🚀 LeetCode Clone - Complete Project Documentation

Project Overview: A full-stack competitive programming platform built with modern technologies, featuring real-time code execution, user authentication, and a comprehensive problem management system.

📋 Table of Contents

🎯 Project Overview

This project is a comprehensive LeetCode clone that provides a platform for competitive programming practice. It includes user authentication, problem creation, code submission, real-time execution, and result evaluation.

Key Objectives:

- Create a scalable competitive programming platform
- Implement secure code execution in sandboxed environments
- Provide real-time feedback on code submissions
- Support multiple programming languages
- Enable user authentication and problem management

🛠️ Technology Stack

Frontend

- Next.js 14.2.3 (React Framework)
- React 18 (UI Library)
- TypeScript (Type Safety)
- Tailwind CSS (Styling)
- Radix UI (Component Library)
- Monaco Editor (Code Editor)
- NextAuth.js (Authentication)
- React Hook Form (Form Management)
- Zod (Schema Validation)

Backend

- Node.js 20 (Runtime)
- TypeScript (Type Safety)
- Express.js (Web Framework)
- Prisma (Database ORM)
- Redis (Message Queue & Caching)
- Winston (Logging)

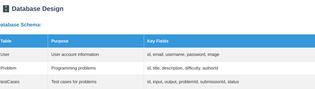
Database & Storage

- PostgreSQL (Primary Database)
- Redis (Message Queue)
- Prisma (Database ORM)

DevOps & Infrastructure

- Docker (Containerization)
- Docker Compose (Orchestration)
- Node.js 22 (Client Runtime)
- Node.js 20 (Worker Runtime)

🏗️ System Architecture



Architecture Components:

1. **Client Service:** Next.js frontend with user interface
2. **Worker Service:** Node.js backend for code execution
3. **Redis Queue:** Message broker for asynchronous processing
4. **PostgreSQL:** Primary database for data persistence
5. **Authentication:** NextAuth.js with OAuth providers

Data Flow:

1. User submits code through the frontend
2. Code is stored in PostgreSQL database
3. Submission reference is pushed to Redis queue
4. Worker picks up submission from queue
5. Worker executes code in sandboxed environment
6. Results are stored back in database
7. Frontend polls for results and displays them

✨ Core Features

🔐 User Authentication

- Google OAuth Integration
- GitHub OAuth Integration
- Email/Password Authentication
- Session Management

📝 Problem Management

- Rich Text Problem Editor
- Test Case Management
- Difficulty Classification
- Problem Publishing

⚡ Code Execution

- Monaco Editor Integration
- Multi-language Support (C++)
- Real-time Compilation
- Test Case Validation

📊 Submission Tracking

- Submission History
- Test Case Results
- Execution Status
- Performance Metrics

🎨 Modern UI/UX

- Responsive Design
- Dark/Light Theme
- Component Library
- Toast Notifications

⚡ Performance

- Asynchronous Processing
- Redis Caching
- Database Optimization
- Containerized Deployment

🗄️ Database Design

Database Schema:

Table	Purpose	Key Fields
User	User account information	id, email, username, password, image
Problem	Programming problems	id, title, description, difficulty, authorId
testCases	Test cases for problems	id, input, output, problemId, submissionId, status
Submission	User code submissions	id, code, language, problemId, userId, status

Relationships:

- **User** → **Problem**: One-to-Many (User can create multiple problems)
- **User** → **Submission**: One-to-Many (User can make multiple submissions)
- **Problem** → **testCases**: One-to-Many (Problem has multiple test cases)
- **Problem** → **Submission**: One-to-Many (Problem can have multiple submissions)
- **Submission** → **testCases**: One-to-Many (Submission has multiple test case results)

⚙️ Code Execution System

Custom Sandbox Implementation: Unlike traditional platforms that use Judge0, this project implements a custom code execution system for better control and security.

Execution Flow:

1. **Code Submission:** User submits code through Monaco Editor
2. **Queue Processing:** Submission is queued in Redis
3. **Worker Processing:** Worker picks up submission
4. **File Creation:** Code is written to temporary files
5. **Compilation:** Code is compiled using system compilers
6. **Execution:** Compiled code runs against test cases
7. **Result Comparison:** Output is compared with expected results
8. **Status Update:** Results are stored in database

Supported Languages:

- **C++:** Fully implemented with g++ compilation
- **JavaScript:** UI support (implementation pending)
- **Python:** UI support (implementation pending)

Security Measures:

- Docker containerization for isolation
- Time and memory limits on execution
- File system restrictions
- Process isolation
- Input validation and sanitization

🔐 Authentication System

Authentication Providers:

- **Google OAuth:** Social login integration
- **GitHub OAuth:** Social login integration
- **Credentials:** Email/password authentication

Security Features:

- bcryptjs password hashing
- JWT token management
- Session-based authentication
- OAuth 2.0 integration

🌐 API Endpoints

Server Actions (Next.js):

Action	Purpose	File Location
compileCode	Submit code for execution	actions/compileCode.ts
checkCompilation	Check submission status	actions/checkCompilation.ts
getCompiledTestCases	Get test case results	actions/getCompiledTestCases.ts
PublishAction	Publish new problem	actions/publish.ts
getSubmission	Get user submissions	actions/getSubmission.ts

🚀 Deployment & DevOps

Docker Configuration:

- **Multi-container setup** with Docker Compose
- **Client container:** Next.js frontend
- **Worker container:** Node.js backend
- **PostgreSQL container:** Database
- **Redis container:** Message queue

Environment Variables:

```
DATABASE_URL=postgresql://postgres:postgres@db:5432/leetcode REDIS_URL=redis://redis:6379
NEXTAUTH_URL=http://localhost:3000 NEXTAUTH_SECRET="password_neextauth" GOOGLE_CLIENT_ID=your_google_client_id
GOOGLE_CLIENT_SECRET=your_google_client_secret GITHUB_CLIENT_ID=your_github_client_id
GITHUB_CLIENT_SECRET=your_github_client_secret
```

Port Configuration:

- **Client:** Port 3000 (Frontend)
- **Worker:** Port 9080 (Backend)
- **PostgreSQL:** Port 5432 (Database)
- **Redis:** Port 6379 (Cache/Queue)
- **Prisma Studio:** Port 5555 (Database GUI)

🎯 Challenges & Solutions

Challenge 1: Secure Code Execution

Problem: Executing untrusted user code safely without compromising system security.

Solution:

- Implemented custom sandbox using Docker containers
- Added time and memory limits
- Restricted file system access
- Process isolation and monitoring

Challenge 2: Asynchronous Processing

Problem: Handling long-running code execution without blocking the user interface.

Solution:

- Implemented Redis message queue
- Separated worker service for code execution
- Real-time status polling
- Non-blocking user experience

Challenge 3: Database Performance

Problem: Managing high-frequency read/write operations for submissions and test cases.

Solution:

- Optimized database schema with proper indexing
- Implemented Redis caching for frequently accessed data
- Used Prisma ORM for efficient queries
- Database connection pooling

Challenge 4: Multi-language Support

Problem: Supporting multiple programming languages with different compilation and execution requirements.

Solution:

- Modular code execution system
- Language-specific compilation commands
- Extensible architecture for new languages
- Standardized input/output handling

🔮 Future Enhancements

Planned Features:

- **Additional Languages:** Python, JavaScript, Java, C# support
- **Advanced Analytics:** User performance tracking and analytics
- **Contest System:** Time-limited coding competitions
- **Discussion Forum:** Problem discussion and solutions sharing
- **Leaderboards:** Global and contest rankings
- **Code Templates:** Pre-built code templates for common patterns
- **Mobile App:** React Native mobile application
- **API Documentation:** Comprehensive API documentation
- **Performance Optimization:** CDN integration, caching strategies
- **Monitoring:** Application performance monitoring and logging

Technical Improvements:

- **Microservices:** Further service decomposition
- **Load Balancing:** Horizontal scaling capabilities
- **CI/CD Pipeline:** Automated testing and deployment
- **Monitoring:** Real-time application monitoring
- **Security:** Enhanced security measures and audits

📊 Project Statistics

Metric	Value
Total Lines of Code	~5000+ lines
Frontend Components	20+ components
Database Tables	4 main tables
API Endpoints	5+ server actions
Docker Containers	4 containers
Supported Languages	1 (C++) + 2 planned

🎓 Learning Outcomes

Technical Skills Developed:

- **Full-Stack Development:** Next.js, Node.js, TypeScript
- **Database Design:** PostgreSQL, Prisma ORM, Redis
- **System Architecture:** Microservices, Message queues
- **DevOps:** Docker, Docker Compose, containerization
- **Security:** Authentication, authorization, code sandboxing
- **Performance:** Caching, optimization, scalability

Soft Skills Developed:

- **Problem Solving:** Complex system design challenges
- **Architecture Design:** Scalable system planning
- **Security Awareness:** Secure coding practices
- **Performance Optimization:** System efficiency considerations
- **Documentation:** Comprehensive project documentation