server.

Offline First Approach

Sunday, September 28, 2025 6:43 AM

Offline-first sync design details (conflict handling & edge cases)

- **Local queue:** client writes every event to local SQLite with a “synced” flag. Evidence blobs referenced by event rows.
- **Batched uploads:** upon connectivity, client packages N events + associated encrypted blobs into an upload, computes signature/HMAC, sends.
- **Server ACK & idempotency:** server returns ACK containing assigned server IDs. Client marks events as synced only after server ACK. If upload fails mid-way, client retries; server must deduplicate by client event ID.
- **Large files:** use chunked upload if needed (snapshot sizes), or only upload compressed small evidence (thumbnails) and retain full-size offline for longer-term per policy.
- **Conflict resolution:** minimal — since client is source of truth for local events, server should accept once verified. For teacher uploads (exam PDF), server versioning handles changes (teachers can re-upload; server stores version history).

What to Implement

Sunday, September 28, 2025 6:44 AM

(what to implement now)

1. **Implement local-first architecture with an on-prem server** (university VM). Do not depend on public cloud for the FYP deliverable. This is most professional and easiest to defend.
2. **Use Electron + NodeJS for client UI + child-process Python for CV** (you're comfortable with Electron and can call Python for OpenCV/MediaPipe).
3. **Local DB: SQLite; Server DB: Postgres.** Use simple REST API and HTTPS.
4. **Implement cryptographic signing of bundles and AES encryption of snapshots** — these are strong technical points you can show to the committee (security work) and are feasible within your timeline.
5. **For FYP-1 (before Oct 17):** deliver a working client that does app-monitoring, local camera detection, local reporting, and sync to a simple server on a VM. That's a strong, demoable slice.

Tech Stack

Sunday, September 28, 2025 6:45 AM

Practical prototype choices (stack)

- Client UI: **Electron (Node.js + React)**
- Local processing: **Python** child process for camera tracking (OpenCV + MediaPipe), communicate via IPC or gRPC.
- Local DB: **SQLite** (node-sqlite3 or better-sqlite3)
- Server: **Node.js (Express)** or **FastAPI**; **PostgreSQL** for metadata.
- File storage on server: filesystem under an encrypted partition, or **MinIO** if you want S3 API.
- Authentication: JWT tokens issued by backend; optionally integrate with university LDAP/SSO later.
- TLS: **Let's Encrypt** if server is reachable; otherwise internal CA.

Cloud Server Wali baat

Sunday, September 28, 2025 6:51 AM

How to do it

- **During development:**
  - Use **VirtualBox** / **VMware** / **WSL2** (if you're on Windows) to create a Linux VM.
  - Install your stack there: Ubuntu + PostgreSQL + backend server.
  - Expose it on your local LAN so multiple laptops (team members) can test against it.
- **Prepare for migration:**
  - Use **Docker** or at least write **deployment scripts (bash, docker-compose, Ansible)** so the whole backend can be redeployed easily.
  - Store configs (DB credentials, keys, JWT secrets) in **environment variables** or .env files, not hardcoded.
  - Keep client → server URL configurable (env file in Electron).
- **Final iteration (deployment at university):**
  - Request an on-prem VM (or a physical lab server).
  - Install same stack as your dev VM.
  - Dump your local DB and restore it on the university server (or start fresh if committee doesn't care about old test data).
  - Update client configs to point to the new server address.

Risks and how to manage them

- **Networking differences:** your home VM might run on NAT/localhost; university VM will be on a LAN with firewall rules. Make sure your API listens on a configurable port and is reachable beyond localhost.
- **Permissions:** deploying to university server may require root/admin rights — request this early so you don't hit a wall last week.
- **Data migration:** if you keep real test/exam logs in dev DB, scrub them before moving. Keep production DB clean.
- **TLS certificates:** on your local VM you can use self-signed certs; on university server you should use Let's Encrypt (or internal CA).