CS6.201: Introduction to Software Systems

Course Project: Identity-Verified Multiplayer Arena

Submission via GitHub Classroom | Deadline: **17 April 2026 23:59**

Phase	Focus Area	Points
Phase 1	The Polyglot Harvester: data scraping and dual-database setup	20
Phase 2	Biometric Authentication Gateway: facial login pipeline	20
Phase 3	The Synchronized Arena: real-time WebSocket game lobby	35
Phase 4	Game State Resolution and Elo Reckoning: rankings and edge cases	25
	Total	**100**

General Instructions

Read all instructions carefully before you begin.

1. **AI Usage Policy.** Generative AI tools (e.g. ChatGPT, GitHub Copilot, Gemini) **are permitted** for boilerplate generation, HTML/CSS styling, and syntax debugging. You must independently architect the system and will be asked to explain your design and integration logic in the final viva. To ensure fairness, no paid services are allowed. Services like GitHub Copilot and Google AI Pro which you got for free with student benefits are allowed, but a paid Claude subscription is not. To let us verify this, you must attach all your LLM chats as screenshots (must be images, not text logs) with the prompts and generated outputs in a separate pdf, and it must show the model used.
2. **Repository Structure.** You have full flexibility over how you organise files and name directories. The only requirement is that the system is functional and satisfies every technical requirement in this document.
3. **Provided Code.** You must implement the system from scratch. The *only* code provided to you is `facial_recognition_module.py`. **Do not modify** its internal logic; treat it as a black-box utility.
4. **Submission.** All projects must be submitted via the official **GitHub Classroom** repository. Submissions by email or any other platform will not be accepted. **Ensure you have at least 40% of the work done by 10 April 2026**. To verify this, we will check your commit history.
5. **Evaluation Authority.** In case of any ambiguity or discrepancy, the text in this document is the final authority for evaluation.
6. **Doubt Resolution.** All project queries must be posted on the shared **HackMD** document. Direct messages to TAs about project doubts will not be entertained.
7. **Package Manager.** For your convenience, we recommend you use `uv` as your Python package manager. Initialise a single `uv` project for the entire repository and list all dependencies. To install the facial recognition dependencies, run:

```
uv add face-recognition numpy Pillow
```

8. **README.md.** Your repository **must** include a detailed `README.md` containing:
   - Your database schemas and instructions for initialising them.
   - The exact commands to start your web server and WebSocket services.
   - Any assumptions made during development.

Project Overview

You will build a **full-stack, real-time multiplayer web application** integrated with a biometric authentication gateway. The project spans five core areas of software systems:

- Autonomous data harvesting (web scraping)
- Polyglot database persistence (relational *and* non-relational)
- Backend API routing
- Real-time bidirectional networking via WebSockets
- Game state synchronisation

Phase 1 The Polyglot Harvester 20 Points

Goal: Populate both databases with the profiles of all eligible players before any user can log in.

Background. You are provided with a CSV file (`batch_data.csv`) containing each student's `uid`, `name`, and their base `website_url`. This phase requires you to scrape profile images from those URLs and persist the data across two different storage technologies. Using multiple storage backends suited to different data types is known as **Polyglot Persistence**.

Task Requirements:

1. **MySQL: Relational Metadata.** Design a normalised relational schema. At minimum, your `users` table must contain the following columns:

Column	Type	Default / Constraint
`uid`	`VARCHAR`	Primary Key
`name`	`VARCHAR`	(none)
`elo_rating`	`INT`	1200
`is_online`	`BOOLEAN`	`FALSE`

2. **MongoDB: Unstructured Asset Storage.** Provision a MongoDB instance *exclusively* for binary asset storage. Relational databases are inefficient for storing large binary blobs, so all scraped profile images will be held in MongoDB.

3. **The Scraper Pipeline.** Write an autonomous Python script that iterates through `batch_data.csv` and, for each student:

   a. Sends an HTTP `GET` request to `<website_url>/images/pfp.jpg`.

   b. On success, simultaneously:

- Executes an `INSERT` into MySQL with the student's metadata.
- Executes an `UPSERT` into MongoDB, storing the image as BSON binary data or a Base64 string, keyed by the student's `uid`. Either format is acceptable; the provided module handles both transparently.

Fault Tolerance (required): Your pipeline must handle HTTP 404 responses, connection timeouts, and missing images *gracefully*. A failure on one student must not crash the entire script.

Phase 2 Biometric Authentication Gateway 20 Points

Goal: Replace traditional password login with facial verification against the profile images collected in Phase 1.

Background. Password authentication is **disabled** in this system. Every login attempt must be verified by comparing a live webcam capture against stored profile images.

Task Requirements:

1. **Frontend: Webcam Capture.** Build a login page that uses the HTML5 API `navigator.mediaDevices.getUserMe` to capture a live frame from the user's webcam. Serialise the frame (e.g. to Base64) and transmit it to your backend via an HTTP `POST` request.
2. **Backend: Image Retrieval.** Upon receiving a login request, your backend (e.g. FastAPI) must fetch *all* stored profile images from MongoDB into memory and pass them to the recognition module.
3. **Facial Recognition: The Black Box.** Call `find_closest_match(login_image_data, db_images_dict)` from `facial_recognition_module.py`, where:
   - `login_image_data` is the webcam frame as raw bytes or a Base64 string.
   - `db_images_dict` is a dictionary mapping `{uid: image_data}` fetched from MongoDB; each value may be raw bytes or a Base64 string.
   - If a match is found with a confidence distance of ≤ 0.7, the function returns the matched `uid` with the highest confidence (i.e. the least distance). Otherwise it returns `None` and the login must be rejected.

   > Do **not** modify the internal logic of `facial_recognition_module.py`. Call it exactly as described above.
4. **Session Management.** On a successful match:
   a. Cross-reference the returned `uid` with MySQL to confirm the user exists.
   b. Establish a secure server-side session for the user.
   c. Set the user's `is_online` flag to `TRUE` in MySQL.

Phase 3 The Synchronized Arena 35 Points

Goal: Build a real-time multiplayer lobby and a playable Tic-Tac-Toe game using WebSockets.

Background. Standard HTTP request-response cycles are too slow for live multiplayer interactions. This phase requires **bidirectional WebSocket communication** (e.g. using `FastAPI`) for both the lobby and the game.

Task Requirements:

1. **Presence System: Live Lobby.** Build a dashboard that displays a live grid of all currently authenticated users (`is_online = TRUE`). The UI must update *dynamically via WebSockets* whenever any user logs in or disconnects, without requiring a manual page refresh.
2. **Challenge Protocol: Matchmaking.**
   - User A clicks on User B in the lobby to send a **Match Challenge**.
   - An asynchronous alert must appear immediately on User B's screen.
   - User B can either **Accept** or **Decline** the challenge in real time.
3. **Room Isolation.** Upon mutual acceptance, the server must create a dedicated WebSocket **Room** exclusive to those two users. All subsequent game traffic for that match must be isolated within this room.
4. **Game State Synchronisation: Tic-Tac-Toe.** Implement a playable Tic-Tac-Toe interface using the following architecture:

Component	Requirement
Authoritative state	Managed **server-side**. The server holds the board array.
Client action	Clients *emit* a desired move to the server.
Server validation	The server checks: is it this player's turn, and is the target cell empty?
Broadcast	After a valid move, the server broadcasts the updated board to both clients in the room.

Anti-cheat requirement: Clients must never be trusted to update the game state directly. All move validation must happen server-side.

Phase 4 Game State Resolution & Elo Reckoning 25 Points

Goal: Maintain a global leaderboard, update player ratings after every match, and handle player disconnects gracefully.

Background. Ratings must update dynamically after every match. Your system must also handle the edge case of a player abandoning a match mid-game by closing their browser.

Task Requirements:

1. **Disconnect Handling.** If a player closes their browser or drops their connection during an active match, the server must:
   a. Detect the severed WebSocket connection.
   b. Award a **forfeit victory** to the remaining player.
   c. Update the Elo ratings and match record in MySQL accordingly.

2. **Elo Rating System.** At the conclusion of every match (win, loss, or draw), update both players' ratings in MySQL using the **standard Elo formula**.

   Step 1: Compute the expected win probability for each player.

$$E = \frac{1}{1 + 10^{\,(R_{\text{opponent}} - R_{\text{player}})\,/\,400}} \tag{1}$$

   Step 2: Update each player's rating.

$$R_{\text{new}} = R_{\text{old}} + K \cdot (S - E) \tag{2}$$

   Parameter definitions:

   - $K = 32$: the maximum rating change per game.
   - S: the match outcome from this player's perspective.
     - $S = 1.0$ for a win
     - $S = 0.5$ for a draw
     - $S = 0.0$ for a loss
   - E: the expected probability of winning, computed in Step 1.

   > Apply the Elo update to **both** players at the end of each match. When computing E, use each player's rating from the *start* of the match; do not use an already-updated value for the second player's calculation.

3. **Global Leaderboard.** Provide a dedicated page that renders a table of **all players in the batch**, sorted in descending order by `elo_rating`. The leaderboard must reflect the most recent ratings from MySQL.