ChatGPT

AI-Driven Proctored Exam Platform – Architecture & Design

An **intelligent online exam system** requires multiple coordinated components: secure authentication, real-time monitoring, data analysis, and user interfaces. Below is a **detailed architecture and design** covering features, data flow, frontend/backend, microservices, ML components, libraries and datasets. All suggested tools are open-source (no paid services).

1. Overview & Objectives

- **Objective:** Build a web-based exam platform that *actively prevents cheating* by analyzing student behavior (video, audio, browsing, inputs) in real time. It should *verify identity, lock the exam environment, monitor for anomalies,* and *alert proctors* to suspicious activity.
- **Unique value:** Goes beyond simple video recording. Uses AI and algorithms (vision, audio, behavioral analysis) to *detect rule violations* (phone use, multiple faces, switching away from exam, speech, etc.) automatically.
- **Use case:** Remote and in-person proctored exams (university finals, certifications).

By design, it "locks down" the browser, captures streams, processes them via AI/algorithms, and logs everything for review. The architecture is modular (microservices) to allow independent scaling of video processing, evaluation, and the core exam engine.

2. Key Features & Components

- **User Authentication & Identity Verification:** Before each exam, students upload a photo/ID and the system verifies it (via face recognition) [1].
- **Exam Management:** Create/schedule exams, assign questions, enrollment. Exam rules (duration, etc.) enforced by backend.
- **Secure Browser Lockdown:** Browser goes fullscreen; right-click, new tab navigation and copy-paste are disabled via JavaScript. Focus/visibility events are monitored (switching tabs or windows triggers flags).
- **Real-Time Monitoring:** Student webcam video, microphone audio, and screen capture (screenshots) are streamed to the backend. WebRTC (open source) can handle media streaming.
- **Event Logging:** Record student actions – tab changes, copy/paste, long idle, login attempts – in a database.
- **Cheating Alerts & Dashboard:** Proctors/admins see a live dashboard of exam status. Suspicious events generate alerts (e.g. "Phone detected", "Multiple faces", "Left screen 15s", etc.).
- **Automated Analysis (AI/Algorithms):** Several detection modules analyze the data streams (see below) and output risk scores.
- **Audit & Reporting:** All logs, flagged video clips, exam responses are stored for post-exam review.

These features ensure a proctoring solution that is *proactive, data-driven, and explainable*.

3. High-Level Architecture (Microservices & Data Flow)

```
Student Browser          Proctor/Admin Browser
    |                          |
    | - Login/Register         | - Admin Login
    | - Face Scan (ID)         | - Dashboard
    | - Join Exam Room         | - Live Alerts / Video
    | - Webcam / Mic / Screen  |
    V                          V
[Auth Service] <--> [User DB]
    |
[Exam Service] -> [Exam DB]
    |
Real-time Service (WebRTC/WebSockets)
    |       |         \
    |       |          -> [Video Processing Service]
    |       |         /
    |       -> [Audio Analysis Service]
    |       |         \
    |       |          -> [Behavior Analysis Service]
    |
Logs/Alerts Service -> [Logs DB]
```

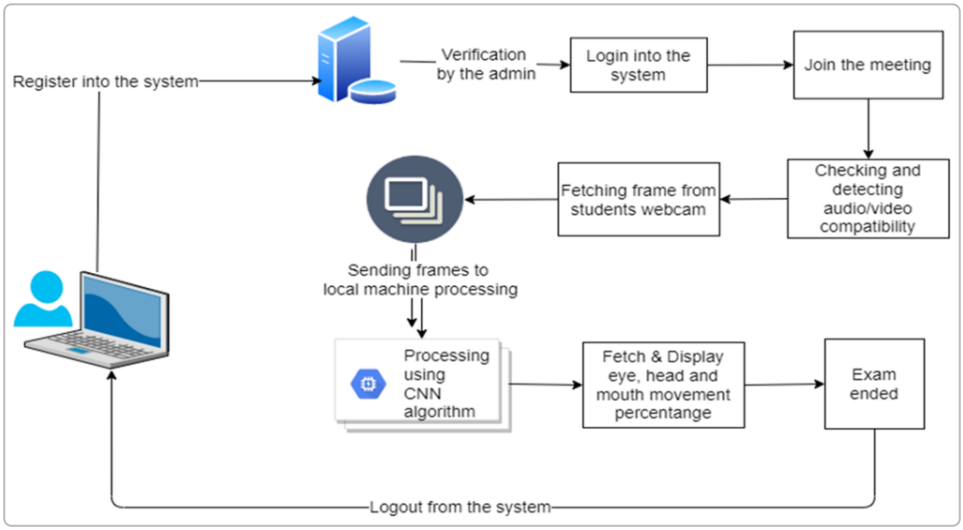

Figure: Example high-level architecture of the proctoring system.

- **Auth Service (SSO):** Manages user login, roles, and pre-exam ID checks (e.g. face-photo vs ID match).
- **Exam Service:** Manages exam metadata (questions, timings), enrollments and exam state.
- **Real-time Service:** A WebRTC server or Socket server (e.g. Kurento or Jitsi for media; or simple Socket.IO) that collects streams and events from each student's browser.
- **Video Processing Service:** Microservice (Python/ML) that takes webcam frames and runs vision models (face detection, multiple faces, eye gaze, phone/object detection).
- **Audio Analysis Service:** Takes audio stream segments and checks for speech using VAD or simple keyword spotting (e.g. forbidden help cues).

- **Behavior Analysis Service:** Listens to browser events (tab changes, key presses, copy/paste events) and computes a risk score.
- **Logs/Alerts Service:** Aggregates outputs from all AI services. It records logs and issues alerts to the Proctor Dashboard.
- **Databases:** Use open-source DBs: e.g. PostgreSQL for relational data (users, exams, logs) and an object store (MinIO or open S3) for storing video/image files.

This **microservice architecture** allows each component to scale independently and use the best language/platform for the task (e.g. Java/Python/Node).

4. Frontend Design

- **Student UI (React or Vue):** A single-page app that shows exam instructions, a webcam view, and the question interface. It uses HTML5 APIs:
- `getUserMedia` for webcam/mic,
- `MediaRecorder` or WebRTC for streaming.
- JavaScript event handlers to capture `document.visibilitychange`, `beforeunload`, `copy/paste` events.
- **Lockdown:** The app goes fullscreen. Right-click and certain key shortcuts are disabled via JS handlers. On any attempt to leave the page (alt-tab, new window), it logs a violation.
- **Real-time Feedback:** If a rule is violated (detected by backend), a warning message appears. The frontend continuously streams media and events and displays a timer.
- **Proctor/Admin UI:** A web dashboard showing all ongoing exams and flagged events. Alerts (colored codes) pop up when cheating is suspected. The admin can click an alert to view video/audio snippets or logs.

Key libraries: React (or Angular), WebRTC frameworks (like SimpleWebRTC or Jitsi), Redux for state, Chart.js/Recharts for dashboards. All are free/open-source.

5. Backend & Microservices

- **Tech stack:** Use **Spring Boot** (Java) or **Node.js (Express)** for core services (Auth, Exam, Logging). Python (Flask/FastAPI) for AI/multimedia services. Run each in Docker containers (open source).
- **Auth Service:** OAuth2 or JWT-based login. Supports ID verification: compare student's live face to stored ID photo using face-recognition (Dlib, FaceNet) [1].
- **Exam Service:** REST APIs to create exams, questions. Enforces rules (no retakes, time limits).
- **Real-time Service:**
- If using WebSockets: handle connections, route audio/video blobs to appropriate analysis services and store chunks in MinIO or disk for logs.
- Alternatively, use a media server (Jitsi/Kurento) to relay streams to analysis components.
- **AI/Analysis Services:**
- **Face & Gaze Service:** Periodically grabs video frames (e.g. 1-2 FPS) from each exam stream. Runs face detection (OpenCV DNN or MediaPipe) to ensure only one face, eyes not closed too long, head orientation (looking away flag).
- **Phone/Object Detection:** Use an object detector like YOLOv7 (open model) trained on "cell phone" class to spot phone in frame.
- **Behavior Service:** Receives JSON events (tab switch, copy/paste, idle time) from the frontend. Implements rule-based checks (e.g. >10s away triggers a warning) and a scoring algorithm.
- **Audio Service:** Uses WebRTC Voice Activity Detector or Silero VAD (open-source) to flag sustained speech (in an exam that should be silent). Possibly record audio clips when speech is detected for review.

- **Alerts & Logging:** A centralized service (Node/Java) listens for alerts (e.g. JSON like `{user, time, type: "phone", confidence:0.9}`) and writes them to a **Logs DB** and pushes notifications via WebSocket to proctor UI.

All microservices communicate via REST or message queues (RabbitMQ or Redis Pub/Sub) for low latency. For example, on receiving a new video frame, the Real-time Service may publish it to a Kafka/RabbitMQ topic that the Face & Gaze Service subscribes to.

6. Data Storage & Logging

- **Relational DB (PostgreSQL):** Store users, exam definitions, enrollments, exam results, and structured logs (time-stamped events, alert entries).
- **Object Storage (MinIO or local file system):** Save media (short video/audio clips, screenshots). For privacy, raw images can be blurred or discarded.
- **Logging DB:** A separate schema/table for exam event logs (timestamp, student, event type, score). All rule violations and AI scores are logged for audit.
- **Indexes & Caching:** Key tables (users, sessions) indexed for fast lookup. Use Redis (open source) for caching session tokens or live presence.
- **Backup/Recovery:** Regular backups of DB. Because media files can be large, consider retention policies (e.g. delete raw video after exam, keep processed events).

By keeping data in open-source systems, we avoid vendor lock-in and costs.

7. Real-Time Communication & Processing

- **WebRTC Streams:** The student's browser sends webcam video and mic audio via WebRTC to the server. We can use an open WebRTC SFU (e.g. Jitsi Videobridge or Janus) to multiplex streams.
- **Screen/Window Capture:** Use the Screen Capture API (with user permission) to send periodic screenshots of the student's screen (or detect if screen sharing started). Could also use `document.visibilitychange` to note if the tab is hidden.
- **Media Pipeline:** Once streams reach the backend, we can do real-time ingestion:
- **Video:** Extract frames every few seconds. Each frame goes to the Vision Service (see above).
- **Audio:** Segment audio stream into short clips (e.g. 5–10s) and run them through a voice detector to check if the student is talking or if there are background voices.
- **Event Channel:** All non-media events (keystrokes, focus) are sent via a WebSocket channel to the Behavior Service instantly.

This setup ensures minimal latency for detection. For example, if a phone is detected in a frame, within seconds the service alerts the proctor.

8. Cheating Detection Modules (AI/ML)

Each detection module outputs a "risk score" or boolean flag. We combine them for final verdict.

- **Identity Check:** At login, use **Face Recognition** (e.g. face_recognition) to match the student's live selfie against a stored ID photo [1]. Reject if confidence is low.
- **Face Presence:** Ensure exactly one face is in the frame. Multiple faces → immediate flag (looking at other person).
- **Eye/Gaze Tracking:** Using MediaPipe Face Mesh or Dlib landmarks, compute gaze vector. If student looks off-screen (eyes/head turned away) for too long, increment risk.

- **Lip/Mouth Movement:** Detect if mouth moves (with lips parted) for long – might indicate talking to someone. The open-source toolkit Dlib or simple facial landmarks can measure lip distance.
- **Head Pose:** From landmarks, compute head orientation (pitch/yaw). Large pitch/yaw suggests looking at notes.
- **Hand Detection:** If the student's hands leave the keyboard (e.g. phone in hand), YOLOv7 can detect "cell phone" or "book" in frame [2].
- **Object Detection:** General objects on desk (notes, textbooks) could also be detected with a trained model.
- **Audio Check:** Use Voice Activity Detection (WebRTC VAD or Silero VAD) on mic input. If any speech is detected (student is not allowed to talk), log an alert.
- **Browser/Tab Behavior:** Using JS events, detect tab switches (`visibilitychange`), window focus loss, and key events. E.g., if the tab loses focus or the user tries to open a new URL, log it. Detect copy/paste via `oncopy` events. If violations exceed a threshold (e.g. 3 tab switches), trigger a warning.
- **Screen Sharing:** If exam allows screen-based questions, require screen sharing. If disabled or blocked, flag it.
- **Answer Pattern (Optional):** For objective-type exams, compare answer timing across students: if two students have suspiciously similar answer patterns/timing (beyond random), flag for instructor review. (This is not a direct collusion network analysis, but a simplistic similarity check.)

Each module runs continuously. Their outputs are numeric scores or flags, and a weighting scheme (or simple rule engine) combines them. For example:

```
Final Risk Score = 0.3*FaceRisk + 0.2*AudioRisk + 0.2*BehaviorRisk +
0.2*ObjectRisk + 0.1*AnswerPattern
```

If **Risk > threshold**, mark "High Risk". If moderate, "Review Needed".

9. ML Models, Algorithms & Data

Models & Libraries

- **Computer Vision:**
- *Face Detection/Recognition:* Dlib or OpenCV's DNN models (e.g. `res10_300x300_ssd`), or MediaPipe Face Detector.
- *Landmark Detection:* Dlib's 68-point face predictor or MediaPipe Face Mesh for eye and head pose landmarks.
- *Eye Tracking:* Compute gaze direction from eye landmarks or use MediaPipe's Iris.
- *Object/Phone Detection:* Use a YOLOv7 or YOLOv8 model (open-source) pre-trained on COCO ("cell phone" class) to detect phones/books on table.

- *Mouth State:* A small CNN (or threshold on lip distance from landmarks) can detect if mouth is open (speaking) vs closed.

- **Audio Processing:**

- *Voice Activity Detection:* WebRTC VAD (C/C++ lib with Python bindings) or Silero VAD for detecting speech segments.

- *Speech Recognition (optional):* If more analysis needed, open-source models like Vosk (Kaldi-based) could transcribe speech to catch prohibited words, but VAD alone may suffice.

- **Behavioral Analysis:**

- *Keystroke Dynamics:* (Advanced) can use simple machine learning (SVM) on keypress timing; but this is optional and complex.

- *Tab/Window Events:* Rule-based scoring (no ML needed).

- **Machine Learning Classifier:**
  All numeric features (counts of events, averaged gaze deviation, audio speech seconds, etc.) can feed into a binary classifier for cheating vs not. The Mendeley dataset [2] suggests models like Random Forest or XGBoost work well. We can similarly train a model using extracted features.
  In fact, Hossen et al. report that "Random Forest and XGBoost achieved high precision/recall" on their proctoring dataset [3]. So starting with XGBoost (open-source) for final scoring is reasonable.
  For simplicity, a logistic regression (scikit-learn) could be used for explainability; however, an ensemble (XGBoost) can capture nonlinear patterns.

Datasets

- **Published Datasets:** We can use public proctoring datasets to bootstrap or evaluate models:
- The **"Students suspicious behaviors"** dataset [2] contains 5,500 labeled records of webcam-derived features (face, hand, gaze, head pose, phone presence). It was gathered by Hossen et al. and includes features useful for training [2] [3].
- Kaggle repositories (e.g. ExamCheatingDataset) and various Roboflow collections provide annotated images of cheating behaviors (looking away, phone, etc.) for training detectors [4] [5].
- **Synthetic Data & Simulation:** If real data is scarce, we simulate sessions: use a webcam and have volunteers act out normal vs cheating behaviors. Tools like MediaPipe make it easy to generate features (see [14†L89-L98]).
- **Data Augmentation:** For vision models, use image augmentation. For behavior logs, generate variations in timing.

Evaluation Metrics

- **Accuracy/F1:** Evaluate the final cheating classifier on labeled sessions (normal vs cheat) with precision/recall emphasis (low false positives is crucial).
- **Detection Latency:** Time between a violation and alert – should be as low as possible (e.g. within 1–2 seconds for critical cheats).
- **False Alarm Rate:** Track how often innocuous behavior triggers alerts. Tune thresholds (e.g. allow short tab switch before alert).
- **User Privacy/Audit:** Ensure we can trace any alert back to raw data (for audit) but not store raw images permanently.

10. Security, Privacy & Ethics

- **Data Privacy:** We process personal video/audio. We will **not store raw media long-term**. Instead, store only extracted features or masked images. For example, blur faces or only store

detection metadata. The Hossen dataset was “fully anonymized, no raw images or personal identifiers” [6], which is a good model.
- **Consent & Transparency:** At login, show privacy notice (required by regulations). Students consent to monitoring.
- **Secure Transmission:** All media and data flows over HTTPS/WebRTC (TLS) to prevent eavesdropping.
- **Access Control:** Only authenticated proctors can view exam streams/logs. Logs/Audit tables in the DB should be protected.
- **Ethical Use:** Clearly define policies on flagging. Any “cheating” flag should be reviewed by a human proctor before penalizing a student (to avoid false positives). Heinrich (2025) notes concerns with opaque AI decision-making in proctoring systems [7]; our design includes explanations (e.g. “Phone detected with 95% confidence”, “Face not found for 10s”) to maintain transparency.

These considerations ensure compliance with open standards (advocated in [17]).

11. Deployment & Scalability

- **Infrastructure:** Deploy on Linux servers or cloud VMs (no AWS managed services). Use Docker for each microservice. Kubernetes or Docker Compose can orchestrate containers.
- **Hardware:** Video processing can be CPU-intensive. A GPU-equipped server (NVIDIA with CUDA) would speed up vision models, but CPU with optimized models (e.g. MobileNet-based) can suffice for small classes.
- **Load Handling:** For large exams (>100 students), replicate the Video/Audio analysis services. Use a message broker (RabbitMQ) to queue frames and scale consumers horizontally.
- **CI/CD:** Use Git for code, Jenkins/GitHub Actions for builds. Store Docker images in a private registry.
- **Monitoring:** Use Prometheus + Grafana (open source) to monitor service health (CPU, memory, queue lengths) and logging (ELK stack for logs).

12. Implementation Plan & Tools

- **Frontend:** React (with hooks) for UI, Tailwind CSS for styling (optional). Use SimpleWebRTC or Socket.IO for realtime events. Browser APIs (WebRTC, Visibility API, Clipboard API) implement environment controls.
- **Backend:**
- **Languages:** Java/Spring Boot for core; Python (Flask/FastAPI) for AI.
- **Database:** PostgreSQL (with PostGIS if any geolocation needed). MinIO for object store (S3-compatible).
- **Messaging:** RabbitMQ or Redis Pub/Sub for inter-service communication.
- **AI Libraries:**
- Python: OpenCV, Dlib, MediaPipe, PyTorch/TensorFlow (for custom models), Scikit-learn, XGBoost.
- YOLOv7: ultralytics/yolov7 (free) for object detection.
- Face-Recognition: ageitgey/face_recognition (wraps dlib).
- Gaze/Head Pose: OpenCV solves PnP with 3D face model, or MediaPipe landmarks.
- Voice VAD: webrtcvad.
- **Dev Tools:** Git/GitHub (open source). Postman for API testing.
- **Datasets:** Incorporate labeled images/videos from Kaggle/Roboflow. For text-based log simulation, generate synthetic JSON logs.

- **Timeline:** Start with MVP: simple face monitoring + tab lock. Then add object detection and audio analysis. Finally, refine ML scoring and add dashboard UI.

13. References & Data Sources

The design above follows current research and tools. For example, Hossen et al. (2025) created a 5,500-sample proctoring behavior dataset using MediaPipe/OpenCV and found multi-cue models (Random Forest, XGBoost) to work well [2] [3]. Heinrich's 2025 review highlights the importance of biometric checks and privacy in proctoring [7] [1]. Open-source projects (e.g. "Proct-Xam" on GitHub [8]) already implement vision features like eye/head tracking and screen share enforcement.

By combining these insights, this architecture ensures a **comprehensive, AI-enhanced proctoring system** that uses only open-source components, covers all critical exam steps, and requires no third-party subscriptions.

---

[1] [7] A Systematic-Narrative Review of Online Proctoring Systems and a Case for Open Standards | Open Praxis
https://openpraxis.org/articles/10.55982/openpraxis.17.3.836

[2] [3] [6] Students suspicious behaviors detection dataset for AI-powered online exam proctoring - Mendeley Data
https://data.mendeley.com/datasets/39xs8th543/1

[4] [5] Top Cheating Datasets and Models | Roboflow Universe
https://universe.roboflow.com/search?q=class%3Acheating

[8] GitHub - lavsharmaa/proctxam-ai-proctoring: Proct-Xam - AI Based Proctoring which can help supervisor in tracking eye, mouth and head movement of the student
https://github.com/lavsharmaa/proctxam-ai-proctoring