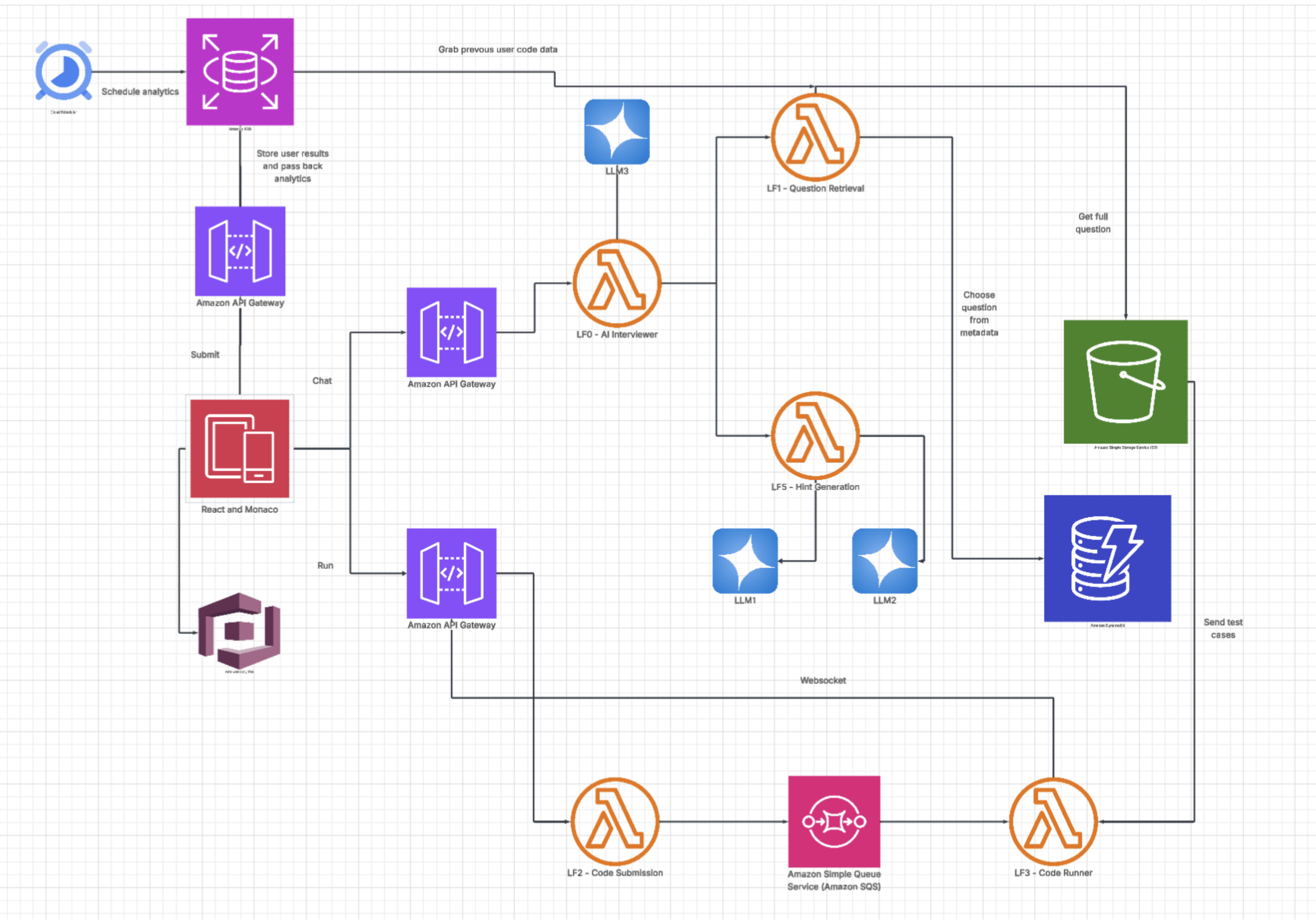


Build a realtime coding interview platform that has an AI Interviewer, comes up with Assessment Questions in realtime,based on the difficulty level, question type. A coding environment also needs to be built and toolcalling to be added to provide only context aware and limited hints/suggestions.

Project Architecture Overview

Front End

- React SPA – A modern, responsive single-page application that provides the user interface. Hosted on AWS for scalability and easy deployment.
- Monaco Editor – A browser-based code editor (same core as VS Code) for users to view, edit, and run code interactively.
- Amazon Cognito – Manages user authentication and authorization securely.
- Amazon S3 (Static Hosting) – Hosts the React frontend assets and serves them through AWS CloudFront (optional for performance).
- CloudFront - Serves static frontend files globally with low latency.

Databases & Storage

- Amazon S3
  - Stores questions, AI-generated question variants, solutions, templates, and test cases.
  - Also stores user-submitted code files and execution results.
  - Chosen for its scalability, durability, and native AWS integration.
- Amazon DynamoDB
  - Stores metadata about each question (difficulty, topic, variant relationships, and S3 object paths).
  - Tracks execution metadata such as user progress, run status, and test results summaries for fast retrieval.
  - Provides O(1) or O(log n) lookups ideal for the low-latency operations between frontend and backend.
- Amazon RDS (Optional)
  - Stores structured user state data or more relational data (e.g., user history, stats).
  - Could be replaced entirely by DynamoDB if time is limited.
- Amazon CloudWatch
  - Collects logs and metrics from Lambda functions for debugging, performance monitoring, and alerting.
- AWS Systems Manager Parameter Store
  - Stores and manages versioning for all LLM prompts (LLM1, LLM2, LLM3) and feature toggles, allowing prompt updates without code deployment.

AI Layer

- LLM1 – Generates hints for coding questions based on the current code, problem description, and test results. Utilizes Function Calling/Tool Use to enforce limited, structured hints.
- LLM2 – Validates the quality and relevance of the hint produced by LLM1.
  - If the hint is invalid or low-quality, LLM1 retries up to a limit before returning a default fallback hint.
- LLM3 – Interview controller (inside LF0)

    - Trained/prompted to act like a professional interviewer, constrained by context retrieved from DynamoDB.

Backend

- Amazon API Gateway
    - Entry point for all client requests (fetching questions, running code, requesting hints).
    - Routes requests securely to corresponding AWS Lambda functions.
    - Configured to handle both standard REST API calls and WebSocket connections for real-time result pushing.
- AWS Lambda Functions
  Each Lambda function (LF) handles a specific backend task:

1. LF0 – AI Interviewer
    - Handles user messages (from chat UI) and interprets intent.
    - Only operates on data retrieved from DynamoDB and S3 (no hallucinations).
    - Uses a system prompt + retrieval pipeline (RAG style) retrieved from Parameter Store to ground its responses.
    - Filters user input through an intent classifier (implemented inside LF0) to reject out-of-scope queries like “what should I have for breakfast”.
    - Calls LF1 and LF5
    - Retrieves feedback returned from LF4
2. LF1 – Question Retrieval
    - Triggered when a user requests a question.
    - Queries DynamoDB for metadata and retrieves the actual question file from S3.
    - Returns the question and any associated files to the frontend.
3. LF2 – Code Submission
    - Triggered when a user runs their code.
    - Uploads the submitted code to S3.
    - Sends a message to an SQS queue to decouple execution from user interaction (avoiding Lambda timeout).
4. LF3 – Code Runner / Evaluator
    - Triggered by messages in SQS.
    - Retrieves the code and corresponding test cases from S3.
    - Executes the code in a secure, isolated environment (e.g., AWS Lambda with strict resource limits, locked-down filesystem/network access, or ECS Fargate container).
    - Stores execution results (pass/fail, errors, output) in S3 and metadata in DynamoDB.
    - After storing results, sends a message via AWS API Gateway WebSockets to the specific connected client, notifying them results are ready.
5. LF4 – Polling for Results
    - Invoked periodically by frontend polling or event triggers.

    - Checks DynamoDB for execution metadata and retrieves the full results from S3 to return to the user.
6. LF5 – Hint Generation
    - Called when a user requests a hint.
    - Sends question context and code to LLM1 to generate a hint.
    - Validates the hint with LLM2 and returns the final result to the frontend.

Flow Summary

1. User logs in via Amazon Cognito.
2. User starts interview → Frontend chat UI sends message to LF0 (AI Interviewer) through API Gateway.
3. LF0 interprets intent:
   • If requesting a question → calls LF1 (Question Retrieval) → queries DynamoDB → fetches question from S3 → returns question to user via chat.
   • If requesting a hint → calls LF5 (Hint Generation) → passes context to LLM1 + LLM2 → returns validated hint via chat.
   • If off-topic → gently redirects conversation toward coding tasks.
4. Frontend displays retrieved question in Monaco Editor.
5. User writes and runs code → LF2 (Code Submission) uploads code to S3 and sends execution job to SQS.
6. LF3 (Code Runner) consumes SQS messages, retrieves code and test cases from S3, executes securely, and stores results in S3 with metadata in DynamoDB.
7. ~~Frontend polls → LF4 (Result Retrieval) fetches execution metadata and displays test results to the use~~r. LF3 immediately pushes results back to the user via the established WebSocket connection.
8. User may request additional hints → LF0 → LF5.
9. When all test cases pass → Frontend notifies LF0, which logs completion in DynamoDB and offers the next question.
10. Session ends → LF0 summarizes performance (questions completed, hints used) using data from DynamoDB.

Why This Architecture Works

- Scalable: Each Lambda is independent and event-driven.
- Fault-tolerant: SQS decouples code execution from UI latency.
- Cost-efficient: Pay-per-use model for Lambda, S3, and DynamoDB.
- Maintainable: Clear separation of responsibilities between components.
- AWS-native: All services integrate seamlessly within the AWS ecosystem.

- **Person A – Frontend Lead (React, Cognito): Marvin**

  Leads the development of the user interface, including authentication, coding environment, and interviewer chat. Secondary responsibility includes integrating frontend interactions with the AI interviewer logic (LF0, LF5) to ensure smooth user experience.

- **Person B – AI Logic Lead (LF0, LF1, LF5): Jesse**

  Oversees the design and implementation of the AI interviewer system, including question selection, hint generation, and conversation control using LLM-based functions. Secondary focus on DynamoDB and API Gateway ensures AI outputs remain grounded in stored question data.

- **Person C – Backend Infrastructure Lead (LF2, LF3, SQS): Ethan**

  Manages backend execution pipelines, including Lambda orchestration, message queuing, and any code submission handling. As secondary for frontend communication (WebSocket/polling), they ensure reliable synchronization between backend events and the user interface.

- **Person D – Data & Integration Lead (DynamoDB, CloudWatch, RDS): Conor**

  Maintains data consistency, user performance tracking, and system monitoring. Also supports LF3 integration to ensure reliable Lambda operation and secure execution environments.

Role	Front	AI	Infra	Data
Lead	Marvin	Jesee	Ethan	Conor
Secondary	Ethan	Marvin	Conor	Jesse

Diagrams

Main

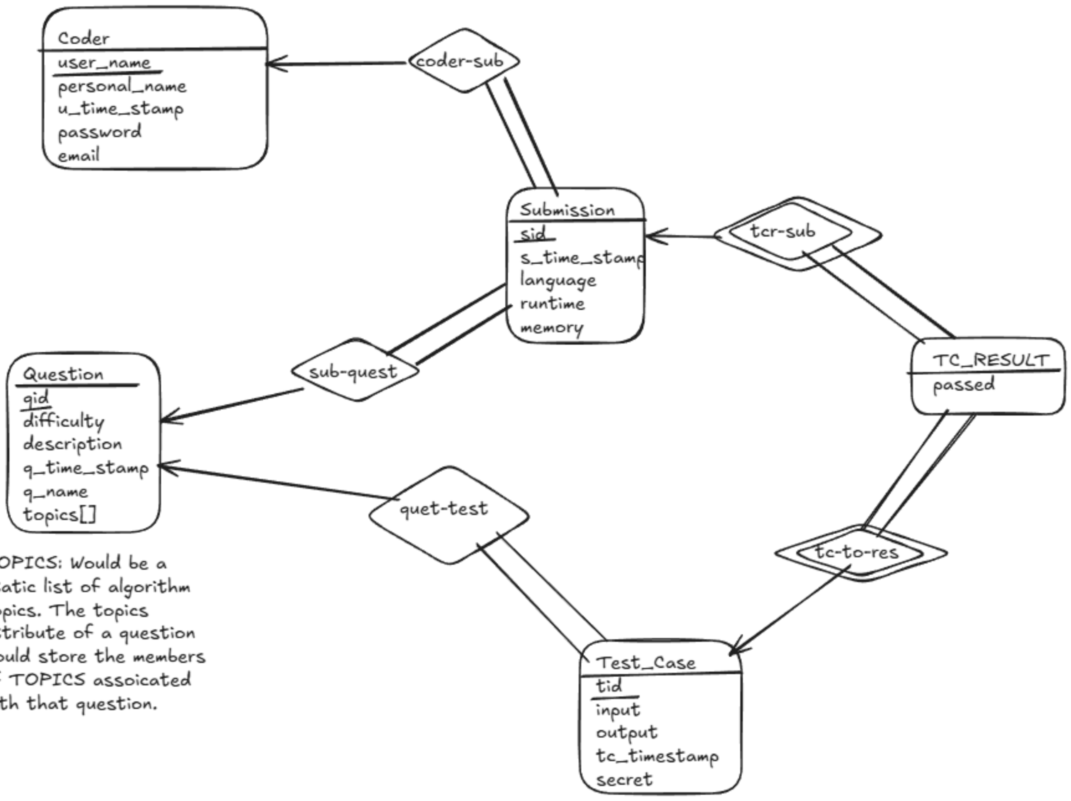
```
flowchart LR
    %% Frontend/UI Layer
    A["React SPA - Monaco Editor"] -->|"REST / WS"| APIGW["API Gateway (REST/WS)"]

    %% Authentication
    APIGW -->|"JWT Auth"| COG["Amazon Cognito"]

    %% Backend Lambda Functions (Expanded Section)
    subgraph Lambda_Functions ["AWS Lambda Functions"]
        L0["LF0 - AI Interviewer Controller"]
        L1["LF1 - Question Retrieval"]
        L2["LF2 - Code Submission"]
        L3["LF3 - Code Runner (Secure Sandbox)"]
        L5["LF5 - Hint Generation"]
    end
    APIGW --> Lambda_Functions

    %% Explicit Calls from L0 to L1 and L5
    L0 --> L1
    L0 --> L5

    %% AI Layer Interactions (Expanded LLM Section)
    subgraph AI_Layer ["AI Components"]
        LLM1["LLM1 - Hint Generator"]
        LLM2["LLM2 - Hint Validator"]
        LLM3["LLM3 - Interview Controller"]
    end
    L5 --> LLM1
    L5 --> LLM2
    L0 --> LLM3

    %% Code Execution Path
    L2 --> SQS["SQS Queue"]
    SQS --> L3
    L3 --> S3_ARTIFACTS["S3 - Code, Results, Logs"]

    %% Data Storage and Logs
    L0 & L1 & L2 & L5 --> DYNAMO_DB["DynamoDB - Metadata/Logs"]
    L1 & L3 --> S3_ARTIFACTS
    L0 & L5 --> PARAM_STORE["Parameter Store - Prompts"]

    %% Optional/Monitoring Components
    Lambda_Functions --> CW["CloudWatch"]
    Lambda_Functions -.-> RDS["RDS (Optional)"]
```

Code flow execution:

```
flowchart TD
   subgraph Frontend["Frontend (React SPA)"]
       UI_Editor["Monaco Editor/UI"]
   end

   subgraph Backend_Sync["Synchronous Backend (API Gateway & Lambda)"]
       AGW["API Gateway (REST)"]
       LF2["LF2 - Code Submission Handler"]
   end

   subgraph Backend_Async["Asynchronous Backend (Execution & Notification)"]
       SQS["SQS Queue (Execution Jobs)"]
       LF3["LF3 - Code Runner (Secure Sandbox)"]
   end

   subgraph Data["Data & Storage"]
       S3["S3 (Code, Test Cases, Results)"]
       DDB["DynamoDB (Metadata, Logs)"]
   end

   subgraph Realtime["Real-time Notification"]
       AGW_WS["API Gateway (WebSockets)"]
   end

   %% Flows

   %% 1. User Submission (Sync Path)
   UI_Editor -- "1. User submits code" --> AGW
   AGW -- "2. Invokes LF2" --> LF2
   LF2 -- "3a. Uploads code to S3" --> S3
```

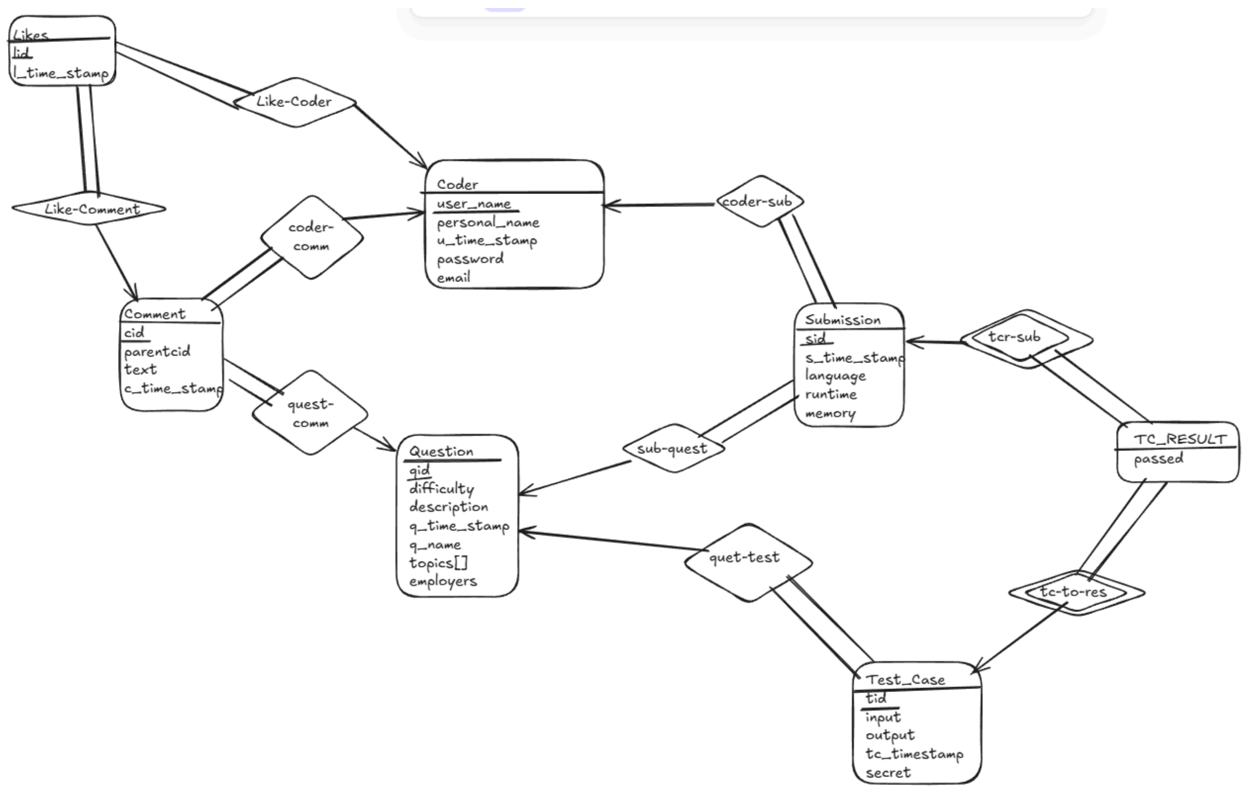
```
LF2 -- "3b. Enqueues job in SQS" --> SQS
LF2 -- "4. Returns 'Accepted' status to UI" --> UI_Editor

%% 5. Execution (Async Path)
SQS -- "5. Triggers LF3 Execution" --> LF3
LF3 -- "6a. Retrieves code/tests from S3" --> S3
LF3 -- "6b. Executes securely" --> DDB
DDB -- "7. Stores Result Metadata" --> S3
S3 -- "7. Stores Full Results" --> LF3

%% 8. Notification (WebSocket Push)
LF3 -- "8. Pushes message via WebSocket API" --> AGW_WS
AGW_WS -- "9. Notifies connected UI instance" --> UI_Editor
```

Simple ER Diagram:



TOPICS: Would be a static list of algorithm topics. The topics attribute of a question would store the members of TOPICS assoicated with that question.

Complicated ER Diagram



Simple DB Schema:

```
CREATE TABLE Coder
(
   user_name VARCHAR(50) PRIMARY KEY,
   personal_name VARCHAR(50) NOT NULL,
   passwd VARCHAR(50) NOT NULL,
   email VARCHAR(100) NOT NULL UNIQUE,
   u_time_stamp TIMESTAMP DEFAULT CURRENT TIMESTAMP
);
CREATE TABLE Question
(
   qid SERIAL PRIMARY KEY,
   q_name VARCHAR(100) NOT NULL,
   difficulty varchar(16) NOT NULL
   CHECK (difficulty IN ("Easy", "Medium", "Hard")
   q_time_stamp TIMESTAMP DEFAULT CURRENT TIMESTAMP
);
CREATE TABLE Submission
(
   s_time_stamp at TIMESTAMP DEFAULT CURRENT TIMESTAMP
   user_name VARCHAR(50) REFERENCES Coder(user_name)
   ON DELETE CASCADE,
   qid INTEGER REFERENCES Question(qid) ON DELETE CASCADE,
   PRIMARY KEY(user_name, qid, s_time_stamp)
);
CREATE TABLE Test_Case
(
   tid SERIAL PRIMARY KEY,
   qid INTEGER NOT NULL REFERENCES Question(qid),
   input VARCHAR(1048576) NOT NULL,
   output VARCHAR(1048576) NOT NULL,
   tc_time_stamp at TIMESTAMP DEFAULT CURRENT TIMESTAMP,
)
CREATE TABLE TC_Result
(
   sid INTEGER REFERENCES Submission(sid),
   tid INTEGER REFERENCES Test_Case(tid),
   PRIMARY KEY(sid, tid),
   passed BOOLEAN NOT NULL
)
```