ChatGPT

Face-Monitor App Overview and Architecture

The **face-monitor** project is a FastAPI-based webcam proctoring tool. Its `app` directory likely contains a Python FastAPI backend and a simple frontend (HTML/JS) for video capture. The backend probably exposes a WebSocket endpoint to receive frames and runs a face-detection model (e.g. OpenCV or a deep learning face-recognition library) to monitor test-takers in real time. This follows the pattern in similar apps: for example, a FastAPI app can use WebSockets and OpenCV to "seamlessly detect human faces in real-time camera video streams" [1]. In practice, the folder might include an `app.py` or `main.py` (initializing the FastAPI instance), route modules, and a machine-learning module with a face detector. The **frontend** likely consists of an `index.html` and a `script.js` that capture the user's webcam via JavaScript and send frames to the server. Indeed, a typical setup uses an HTML page and a JS script that "connect[ ] to a WebSocket in the browser. It will handle sending requests and receiving responses to and from our API" [2]. Data (e.g. recognized faces or alerts) may be logged or stored for analysis. In summary, the architecture is roughly:
- **Backend (FastAPI/Python):** WebSocket and/or REST endpoints, face-detection/recognition logic (e.g. using OpenCV, Haar cascades or a DNN) [1]. Python FastAPI is a common choice for such APIs [3].
- **Frontend (HTML/JS):** Web page that captures webcam video (via WebRTC `getUserMedia`) and streams frames or images to the FastAPI server over WebSocket [2].
- **Models/Data:** A face detection model or library. Possibly user management (e.g. student profiles) if implemented. Data storage for logs or alerts.

Overall, face-monitor looks like a standard asynchronous FastAPI app with real-time video processing. The WebSocket connection and front-end JS form the live stream pipeline, and the FastAPI backend runs face analysis on each frame to flag issues (e.g. no face, multiple faces, or lack of attention).

AI Models and APIs for Question Generation

To add **topic-wise exam question generation**, we can leverage modern large language models (LLMs) or specialized NLP services. The two main options are cloud APIs (e.g. OpenAI, Cohere, Anthropic) or open-source models (via Hugging Face, Replicate, etc).

- **OpenAI (ChatGPT/GPT-4, GPT-3.5):** OpenAI's GPT family offers state-of-the-art text generation and is easy to integrate via their Python SDK or HTTP API [4]. For example, GPT-4 can generate high-quality questions given a prompt (subject, chapter, etc.). It's a "top choice" in generative AI [4]. Note that API usage has costs per token and requires handling sensitive data carefully.
- **Anthropic Claude / Amazon Bedrock:** Anthropic Claude (especially Claude v2 or Claude 3) is a powerful LLM accessible via Anthropic's API or through AWS Bedrock. AWS Bedrock provides a single API with many models (Anthropic Claude, Cohere Command, Meta Llama 2, AI21 Jurassic, etc.) [5]. For instance, an AWS example used Claude on Bedrock to "generate exam questions and answers" from lecture PDF files [6]. This shows that Claude or similar models can be used for automatic question generation.
- **Perplexity.ai and TogetherAI:** These platforms aggregate multiple LLMs (including GPT-4o, GPT-4 Turbo, Claude-3) behind a unified API [7]. They can be used interchangeably to call GPT or Claude

models without vendor lock-in. Using such services, you could access GPT-4 or Claude through a single integration.

- **Hugging Face / Open-Source Models:** If self-hosting is preferred (for cost or privacy), you can use open models via Hugging Face or Replicate. For example, HF Transformers supports T5, BLOOM, GPT-Neo, LLaMA, Mistral, etc., for text generation [8]. You could either call HF's Inference API or run a model in a Docker container. Models like **LLaMA 2** or **Mistral** can be deployed on a capable GPU/CPU server to generate questions. This approach avoids per-call fees but requires infrastructure.

Recommendations: For ease of development and quality, starting with **OpenAI's GPT-4 (via the `openai` Python library)** is sensible [4]. It produces high-quality text and handles instruction-based prompts well. If budget or data privacy is a concern, consider **Cohere Command** or **Claude via AWS Bedrock** [5]. For an open-source path, using Hugging Face's Transformers (with a model like LLaMA2 or Mistral) is possible. All these approaches can be driven by passing a subject/chapter prompt to the LLM.

Structuring Topic-Based Question Generation

To generate questions by subject or chapter, we should add a **new API endpoint** and an internal prompt/template system. For example, create a FastAPI route like `POST /generate_questions` that accepts JSON with keys like `subject`, `chapter`, `num_questions`, and maybe `difficulty` or `question_type`. Internally, this handler would:

- **Map the topic:** Validate that `subject` and `chapter` correspond to a real topic (perhaps stored in a small database or config). Optionally fetch any reference text if using RAG.
- **Build the prompt:** Use a template such as, *"Generate {num} exam questions (with answers) on in . Include a mix of multiple-choice and short-answer formats, and cover Bloom's taxonomy levels."* This prompt explicitly specifies topic and output format.
- **Call the AI API:** Send the prompt to the chosen LLM endpoint (e.g. OpenAI's `create_chat_completion`). For example, with OpenAI's Python client:

```
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[{"role": "system", "content": "You are an educational
question generator."},
              {"role": "user",   "content": prompt}]
)
questions = response.choices[0].message.content
```

- **Process the output:** Return the generated questions (and answers) to the caller, possibly parsing them into structured JSON.

To allow **topic-based** generation, the integration could also include: - A table or file of subjects/chapters (so the front-end can present dropdowns). - (Optional) If available, pre-loaded curriculum text or lecture notes could be retrieved to augment the prompt (e.g. via embeddings or RAG) so the questions are grounded in actual content.

In effect, the structure is: **Frontend** ⇒ selects a subject/chapter ⇒ **FastAPI endpoint** ⇒ calls LLM with that context ⇒ returns questions. An example flow (similar to AWS's design) is shown below: the educator uploads or selects content, the system invokes an LLM to *"generate exam questions"* from it [6]. We would simplify it by directly using the topic name in the prompt.

Frontend and Backend Updates

Frontend changes: Add a new page or section in the UI for question generation. This should include: - A form (dropdowns or text fields) for *Subject*, *Chapter/Topic*, *Number of questions*, and perhaps *Question type (e.g. MCQ/short-answer)*. - A "Generate" button that sends these to the backend. - An area to display the returned questions and answers (e.g. collapsible Q&A, or a list).

If the existing app is plain HTML/JS, we can use `fetch` or AJAX to call the new `/generate_questions` API. If a modern framework (React/Vue) is used, add a component for this form. Show a loading indicator while waiting. Also provide a way to refresh/regenerate or copy the output.

Backend changes: In the FastAPI backend: - Create new Pydantic request/response models (e.g. a `QuestionRequest` model with subject, chapter, etc., and a `QuestionResponse`). - Implement the `/generate_questions` endpoint (async def). This will validate input and then call the AI service. For long calls (GPT-4 may take a few seconds), you can either await it directly or use a background task. - Install any needed libraries: e.g. `openai` (via `pip install openai`), or `cohere` if using Cohere. - Securely store API keys (as environment variables or secrets) and use them in the code. - (Optional) Update the data layer: if you want to save generated questions, add a database table (e.g. in SQLite/Postgres) to cache them by topic or save user history. Use SQLAlchemy or Tortoise ORM if needed. - Unit-test the endpoint with mock topics and verify output format. - Update CORS or auth if needed (the new endpoint may need to be accessible from the front-end domain).

Tools/Frameworks: You'll continue using **FastAPI/Python** for the backend. For the AI call, use the official client (e.g. `openai` SDK, or `huggingface` for Transformers, or `requests` for Hugging Face Inference API). On the front end, use JavaScript's Fetch API or a library like Axios for HTTP calls. You might also use **LangChain** or **Promptify** if you need advanced prompt orchestration, but for a first version simple prompts suffice.

Educational Best Practices

Bloom's Taxonomy provides a hierarchy of cognitive skills from basic recall to higher-order thinking (Remember → Understand → Apply → Analyze → Evaluate → Create) [9] [10]. When generating exam questions, ensure variety across these levels. For example, ask some factual (recall) questions, some conceptual (explain/understand) questions, and some higher-order (analyze or apply) problems. Questgen's AI, for instance, explicitly targets all Bloom's levels in its outputs [9].

Other best practices include: - **Question types:** Mix formats (multiple-choice, true/false, short answer, etc.) to assess different skills. Avoid overloading with one type. - **Difficulty balance:** Offer easy, medium, and challenging questions. You might allow specifying difficulty or design the prompt to include a range. - **Clear wording:** Write questions clearly and unambiguously. Check that AI didn't produce any factual errors or misleading distractors. Humans should review and edit the AI output. - **Curriculum alignment:** Ensure

questions match the specified topic. The AWS example emphasized generating *curriculum-aligned assessments* [6] . Prompt engineering should include the subject/chapter so content stays on-target. - **Avoid bias and errors:** AI may hallucinate facts. Always validate the correct answers. For fairness, ensure questions are culturally appropriate and free of bias. - **Answer key:** Provide answers (and explanations if possible) for any student self-check. Automated generation can include answers as part of the output. - **Use strong verbs:** Follow Bloom's verbs (e.g., "list", "explain", "compare", "design") to specify cognitive level. This helps the AI create questions at the intended level [11] .

By following Bloom's framework and these practices, generated questions will better assess learning objectives and stimulate deeper thinking [9] [10] .

Step-by-Step Implementation Plan

1. **Audit the existing app.** Examine the face-monitor code (FastAPI routes, frontend files) to understand where to integrate. Identify where to add new files (HTML/JS) and API logic.
2. **Define question specs.** Decide how teachers select topics: hard-coded subjects/chapters or dynamic listing (e.g. from a database). Prepare any lookup tables or JSON config of available topics.
3. **Choose AI model/API.** Based on requirements, pick a service (e.g. OpenAI GPT-4). Obtain API credentials. Install the library (`pip install openai`) or set up necessary environment (e.g. Hugging Face credentials).
4. **Design prompts.** Write a clear prompt template for question generation, including placeholders for subject/chapter, number of Q's, etc. Test prompts manually in the playground to refine.
5. **Implement backend endpoint.** In FastAPI, create a new POST route (e.g. `/generate_questions`) with a Pydantic model for input. In the handler, format the prompt and call the AI API (async).
6. **Process and return results.** Parse the LLM response into a structured format (e.g. JSON array of questions and answers). Handle errors (timeouts, token limits) gracefully. Return the data to the client.
7. **Build frontend interface.** Add a page or modal in the web UI with form fields for subject, chapter, number of questions, etc. Use JavaScript to `fetch` the new API and display the results. Include loading indicators and error messages.
8. **Incorporate Bloom's taxonomy.** Optionally, allow specifying a Bloom's level or ensure the prompt explicitly covers multiple levels. Include guidance text reminding users about question variety.
9. **Test end-to-end.** Run the app locally. Try generating questions on sample topics to check relevance and correctness. Adjust the prompt or parameters as needed.
10. **Review and refine questions.** Have educators or team members examine the AI-generated questions. Revise the prompt and template to fix any systematic issues (e.g. too many trivial questions).
11. **Enhance as needed:** Consider adding caching (so repeated requests for the same topic reuse answers), rate-limiting, or a job queue (e.g. Celery) if generation is slow.
12. **Deploy and secure.** Add the API keys securely (environment vars). Deploy the updated FastAPI app (Docker, AWS/GCP, etc.). Ensure the endpoint is protected (authentication if needed) to prevent abuse.
13. **Documentation:** Document the new feature for end users (how to select topics, interpret questions) and for developers (how prompts are constructed, where to plug in API keys).

By following these steps and using tools like FastAPI, the OpenAI (or equivalent) SDK, and a simple frontend framework or vanilla JS, the face-monitor app can be extended into a comprehensive exam platform that both proctors students and generates relevant practice questions on demand.

Sources: FastAPI face-detection example [1] [2]; AI quiz generation and Bloom's Taxonomy guidance [9] [10] [12].

---

[1] [2] FastAPI & OpenCV: A Comprehensive Guide to Real-Time Face Detection | by Sam Akan | Medium
https://medium.com/@samakan061/fastapi-opencv-a-comprehensive-guide-to-real-time-face-detection-c12779295f16

[3] [8] GitHub - TINBYTE/RAG_SQL_QUIZ_GENERATION: An AI-powered platform that dynamically generates and grades exam questions using Retrieval-Augmented Generation (RAG). It leverages NLP, document retrieval, and a user-friendly interface for seamless exam creation
https://github.com/TINBYTE/RAG_SQL_QUIZ_GENERATION

[4] [7] Best Generative AI APIs in 2025 | Eden AI
https://www.edenai.co/post/best-generative-ai-apis

[5] [6] [12] Build a serverless exam generator application from your own lecture content using Amazon Bedrock | AWS Machine Learning Blog
https://aws.amazon.com/blogs/machine-learning/build-a-serverless-exam-generator-application-from-your-own-lecture-content-using-amazon-bedrock/

[9] AI-powered Bloom's Taxonomy Quiz Generator
https://www.questgen.ai/ai-blooms-taxonomy-quiz-generator

[10] [11] Bloom's Taxonomy Graphic Description - Center for Instructional Technology and Training - University of Florida
https://citt.ufl.edu/resources/the-learning-process/designing-the-learning-experience/blooms-taxonomy/blooms-taxonomy-graphic-description/