Technology Stack and ML/DL Algorithms for Real-Time 1v1 Coding Challenge Platform

August 5, 2025

Contents

1 Overview

This document outlines the technology stack and machine learning/deep learning algorithms for a scalable, real-time 1v1 coding challenge platform. The platform supports multiple programming languages, real-time code editing, matchmaking, instant feedback, communication, leaderboards, and advanced features like plagiarism detection and automated test case generation. Below, we detail the front-end, back-end, database, and other technologies, followed by a step-by-step breakdown of the tech stack requirements for each feature.

2 Technology Stack

2.1 Front-End

- **React.js**: For building a dynamic, component-based user interface (UI) for the code editor, leaderboard, and real-time communication features.
- **Tailwind CSS**: For responsive and modular styling of UI components.
- **CodeMirror**: For a customizable, real-time code editor supporting syntax highlighting for multiple programming languages.
- **Monaco Editor**: Alternative to CodeMirror for advanced code editing features, including IntelliSense, used in VS Code.
- **WebSocket (via Socket.IO)**: For real-time code synchronization and communication between users.
- **Chart.js**: For visualizing performance analytics and leaderboard data.
- **Babel**: For modern JavaScript syntax and JSX support in React.

2.2 Back-End

- **Node.js with Express.js**: For building a scalable, RESTful API to handle user requests, authentication, and matchmaking.
- **WebSocket (Socket.IO)**: For real-time bidirectional communication for code syncing, chat, and spectator views.
- **Docker**: For containerizing the code execution environment to ensure security and scalability.
- **NGINX**: For load balancing and serving static assets.
- **Redis**: For caching user sessions, matchmaking queues, and real-time leaderboard updates.
- **RabbitMQ**: For message queuing to handle asynchronous tasks like test case validation and plagiarism checks.

2.3 Database

- **PostgreSQL**: For storing user profiles, challenge data, leaderboards, and performance analytics due to its relational structure and scalability.
- **MongoDB**: For storing unstructured data like code submissions, test case results, and analytics logs.

- **Elasticsearch**: For fast searching and indexing of coding challenges and user submissions.

2.4 Other Technologies

- **Judge0**: For secure, sandboxed code execution supporting multiple programming languages.
- **AWS ECS/EC2 or Kubernetes**: For orchestrating scalable containerized environments for code execution and back-end services.
- **AWS S3**: For storing large datasets like test cases and user-submitted code.
- **WebRTC**: For real-time voice/video communication in collaborative debugging.
- **GitHub API**: For integrating user code repositories for custom challenges.

2.5 Machine Learning/Deep Learning Algorithms

- **Plagiarism Detection**:
  - *Algorithm*: Moss (Measure of Software Similarity) combined with a Siamese Neural Network.
  - *Description*: Moss tokenizes code and compares structural similarity, while a Siamese Neural Network trained on code embeddings (e.g., CodeBERT) detects semantic similarities. The network uses a contrastive loss function to differentiate between original and plagiarized code.
  - *Tech*: Python, PyTorch/TensorFlow, CodeBERT, Moss.
- **Code Quality Assurance**:
  - *Algorithm*: Linting with Deep Code Review (LSTM-based model).
  - *Description*: An LSTM model trained on code repositories analyzes code for style, readability, and best practices. Tools like ESLint (JavaScript) or Pylint (Python) are integrated for static analysis.
  - *Tech*: Python, TensorFlow, ESLint, Pylint, SonarQube.
- **Automated Test Case Generation**:
  - *Algorithm*: Genetic Algorithm with Reinforcement Learning (RL).
  - *Description*: A genetic algorithm generates diverse test cases by mutating input sets, guided by an RL agent (e.g., DQN) to maximize code coverage. The RL agent learns to prioritize test cases that expose edge cases.
  - *Tech*: Python, DEAP (for genetic algorithms), Gym (for RL), JaCoCo (for code coverage).

3 Step-by-Step Tech Stack Requirements

3.1 Room Creation for Pairing Two Users with Similar/Nearby Ratings

- **Requirements**: Matchmaking based on user ratings (e.g., Elo rating system). Real-time room creation with WebSocket for live updates.
- **Tech Stack**:

  - *Back-End*: Node.js with Express.js to handle room creation API. Redis for storing user ratings and matchmaking queues.
  - *Algorithm*: Elo rating system to calculate user ratings and match users within a rating range (ś50 points).
  - *WebSocket (Socket.IO)*: To notify users when a match is found and create a room.
  - *Database*: PostgreSQL to store user ratings and match history.

- **Steps**:
  1. User joins matchmaking queue via a POST request to `/api/matchmaking/join`.
  2. Redis stores user ID and rating in a sorted set for quick lookup.
  3. Back-end runs a matchmaking algorithm every 5 seconds to pair users with similar ratings.
  4. On match, create a room ID in Redis and notify users via Socket.IO.
  5. Store room metadata (user IDs, challenge ID) in PostgreSQL.

3.2 Collecting and Managing Questions

- **Requirements**: Store and categorize coding challenges by difficulty, topic, and language. Admin interface for adding/editing questions.
- **Tech Stack**:
  - *Back-End*: Node.js with Express.js for question management APIs.
  - *Database*: MongoDB for flexible schema to store questions, test cases, and metadata. Elasticsearch for full-text search.
  - *Front-End*: React.js for admin dashboard to manage questions.
- **Steps**:
  1. Admin submits a question via React form, including description, difficulty, and sample test cases.
  2. Back-end validates input and stores in MongoDB with fields: `title`, `description`, `difficulty`, `tags`, `test_cases`.
  3. Elasticsearch indexes question metadata for fast search by keywords or difficulty.
  4. API endpoints (`/api/questions/search`, `/api/questions/add`) handle retrieval and addition.

3.3 Developing the Code Editor/Environment

- **Requirements**: Real-time code editor supporting multiple languages with syntax highlighting. Secure code execution with test case validation.
- **Tech Stack**:
  - *Front-End*: CodeMirror or Monaco Editor for real-time editing with syntax highlighting.
  - *Back-End*: Judge0 for sandboxed code execution. Node.js for API to submit code.
  - *WebSocket (Socket.IO)*: For real-time code synchronization between users.
  - *Docker*: For isolated code execution environments.

- **Steps**:
  1. Integrate CodeMirror/Monaco Editor in React for language-specific syntax highlighting.
  2. User types code; changes are synced via Socket.IO to other user/spectators.
  3. On submission, POST request sends code to `/api/submit` endpoint.
  4. Node.js forwards code to Judge0, which runs it in a Docker container with test cases.
  5. Return execution results (pass/fail, runtime, memory) to the front-end.

3.4 Generating Test Cases

- **Requirements**: Automatically generate diverse test cases to cover edge cases. Maximize code coverage.
- **Tech Stack**:
  - *ML/DL*: Genetic Algorithm with RL (Python, DEAP, Gym).
  - *Back-End*: Node.js to trigger test case generation.
  - *Tools*: JaCoCo for code coverage analysis.
- **Steps**:
  1. Parse the problems input/output format and constraints using a Python script.
  2. Genetic algorithm generates initial test case population (e.g., random inputs).
  3. RL agent evaluates test cases, rewarding those that increase code coverage (via JaCoCo).
  4. Store generated test cases in MongoDB with problem ID.
  5. Back-end API retrieves test cases for code execution.

3.5 Validating Test Cases

- **Requirements**: Ensure test cases are correct and cover edge cases. Provide instant feedback on submission.
- **Tech Stack**:
  - *Back-End*: Judge0 for executing code against test cases.
  - *Database*: MongoDB to store test case results.
  - *Queue*: RabbitMQ for asynchronous validation.
- **Steps**:
  1. On code submission, queue test case execution in RabbitMQ.
  2. Judge0 runs code against each test case in a Docker container.
  3. Compare output with expected output; store results in MongoDB.
  4. Notify user via Socket.IO with pass/fail status and feedback.

3.6 Plagiarism Checking

- **Requirements**: Detect code similarity across submissions. Flag potential plagiarism in real-time.
- **Tech Stack**:
  - *ML/DL*: Siamese Neural Network with CodeBERT embeddings (Python, PyTorch).
  - *Tools*: Moss for structural similarity.
  - *Database*: MongoDB to store code submissions.
- **Steps**:
  1. On submission, preprocess code to remove comments and normalize formatting.
  2. Moss tokenizes code and compares it with historical submissions.
  3. CodeBERT generates embeddings; Siamese network computes similarity scores.
  4. If similarity exceeds threshold (e.g., 90%), flag submission and notify admin.
  5. Store results in MongoDB for audit.

3.7 Collaborative Debugging

- **Requirements**: Real-time code sharing and debugging with voice/video support. Highlight errors and breakpoints.
- **Tech Stack**:
  - *Front-End*: CodeMirror/Monaco Editor for shared editing.
  - *WebSocket (Socket.IO)*: For real-time code syncing.
  - *WebRTC*: For voice/video communication.
  - *Back-End*: Node.js to manage debugging sessions.
- **Steps**:
  1. Users join a debugging session via a unique room ID.
  2. CodeMirror syncs code changes via Socket.IO.
  3. WebRTC establishes peer-to-peer voice/video connection.
  4. Back-end API logs debugging session metadata in PostgreSQL.
  5. Highlight runtime errors using Judge0 output.

3.8 Updating User Ratings

- **Requirements**: Update user ratings based on challenge outcomes using Elo system. Store historical ratings for analytics.
- **Tech Stack**:
  - *Back-End*: Node.js for rating calculations.
  - *Database*: PostgreSQL for storing ratings and match history.
  - *Cache*: Redis for fast access to current ratings.
- **Steps**:

1. After a match, calculate rating changes using Elo formula based on win/loss and expected outcome.
2. Update user ratings in PostgreSQL and Redis.
3. Store match details (winner, rating change) in PostgreSQL.
4. Notify users of rating updates via Socket.IO.

3.9 Timer Settings

- **Requirements**: Set challenge timers with real-time countdown. Handle time-out penalties.
- **Tech Stack**:
  - *Front-End*: React.js for displaying countdown timer.
  - *WebSocket (Socket.IO)*: For syncing timer across users.
  - *Back-End*: Node.js to manage timer logic.
- **Steps**:
  1. On room creation, set timer duration (e.g., 30 minutes) in back-end.
  2. Broadcast timer start via Socket.IO to both users.
  3. React updates countdown UI in real-time.
  4. On timeout, penalize user (e.g., deduct rating) and end match.

3.10 Real-Time Communication Setup

- **Requirements**: Support text chat, voice, and video for users and spectators.
- **Tech Stack**:
  - *WebSocket (Socket.IO)*: For text chat and code syncing.
  - *WebRTC*: For voice/video communication.
  - *Front-End*: React.js for chat UI.
  - *Back-End*: Node.js to manage communication sessions.
- **Steps**:
  1. Users join a room; Socket.IO establishes a WebSocket connection.
  2. Text messages are sent via Socket.IO and displayed in React chat UI.
  3. WebRTC sets up peer-to-peer voice/video streams for users/spectators.
  4. Store chat logs in MongoDB for moderation.

3.11 Leaderboard and Analytics Generation

- **Requirements**: Display real-time leaderboards and user performance analytics. Track metrics like win rate, average solving time, and code quality.
- **Tech Stack**:
  - *Front-End*: Chart.js for analytics visualization.
  - *Back-End*: Node.js for aggregating analytics.
  - *Database*: PostgreSQL for leaderboard data, MongoDB for analytics logs.

  - *Cache*: Redis for real-time leaderboard updates.

- **Steps**:
  1. After each match, store performance metrics (win/loss, time, score) in MongoDB.
  2. Aggregate metrics (e.g., win rate) using Node.js and store in PostgreSQL.
  3. Redis maintains a sorted set for leaderboard rankings.
  4. React with Chart.js displays leaderboard and analytics charts.

3.12 User Authentication/Authorization

- **Requirements**: Secure user login, registration, and role-based access (e.g., user, admin). Support inviting friends and creating custom challenges.
- **Tech Stack**:
  - *Back-End*: Node.js with Express.js for auth APIs.
  - *Auth*: JSON Web Tokens (JWT) for session management.
  - *Database*: PostgreSQL for user credentials and roles.
  - *Cache*: Redis for session storage.
- **Steps**:
  1. User registers/logs in via `/api/auth/register` or `/api/auth/login`.
  2. Node.js generates JWT and stores session in Redis.
  3. For custom challenges, API allows users to invite friends via email or user ID.
  4. PostgreSQL stores user roles (e.g., admin for question management).
  5. JWT verifies access to protected routes (e.g., `/api/challenges/create`).

4 Scalability Considerations

- **Horizontal Scaling**: Use Kubernetes or AWS ECS to scale Node.js servers and Judge0 containers.
- **Load Balancing**: NGINX distributes traffic across servers.
- **Caching**: Redis caches frequently accessed data (ratings, leaderboards).
- **Database Sharding**: PostgreSQL sharding for user data; MongoDB for submissions.
- **Asynchronous Processing**: RabbitMQ handles time-intensive tasks like test case generation and plagiarism checks.

5 Conclusion

The proposed technology stack leverages modern web technologies (React, Node.js, WebSocket) and robust databases (PostgreSQL, MongoDB) to ensure scalability and real-time performance. Machine learning and deep learning algorithms like Siamese Networks, LSTMs, and Genetic Algorithms enhance features such as plagiarism detection and test case generation. This architecture supports all required features while maintaining security, scalability, and a seamless user experience.