Project Analysis Report

Video Name: Build and Deploy a LeetCode Clone with React: Next JS, TypeScript, Tailwind CSS, Firebase **Video Links:**

- Part 1: https://www.google.com/search?q=https://youtu.be/igqiduZR-Gg
- Part 2: https://www.youtube.com/watch?v=cSYDnjOTlQk

1. Executive Summary

This project is a full-stack **LeetCode Clone** designed to replicate the core functionality of the popular coding interview platform. It provides a collaborative environment where users can browse algorithmic problems, write and execute code in a browser-based editor, and track their progress.

The application features a robust authentication system, a real-time database for storing problem metadata and user progress (likes, dislikes, solved status), and a rich code editing interface. Unlike traditional clones that might use a heavy backend for code execution, this project cleverly utilizes **client-side code evaluation** to simulate a judging environment, keeping the architecture serverless and cost-effective.

2. Tech Stack Overview

Component	Technology Used	Reason for Choice
Frontend	Next.js (React)	Server-side rendering (SSR) for SEO and fast initial load, file-based routing.
Language	TypeScript	Strong typing to prevent runtime errors and improve developer experience.
Styling	Tailwind CSS	Rapid UI development with utility classes; Dark mode support.
Backend / DB	Firebase	Serverless architecture providing Authentication and Firestore (NoSQL database).
State Management	Recoil	Efficient, atom-based state management for global UI states (modals, user data).
Code Editor	CodeMirror	Lightweight, extensible text editor component for the coding workspace.
Deployment	Vercel	Optimized hosting for Next.js applications.

3. Frontend Architecture

Core Technologies

- **Framework:** Next.js (Pages Router approach).
- **State Management:** Recoil (Atoms for Auth Modal, Problem State).
- **Notifications:** `react-toastify` for success/error alerts.
- **Layout Engine:** `react-split` for the resizable split-screen workspace.

Key Features

- **Problem Explorer:** A sortable list of coding problems with difficulty tags and solution status.
- **Coding Workspace:** A split-view interface:
  - **Left Panel:** Problem description, examples, and constraints (rendered HTML).
  - **Right Panel:** Interactive code editor (`@uiw/react-codemirror`) and test case runner.
- **Authentication Modals:** Custom-built modals for Login, Signup, and Password Reset (managed via Recoil state).
- **Video Solutions:** Embedded YouTube player for video walkthroughs using `react-youtube` .
- **Settings Modal:** Allows users to customize editor settings (font size, etc.).

4. Backend Architecture (Serverless)

Core Technologies

- **Platform:** Firebase (Backend-as-a-Service).
- **Authentication:** Firebase Auth (Email/Password).
- **Database:** Cloud Firestore.

Data Logic

- **User Management:** Handles user registration and session persistence.
- **Problem Metadata:** Stores dynamic data such as:
  - `likes` : Counter for problem likes.
  - `dislikes` : Counter for problem dislikes.
  - `starred` : Boolean flag for user favorites.
- **Transactional Updates:** Uses Firestore Transactions to ensure data consistency when updating like/dislike counts concurrently.

Code Execution Engine (Client-Side)

Instead of a dedicated backend compilation server (like Piston or Judge0), this project uses a clever **local execution model**:

1. **Boilerplate Code:** Each problem comes with a starter function signature.

2. **Handler Functions:** A local `handler` function is defined for each problem.
3. **Evaluation:** The user's code string is converted to a function using `new Function()` and passed to the handler.
4. **Assertion:** The handler runs the user's function against pre-defined inputs and compares the output with expected results directly in the browser.

5. Database Information

System

- **Database:** Google Cloud Firestore (NoSQL).

Schema / Collections

`users` **Collection:**

- `uid` (Document ID): Unique User ID.
- `email` : User's email address.
- `displayName` : User's name.
- `createdAt` : Timestamp.
- `updatedAt` : Timestamp.
- `likedProblems` : Array of strings (Problem IDs).
- `dislikedProblems` : Array of strings.
- `solvedProblems` : Array of strings.
- `starredProblems` : Array of strings.

`problems` **Collection:**

- `id` (Document ID): Problem slug (e.g., "two-sum").
- `title` : "Two Sum".
- `difficulty` : "Easy" | "Medium" | "Hard".
- `category` : "Array", "DP", etc.
- `order` : Integer (for sorting).
- `videoId` : String (YouTube Video ID).
- `link` : String (Link to LeetCode original).
- `likes` : Number.
- `dislikes` : Number.

6. Folder Structure

```
root/
├── components/
│   ├── Buttons/            # Logout, OAuth buttons
```

```
│   ├── Modals/             # AuthModal, SettingsModal, ResetPassword
│   ├── Navbar/             # Top navigation
│   ├── Workspace/          # Main coding area
│   │   ├── Playground/     # Editor & Test Cases
│   │   └── ProblemDescription/
│   └── Skeletons/          # Loading states
├── firebase/               # Firebase config & init
├── hooks/                  # Custom hooks (useLocalStorage, useHasMounted)
├── mockProblems/           # Static problem data (descriptions, test cases)
├── pages/
│   ├── auth/               # Authentication page
│   ├── problems/
│   │   └── [pid].tsx       # Dynamic problem page (SSR/SSG)
│   ├── _app.tsx            # Main app wrapper (RecoilRoot, ToastContainer)
│   └── index.tsx           # Home page (Problem List)
├── styles/                 # Global CSS (Tailwind imports)
├── atoms/                  # Recoil atoms (authModalAtom, etc.)
└── utils/                  # Utility functions & types
```

7. References & Resources

- **GitHub Repository:** (Likely "leetcode-clone" by burakorkmez or similar)
- **Libraries:**
  - `recoil` : State Management.
  - `react-split` : Resizable panels.
  - `@uiw/react-codemirror` : Code Editor.
  - `react-confetti` : Success animation.
  - `react-firebase-hooks` : Simplified Firebase hooks.