**Phase 1: Core Functionality - Scenario & Dataset Generation MVP**

**1. Project Setup & Environment**

* Install Python 3.x
* Create a virtual environment
* Install Flask
* Install google-generativeai
* Install Pandas
* Install Faker
* Initialize a Git repository
* Create a basic Flask app structure (e.g., app.py, templates folder, static folder)

**2. Gemini API Exploration & Integration**

* Review Gemini API documentation thoroughly.
* Obtain a Google Cloud API key.
* Install the google-generativeai library.
* Test basic text generation with simple prompts to understand API behavior.
* Explore different Gemini models (if available) to determine the best fit for scenario generation.

**3. Scenario Generation Logic with Gemini**

* Define prompt templates for different data domains (e.g., e-commerce, healthcare, finance). These templates will guide Gemini to generate realistic scenarios. Example template for e-commerce:
* "Generate a realistic data analysis scenario for an e-commerce company. Include a business problem or question they need to answer using data. Specify the type of data they have available. The scenario should require data cleaning and analysis.
* Business Task: [Insert Business Task Here]
* Data Provided: [Insert Data Description Here]"
* Experiment with prompt variations to control the output's complexity, length, and specific details.
* Implement a Python function (e.g., generate\_scenario()) that:
  + Takes a domain (optional for now) as input.
  + Constructs the prompt using the appropriate template and domain.
  + Calls the Gemini API to generate the scenario text.
  + Retrieves the scenario text from the API response.
* Parse the Gemini output to extract key components:
  + **Business Task:** The specific problem or question the data analyst needs to address.
  + **Data Provided:** A description of the data available to solve the problem.
* Implement robust error handling for potential API issues (e.g., timeouts, invalid requests).

**4. Dataset Generation Logic**

* Design data schemas (column names, data types) based on common data types used in analysis:
  + Numerical (integers, floats)
  + Categorical (strings, booleans)
  + Date/Time
* Implement a Python function (e.g., generate\_dataset()) that:
  + Takes the scenario information (from generate\_scenario()) as input, specifically the "Data Provided" part.
  + Creates a Pandas DataFrame based on the described data.
  + Uses Faker to populate DataFrame columns with realistic data according to the data types.
* Implement logic to introduce common data quality issues:
  + **Missing values:** Randomly introduce NaN values in different columns.
  + **Outliers:** Generate values significantly outside the normal range for numerical columns.
  + **Inconsistent formatting:** Vary date/time formats, string casing, etc.

**5. Backend Integration - Scenario & Data**

* Create Flask routes:
  + /generate\_scenario: Handles scenario generation requests.
  + /download\_data: Handles data download requests.
* Implement a Python function (e.g., create\_scenario\_and\_data()) to orchestrate the process:
  + Call generate\_scenario() to get the scenario.
  + Pass the scenario information to generate\_dataset().
  + generate\_dataset() creates and returns the DataFrame.
* Implement logic within the /download\_data route to:
  + Convert the DataFrame to a CSV file in memory.
  + Serve the CSV file for download using Flask's send\_file function.

**6. Basic Front-End Development**

* Create index.html:
  + Include a title (e.g., "Data Analyst Client Simulator").
  + Write a brief description of the application.
  + Add a button labeled "Generate Scenario".
* Implement a JavaScript function (e.g., getScenario()) that:
  + Is triggered by the "Generate Scenario" button.
  + Makes an AJAX request to the /generate\_scenario route.
* Implement logic to display the generated scenario text on the page:
  + Create a designated area (e.g., a <div>) in index.html.
  + Update the content of this area with the scenario text from the API response.
* Create a download link for the CSV file. This link should point to the /download\_data route and be updated after each scenario generation.
* Implement basic JavaScript error handling to display alerts for:
  + Failed API requests.
  + Errors during scenario or data generation.

**7. Testing & Refinement - Scenario & Data**

* Generate multiple scenarios across different potential domains and evaluate their realism.
* Inspect generated datasets to ensure:
  + Data types are correct.
  + Data quality issues (missing values, outliers, inconsistencies) are present and realistic.
  + The data aligns with the generated scenario description.
* Thoroughly test the "Generate Scenario" button and CSV download link.
* Test error handling for various scenarios (e.g., API errors, invalid data requests).

**8. Documentation & Initial Tracking Implementation**

* Add clear comments to the Python code to explain the purpose of each function and section.
* Create a README.md file:
  + Provide a project overview.
  + Include detailed setup instructions (dependencies, environment setup, running the app).
* Implement basic logging using Python's logging module to record:
  + Successful scenario generation events.
  + Dataset generation events.
  + Any errors encountered.

**Phase 2: Refinements and Preparation for Future Features**

**1. Enhance Scenario Variety & Realism by Domain**

* Research domain-specific terminology and common business tasks in areas like:
  + E-commerce
  + Finance
  + Healthcare
  + Marketing
  + etc.
* Refine Gemini prompt templates to incorporate domain-specific language and tasks.
* Experiment with techniques to generate more complex scenarios:
  + Multi-step tasks
  + Scenarios requiring multiple datasets
  + Scenarios requiring external data sources (mentioning them without actually providing them).
* Review generated scenarios for domain relevance, accuracy, and overall realism.

**2. Improve Dataset Logic & Size Control Foundation**

* Analyze the dataset generation logic (generate\_dataset()) for areas to improve consistency and data quality.
* Research Pandas functionalities for creating DataFrames with a specific number of rows (e.g., using pd.DataFrame.sample or generating data row by row until a size limit is reached).
* Design the structure for storing dataset size preferences (even though this won't be user-configurable yet). This could be a configuration variable or a part of the scenario data.

**3. Plan for Domain Selection UI**

* Research common UI patterns for category selection:
  + Dropdown menus
  + Lists (radio buttons, checkboxes)
* Define the list of data domains that will be offered initially.
* Design the data model for storing and retrieving domain information (e.g., a simple list, a dictionary, or a database table if needed).
* Outline the backend logic for filtering or prioritizing scenarios based on the selected domain.

**4. Front-End UI/UX Improvements (Basic Styling)**

* Choose a basic CSS framework (e.g., Bootstrap, Tailwind CSS - for simple styling) or define a simple color palette, fonts, and spacing rules.
* Apply styling to improve:
  + Readability of text.
  + Visual appeal of the page.
  + Clarity of the scenario and data sections.
* Ensure consistent font sizes, spacing, and alignment throughout the interface.

**5. Code Refactoring and Modularization**

* Identify logical modules within the codebase:
  + Scenario generation module
  + Dataset generation module
  + API interaction module
  + Flask route handlers
* Refactor code into separate functions or classes within these modules to improve organization and maintainability.
* Add more detailed comments to explain complex logic within functions.

**6. Advanced Testing & Error Handling (Phase 1 Features)**

* Implement more specific error handling for Gemini API calls:
  + Handle different HTTP status codes (e.g., 400, 500).
  + Handle API rate limits gracefully.
* Implement more comprehensive logging of errors and warnings, including timestamps and relevant context.
* Write unit tests (using a framework like unittest or pytest) for:
  + Scenario generation functions.
  + Dataset generation functions.
  + API interaction functions.

**Phase 3: (Future) Implementing Domain Selection and Dataset Size Control**

**1. Develop Front-End for Domain Selection**

* Implement an HTML dropdown element (<select>) for domain selection.
* Populate the dropdown with the list of available domains.
* Implement JavaScript to capture the selected domain and pass it to the backend.

**2. Develop Front-End for Dataset Size Slider**

* Implement an HTML range input element (<input type="range">) to act as a slider.
* Define the minimum and maximum values for the slider (e.g., 100 to 10,000 rows).
* Implement JavaScript to:
  + Display the currently selected dataset size.
  + Capture the selected size and pass it to the backend.

**3. Backend Integration - Domain Selection**

* Modify the /generate\_scenario Flask route to accept the selected domain as a parameter.
* Update the generate\_scenario() function (or a related function) to use the selected domain:
  + Select the appropriate prompt template based on the domain.
  + Potentially filter a list of pre-generated scenarios (if you have them) by domain.

**4. Backend Integration - Dataset Size Control**

* Modify the /generate\_scenario or a new route to accept the desired dataset size as a parameter.
* Update the generate\_dataset() function to:
  + Use the size parameter to control the number of rows in the generated DataFrame.
  + Use Pandas functions to efficiently create a DataFrame of the specified size.

**5. Test Domain Selection & Size Control**

* Test generating scenarios for each available domain and verify:
  + Relevance of the scenarios to the selected domain.
  + Correct data types and content in the generated datasets.
* Test the dataset size slider with various values and verify that the generated datasets have the correct number of rows.

**Phase 4: (Future) Implementing the Interactive AI Client**

**1. Explore Gemini for Conversational AI**

* Research Gemini's conversational API endpoints and capabilities.
* Experiment with sending user queries and receiving responses in a conversational manner.
* Explore options for maintaining conversation history with the API (if supported).

**2. Backend Integration - Gemini Interaction**

* Create a new Flask route (e.g., /chat) to handle user queries.
* Implement a Python function that:
  + Receives user input from the front-end.
  + Sends the user's query to the Gemini API (potentially including conversation history).
  + Receives and processes the Gemini API's response.

**3. Front-End - Chat Interface Development**

* Create HTML elements for:
  + A chat input field (e.g., <input type="text"> or <textarea>).
  + A message display area (e.g., a <div> to hold messages).
* Implement JavaScript to:
  + Send user messages from the input field to the /chat route.
  + Display AI responses in the message display area, differentiating between user and AI messages.

**4. Implement Conversation State Management**

* Explore methods for storing conversation history:
  + **In-memory:** Simpler for development but loses history on server restart.
  + **Database:** More robust for production but requires database setup.
* Implement logic to:
  + Store user and AI messages.
  + Retrieve conversation history when interacting with the Gemini API to provide context.

**5. Test Interactive AI Client**

* Test basic question-and-answer interactions with the AI.
* Verify that the AI client understands data analysis-related questions in the context of the generated scenario and dataset.
* Test the handling of conversation history to ensure the AI maintains context.

**Phase 5: (Future) Feedback Mechanism and Portfolio Building Support**

**1. Define Feedback Criteria for Data Analysis**

* Define specific, measurable, achievable, relevant, and time-bound (SMART) criteria for evaluating:
  + **Data cleaning:** Correctness, completeness, handling of missing values, outliers, and inconsistencies.
  + **Analysis techniques:** Appropriateness of methods, correctness of implementation.
  + **Interpretation:** Accuracy of conclusions, insights drawn from the data.
  + **Communication:** Clarity of code, reports, and visualizations.

**2. Develop Feedback Logic (Potentially with Gemini)**

* Explore Gemini's capabilities for:
  + Code analysis (to assess code quality and correctness).
  + Natural language understanding (to evaluate the clarity and accuracy of written reports).
* Implement logic (potentially using Gemini or other tools) to:
  + Analyze user submissions (code, reports) against the defined feedback criteria.
  + Generate feedback messages that are specific, constructive, and actionable.

**3. Backend Integration - Feedback Mechanism**

* Create a Flask route to receive user submissions (e.g., code files, reports).
* Implement logic to trigger the feedback analysis process.
* Store and retrieve feedback messages (potentially associated with user accounts or submissions).

**4. Front-End - Feedback Display**

* Design UI elements to display feedback messages clearly and concisely.
* Consider highlighting or annotating specific parts of the user's submission where feedback is provided.

**5. Implement Portfolio Idea Generation**

* Analyze generated scenarios to extract potential data analysis project ideas.
* Implement logic to suggest relevant project ideas to the user, based on:
  + Scenarios they've worked with.
  + Their stated interests (if you implement user profiles).
  + Popular or trending data analysis topics.

**6. Implement User Account Management (If Needed)**

* Design a database schema for user accounts (if storing user data is required).
* Implement user registration and login functionality.
* Implement features for storing user progress, submissions, and feedback (optional).

**7. Plan for Deployment and Scaling**

* Research different deployment options:
  + **Heroku:** Platform as a Service (PaaS), good for simple deployments.
  + **AWS, Google Cloud:** Infrastructure as a Service (IaaS), more control but more complex.
* Consider potential scaling challenges (increased traffic, larger datasets) and plan for solutions (e.g., database optimization, load balancing).