Project Documentation: Coding Judge (Remote Code Execution Engine)

Section 1: Technology Stack & Dependencies

This section details the core technologies and specific libraries used in the project, explaining **what** they are and **why** they were chosen for this specific use case.

1. Core Technologies

- Node.js (Runtime):
  Chosen for its Event-Driven, Non-Blocking I/O model. Since a Coding Judge handles many concurrent file uploads and network requests, Node.js ensures the main thread never freezes while waiting for Input/Output operations.
- Express.js (Framework):
  A minimal web framework used to structure the REST API, manage routes, and handle middleware (authentication, validation).
- MongoDB (Database):
  A NoSQL database used for its flexible schema. It stores metadata (User profiles, Question details, Submission results) where the structure might evolve over time.
- Redis (In-Memory Store):
  Used as the Message Broker. It acts as a temporary "waiting room" for code submissions before the Worker picks them up. It is crucial for decoupling the API from the heavy execution engine.
- C++ (G++ Compiler):
  The underlying engine used to actually compile and execute the user's code. We use the child_process module to spawn system commands that invoke G++.

- Docker & Containerization
  - What is it? A platform to package the application and all its dependencies (Node.js, G++ compiler, Redis client) into a "Container."
  - Why we used it:
    - Consistency: It solves the "It works on my machine" issue. The C++ compiler version is locked inside the Linux container, so code runs exactly the same on Windows, Mac, or Cloud Servers.
    - Microservices: We use `docker-compose` to orchestrate 4 separate services (API, Worker, Redis, MongoDB) that communicate over a virtual internal network, mimicking a real production cluster.

2. Key NPM Dependencies

- bull:
  - What it is: A powerful queue system based on Redis.
  - Why we used it: To manage the Submission Queue. It handles "Producer-Consumer" logic, ensuring that if 100 users submit code at once, the server doesn't crash; instead, bull lines them up to be processed one by one (or in parallel batches).
- multer:
  - What it is: Middleware for handling multipart/form-data.
  - Why we used it: Standard JSON requests cannot send files. Multer intercepts the request, extracts the binary file (e.g., code.cpp), and securely saves it to the local disk (uploads/ folder).
- jsonwebtoken (JWT):
  - What it is: A library to generate and verify tokens.
  - Why we used it: For Stateless Authentication. Instead of storing sessions in server RAM (which is hard to scale), we issue a signed token. The server verifies the signature on every request to know who the user is.
- express-validator:
  - What it is: A set of express.js middlewares that wraps validator.js.
  - Why we used it: To sanitize and validate inputs before they reach the controller. For example, ensuring question_id is a valid MongoDB ID prevents database crashes.
- child_process (Built-in Node Module):
  - What it is: Allows Node.js to run shell commands (like Terminal commands).

- Why we used it: This is the bridge between JavaScript and the OS. We use it to run g++ main.cpp -o app and ./app directly from the code.

3. Deep Dive: Workers & Scalability

One of the most critical design choices in this project is the Worker Architecture.

- What is a "Worker"?

  A Worker is a separate Node.js process (running worker.js) that acts as a "Consumer." It does not handle HTTP requests from users. Its only job is to listen to the Redis Queue, pull out a pending submission, and execute the heavy C++ compilation task.
- How are Workers assigned to Submissions? We use the Competing Consumers Pattern.
  1. The API (Producer) pushes a Job ID into Redis.
  2. Redis holds the job in a First-In-First-Out (FIFO) queue.
  3. The Worker (Consumer) is constantly "polling" Redis. When it sees a job, it "claims" it.
  4. Locking: Once a worker claims a job, Redis locks it so no other worker can touch it. This prevents double-execution.
- How do we handle high traffic (Scaling)? If 1,000 users submit code at once, a single worker will take too long. To scale this:
  - Horizontal Scaling: We simply open more terminals and run node worker.js multiple times.
  - Logic: Redis is smart enough to distribute the jobs among 5, 10, or 100 workers automatically. We don't need to change a single line of code to scale up; we just add more Worker instances.

Section 2: End-to-End System Workflow (Deep Dive)

This section traces the exact lifecycle of data, highlighting key architectural decisions.

The Docker Architecture (Microservices & Shared Volumes)

When running in Docker, the system is split into 4 Isolated Containers:

1. judge_mongo: The Database.
2. judge_redis: The Message Queue.
3. judge_api: The REST API (Producer).
4. judge_worker: The Code Executor (Consumer).

💡 **ARCHITECTURAL CHALLENGE:**

The "Split Brain" Problem Since the API and Worker are in different containers (like different laptops), if the API saves a file to uploads/, the Worker *cannot* see it by default.

The Solution: **Shared Volumes** We use a Docker Volume named shared_processing. This acts as a "Shared Network Drive" mounted to /app/processing in both containers.

- Step 1: API container saves code.cpp to the shared volume.
- Step 2: Worker container instantly sees code.cpp in its folder because they share the same physical storage.

Part A: The Admin Lifecycle (Creating a Question)

1. Authentication & Authorization

- The Admin sends a POST request to create a question.
- Middleware: The server checks the JWT. It queries MongoDB to confirm the user has the admin role before allowing access.

2. File Handling (The "Split Storage" Strategy)

- The request contains Text Data (title) and Binary Files (input.txt, solution.txt).
- Action: Multer streams the files directly to the server's Local Disk (uploads/ folder).
- Action: The Controller saves the *File Path* (string) and the Title to MongoDB.

💡 ARCHITECTURAL DECISION: Why Local Storage instead of MongoDB?

1. Compiler Requirement: The C++ compiler (g++) requires a physical file path on the Operating System to run (e.g., C:\files\code.cpp). It cannot compile text stored inside a database cell or a variable.
2. If we will store code files in mongo db as object part,then first we will have to convert them to a cpp file then online compiler can run them this is a complicated task so store files in firebase storage or local storage.
3. Performance: Databases are optimized for querying structured data (JSON), not for storing large binary blobs (BLOBs). Storing files in MongoDB increases the database size rapidly, slowing down queries and backups.
4. Cost: Disk storage is significantly cheaper than Database storage (RAM/CPU resources).

Part B: The User Submission Lifecycle

1. Submission Reception (The Producer)

- User uploads code.cpp.
- Validation: Server checks if the file exists and the user is logged in.
- Queueing: Instead of running the code immediately, the API adds a "Job" to the Redis Queue and immediately responds to the user: *"Submission Queued"*.

💡 ARCHITECTURAL DECISION: Why use a Redis Queue? If we ran the C++ compiler inside the API route, the server would "freeze" for 1-2 seconds per user. If 100 users submitted at once, the server would crash. The Queue allows the API to remain Non-Blocking and fast, accepting thousands of requests while the Workers process them at their own pace.

2. The Worker (The Consumer)

- Job Detection: The Worker (listening to Redis) pulls the job.
- File Reconstruction:

- It reads the user's code from the job data and writes it to processing/sub.cpp.
- It fetches the Admin's input.txt and solution.txt from the Local Disk (using the paths stored in MongoDB).
-

3. Execution (The Sandbox)

- Compilation: Worker runs g++ sub.cpp -o a.exe. If this fails, verdict is COMPILATION ERROR.
- Runtime: Worker runs a.exe < input.txt > output.txt.
  - Security: We use a timeout command to kill the process if it runs too long (preventing infinite loops).
- Judgment: Worker compares output.txt vs solution.txt.

4. Result Storage

- The Worker updates MongoDB with the final verdict (ACCEPTED / WRONG ANSWER) and execution time.
- Cleanup: The Worker deletes the temporary files (.cpp, .exe, .txt) from Local Storage to prevent the disk from filling up.

Part C: The Docker Workflow (Under the Hood)

When running with Docker, the application workflow changes from running on a single operating system to interacting across four isolated environments (containers). Here is exactly how it works:

1. The Build Phase ("Baking the Image")

When you run docker compose up --build, Docker reads the Dockerfile.

- Base Layer: It downloads a lightweight Linux OS (node:18-slim) instead of using your Windows OS.
- Tooling: It runs apt-get install g++ inside this Linux layer. This guarantees that the compiler is always available, even if the host computer (yours) doesn't have G++ installed.
- Result: It creates a reusable "Image"—a frozen snapshot of the application with all dependencies installed.

2. The Orchestration Phase ("The Private Network")

Docker Compose spins up 4 separate containers: judge_mongo, judge_redis, judge_api, and judge_worker.

- Internal DNS: Docker creates a private virtual network. Inside this network, the containers call each other by name.
  - The API connects to mongodb://mongo:27017 (not localhost).
  - The Worker connects to redis:6379.
- Isolation: These ports are not exposed to the outside internet (except for the API on port 5000), making the database secure by default.

3. The Execution Phase ("The Shared Volume")

This is the most critical part of the Docker workflow for this project.

- The Problem: Since the API and Worker are in different containers (like two different laptops), the Worker normally cannot see files uploaded to the API.
- The Solution (Volumes): We map a folder on the host machine (your computer) to /app/processing inside both containers.
  1. User Upload: API Container receives code.cpp and saves it to /app/processing/code.cpp.
  2. Magic Sync: Because of the shared volume, this file instantly appears in the Worker Container's /app/processing/ folder.
  3. Execution: The Worker compiles the code using its internal Linux G++ compiler (not your Windows one) and executes it safely within the container's isolated boundaries.

Section 3: The "Live Demo" Script (How to Present)

Scenario: You haven't touched the project in 2 months. You are in the interview, sharing your screen. Follow these exact steps to look like a pro.

Phase 1: The Setup (Do this BEFORE sharing screen if possible)

1. Start the Infrastructure:
    - MongoDB: Ensure your local MongoDB is running (or you have the cloud URL in .env).
    - Redis: Open a terminal and run redis-cli ping. If it doesn't say PONG, start the Redis service.
2. Verify Environment: Check your .env file. Ensure PORT=5000 and MONGO_URL are correct.

Phase 2: Launching the Application (Share Screen Now)

Option A: The Docker Way (Recommended)

1. Open Terminal: Run the orchestrator command:
    - *docker compose up --build*
2. Explain the Logs: Point to the colorful scrolling text.
    - *"Here you can see the Redis container initializing..."*
    - *"Here is MongoDB waiting for connections..."*
    - *"And finally, the API and Worker containers have connected to the network."*

Option B: The manual Way

Tell the interviewer: *"I will now start the microservices: the API Server and the Judge Worker."*

1. **Open VS Code Terminal 1 (The API):**

    Bash
    npm run dev
    Wait for: "Server Running on Port 5000" and "MongoDB Connected".
2. **Open VS Code Terminal 2 (The Worker):**

    Bash
    node worker.js
    Wait for: "Worker is running and waiting for jobs..."

Phase 3: Generating Access Tokens

Tell the interviewer: *"Since the system is secure, I need to generate JWT tokens for both an Admin and a User using Google OAuth."*

1. Open Browser: Go to http://localhost:5000/api/admin/auth.
    - Sign in. Copy the Token displayed on the screen.
    - Paste it into a Notepad/Sticky Note labeled "ADMIN TOKEN".
2. Open Browser: Go to http://localhost:5000/api/user/auth.
    - Sign in. Copy the Token.
    - Paste it into the Notepad labeled "USER TOKEN".

Phase 4: The Admin Flow (Creating a Problem)

Tell the interviewer: *"First, acting as an Admin, I will upload a new coding problem with test cases."*

1. Open Postman.
2. Create Request: POST http://localhost:5000/api/admin/questions/create
3. Auth: Tab > Bearer Token > Paste ADMIN TOKEN.
4. Body (form-data):
    - title: "Sum of Two"
    - content:”return sum of two integers.
    - time_limit:1
    - input_file: [Upload input.txt with content 5 10]
    - solution_file: [Upload output.txt with content 15]
5. Hit Send.

6. Action: Copy the _id from the response. Say: *"The question is now stored in the database and files are on the disk."*

Phase 5: The User Flow (Solving the Problem)

Tell the interviewer: *"Now, I will switch roles to a User and submit a C++ solution."*

1. New Request: POST http://localhost:5000/api/user/submission
2. Auth: Tab > Bearer Token > Paste USER TOKEN.
3. Body (form-data):
    - question_id: [Paste the ID you copied]
    - submission_file: [Upload code.cpp]
    - *Code content:*

```
C++
#include <iostream>
using namespace std;
int main() { int a, b; cin >> a >> b; cout << a + b; return 0; }
```

4. **Hit Send.**

5. **Show the Magic:**
    - Point to Terminal 2 (Worker) immediately.
    - Show the logs: Processing Job... Compiled... Executed... Verdict: ACCEPTED.
    - Point to Postman Response: Show status: "ACCEPTED".

6: Verification (History API)

Tell the interviewer: *"Finally, the user can see their past submissions to track progress."*

1. New Request: GET http://localhost:5000/api/user/submission/history
2. Auth: Bearer Token > Paste USER TOKEN.
3. Hit Send.
4. Explain: Show the JSON array. *"This fetches data from MongoDB, populating the question details to show the user exactly what they solved."*

*7: The Security Demo (The Sandbox)***

A key feature of an Online Judge is security. I will now attempt to break the system with an Infinite Loop.

- Switch to Postman:POST /api/user/submission
- Upload File: Select a C++ file with this content:
    - *cpp int main() { while(true); }*
- Hit Send.
- Action:Quickly switch to the Worker Terminal.
- Point out:"See? The worker started the job, but after 1 second, it forcefully killed the process due to the Time Limit. The server is still running safely."*
- Show History:Check the history API to show the `TIME LIMIT EXCEEDED` status.

Section 4: Key Concepts & General Use Cases

This section explains the core technologies used in this architecture, detailing what they are and how they are generally used in the software industry.

1. Redis (Remote Dictionary Server)

- What is it? Redis is an open-source, in-memory data structure store. unlike traditional databases (SQL/Mongo) that save to a hard drive, Redis keeps data in RAM. This makes it incredibly fast (microseconds) but means data can be lost if the server loses power (volatility).
- Analogy: Think of your hard drive as a Library (slow to find books, but safe storage) and Redis as your Desk (very limited space, but instant access to whatever is on it).
- General Industry Use Cases:
    - Caching: Storing frequently accessed data (like User Profiles or Product Prices) so the app doesn't have to query the slow database every time.
    - Session Management: Storing "Logged In" tokens for millions of users because checking RAM is faster than checking a disk.
    - Real-Time Leaderboards: Gaming companies use Redis sorted sets to track scores in real-time because it handles rapid updates easily.

2. Workers (Background Jobs)

- What is it? A "Worker" is a separate process or server dedicated to running long-running tasks in the background, keeping the main Web Server (API) free to respond to user requests instantly.
- Analogy: In a restaurant, the Waiter (API) takes your order and immediately goes to the next table. The Chef (Worker) cooks the food in the back. If the waiter had to go into the kitchen and cook every meal himself, he could only serve one customer at an hour.
- General Industry Use Cases:
    - Email/Notifications: When you sign up for a service, the "Welcome Email" is sent by a worker so you don't stare at a loading screen.
    - Image Processing: When you upload a photo to Instagram, a worker resizes it and applies filters in the background.
    - Report Generation: Generating a large PDF bank statement is done by a worker so the banking app remains responsive.

3. Bull (The Queue Manager)

- What is it? Bull is a Node.js library that manages queues based on Redis. While Redis acts as the storage, Bull provides the logic to manage the flow of work (producer-consumer pattern).
- Analogy: Redis is the physical waiting room chairs. Bull is the Receptionist who calls your name, makes sure people don't cut in line, and calls you back if you missed your turn.
- General Industry Use Cases:
    - Rate Limiting: Ensuring a system only processes 50 requests per second to prevent crashing external APIs.
    - Retries: If a payment fails due to a network error, Bull can automatically retry the job 3 times before giving up.
    - Scheduled Tasks: Running jobs at specific times (e.g., "Send a newsletter every Monday at 9 AM").

4. Sandboxing

- What is it? Sandboxing is a security mechanism for separating running programs. It executes code in a restricted environment with limited access to the host machine's resources (files, network, memory).
- Analogy: It is like a Bomb Disposal Unit. If you find a suspicious package, you don't open it in the lobby. You put it inside a reinforced steel chamber (Sandbox) so that if it explodes, it doesn't destroy the building.
- General Industry Use Cases:
    - Web Browsers: Chrome runs every tab in a sandbox. If one tab crashes or has a virus, it cannot steal data from other tabs or crash your entire browser.
    - SaaS Plugins: Tools like Figma or Shopify allow developers to write plugins. These plugins run in a sandbox so they can't delete user data or hack the main platform.
    - Mobile Apps: Android and iOS sandbox every app so "Flappy Bird" cannot read your Banking App's data.

5. Docker Volumes

- **What is it?** Imagine two people (Containers) living in different houses. They cannot see what's in each other's fridge. A **Volume** is like a magic tunnel connecting their fridges. If Person A puts a cake in, Person B can take it out.
- **Why we used it:** Our API receives the file, but the Worker needs to run it. We use a Volume to share the file between the two separate containers.

Two show actual Queue funtionality

You are asking a very sharp question about the system architecture.
Currently, you are seeing ACCEPTED in Postman because your code is running in **Synchronous Mode** (Blocking). The API server is politely waiting for the Worker to finish before replying to you.
To see the **"Queued"** message and the **PENDING** status, we must switch the API to **Asynchronous Mode** (Non-Blocking).
Here is exactly what happens in both scenarios:
Scenario 1: Current Setup (Synchronous / Blocking)

- **Behavior:** The API adds the job to the queue and **pauses** (await job.finished()). It waits until the Worker says "I'm done!"
- **Postman Response:** You wait 1-2 seconds, then get the final result:
  JSON
  { "status": "ACCEPTED", "execution_time": 0.05 }
- **Why use this?** Good for simple sites where users want immediate feedback.
  Scenario 2: Heavy Load Setup (Asynchronous / Non-Blocking)
- **Behavior:** The API adds the job to the queue and **replies immediately**. It does NOT wait for the worker.
- **Postman Response:** You get an instant response (in milliseconds):
  JSON

```
{
  "message": "Submission Queued! We are processing it in the background.",
  "status": "PENDING"
}
```

- **What happens to the Verdict?** Since the worker hasn't finished yet, the status in the database is set to **PENDING**.
- **How does the user see "ACCEPTED"?** The user must check the **History API** (GET /history) a few seconds later. By then, the worker will have finished and updated the database from PENDING → ACCEPTED.

How to see the "Queued" & "PENDING" status right now
To experience this, you need to swap the code in controllers/user/submission.js.

1. **Open controllers/user/submission.js 2. Comment out the "Blocking" logic and Uncomment the "Non-Blocking" logic.**
2. Now Uncomment the async function submitFile (there are two functions with same name),uncomment the one here working of redis queue is written
3. Now on submission the api will reply immediately without waiting the submission is queued.
4. After on worker terminal the output will displaced a bit late based on the queue size.
5. After output display when you will check the submission history then it will show the actual result of compilation.
6. If you use the another function commenting this one the api will wait until execution I done, and as the output is displaced on worker terminal the response will be send on postman.

The Experiment

1. **Restart Server:** npm run dev.
2. **Restart Worker:** node worker.js.
3. **Postman:** Send a submission.
4. **Result:**
   - **Postman** immediately returns: "status": "PENDING".
   - **Terminal 2 (Worker)** wakes up a moment later: Job Received... Compiled... Verdict: ACCEPTED.
   - **Postman History:** If you check GET /history now, that same submission will now say "status": "ACCEPTED".

Section 5: Challenges & Technical Solutions

This section outlines the critical engineering hurdles encountered during development and the specific architectural patterns used to solve them.

1. Challenge: The "Blocking Server" Problem

- **The Issue:** Compiling C++ code is CPU-intensive and takes 1-2 seconds. In a standard REST API, if the server executes this code in the main thread, it "freezes" the entire application. During this 2-second window, no other user can log in or submit code.
- **The Solution: Asynchronous Processing (Queues)**
    - We decoupled the **API (Producer)** from the **Execution Engine (Consumer)** using **Redis** and **Bull**.
    - Now, when a user submits code, the API simply adds a "Job" to the queue and returns 200 OK immediately. A separate **Worker Process** picks up the job and runs it in the background. This allows the API to handle thousands of requests per second without lagging.

2. Challenge: The "Split-Brain" File Storage (Docker)

- **The Issue:** In a containerized environment, the **API** and **Worker** run in separate isolated containers (like two different computers). When the API saved an uploaded file to its local uploads/ folder, the Worker container could not access it to compile the code.
- **The Solution: Shared Docker Volumes**
    - We implemented a **Docker Volume** strategy. We mapped a single host directory (./processing) to /app/processing inside **both** containers.
    - This acts as a "Network Shared Drive," allowing the API to write a file and the Worker to instantly read it, maintaining data consistency across microservices.

3. Challenge: Handling Malicious Code (Security)

- **The Issue:** Allowing users to run C++ code on our server is dangerous. A user could submit an infinite loop (while(true)) or a fork bomb, causing the CPU to spike to 100% and crashing the server for everyone.
- **The Solution: Sandboxing & Timeouts**
    - We implemented a strict **Time Limit Enforcement** mechanism.
    - On Linux/Docker, we wrap execution in a timeout command. On Windows/Node, we use child_process.exec with a timeout parameter.
    - If a process exceeds 2 seconds, the Worker forcefully kills the process (SIGKILL) and marks the submission as TIME LIMIT EXCEEDED, protecting the server's availability.

4. Challenge: File Locking on Windows (EBUSY Error)

- **The Issue:** During testing on Windows, the Worker frequently crashed with EBUSY: resource busy or locked when trying to delete temporary files. This happened because the Operating System hadn't fully released the file lock from the C++ executable before the cleanup script ran.
- **The Solution: Graceful Cleanup Strategy**
    - We implemented a **Resource Release Delay**. The worker now waits 100ms after execution finishes before attempting deletion.
    - We also wrapped the cleanup logic in a try-catch block. If a file is locked, the system logs a warning instead of crashing the entire worker process.

5. Challenge: Data Consistency in Async Flows

- **The Issue:** Because the system is asynchronous, the user receives a "Success" response before the code is actually compiled. If we didn't handle this, the user would see "Status: NULL" or missing data immediately after submission.
- **The Solution: The "Pending" State Pattern**

- We introduced a PENDING status in the MongoDB schema.
- **Flow:**
    1. API creates a DB Record (status: PENDING) -> Returns ID to user.
    2. Worker processes job -> Updates DB Record (status: ACCEPTED).
- This ensures the database always has a valid state, and the user can poll the **History API** to see the status update in real-time.