Okay, here is a project report draft tailored specifically to the "LeetCode AI Assistant" project we've discussed, following the detailed structure you provided. I've filled in the details based on the components and functionality we implemented (Streamlit UI, two databases, agents, Gemini API integration).

You will need to **fill in the bracketed placeholder text** [...] with your specific details, refine the descriptions, and potentially add more specific examples or challenges you faced.

I've also included example **Mermaid code snippets** for the diagrams in Section 2. You can copy these snippets into Mermaid live editors, Markdown files that support Mermaid, or tools like Obsidian or Notion to visualize them.

## ****Personalized LeetCode DSA Assistant with Gemini API****

### ****Project Report****

### ****Abstract****

This project presents the design and implementation of a Personalized LeetCode Data Structures and Algorithms (DSA) Assistant. The system aims to provide tailored learning support by analyzing a user's LeetCode submission history and source code. It utilizes a combination of web scraping (via LeetCode's GraphQL API), local data storage (SQLite), rule-based local analysis agents (for coding style and topic coverage), and advanced natural language processing via the Google Gemini API. Users interact with the assistant through a Streamlit web interface, asking DSA-related questions, requesting analysis of specific problems (optionally providing code), and receiving context-aware responses that incorporate insights derived from their unique LeetCode activity patterns. The assistant leverages fetched metadata (difficulty, tags, submission status) and analyzed source code (language, basic patterns) to generate detailed, pedagogical answers, including algorithm explanations, test case generation, and code feedback, ultimately aiming to enhance the user's DSA learning journey.

### ****Table of Contents****

* 1. Introduction ................................................................... [Page No.]
  2. 1.1 Objective of the project ........................................ [Page No.]
  3. 1.2 Brief description of the project ........................... [Page No.]
  4. 1.3 Technology used .................................................. [Page No.]
  5. 1.4 H/W Requirement ............................................... [Page No.]
  6. 1.5 Software Requirement .......................................... [Page No.]
  7. 1.6 Organization Profile (if applicable) .................... [Page No.]
  8. Design Description ........................................................... [Page No.]
  9. 2.1 Flow Chart ............................................................. [Page No.]
  10. 2.2 Data Flow Diagrams (DFDs) ................................. [Page No.]
  11. 2.3 Entity Relationship Diagram (E-R Diagram) ........ [Page No.]
  12. Project Description .......................................................... [Page No.]
  13. 3.1 Data Base .............................................................. [Page No.]
  14. 3.2 Table Description .................................................. [Page No.]
  15. 3.3 File/Database Design .......................................... [Page No.]
  16. 3.4 Input/Output Form Design .................................. [Page No.]
  17. Testing & Tools used ..................................................... [Page No.]
  18. Implementation & Maintenance ..................................... [Page No.]
  19. Conclusion and Future Work ........................................ [Page No.]
  20. Outcome .......................................................................... [Page No.]
  21. Bibliography .................................................................. [Page No.]
* [Appendices (if any)] ..................................................... [Page No.]

(Note: Generate page numbers using your word processor)

### ****1. Introduction****

#### ****1.1 Objective of the project****

* **Primary Goal:** To develop an intelligent, personalized assistant that leverages a user's LeetCode submission history and source code to provide tailored help, explanations, and feedback for Data Structures and Algorithms (DSA) learning and problem-solving.
* **Problem Statement:** Generic DSA resources lack personalization. Learners often struggle to understand their weaknesses, identify patterns in mistakes, or get contextual help related to their specific coding style and past attempts. This project aims to bridge that gap by creating an assistant that learns from the user's data.
* **Scope:** The project includes:
  + Fetching user submission history (metadata) from LeetCode via its API.
  + Storing submission metadata and problem details (tags, difficulty) in a local database.
  + Storing user-provided source code (via a separate script) in another local database.
  + Developing local agents to analyze language preferences, basic code patterns (keywords), and topic coverage (tags).
  + Integrating with the Google Gemini API to generate natural language responses based on user queries, agent analysis, problem context, and submission history.
  + Providing an interactive web interface using Streamlit for user interaction.
  + Out of Scope: Real-time submission monitoring, automated code grading, advanced static/dynamic code analysis (beyond keywords), user account management across multiple machines.

#### ****1.2 Brief description of the project****

This project, the "Personalized LeetCode DSA Assistant," is a desktop application built with Python and Streamlit. It acts as an interactive chatbot designed to aid users in their DSA practice on LeetCode. The core functionality involves scraping the user's submission history (status, language, runtime, memory) and associated problem metadata (tags, difficulty) directly from LeetCode using their GraphQL API (requiring a user-provided session cookie). This data is stored locally in an SQLite database (leetcode\_submissions.db).

Separately, the user can populate another SQLite database (leetcode\_solutions.db) with their actual solution code (e.g., using the provided convert\_to\_db.py script).

When the user interacts with the assistant via the Streamlit interface, they can ask general DSA questions or specific questions about LeetCode problems (optionally providing a URL and/or their code). The system's DSAAssistantAgent gathers context by:

1. Querying the databases for the user's history with the specific problem (if a URL is provided).
2. Running local analysis agents (MindsetAgent, DSACategoryAgent) on the stored data to infer preferred languages, topic coverage based on tags, and rudimentary coding patterns based on keywords in the stored code.
3. Constructing a detailed prompt containing the user's query, the gathered context (profile, history, patterns), and specific instructions for response generation.
4. Sending this prompt to the Google Gemini API.

The Gemini API generates a response tailored to the user's query and context, following instructions to provide explanations, compare algorithms, generate relevant test cases, offer code feedback (or full solutions), and incorporate insights from the user's profile. This response is then displayed back to the user in the Streamlit UI.

#### ****1.3 Technology used****

* **Programming Languages:** Python 3.9+
* **Frameworks/Libraries:**
  + Streamlit (for Web UI)
  + Requests (for LeetCode API calls)
  + Google Generative AI SDK (google-generativeai) (for Gemini API)
  + python-dotenv (for environment variables)
  + Collections (Counter), re (for agent analysis)
* **Database:** SQLite 3
* **APIs:** LeetCode GraphQL API, Google Gemini API
* **Development Tools:** VS Code, Git, Standard Python Debugger
* **Other:** JSON (for data parsing), sqlite3 Python module

#### ****1.4 H/W Requirement****

* **Development/Execution Machine (Minimum):**
  + CPU: Intel Core i3 / AMD Ryzen 3 or equivalent
  + RAM: 8 GB
  + Storage: ~100 MB free space (for Python, libraries, and growing databases)
  + Other: Active Internet Connection (for API calls)
* **Development/Execution Machine (Recommended):**
  + CPU: Intel Core i5 / AMD Ryzen 5 or equivalent
  + RAM: 16 GB
  + Storage: 500 MB+ free space

#### ****1.5 Software Requirement****

* **Operating System:** Windows 10/11, macOS 11+, Linux (e.g., Ubuntu 20.04+)
* **Runtime Environments:** Python 3.9 - 3.11 recommended
* **Database Management System (Optional):** DB Browser for SQLite (for inspecting databases)
* **Web Browser (for UI):** Google Chrome, Firefox, Edge (latest versions)
* **Key Dependencies:** As listed in requirements.txt (Streamlit, Requests, google-generativeai, python-dotenv, etc.)
* **API Keys/Credentials:**
  + Google API Key for Gemini (obtained from Google AI Studio/Cloud Console).
  + Active LeetCode Session Cookie (LEETCODE\_SESSION).
  + Stored in a .env file in the project root.

#### ****1.6 Organization Profile (if applicable)****

* [If this project was part of coursework, research, or work for an organization, describe it here. Otherwise, state "This project was developed independently."]

### ****2. Design Description****

(Note: You need to generate the actual diagrams using a Mermaid tool based on these code snippets and embed the resulting images or use Markdown rendering that supports Mermaid.)

#### ****2.1 Flow Chart****

* **Purpose:** This flowchart illustrates the main interaction loop when a user submits a query to the assistant.
* **Mermaid Code:**

Code snippet

graph TD

A[User Enters Query & Optional URL] --> B{Assistant Ready?};

B -- Yes --> C[Gather Context];

B -- No --> D[Display Error/Status to User];

C --> E{Problem URL Provided?};

E -- Yes --> F[Check Problem History (DB)];

E -- No --> G;

F --> G[Run Local Agents (Mindset/Category Analysis on DBs)];

G --> H[Construct Detailed Prompt (Query + Context + Instructions)];

H --> I[Send Prompt to Gemini API];

I --> J[Receive Response from Gemini];

J --> K[Display Response in UI];

K --> A;

D --> A;

subgraph Context Gathering

direction LR

F

G

end

style D fill:#f9f,stroke:#333,stroke-width:2px

* **Description:** The flow starts with user input. The system checks readiness, gathers context (problem history, local agent analysis), builds a prompt, queries the Gemini API, and displays the response. If not ready or if errors occur, appropriate status is shown.

#### ****2.2 Data Flow Diagrams (DFDs)****

* **Purpose:** To show data movement between the system, external entities, and internal data stores.
* **Context Diagram (Level 0) - Mermaid Code:**

Code snippet

graph TD

subgraph "External Entities"

U[User]

L[LeetCode API]

G[Gemini API]

end

subgraph "System"

P((Personalized LeetCode Assistant))

end

U -- Query/URL/Code --> P;

P -- Formatted Response/Status --> U;

P -- Cookie --> L;

L -- Submission/Problem Data --> P;

P -- API Key + Prompt --> G;

G -- LLM Response --> P;

style U fill:#lightgrey,stroke:#333

style L fill:#lightblue,stroke:#333

style G fill:#lightgreen,stroke:#333

* **Description (Context):** The main system interacts with the User (inputs query/URL, receives response), the LeetCode API (sends cookie, receives data), and the Gemini API (sends key/prompt, receives response).
* **Level 1 DFD (Simplified) - Mermaid Code:**

Code snippet

graph TD

subgraph "External Entities"

U[User]

L[LeetCode API]

G[Gemini API]

end

subgraph "Processes"

P1(1. Handle User Interaction / UI);

P2(2. Scrape LeetCode Data);

P3(3. Store/Retrieve Data);

P4(4. Analyze Local Data);

P5(5. Generate AI Response);

end

subgraph "Data Stores"

DS1[D1: Submissions DB];

DS2[D2: Solutions DB];

end

U -- Query/URL/Code/ScrapeCmd --> P1;

P1 -- Formatted Response/Status --> U;

P1 -- Scrape Trigger --> P2;

P1 -- Query/Context Request --> P5;

P1 -- Agent Results --> P5;

P1 -- History Request --> P3;

P3 -- History Data --> P1;

P2 -- Cookie --> L;

L -- Raw Submission/Problem Data --> P2;

P2 -- Parsed Data --> P3;

P3 -- Data for Analysis --> P4;

P4 -- Agent Analysis Results --> P1;

P3 -- Stored Code --> P4;

P3 -- Stored Tags/History --> P4;

P5 -- Context + Query --> P5; # Internal processing

P5 -- Prompt --> G;

G -- LLM Response --> P5;

P5 -- Generated Response --> P1;

P3 <--> DS1;

P3 <--> DS2;

P4 <--> DS1;

P4 <--> DS2;

style DS1 fill:#f9f,stroke:#333

style DS2 fill:#f9f,stroke:#333

* **Description (Level 1):** This breaks down the system into major functions: UI handling, data scraping, data storage/retrieval, local analysis via agents, and AI response generation using Gemini. It shows data flow between these processes and the two data stores.

#### ****2.3 Entity Relationship Diagram (E-R Diagram)****

* **Purpose:** Models the structure of the two SQLite databases.
* **Mermaid Code:**

Code snippet

erDiagram

PROBLEMS {

TEXT title\_slug PK "Unique problem identifier"

TEXT problem\_name

TEXT difficulty

TEXT tags "JSON list of tags"

}

SUBMISSIONS {

INTEGER submission\_id PK "Auto-incrementing ID"

TEXT submission\_uid UK "LeetCode's submission ID"

TEXT title\_slug FK "Links to PROBLEMS"

TEXT submission\_timestamp

TEXT status

TEXT language

TEXT runtime

TEXT memory

}

SOLUTIONS {

TEXT title\_slug PK "Unique problem identifier"

INTEGER problem\_number

TEXT title

TEXT url

TEXT language

TEXT code "Actual source code"

}

PROBLEMS ||--o{ SUBMISSIONS : "has"

PROBLEMS ||--|| SOLUTIONS : "describes" # Assuming title\_slug links them

%% Note: There's an implicit link between PROBLEMS and SOLUTIONS via title\_slug

%% but they are in different DB files in this project design.

%% The ERD shows the logical relationship.

* **Description:** This ERD shows the three main tables across the two databases. PROBLEMS stores metadata. SUBMISSIONS stores individual attempts and links to PROBLEMS via title\_slug (one-to-many). SOLUTIONS stores the source code, also identified by title\_slug (effectively one-to-one with problems for which code is stored), and contains redundant fields like title/URL for convenience as it's in a separate DB.

### ****3. Project Description****

#### ****3.1 Data Base****

* **Names:**
  + leetcode\_submissions.db
  + leetcode\_solutions.db
* **Type:** SQLite 3 (Relational, file-based)
* **Rationale:** SQLite was chosen for its simplicity, ease of setup (no separate server needed), portability (single file databases), and adequacy for a single-user desktop application context. The separation into two databases was a practical decision: leetcode\_submissions.db is populated automatically by the API scraper, while leetcode\_solutions.db requires a separate process (manual or scripted) to acquire the source code, which the LeetCode API doesn't provide directly.

#### ****3.2 Table Description****

* **Table Name:** problems (in leetcode\_submissions.db)
  + **Purpose:** Stores metadata about LeetCode problems fetched via API.
  + **Columns:**
    - title\_slug (TEXT, PRIMARY KEY): Unique identifier slug from LeetCode URL.
    - problem\_name (TEXT): Official title of the problem.
    - difficulty (TEXT): Difficulty level (Easy, Medium, Hard).
    - tags (TEXT): JSON string containing a list of topic tags (e.g., '["array", "hash-table"]').
* **Table Name:** submissions (in leetcode\_submissions.db)
  + **Purpose:** Stores records of individual submission attempts by the user.
  + **Columns:**
    - submission\_id (INTEGER, PRIMARY KEY AUTOINCREMENT): Internal DB sequence ID.
    - submission\_uid (TEXT, UNIQUE): LeetCode's unique ID for the submission.
    - title\_slug (TEXT, FOREIGN KEY -> problems.title\_slug): Links to the problems table.
    - submission\_timestamp (TEXT): Unix timestamp string of submission time.
    - status (TEXT): Submission result (e.g., 'Accepted', 'Wrong Answer', 'Time Limit Exceeded').
    - language (TEXT): Language used (e.g., 'python3', 'cpp', 'java').
    - runtime (TEXT): Execution time reported by LeetCode (e.g., '45 ms').
    - memory (TEXT): Memory usage reported by LeetCode (e.g., '16.3 MB').
* **Table Name:** solutions (in leetcode\_solutions.db)
  + **Purpose:** Stores the actual source code of user submissions, obtained via a separate process (e.g., convert\_to\_db.py script).
  + **Columns:**
    - problem\_number (INTEGER): The LeetCode problem number (e.g., 1, 27).
    - title (TEXT): Problem title (redundant but useful for standalone viewing).
    - url (TEXT): Full LeetCode URL (redundant).
    - title\_slug (TEXT, PRIMARY KEY): Unique slug, used to link logically to problems/submissions.
    - language (TEXT): Language of the stored source code.
    - code (TEXT): The complete source code content.

#### ****3.3 File/Database Design****

* **Schema Rationale:** The design uses two separate SQLite files primarily due to the different data acquisition methods. The scraper (main.py) populates leetcode\_submissions.db with readily available API data. The source code requires manual extraction or a different script (convert\_to\_db.py) and is stored in leetcode\_solutions.db. While logically related (via title\_slug), keeping them separate simplifies the data loading processes.
* **Normalization:** The leetcode\_submissions.db schema is partially normalized. problems and submissions are separate tables linked by title\_slug, reducing redundancy of problem metadata (satisfies aspects of 2NF/3NF). The solutions table in the other database contains some redundant information (title, url) for ease of use when accessing code directly but uses title\_slug as a key for potential joins if both databases were attached.
* **Indexing:** Primary Keys (title\_slug, submission\_id) and Unique Keys (submission\_uid) are automatically indexed by SQLite. A foreign key relationship is defined between submissions.title\_slug and problems.title\_slug, which SQLite can use for query optimization but doesn't enforce by default (though good practice to define). No additional custom indexes were deemed necessary for the current query patterns (primarily lookups by title\_slug).
* **Data Integrity:** UNIQUE constraint on submission\_uid prevents duplicate submission entries from the scraper. PRIMARY KEY constraints enforce uniqueness for problems and solutions based on title\_slug. Data type affinity is handled by SQLite. JSON validity for tags relies on the fetching/storing process.

#### ****3.4 Input/Output Form Design****

* **Main User Interface:** Implemented using Streamlit.
* **Key Components:**
  + **Sidebar:**
    - Purpose: Controls, data management, status display.
    - Inputs: "Scrape LeetCode Submissions (API)" button.
    - Outputs: Database status indicators (existence, counts, load status), Agent status (active, limited, disabled), API key/cookie status messages.
  + **Main Chat Area:**
    - Purpose: Primary user interaction zone.
    - Inputs:
      * Text Input for LeetCode Problem URL (Optional).
      * Text Area for User Question / Code Submission.
      * "Get Answer" Button.
    - Outputs:
      * Markdown display area showing the formatted response from the DSA Assistant (Gemini).
      * Status indicators (e.g., spinner during processing).
  + **Conversation History:**
    - Purpose: Displays past interactions in the current session.
    - Outputs: Displays user query (formatted), assistant response (formatted), optional expander to view the raw context sent to the LLM for that interaction.
* **Design Aids:** No formal wireframes used; layout determined by Streamlit's default components and structure (sidebar, main area, columns, containers, expanders).

### ****4. Testing & Tools used****

* **Testing Strategy:** Due to the interactive and API-dependent nature, testing was primarily **manual and exploratory**. Key focus areas were:
  + Correctness of data scraping and storage.
  + Accuracy of local agent analysis (language counts, tag counts).
  + Effectiveness of prompt engineering in guiding the LLM response.
  + Robustness of the application against common errors (missing keys, invalid inputs, API failures).
  + Usability of the Streamlit interface.
* **Types of Tests Performed:**
  + **Functional Testing:** Verified scraping initiation, database creation/updates, agent execution, chat query submission, response display, history rendering. Tested different query types (general DSA, specific problem, code analysis).
  + **Integration Testing:** Ensured data flow: LeetCode API -> Scraper -> submissions.db -> DSACategoryAgent -> Assistant -> Prompt. Also: solutions.db -> MindsetAgent -> Assistant -> Prompt. Tested interaction with Gemini API.
  + **Usability Testing:** Informal assessment of UI clarity, ease of providing input, readability of responses.
  + **Error Handling Testing:** Tested scenarios: missing .env file, missing API keys/cookie, incorrect cookie, non-existent database files, invalid URLs, Gemini API errors (rate limits, content blocking - observed via logs/UI errors).
* **Test Cases:** Developed informally based on features and potential errors. Examples:
  + TC1: Start fresh -> Run scraper -> Verify submissions.db created and populated -> Verify sidebar status update.
  + TC2: Run convert\_to\_db.py -> Verify solutions.db created -> Start app -> Verify language analysis works.
  + TC3: Ask general DSA question -> Verify sensible response from Gemini.
  + TC4: Provide valid LeetCode URL -> Ask question -> Verify problem context (history, tags) appears in LLM context and response.
  + TC5: Paste code + URL -> Ask "What's the issue?" -> Verify detailed response following the preferred format (diagnosis, test cases, code help).
  + TC6: Input invalid URL -> Verify graceful handling.
  + TC7: Run without API Key -> Verify AI features disabled.
* **Tools Used:**
  + **Manual Testing:** Direct interaction via the Streamlit web interface.
  + **Debugging:** Python's print() statements, VS Code debugger.
  + **Database Inspection:** DB Browser for SQLite.
  + **API Testing (Manual):** Observing network requests/responses for LeetCode and Gemini (less formal).
  + **Version Control:** Git.

### ****5. Implementation & Maintenance****

#### ****5.1 Implementation****

* **Environment Setup:** Requires cloning the project repository, creating a Python virtual environment, installing dependencies via pip install -r requirements.txt (assuming one is created), and creating a .env file in the root directory containing GOOGLE\_API\_KEY and LEETCODE\_SESSION\_COOKIE. The convert\_to\_db.py script needs to be run separately with the source code data file (data.txt or similar).
* **Deployment:** The application is designed primarily for **local execution**. Deployment to cloud platforms (like Streamlit Community Cloud, Google Cloud Run) would require additional configuration for persistent storage (managing the SQLite files or migrating to a cloud database) and secure handling of environment variables. [Specify if any deployment was actually attempted or achieved, e.g., "Tested locally on Windows/Linux."].
* **Challenges Faced:**
  + Obtaining reliable submission data (LeetCode API can be rate-limited or change).
  + Acquiring source code (API doesn't provide it, requiring manual steps/separate scripts).
  + **Prompt Engineering:** Iteratively refining the prompt sent to Gemini to achieve the desired detailed and structured output format was a significant challenge, requiring experimentation.
  + Managing state and dependencies between the two separate databases.
  + Robust error handling for external API calls and file system operations.

#### ****5.2 Maintenance****

* **Updates & Bug Fixing:** Code managed via Git. Updates or bug fixes require manual code changes, testing, and potentially re-running the application locally. No automated update mechanism is built-in.
* **Monitoring:** Limited to observing terminal output (print statements, errors) during local execution. Deployed versions would need platform-specific monitoring.
* **Data Backup:** User is responsible for manually backing up the leetcode\_submissions.db and leetcode\_solutions.db files.
* **Dependency Management:** Dependencies are listed in requirements.txt (or should be). Updates require manual checks and potential code adjustments using pip.
* **User Support:** No formal support channel; issues would typically be handled via the development environment (e.g., GitHub issues if hosted).

### ****6. Conclusion and Future Work****

#### ****6.1 Conclusion****

This project successfully demonstrated the feasibility of creating a personalized DSA assistant by integrating LeetCode user data with a large language model (Google Gemini). A functional prototype was developed using Python and Streamlit, capable of scraping submission history, analyzing stored source code for basic patterns, and generating context-aware responses via the Gemini API based on detailed prompt instructions. The system effectively utilizes the user's LeetCode activity (tags, languages, submission status) and problem context to provide more relevant and potentially more helpful DSA learning support compared to generic resources. The core objective of building a personalized, interactive learning tool was achieved at a proof-of-concept level. Key learnings include the importance of robust data handling, the power and challenge of prompt engineering for controlling LLM output, and the practicalities of working with external APIs and local data storage in a desktop application context.

#### ****6.2 Future Work****

* **Enhanced Code Analysis:** Replace basic keyword counting in MindsetAgent with Abstract Syntax Tree (AST) parsing for more sophisticated analysis of code structure, complexity, and specific algorithm implementations.
* **Error Message Analysis:** Modify the scraper (if possible, though difficult with current API) or data input methods to store actual compiler/runtime errors from LeetCode submissions. Use this data for more precise feedback on why submissions failed.
* **Improved Scraping:** Investigate more robust methods for LeetCode data scraping, potentially handling pagination more reliably or exploring alternative (unofficial) API endpoints if necessary and feasible. Add retries and better rate limit handling.
* **Visualization:** Integrate simple charts (e.g., using Streamlit's native charting or libraries like Plotly) to visualize user progress, such as tag frequency over time or submission status distribution.
* **User Accounts & Cloud Sync:** Implement user authentication and store database files in cloud storage (e.g., Google Drive, Dropbox API, or a cloud DB) to allow usage across multiple devices.
* **Testing Automation:** Develop a suite of automated tests (unit tests for agents/helpers, possibly integration tests using mocks for APIs) to improve reliability and facilitate refactoring.
* **Direct Code Input:** Allow users to paste code directly into the chat interface for analysis without needing the convert\_to\_db.py step, perhaps storing it temporarily for the session.

### ****7. Outcome****

* **Nature of Outcome:** Functional Proof-of-Concept Application.
* **Progress Detail:**
  + The application runs locally, successfully scraping LeetCode data, performing basic analysis, and interacting with the Gemini API to provide contextual responses via a Streamlit interface.
  + The codebase, including main.py, agents.py, and convert\_to\_db.py, demonstrates the core architecture and functionality. [Optionally add: Link to your Git repository if public]
  + The immediate next step is further refinement of the LLM prompts based on more extensive testing and potentially implementing more advanced local analysis techniques (like AST parsing).

### ****8. Bibliography****

* **Core Technologies:**
  + Python Software Foundation. (2025). Python Language Reference, version 3.x. Retrieved from <https://docs.python.org/3/>
  + Streamlit Inc. (2025). Streamlit Documentation. Retrieved from <https://docs.streamlit.io/>
  + SQLite Consortium. (2025). SQLite Documentation. Retrieved from <https://www.sqlite.org/docs.html>
  + Reitz, K. Requests: HTTP for Humans™. Retrieved from <https://requests.readthedocs.io/en/latest/>
  + Google AI. (2025). Google AI Gemini API Documentation. Retrieved from <https://ai.google.dev/docs>
  + The Python Package Index (PyPI). python-dotenv. Retrieved from <https://pypi.org/project/python-dotenv/>
* **Data Source:**
  + LeetCode. (2025). LeetCode Online Judge. Retrieved from <https://leetcode.com/>
* **Conceptual References (Examples - Add specific books/papers you used):**
  + Leetcode.com
  + Google.AiStudio
  + langchain.com
  + loggers.com
  + streamlit.com