A Comprehensive Development Roadmap for a MERN Stack Interview Preparation Platform

I. Foundational Setup & Project Architecture

The initial phase of any software development project is dedicated to establishing a robust and scalable foundation. The architectural decisions made at this stage have a profound impact on the entire development lifecycle, influencing developer productivity, maintainability, and the application's ability to evolve. This section outlines a meticulous, step-by-step process for setting up the development environment, structuring the project, and configuring the necessary tooling for the interview preparation platform. The approach prioritizes modern development practices, separation of concerns, and security from the outset.

Phase 1: Environment and Tooling

A standardized and correctly configured development environment is a prerequisite for efficient and error-free development. This phase ensures that all necessary software is installed and the foundational services, such as the database and version control, are properly initialized before any application-specific code is written.

Prerequisites and Core Installations

The MERN stack is built entirely on JavaScript technologies, with Node.js serving as the core runtime environment for the backend. The first step is to install the Long-Term Support (LTS) version of Node.js, which includes the Node Package Manager (npm). Using the LTS version ensures stability and access to the latest features while avoiding the potential instability of the most current release.
Simultaneously, version control must be established. Git is the industry standard for source code management, and initializing a Git repository is a non-negotiable first action. This practice safeguards against code loss, provides a detailed history of changes, and is essential for any future collaboration or deployment pipelines.

MongoDB Atlas Cloud Database Setup

While it is possible to run a MongoDB instance locally, a cloud-based Database-as-a-Service (DBaaS) like MongoDB Atlas is the superior choice for this project. It abstracts away the complexities of database administration, offers a generous free tier, and provides enterprise-grade features such as automated backups, security, and scalability.
The setup process involves creating a free account on the MongoDB Atlas platform, deploying a new cluster (the free M0 tier is sufficient for development and initial deployment), and configuring network access rules to allow connections from your development IP address. The most critical output of this process is the database connection string (URI), a unique identifier

that the backend server will use to connect to the cluster. This string contains sensitive credentials and must be handled securely.

Technology Stack & Tools Summary

To provide a clear and consolidated overview of the project's technical composition, the following table outlines the complete technology stack. This serves as a definitive reference for all components, libraries, and tools that will be utilized.

Category	Technology/Tool	Version (Recommended)	Rationale
Runtime	Node.js	LTS (e.g., v20.11.0+)	JavaScript runtime for the backend server.
Database	MongoDB Atlas	Latest	Cloud-native, scalable NoSQL database with a robust free tier.
Backend Framework	Express.js	~4.18.2	Minimalist and flexible web framework for Node.js to build the REST API.
DB Object Modeling	Mongoose	~8.0.0	Provides schema validation, data modeling, and simplifies MongoDB interactions.
Frontend Library	React	~18.2.0	A component-based UI library for building modern, interactive user interfaces.
Frontend Tooling	Vite	~5.0.0	A modern, exceptionally fast build tool and development server for React projects.
API Client	Axios	~1.6.0	A promise-based HTTP client for seamless frontend-backend communication.
Utility	CORS	~2.8.5	Node.js middleware to enable Cross-Origin Resource Sharing for the API.
Utility	dotenv	~16.3.1	Manages environment variables, keeping sensitive data like database URIs out of source code.
Version Control	Git	Latest	The standard for

Category	Technology/Tool	Version (Recommended)	Rationale
			tracking changes and managing the project's history.

Cursor AI Prompts: Environment Setup

The following prompts can be used with Cursor AI to guide the initial setup process.

1. **Initialize Git Repository:**
   ```
   Initialize a new Git repository in the current directory.
   ```
2. **Create .gitignore:**
   ```
   Create a standard.gitignore file for a MERN stack project. It
   should ignore node_modules directories in both the client and
   server,.env files, and production build artifacts.
   ```

Phase 2: Project Scaffolding

A well-organized project structure is crucial for maintainability. The standard practice for MERN applications is to adopt a monorepo-style structure with distinct directories for the client-side and server-side code. This separation of concerns simplifies development, dependency management, and future deployment strategies.

Monorepo Directory Structure

The project will begin with a root directory, named interview-prep-platform or similar. Inside this root, two primary subdirectories will be created:

- server: This will house the entire Node.js and Express.js backend application, including API routes, database models, and server configuration.
- client: This will contain the complete React frontend application, including components, styles, and state management logic.

This structure ensures that the backend and frontend are decoupled, communicating only through the defined API contract.

Backend Initialization

Within the server directory, the Node.js project is initialized by running the command npm init -y. This command generates a package.json file, which will track the project's metadata and dependencies. Following initialization, the core backend dependencies are installed:

- express: The web server framework.
- mongoose: The ODM for interacting with MongoDB.
- cors: Middleware to handle cross-origin requests from the React client.
- dotenv: To load environment variables from a .env file, which is critical for securely managing the MongoDB Atlas connection string.

Frontend Initialization with Vite

For the frontend, the React application will be scaffolded using Vite. While older guides often reference create-react-app (CRA), Vite represents a significant advancement in frontend tooling. CRA's development process relies on bundling the entire application with a tool like Webpack before serving it, which can lead to slow startup times and sluggish updates (Hot Module Replacement).
Vite takes a different approach by leveraging native ES modules in the browser. This allows it to serve code directly without a bundling step during development, resulting in a near-instantaneous server start and incredibly fast updates. This dramatic improvement in the developer feedback loop directly translates to higher productivity and a more pleasant development experience. The choice to use Vite is a strategic one, prioritizing modern, efficient tooling over legacy standards. The application is created by running npm create vite@latest client --template react in the project's root directory.

Cursor AI Prompts: Project Scaffolding

The following prompts will generate the directory structure and initialize both the backend and frontend projects.

1. **Create Project Structure:**
   ```
   Create a root directory for a MERN project. Inside this root
   directory, create two subdirectories: 'server' for the backend and
   'client' for the frontend.
   ```
2. **Initialize Backend:**
   ```
   Navigate into the 'server' directory. Initialize a new Node.js
   project and install the following dependencies: express, mongoose,
   cors, and dotenv.
   ```
3. **Initialize Frontend with Vite:**
   ```
   Navigate to the project's root directory. Use Vite to create a new
   React application inside the 'client' directory. After creation,
   navigate into the 'client' directory and install axios.
   ```

II. Backend Development: API and Data Persistence

The backend serves as the application's core, responsible for managing business logic, interacting with the database, and exposing data to the frontend via a well-defined Application Programming Interface (API). This section details the design of the database schemas, the implementation of RESTful API endpoints, and the strategy for populating the database with the initial educational content.

Phase 3: Database Schema Design

The content specified in the user's mockups—LeetCode problems, High-Level Design topics, and Low-Level Design topics—is structurally distinct. A NoSQL database like MongoDB offers schema flexibility, but enforcing a consistent structure at the application layer using Mongoose is a best practice for ensuring data integrity and predictability. Attempting to use a single, generic

schema for all three content types would result in an inefficient data model with numerous optional or irrelevant fields for any given document. Therefore, creating three separate, specialized Mongoose schemas is the optimal approach. Each schema will map to its own collection in the MongoDB database.
The following tables provide a definitive blueprint for these data structures, detailing the fields, data types, and validation rules for each. This serves as the foundational contract between the application logic and the database.

Table 1: MongoDB Collection Schemas

LeetCodeProblem Schema This schema is designed to capture all relevant details for a competitive programming problem, modeled after the structure of the "Two Sum" problem analysis. The use of a slug provides a clean, URL-friendly identifier for API requests.

Field Name	Data Type	Required	Description
title	String	Yes	The name of the problem (e.g., "Two Sum").
slug	String	Yes, Unique	URL-friendly identifier (e.g., "two-sum"). Used for API lookups.
category	String	Yes	The Blind 75 category (e.g., "Array", "String", "Graph").
difficulty	String	Yes	"Easy", "Medium", or "Hard".
problemStatement	String	Yes	The full description of the problem. Supports Markdown for formatting.
example	String	Yes	An example input/output. Supports Markdown.
constraints	String	Yes	List of constraints. Supports Markdown.
solution	Object	Yes	An embedded object containing the C++ solution and its analysis.
solution.cppCode	String	Yes	The complete, formatted C++ solution code.
solution.explanation	String	Yes	A step-by-step explanation of the solution's logic. Supports Markdown.
solution.timeComplexity	String	Yes	The time complexity analysis of the solution,

Field Name	Data Type	Required	Description
			e.g., "O(n)".
solution.spaceComplexity	String	Yes	The space complexity analysis of the solution, e.g., "O(n)".

HLDTopic Schema This schema is tailored for High-Level Design concepts, focusing on principles, architectural diagrams, and trade-offs, as derived from common HLD interview questions.

Field Name	Data Type	Required	Description
title	String	Yes	The name of the HLD topic (e.g., "Design a Load Balancer").
slug	String	Yes, Unique	URL-friendly identifier (e.g., "design-load-balancer").
category	String	Yes	A broad category for organization (e.g., "Scalability", "Databases").
coreConcepts	String	Yes	A detailed explanation of the fundamental principles. Supports Markdown.
architectureDiagram	Object	No	An optional object for linking to a diagram image.
architectureDiagram.url	String	No	The URL of the diagram, hosted on an image service.
architectureDiagram.alt	String	No	Descriptive alt text for the diagram for accessibility.
tradeoffs	String	Yes	A discussion of the pros and cons of different design choices. Supports Markdown.

LLDTopic Schema This schema is structured for Low-Level Design topics, particularly design patterns, emphasizing their intent, structure, and practical application with code examples.

Field Name	Data Type	Required	Description
title	String	Yes	The name of the LLD topic (e.g., "Singleton Design Pattern").
slug	String	Yes, Unique	URL-friendly identifier (e.g., "singleton-pattern").

Field Name	Data Type	Required	Description
category	String	Yes	"Design Pattern" or "Object-Oriented Design Problem".
intent	String	Yes	The purpose and problem that the pattern or design solves. Supports Markdown.
structure	String	Yes	An explanation of the structure, potentially including UML diagrams as Markdown images.
useCases	String	Yes	Real-world scenarios and examples where the design is applicable. Supports Markdown.
cppExample	String	Yes	A complete code example demonstrating the implementation in C++.

Cursor AI Prompts: Schema and Model Creation

1. **Create LeetCode Schema and Model:**
```
In the 'server' directory, create a 'models' folder. Inside,
create a file named 'LeetCodeProblem.js'. In this file, use
Mongoose to define a schema and create a model for
'LeetCodeProblem'. The schema should have the following fields:
title (String, required), slug (String, required, unique),
category (String, required), difficulty (String, required),
problemStatement (String, required), example (String, required),
constraints (String, required), and a nested 'solution' object.
The solution object must contain: cppCode (String, required),
explanation (String, required), timeComplexity (String, required),
and spaceComplexity (String, required). Export the model.
```

2. **Create HLD Schema and Model:**
```
In the 'server/models' directory, create a file named
'HLDTopic.js'. Use Mongoose to define a schema and create a model
for 'HLDTopic'. The schema should include: title (String,
required), slug (String, required, unique), category (String,
required), coreConcepts (String, required), an optional
'architectureDiagram' object with 'url' and 'alt' string fields,
and tradeoffs (String, required). Export the model.
```

3. **Create LLD Schema and Model:**
```
In the 'server/models' directory, create a file named
'LLDTopic.js'. Use Mongoose to define a schema and create a model
for 'LLDTopic'. The schema must contain: title (String, required),
slug (String, required, unique), category (String, required),
intent (String, required), structure (String, required), useCases
(String, required), and cppExample (String, required). Export the
model.
```

Phase 4: API Endpoint Implementation

With the data models defined, the next step is to build the API that allows the frontend to interact with the database. This involves setting up the Express server and defining specific routes for fetching content.

Server and Database Connection

The entry point for the backend will be a server.js file in the server directory's root. This file will be responsible for:

1. Importing Express and other necessary modules.
2. Creating an instance of the Express application.
3. Applying essential middleware: cors() to allow requests from the frontend (which runs on a different port) and express.json() to parse incoming JSON request bodies.
4. Using the dotenv package to securely load the ATLAS_URI from a .env file. This file will be excluded from Git to prevent credentials from being exposed.
5. Establishing a connection to the MongoDB Atlas database using Mongoose.
6. Defining a "home" route and starting the server to listen on a specified port (e.g., 5000).

API Design and Routing

A modular routing structure is essential for maintainability. A routes directory will be created within server. Inside this directory, separate files (leetcode.js, hld.js, lld.js) will define the API endpoints for each content type. These router files will be imported and mounted in server.js under a base path, such as /api.
The design of the API endpoints is directly influenced by the UI requirements. The application needs to display lists of topics in the side panels and the full content of a single selected topic in the center. A naive approach would be to have a single endpoint for each category (e.g., /api/leetcode) that returns the complete data for all problems. However, this is highly inefficient. The payload for 75 LeetCode problems, including full problem statements and code solutions, would be excessively large, leading to slow initial page loads and a poor user experience.
A more performant architecture separates these concerns. For each content type, two endpoints will be created:

1. A "list" endpoint (e.g., GET /api/leetcode) that returns a lightweight array of objects containing only the fields necessary for the panel view: title, slug, category, and difficulty. This minimizes the initial data load.
2. A "detail" endpoint (e.g., GET /api/leetcode/:slug) that is called only when a user clicks a specific topic. This endpoint fetches and returns the complete document for that single

item.

This design ensures that the application is fast and responsive, fetching heavy content only when explicitly requested by the user.

Table 2: REST API Endpoint Definitions

This table serves as the formal specification for the API, defining the contract between the frontend and backend.

Method	Endpoint	Description	Response Body (Success 200)
GET	/api/leetcode	Fetches a lightweight list of all LeetCode problems for the side panel.	[{ title, slug, category, difficulty },...]
GET	/api/leetcode/:slug	Fetches the full details for a single LeetCode problem based on its slug.	{ title, slug, category, problemStatement, example, constraints, solution: {... } }
GET	/api/hld	Fetches a lightweight list of all HLD topics for the side panel.	[{ title, slug, category },...]
GET	/api/hld/:slug	Fetches the full details for a single HLD topic based on its slug.	{ title, slug, category, coreConcepts, architectureDiagram, tradeoffs }
GET	/api/lld	Fetches a lightweight list of all LLD topics for the side panel.	[{ title, slug, category },...]
GET	/api/lld/:slug	Fetches the full details for a single LLD topic based on its slug.	{ title, slug, category, intent, structure, useCases, cppExample }

Cursor AI Prompts: API Implementation

1. **Setup Express Server and DB Connection:**
```
In 'server/server.js', create a basic Express server. Use the
'cors' and 'express.json' middleware. Use 'dotenv' to load
environment variables. Create a '.env' file in the 'server'
directory and add variables for PORT=5050 and
ATLAS_URI="your_mongodb_atlas_connection_string". Use Mongoose to
connect to the database using the ATLAS_URI. Start the server to
listen on the specified port.
```

2. **Create LeetCode API Routes:**
```
In the 'server' directory, create a 'routes' folder. Inside,
create 'leetcode.js'. In this file, create an Express router.
Import the 'LeetCodeProblem' model.
```

```
1. Create a GET route for '/' that fetches all LeetCode problems
from the database. Use.select('title slug category difficulty') to
retrieve only the fields needed for the list view. Send the
results as a JSON response.
2. Create a GET route for '/:slug' that finds a single problem by
its 'slug' field. Fetch the full document and send it as a JSON
response. Handle cases where the problem is not found with a 404
status.
Export the router.
```

3. **Create HLD and LLD API Routes (and mount all routes):**

```
Following the pattern from the 'leetcode.js' file, create 'hld.js'
and 'lld.js' in the 'server/routes' directory.
- In 'hld.js', create two GET endpoints for HLD topics: one to get
a list (title, slug, category) and one to get a single topic by
slug. Use the 'HLDTopic' model.
- In 'lld.js', create two GET endpoints for LLD topics: one to get
a list (title, slug, category) and one to get a single topic by
slug. Use the 'LLDTopic' model.
Finally, in 'server/server.js', import all three routers and mount
them under the '/api' path. For example: app.use('/api/leetcode',
leetcodeRouter);
```

Phase 5: Database Seeding

To populate the application with its initial content without manual data entry, a seeding script is the most efficient method. This involves creating a standalone script that reads from local data files and populates the MongoDB collections.

Seeding Strategy

A file named seed.js will be created in the server directory. This Node.js script will:

1. Connect to the MongoDB Atlas database using the same connection logic as server.js.
2. Import the three Mongoose models (LeetCodeProblem, HLDTopic, LLDTopic).
3. Read data from three corresponding JSON files: data/leetcode.json, data/hld.json, and data/lld.json.
4. For each collection, it will first delete all existing documents to prevent duplication on subsequent runs.
5. It will then use the Model.insertMany() method to bulk-insert the array of documents from the JSON files into the respective collections.
6. Finally, it will close the database connection.

This script is executed once from the command line (node seed.js) to set up the initial state of the database.

Content Preparation

The JSON data files must be meticulously prepared to match their corresponding Mongoose schemas. The content itself will be curated from the provided research materials. For example, the leetcode.json file will contain an array of 75 objects, each representing a problem from the Blind 75 list. The hld.json file will contain objects for topics like "Load Balancing," "Caching," and "Databases". The lld.json file will cover topics like "Singleton Pattern" and "Design a Parking Lot".

Cursor AI Prompts: Database Seeding

1. **Create Seed Data Files:**
   ```
   In the 'server' directory, create a 'data' folder. Inside, create
   three JSON files: 'leetcode.json', 'hld.json', and 'lld.json'.
   - Populate 'leetcode.json' with an array containing one sample
   LeetCode problem object for "Two Sum", matching the
   LeetCodeProblem schema exactly.
   - Populate 'hld.json' with an array containing one sample HLD
   topic object for "Load Balancing", matching the HLDTopic schema.
   - Populate 'lld.json' with an array containing one sample LLD
   topic object for "Singleton Pattern", matching the LLDTopic
   schema.
   ```

2. **Create the Seeding Script:**
   ```
   In the 'server' directory, create a file named 'seed.js'. Write a
   Node.js script that does the following:
   1. Connects to the MongoDB database using Mongoose and the
   ATLAS_URI from the.env file.
   2. Imports the three Mongoose models: LeetCodeProblem, HLDTopic,
   and LLDTopic.
   3. Reads the data from the three JSON files in the 'data'
   directory.
   4. Creates an async function to perform the seeding. Inside this
   function, use a try/catch block.
   5. In the try block, first delete all existing documents from all
   three collections.
   6. Then, use Model.insertMany() to insert the data from the JSON
   files into their respective collections.
   7. Log a success message to the console.
   8. In the catch block, log any errors.
   9. Finally, close the database connection.
   Execute the async function.
   ```

III. Frontend Development: User Interface & Interactivity

The frontend is the user-facing component of the application, responsible for presenting data and handling user interactions. This section details the process of translating the UI mockups into a dynamic and responsive interface using React. The focus is on creating a clean component architecture, managing application state effectively, and fetching data from the backend API to populate the UI.

Phase 6: Core Layout Implementation

The specified UI consists of a complex, multi-panel layout. Modern CSS provides powerful tools for creating such structures without resorting to fragile, outdated techniques. A combination of CSS Grid and Flexbox offers the best approach for this design.

Macro Layout with CSS Grid

CSS Grid is the ideal technology for defining the overall two-dimensional page structure. The main application container, likely in the App.css file, will be set to display: grid. The grid-template-areas property will be used to create a semantic and readable layout definition that directly corresponds to the visual mockup. This approach is declarative and makes future layout modifications straightforward.
An example implementation might look like this:

```
.app-container {
  display: grid;
  height: 100vh;
  grid-template-columns: 250px 1fr 250px;
  grid-template-rows: 1fr auto;
  grid-template-areas:
    "left-panel content right-panel"
    "bottom-panel bottom-panel bottom-panel";
  gap: 1rem;
}

.left-panel { grid-area: left-panel; }
.right-panel { grid-area: right-panel; }
.content-display { grid-area: content; }
.bottom-panel { grid-area: bottom-panel; }
```

Micro Layouts with CSS Flexbox

While CSS Grid excels at the overall page structure, CSS Flexbox is better suited for arranging items along a single axis—either a row or a column. Within each of the three topic panels (left, right, and bottom), Flexbox will be used to style the list of clickable topic links. For instance, display: flex with flex-direction: column will neatly stack the links vertically and allow for easy alignment and spacing.

Phase 7: Component Architecture

A key principle of React development is breaking down the UI into small, reusable, and single-responsibility components. This approach, known as componentization, enhances code readability, simplifies maintenance, and promotes reusability.

Table 3: React Component Hierarchy

This table outlines the primary components of the application, their specific responsibilities, and the flow of data (props) and actions (callbacks) between them. This serves as an architectural blueprint for the frontend.

Component Name	Responsibility	Props Received	State Managed
App	The main application container. Manages the overall layout and application-level state. Fetches data for the selected topic.	-	selectedTopic (object with type and slug), topicData, loading, error
TopicPanel	A reusable component that displays a list of topics for a given category (LeetCode, HLD, or LLD).	title (String), topics (Array), onTopicSelect (Function)	activeTopicSlug (String)
ContentDisplay	A container that conditionally renders the appropriate view based on the selected topic's data. Displays loading and error states.	topicData (Object), loading (Boolean), error (Object)	-
LeetCodeView	Renders the detailed layout for a LeetCode problem, including statement, examples, and solution.	data (Object)	-
HLDView	Renders the detailed layout for an HLD topic, including concepts and diagrams.	data (Object)	-
LLDView	Renders the detailed layout for an LLD topic, including intent, structure, and code examples.	data (Object)	-
CodeBlock	A specialized component for displaying code with syntax highlighting.	code (String), language (String)	-

Phase 8: State Management and Data Fetching

The application's interactivity hinges on how it manages state and communicates with the backend API.

Centralized State Management

For an application of this scale, a simple and effective state management pattern is "lifting state up." The highest-level common ancestor component, App.js, will own the most critical pieces of state: the currently selected topic, the data for that topic, and the loading/error status of the API request.
When a user clicks a topic link inside a TopicPanel component, it will not fetch the data itself. Instead, it will invoke a callback function (onTopicSelect) passed down from App.js as a prop. This function will update the selectedTopic state in the App component with the type (e.g., 'leetcode') and slug (e.g., 'two-sum') of the clicked item.

Data Fetching Lifecycle

The App component will use a useEffect hook to react to changes in the selectedTopic state. When this state changes, the effect will trigger, initiating an API call using Axios to the corresponding detail endpoint (e.g., GET /api/leetcode/two-sum). During this process, the App component will manage the full lifecycle:

1. Set loading to true and error to null.
2. Make the API request.
3. If the request is successful, update the topicData state with the response and set loading to false.
4. If the request fails, update the error state with the error information and set loading to false.

This centralized data fetching logic keeps the child components "dumb" and presentational, simplifying the overall architecture.

Initial Data Load for Panels

Upon initial application load, the App component will also use a useEffect hook (with an empty dependency array) to make three initial API calls to the list endpoints: /api/leetcode, /api/hld, and /api/lld. The resulting arrays of topic lists will be stored in state and passed down as props to the three TopicPanel instances.

Phase 9: Content Rendering

The final step is to display the fetched data in a well-formatted and user-friendly manner, matching the mockups.

Conditional and Specialized Rendering

The ContentDisplay component will act as a router. Based on the category or type of the topicData it receives, it will conditionally render the appropriate specialized view component: LeetCodeView, HLDView, or LLDView. This ensures that each content type can have its own unique layout and structure without cluttering a single, monolithic component.

Rich Content Formatting

The data stored in MongoDB for explanations and problem statements is intended to be

formatted using Markdown. To render this correctly in the browser, a library such as react-markdown will be used. This library parses a Markdown string and converts it into the corresponding HTML elements.
For displaying the C++ code solutions, it is essential to provide syntax highlighting for readability, as shown in the mockup. A library like react-syntax-highlighter is perfectly suited for this task. The CodeBlock component will wrap this library, accepting the code string and language as props, and rendering a beautifully formatted and colored code block.

Cursor AI Prompts: Frontend Development

1. **Create Main Layout with CSS Grid:**
```
In 'client/src/App.css', create a three-panel layout for a
container with the class 'app-container'. Use CSS Grid with
'grid-template-areas' to define four areas: 'left-panel',
'right-panel', 'content', and 'bottom-panel'. The layout should
have three columns (e.g., 250px 1fr 250px) and two rows (e.g., 1fr
auto). Assign the grid areas to their respective positions.
```

2. **Create App Component Structure:**
```
In 'client/src/App.js', create the main component structure. It
should render four divs with the class names corresponding to the
CSS Grid areas: 'left-panel', 'right-panel', 'content-display',
and 'bottom-panel'. Import 'App.css'.
```

3. **Create TopicPanel Component:**
```
In 'client/src/components', create a new file 'TopicPanel.js'.
This component should accept props: 'title', 'topics' (an array of
objects with title and slug), and 'onTopicSelect' (a function). It
should render the title in an h3 tag. Below the title, it should
map over the 'topics' array and render a clickable button or div
for each topic. When a topic is clicked, it should call the
'onTopicSelect' function, passing back the topic's type and slug.
```

4. **Implement State and Initial Data Fetching in App.js:**
```
In 'client/src/App.js', implement the following using React hooks:
1. Create state variables for 'leetcodeTopics', 'hldTopics', and
'lldTopics', initialized as empty arrays.
2. Use a 'useEffect' hook that runs once on component mount to
fetch data from '/api/leetcode', '/api/hld', and '/api/lld' using
Axios.
3. Populate the respective state variables with the fetched data.
4. Render three instances of the 'TopicPanel' component in the
left, right, and bottom panels, passing the appropriate title and
topic lists to each.
```

5. **Implement Selected Topic State and Data Fetching:**
```
In 'client/src/App.js', add the following logic:
1. Create state variables for 'selectedTopic' (initially null),
```

'topicData' (initially null), 'loading' (initially false), and 'error' (initially null).
2. Create a handler function 'handleTopicSelect(type, slug)' that updates the 'selectedTopic' state. Pass this function as the 'onTopicSelect' prop to each 'TopicPanel'.
3. Create a 'useEffect' hook that depends on 'selectedTopic'. When 'selectedTopic' changes, this effect should:
    a. Set loading to true.
    b. Construct the correct API URL based on the topic's type and slug (e.g., `/api/${type}/${slug}`).
    c. Fetch the detailed data using Axios.
    d. On success, set 'topicData' with the result and set loading to false.
    e. On failure, set the 'error' state and set loading to false.

6. **Create ContentDisplay and View Components:**
1. Create a 'ContentDisplay.js' component in 'client/src/components'. It should receive 'topicData', 'loading', and 'error' as props. It should display a loading message if loading is true, an error message if an error exists, and a welcome message if no topic is selected. If 'topicData' exists, it should conditionally render 'LeetCodeView', 'HLDView', or 'LLDView' based on a property like 'topicData.category' or 'topicData.type' (you'll need to add a 'type' field to your API responses for this).
2. Create placeholder components for 'LeetCodeView.js', 'HLDView.js', and 'LLDView.js'. Each should accept a 'data' prop and display the title for now.
3. Integrate the 'ContentDisplay' component into 'App.js' inside the 'content-display' div, passing the relevant state variables as props.

7. **Implement LeetCodeView with Syntax Highlighting:**
Flesh out the 'LeetCodeView.js' component.
1. Install 'react-markdown' and 'react-syntax-highlighter'.
2. The component should receive the full LeetCode problem data object as a prop.
3. Render the problem's title, difficulty, and category.
4. Use the 'ReactMarkdown' component to render the 'problemStatement', 'example', and 'constraints'.
5. Create a 'CodeBlock.js' component that uses 'react-syntax-highlighter' to display the 'solution.cppCode' with C++ language highlighting.
6. Use 'ReactMarkdown' again to render the 'solution.explanation'.
7. Display the time and space complexity.

IV. Appendix: Comprehensive Cursor AI Prompt Library

This appendix consolidates all Cursor AI prompts from the preceding sections into a single, sequential library. This serves as an actionable playbook for generating the core structure and logic of the application from start to finish.

Foundational Setup

1. **Initialize Git Repository:** Initialize a new Git repository in the current directory.
2. **Create .gitignore:** Create a standard.gitignore file for a MERN stack project. It should ignore node_modules directories in both the client and server,.env files, and production build artifacts.
3. **Create Project Structure:** Create a root directory for a MERN project. Inside this root directory, create two subdirectories: 'server' for the backend and 'client' for the frontend.
4. **Initialize Backend:** Navigate into the 'server' directory. Initialize a new Node.js project and install the following dependencies: express, mongoose, cors, and dotenv.
5. **Initialize Frontend with Vite:** Navigate to the project's root directory. Use Vite to create a new React application inside the 'client' directory. After creation, navigate into the 'client' directory and install axios.

Backend Development

1. **Create LeetCode Schema and Model:** In the 'server' directory, create a 'models' folder. Inside, create a file named 'LeetCodeProblem.js'. In this file, use Mongoose to define a schema and create a model for 'LeetCodeProblem'. The schema should have the following fields: title (String, required), slug (String, required, unique), category (String, required), difficulty (String, required), problemStatement (String, required), example (String, required), constraints (String, required), and a nested 'solution' object. The solution object must contain: cppCode (String, required), explanation (String, required), timeComplexity (String, required), and spaceComplexity (String, required). Export the model.
2. **Create HLD Schema and Model:** In the 'server/models' directory, create a file named 'HLDTopic.js'. Use Mongoose to define a schema and create a model for 'HLDTopic'. The schema should include: title (String, required), slug (String, required, unique), category (String, required), coreConcepts (String, required), an optional 'architectureDiagram' object with 'url' and 'alt' string fields, and tradeoffs (String, required). Export the model.
3. **Create LLD Schema and Model:** In the 'server/models' directory, create a file named 'LLDTopic.js'. Use Mongoose to define a schema and create a model for 'LLDTopic'. The schema must contain: title (String, required), slug (String, required, unique), category (String, required), intent (String, required), structure (String, required), useCases (String, required), and cppExample (String, required). Export the model.
4. **Setup Express Server and DB Connection:** In 'server/server.js', create a basic Express server. Use the 'cors' and 'express.json' middleware. Use 'dotenv' to load environment variables. Create a '.env' file in the 'server' directory and add variables for PORT=5050 and ATLAS_URI="your_mongodb_atlas_connection_string". Use Mongoose to connect to

the database using the ATLAS_URI. Start the server to listen on the specified port.

5. **Create LeetCode API Routes:** In the 'server' directory, create a 'routes' folder. Inside, create 'leetcode.js'. In this file, create an Express router. Import the 'LeetCodeProblem' model. 1. Create a GET route for '/' that fetches all LeetCode problems from the database. Use.select('title slug category difficulty') to retrieve only the fields needed for the list view. Send the results as a JSON response. 2. Create a GET route for '/:slug' that finds a single problem by its 'slug' field. Fetch the full document and send it as a JSON response. Handle cases where the problem is not found with a 404 status. Export the router.
6. **Create HLD and LLD API Routes (and mount all routes):** Following the pattern from the 'leetcode.js' file, create 'hld.js' and 'lld.js' in the 'server/routes' directory. In 'hld.js', create two GET endpoints for HLD topics: one to get a list (title, slug, category) and one to get a single topic by slug. Use the 'HLDTopic' model. In 'lld.js', create two GET endpoints for LLD topics: one to get a list (title, slug, category) and one to get a single topic by slug. Use the 'LLDTopic' model. Finally, in 'server/server.js', import all three routers and mount them under the '/api' path. For example: app.use('/api/leetcode', leetcodeRouter);
7. **Create Seed Data Files:** In the 'server' directory, create a 'data' folder. Inside, create three JSON files: 'leetcode.json', 'hld.json', and 'lld.json'. Populate 'leetcode.json' with an array containing one sample LeetCode problem object for "Two Sum", matching the LeetCodeProblem schema exactly. Populate 'hld.json' with an array containing one sample HLD topic object for "Load Balancing", matching the HLDTopic schema. Populate 'lld.json' with an array containing one sample LLD topic object for "Singleton Pattern", matching the LLDTopic schema.
8. **Create the Seeding Script:** In the 'server' directory, create a file named 'seed.js'. Write a Node.js script that does the following: 1. Connects to the MongoDB database using Mongoose and the ATLAS_URI from the.env file. 2. Imports the three Mongoose models: LeetCodeProblem, HLDTopic, and LLDTopic. 3. Reads the data from the three JSON files in the 'data' directory. 4. Creates an async function to perform the seeding. Inside this function, use a try/catch block. 5. In the try block, first delete all existing documents from all three collections. 6. Then, use Model.insertMany() to insert the data from the JSON files into their respective collections. 7. Log a success message to the console. 8. In the catch block, log any errors. 9. Finally, close the database connection. Execute the async function.

Frontend Development

1. **Create Main Layout with CSS Grid:** In 'client/src/App.css', create a three-panel layout for a container with the class 'app-container'. Use CSS Grid with 'grid-template-areas' to define four areas: 'left-panel', 'right-panel', 'content', and 'bottom-panel'. The layout should have three columns (e.g., 250px 1fr 250px) and two rows (e.g., 1fr auto). Assign the grid areas to their respective positions.
2. **Create App Component Structure:** In 'client/src/App.js', create the main component structure. It should render four divs with the class names corresponding to the CSS Grid areas: 'left-panel', 'right-panel', 'content-display', and 'bottom-panel'. Import 'App.css'.
3. **Create TopicPanel Component:** In 'client/src/components', create a new file 'TopicPanel.js'. This component should accept props: 'title', 'topics' (an array of objects with title and slug), and 'onTopicSelect' (a function). It should render the title in an h3 tag. Below the title, it should map over the 'topics' array and render a clickable button or div for

each topic. When a topic is clicked, it should call the 'onTopicSelect' function, passing back the topic's type and slug.

4. **Implement State and Initial Data Fetching in App.js:** In 'client/src/App.js', implement the following using React hooks: 1. Create state variables for 'leetcodeTopics', 'hldTopics', and 'lldTopics', initialized as empty arrays. 2. Use a 'useEffect' hook that runs once on component mount to fetch data from '/api/leetcode', '/api/hld', and '/api/lld' using Axios. 3. Populate the respective state variables with the fetched data. 4. Render three instances of the 'TopicPanel' component in the left, right, and bottom panels, passing the appropriate title and topic lists to each.
5. **Implement Selected Topic State and Data Fetching:** In 'client/src/App.js', add the following logic: 1. Create state variables for 'selectedTopic' (initially null), 'topicData' (initially null), 'loading' (initially false), and 'error' (initially null). 2. Create a handler function 'handleTopicSelect(type, slug)' that updates the 'selectedTopic' state. Pass this function as the 'onTopicSelect' prop to each 'TopicPanel'. 3. Create a 'useEffect' hook that depends on 'selectedTopic'. When 'selectedTopic' changes, this effect should: a. Set loading to true. b. Construct the correct API URL based on the topic's type and slug (e.g., \/api/${type}/${slug}`). c. Fetch the detailed data using Axios. d. On success, set 'topicData' with the result and set loading to false. e. On failure, set the 'error' state and set loading to false.`
6. **Create ContentDisplay and View Components:** 1. Create a 'ContentDisplay.js' component in 'client/src/components'. It should receive 'topicData', 'loading', and 'error' as props. It should display a loading message if loading is true, an error message if an error exists, and a welcome message if no topic is selected. If 'topicData' exists, it should conditionally render 'LeetCodeView', 'HLDView', or 'LLDView' based on a 'type' property you will add to the data. 2. Create placeholder components for 'LeetCodeView.js', 'HLDView.js', and 'LLDView.js'. Each should accept a 'data' prop and display the title for now. 3. Integrate the 'ContentDisplay' component into 'App.js' inside the 'content-display' div, passing the relevant state variables as props.
7. **Implement LeetCodeView with Syntax Highlighting:** Flesh out the 'LeetCodeView.js' component. 1. Install 'react-markdown' and 'react-syntax-highlighter'. 2. The component should receive the full LeetCode problem data object as a prop. 3. Render the problem's title, difficulty, and category. 4. Use the 'ReactMarkdown' component to render the 'problemStatement', 'example', and 'constraints'. 5. Create a 'CodeBlock.js' component that uses 'react-syntax-highlighter' to display the 'solution.cppCode' with C++ language highlighting. 6. Use 'ReactMarkdown' again to render the 'solution.explanation'. 7. Display the time and space complexity.

Works cited

1. How To Use MERN Stack: A Complete Guide - MongoDB, https://www.mongodb.com/resources/languages/mern-stack-tutorial 2. Setting Up Your First MERN Stack Project: A Step-by-Step Tutorial - Medium, https://medium.com/@sindoojagajam2023/setting-up-your-first-mern-stack-project-a-step-by-step-tutorial-0a4f88fa4e98 3. MERN Stack Project SetUp - A Complete Guide - GeeksforGeeks, https://www.geeksforgeeks.org/git/mern-stack-project-setup-a-complete-guide/ 4. How to Build a MERN Stack To-Do App - freeCodeCamp, https://www.freecodecamp.org/news/how-to-build-a-mern-stack-to-do-app/ 5. Mongoose v8.18.0: Schemas, https://mongoosejs.com/docs/guide.html 6. Mongoose Schemas Creating a

Model - GeeksforGeeks,
https://www.geeksforgeeks.org/mongodb/mongoose-schemas-creating-a-model/ 7. Blind 75 LeetCode Questions - Discuss,
https://leetcode.com/discuss/post/460599/blind-75-leetcode-questions/ 8. Two Sum Problem on LeetCode in C++ - Scaler Topics, https://www.scaler.com/topics/two-sum-problem/ 9. Comparing brute force vs. optimized solutions to show progression - Design Gurus,
https://www.designgurus.io/answers/detail/comparing-brute-force-vs-optimized-solutions-to-show-progression 10. High-Level Design(HLD) Interview Questions - System Design - GeeksforGeeks,
https://www.geeksforgeeks.org/system-design/top-high-level-designhld-interview-questions-2024/ 11. Top System Design Interview Questions (2025) - InterviewBit,
https://www.interviewbit.com/system-design-interview-questions/ 12. Mastering Load Balancing for System Design Interviews - DEV ...,
https://dev.to/codewithved/mastering-load-balancing-for-system-design-interviews-4b1i 13. 25 Low-Level Design Interview Questions You Must Know - Final Round AI,
https://www.finalroundai.com/blog/low-level-design-interview-questions 14. Last-Minute Preparation of Low Level Design Interviews for Beginners - Medium,
https://medium.com/@prashant558908/low-level-design-last-minute-interview-preparation-guide-899a202411cd 15. Singleton - Refactoring.Guru,
https://refactoring.guru/design-patterns/singleton 16. How to Build a RESTful API Using Node, Express, and MongoDB - freeCodeCamp,
https://www.freecodecamp.org/news/build-a-restful-api-using-node-express-and-mongodb/ 17. Blind 75 Coding Questions (With Answers) – Ace Your Interviews - Design Gurus,
https://www.designgurus.io/blind75 18. How to Create Dynamic Layouts with CSS Grid Template Areas - PixelFreeStudio Blog,
https://blog.pixelfreestudio.com/how-to-create-dynamic-layouts-with-css-grid-template-areas/ 19. CSS Grid Layout Guide, https://css-tricks.com/snippets/css/complete-guide-grid/ 20. Beyond Pretty Colors: Mastering CSS3 Layouts with Flexbox & Grid for Responsive Design | by Harshit Rai | Jul, 2025 | Medium,
https://medium.com/@raiharshit121/beyond-pretty-colors-mastering-css3-layouts-with-flexbox-grid-for-responsive-design-ad136093ebda 21. CSS Flexbox Layout Guide,
https://css-tricks.com/snippets/css/a-guide-to-flexbox/