**DACS Project Master Guide - Phase**

**I. Task 1: Project Setup & Environment**

**Task ID: task1**

**Goal: Establish the foundational development environment for the DACS project by installing all necessary tools, libraries, and files.**

**Estimated Time: 1 Week**

**Description: This task will get you started by setting up all the requirements of the DACS project on your own computer. This includes Python, a code repository, a virtual environment, and a basic structure for the app.**

**Subtasks:**

**1.1 Install Python 3.x (or a specific version you choose):**

* **Action: Download the Python 3.10+ installer from** [**https://www.python.org/downloads/**](https://www.python.org/downloads/)**, making sure to add Python to your system’s PATH.**
* **Testing: Verify the installation by running python --version (or python3 --version) in your command line.**
* **Documentation:** 
  + **Record the specific Python version installed in your "Technology Choices" document in ClickUp.**
  + **Include the Python version and the installation steps in your README.md file.**

**1.2 Create a Virtual Environment for the Project:**

* **Action: Create a virtual environment named venv by using python -m venv venv in your project directory and then activate it using: venv\Scripts\activate (Windows) or source venv/bin/activate (macOS/Linux).**
* **Testing: Verify that the virtual environment is active in your command line, where it should appear as (venv) at the beginning of your prompt.**
* **Documentation: Include the instructions for creating a virtual environment in the README.md file.**

**1.3 Install Required Python Libraries:**

* **Action: In the activated virtual environment, install the required libraries using pip install <package\_name>==<version>. For this project, use pip install flask==2.3.0 google-generativeai==0.3.2 pandas==2.1.4 faker==20.2.2.**
* **Testing: Verify that all packages have been installed by using pip list.**
* **Documentation: List all the packages and their specific versions in your "Technology Choices" document, and include the install command in README.md.**

**1.4 Initialize a Git Repository:**

* **Action: Create a new repository on your chosen platform (GitHub, GitLab, or Bitbucket). In the root of your DACS project folder, use: git init, then git add ., then git commit -m "Initial Commit". Finally, add your remote repository with: git remote add origin <your\_repo\_url>.**
* **Testing: Verify that you can connect to your online repository, and that the git repo has been initialized properly, also test your git connection with git push -u origin main to push your code for the first time.**
* **Documentation: Record the repository URL in the "Resources" section of the PRD. Include the process for setting up the repository in README.md.**

**1.5 Create a Basic Flask Application Structure:**

* **Action:** 
  + **Create an app.py file at the root of your directory.**
  + **Create an empty folder named templates at the root of your directory.**
  + **Create an empty folder named static at the root of your directory.**
  + **In app.py, add the following basic code:**

**Python**

**from flask import Flask**

**app = Flask(\_\_name\_\_)**

**@app.route('/')**

**def hello\_world():**

**return 'Hello, World!'**

**if \_\_name\_\_ == '\_\_main\_\_':**

**app.run(debug=True)**

* **Testing: Run your app by typing python app.py and visit [invalid URL removed] in your web browser and verify that the app is working correctly and displaying the "Hello, World" message.**
* **Documentation: In the README.md document, describe the purpose of app.py, the templates folder, and the static folder.**

**Relationships: None**

**Best Practices:**

* **Use the most current and stable version of Python for development.**
* **Use python -m venv venv in your project directory for the virtual environment.**
* **Install libraries individually using pip and always specify their version (e.g., pip install flask==2.3.0).**
* **Always use clear and concise commit messages.**
* **Create the most basic and understandable file structure.**

**Outputs:**

* **Fully functional dev environment**
* **Initialized git repo**
* **Empty structure of main Flask application**

**II. Task 2: Gemini API Exploration & Integration**

**Task ID: task2**

**Goal: Explore the Gemini API and implement a basic integration with the python codebase for future use.**

**Estimated Time: 1 Week**

**Description: This task will get you familiar with the Gemini API, its requirements, and basic responses.**

**Subtasks:**

**2.1 Review Gemini API Documentation:**

* **Action: Review the Google Gemini API documentation at** [**https://ai.google.dev/**](https://ai.google.dev/)**.**
* **Testing: Ensure that all information you read is up to date and that you understand the different aspects of the API.**
* **Documentation:** 
  + **Create a "Gemini API Documentation" doc in ClickUp.**
  + **Include links to all relevant sections.**
  + **Summarize key information (authentication, endpoints, request parameters, response format, error handling).**

**2.2 Obtain a Google Cloud API Key:**

* **Action: Create a Google Cloud Project if you don't have one already, enable the Gemini API, and create an API key using the Google Cloud Console. Store the API key securely.**
* **Testing: Make a test API call using the key you have generated.**
* **Documentation: Specify what method you are using to store your API key in your documentation ("Gemini API Documentation" or "Secrets").**

**2.3 Install google-generativeai Library:**

* **Action: Install the google-generativeai library (version 0.3.2) using pip install google-generativeai==0.3.2.**
* **Testing: Verify that you can import the library into a python file.**
* **Documentation: Include the specific install command in your README.md, and document the chosen version in Technology Choices.**

**2.4 Test Basic Text Generation with Simple Prompts:**

* **Action: Add a python script to your app.py to test the API using your newly generated API key and a basic prompt.**
* **Testing: Verify that you get a text output from your API call, when running this script using python app.py.**
* **Documentation: Include a code snippet and the API response in your "Research Notes" ClickUp document.**

**2.5 Explore Different Gemini Models:**

* **Action: Test different Gemini models and evaluate their performance.**
* **Testing: Test all available Gemini models, and note any problems or details that are not working for your application.**
* **Documentation: In your "Research Notes," document which models you tested, and explain all pros and cons for each one, and the final choice of model that is best for your use case.**

**Relationships: task1**

**Best Practices:**

* **Focus on endpoints, parameters, and responses.**
* **Use Google Cloud Secret Manager (or OS Environment Variables for local development).**
* **Make sure to test every API call you make.**

**Outputs:**

* **Working understanding of the Gemini API**
* **Functional test of an API call to Gemini**
* **Decided Gemini model.**

**III. Task 3: Scenario Generation Logic with Gemini**

**Task ID: task3**

**Goal: Implement the logic to generate realistic, diverse client scenarios using the Gemini API with specific data domains.**

**Estimated Time: 2 Weeks**

**Description: This task will create a function to call the Gemini API and create your scenarios based on your input data. This will require testing the different prompt types and extracting the appropriate information.**

**Subtasks:**

**3.1 Define Prompt Templates for Different Data Domains:**

* **Action: Create distinct prompt templates, using a clear structure, for every data domain (e-commerce, finance, healthcare, botany, astronomy, sports, and history), including all key aspects (client profile, business task, data available).**
* **Testing: Test that every prompt can generate a response from the API.**
* **Documentation: Include all templates and their descriptions in your "Prompts" ClickUp Doc.**

**3.2 Experiment with Prompt Variations:**

* **Action: Test and explore different ways of formatting the instructions for the Gemini AI, by changing parameters or adding other words.**
* **Testing: Test all new variations and see if they are making the response better or worse.**
* **Documentation: Note your findings in your "Prompts" ClickUp Doc, and include all test information, and code used.**

**3.3 Implement Python Function to Call Gemini API and Retrieve Scenario Text:**

* **Action: Create the generate\_scenario(domain) function that takes the domain and: chooses the correct template, calls the Gemini API, and returns the generated text.**
* **Testing: Test that the chosen template is working with every call to the function, and that you receive an output from the API.**
* **Documentation: Add descriptive comments to your code, specifying what every function is doing, and the code you created for that function.**

**3.4 Parse Gemini Output to Extract Scenario Components:**

* **Action: Create parsing logic to extract the Client Profile, Business Task, and Data Provided using splitting or regular expressions.**
* **Testing: Verify that your extraction logic is able to separate all the data into the different sections, by logging each section individually.**
* **Documentation: Add code comments describing how you have implemented your parsing logic, and what type of structured data is being created.**

**3.5 Handle Potential API Errors:**

* **Action: Add try-catch blocks around your API code to handle errors that may arise, log them, and return a message to the user.**
* **Testing: Introduce errors by changing the API key, and test that they are being caught by your try catch block, and that the details are being properly logged in the app.log file.**
* **Documentation: Describe your error handling process in your Error Handling Strategy ClickUp doc.**

**Relationships: task2**

**Best Practices:**

* **Write specific and clear prompt templates.**
* **Test all prompts with a clear and consistent methodology.**
* **Add try catch blocks to ensure that the API does not break the application.**

**Outputs:**

* **List of different prompt templates**
* **Working generate\_scenario() function.**
* **Function handles and parses API response.**
* **Handles errors effectively.**

**IV. Task 4: Dataset Generation Logic**

**Task ID: task4**

**Goal: Implement a function that generates realistic and diverse datasets based on a data schema, and including a series of data quality issues.**

**Estimated Time: 2 Weeks**

**Description: This task will create functions that can make realistic data, based on the description of the data and the data types. This will include the generation of data with missing values, outliers and inconsistent formatting.**

**Subtasks:**

**4.1 Design Data Schemas:**

* **Action: Define Python dictionaries to represent data schemas for different data types (numerical, categorical, datetime), including their attributes (max, min, range, categories, start date, end date, etc)**
* **Testing: Manually create different data schemas and verify that they have the correct information.**
* **Documentation: Document the purpose and structure of your schemas in your Technology Choices doc in ClickUp.**

**4.2 Implement Python Function to Create Pandas DataFrames:**

* **Action: Create a python function called generate\_dataset(data\_schema, number\_of\_rows) that can receive the schema data and create the dataframe.**
* **Documentation: Use code comments to explain your code, and all steps that it takes.**
* **Testing: Verify that your function can create empty dataframes with the correct schema.**

**4.3 Use Faker to Populate DataFrame Columns with Realistic Data:**

* **Action: Use the Faker library to create different values based on the type of data you need to create, and its parameters.**
* **Documentation: Add comments that specify which Faker functions you are using for the different data types.**
* **Testing: Verify that different data types are being created by your function and that every column is being populated.**

**4.4 Implement Logic to Introduce Missing Values:**

* **Action: Add a flag to your code to generate random NaN values with a specified probability, using None values.**
* **Documentation: Add comments that explain the parameters, the code, and the logic behind adding missing data.**
* **Testing: Run your code, and verify that you are adding missing values randomly using that probability.**

**4.5 Implement Logic to Introduce Outliers:**

* **Action: Add a new method to create outliers by using random numbers from statistical distributions to create new values, and using probability to determine when they should appear.**
* **Documentation: Document what type of distributions you are using, and what methods you are applying, and their respective parameters.**
* **Testing: Verify that these values are not only different, but that they are beyond the scope of the expected values.**

**4.6 Implement Logic to Introduce Inconsistent Formatting:**

* **Action: Create different formats for strings, and datetime values, and apply them randomly.**
* **Documentation: Log all available methods for randomizing format for every data type.**
* **Testing: Verify that your data has both date and strings in different formats.**

**Outputs:**

* **Working data schemas for different datatypes**
* **Working generate\_dataset() function**
* **Introduces data quality issues.**

**V. Task 5: Backend Integration - Scenario & Data**

**Task ID: task5**

**Goal: Create Flask routes for the backend and connect all the python modules and create data flow.**

**Estimated Time: 1 Week**

**Description: This task requires you to create your Flask endpoints, connect your previously created functions, and prepare for data serving.**

**Subtasks:**

**5.1 Create Flask Routes for Scenario Generation and Data Download:**

* **Action: Use the @app.route decorator in your app.py file to create two new routes for generating scenarios, and downloading data:** 
  + **@app.route('/generate\_scenario') will handle all requests for generating a new scenario.**
  + **@app.route('/download\_data') will handle all request for downloading data.**
* **Testing: Test both endpoints by visiting them on your web browser and see that they respond correctly.**
* **Documentation: Add comments that document their purpose and input parameters.**

**5.2 Implement Python Function to Orchestrate Scenario Generation and Dataset Creation:**

* **Action: Create the create\_scenario\_and\_data() function that will call generate\_scenario and generate\_dataset, and pass all information that is needed between them.**
* **Testing: Test this function, by checking its input and output and confirming that they have the correct types, and no errors appear during the process.**
* **Documentation: Add comments to all parts of the function, detailing each step of its execution.**

**5.3 Pass Scenario Information to Dataset Generation Function:**

* **Action: Modify the create\_scenario\_and\_data() to extract the data description from the generated scenario and pass it as a parameter to generate\_dataset().**
* **Testing: Log the text you are passing to the generate\_dataset(), and verify that it matches the data you need to create.**
* **Documentation: Explain the type of data that you are passing to the generate\_dataset(), and document the methods you use for this extraction.**

**5.4 Implement Logic to Generate and Serve CSV File for Download:**

* **Action: Using the /download\_data route, call the create\_scenario\_and\_data function, then use pandas to create the csv, and return a flask.send\_file() using the correct mimetype for csv, and setting the appropriate headers.**
* **Testing: Visit the /download\_data endpoint using your browser, and confirm that you are downloading the csv file as expected, and that you can properly open it.**
* **Documentation: Document all the parameters and variables used to create the csv file, using clear comments.**

**Outputs:**

* **Flask routes to manage scenario generation and serving**
* **Working create\_scenario\_and\_data() function.**
* **CSV download functionality**

**VI. Task 6: Basic Front-End Layout**

**Task ID: task6**

**Goal: Create the basic HTML structure and CSS styling for the front-end.**

**Estimated Time: 1 Week**

**Description: This task will create all basic layouts for your application.**

**Subtasks:**

**6.1 Create index.html File:**

* **Action: Create a file named index.html in your templates folder.**
* **Testing: Make sure that the HTML file has a basic and correct structure that can be displayed on a web browser by opening it directly on the browser.**
* **Documentation: Add code comments for each HTML tag to describe the element and its purpose.**

**6.2 Design Basic Layout with Title, Description, and Button:**

* **Action: Create a file named style.css in your static folder, to style all elements you have created for index.html**
* **Testing: After writing all the code, make sure you refresh the browser, and check that all elements are displayed as you intend.**
* **Documentation: Add comments in your css file, describing the purpose of every rule and style you created.**

**You are absolutely correct! My apologies. Here's the rest of the document, continuing from where I left off:**

**VI. Task 6: Basic Front-End Layout (Continued)**

**Outputs:**

* **Basic HTML structure.**
* **Basic styling.**

**VII. Task 7: Basic Front-End JS Interaction**

**Task ID: task7**

**Goal: Create the functionality for the front end of the DACS application by adding JavaScript to make all elements work properly.**

**Estimated Time: 1 Week**

**Description: This task will create all the JavaScript code necessary to create a working user interface.**

**Subtasks:**

**7.1 Implement JavaScript Function to Trigger Scenario Generation on Button Click:**

* **Action: Add the event listener for the click action on the button, using document.getElementById to get the HTML button, and addEventListener to specify the action that it must perform. Make a GET request to the /generate\_scenario endpoint.**
* **Testing: Verify that your JavaScript code makes a call to the /generate\_scenario endpoint after you press the generate button, use the developer console on your web browser to verify it.**
* **Documentation: Use comments to explain each step and the purpose of all lines in the javascript code.**

**7.2 Implement Logic to Display Scenario Text on the Page:**

* **Action: In your getScenario() javascript function, use the response from the server to update the content of the div element with the id="scenario-output", and display the returned text in that location of the page using the innerHTML property.**
* **Testing: Click the button and verify if the text from the API is being displayed in your web application.**
* **Documentation: Use comments to explain the javascript logic you are using for this section.**

**7.3 Create Download Link for CSV File:**

* **Action: Use the javascript to update the href property of the html a element (with id="download-link"), with the link that you will receive from the API endpoint, and set the name of the file that will be downloaded.**
* **Testing: Check that the download link is being properly updated after each response.**
* **Documentation: Comment on each step, and the purpose of every variable you are using.**

**7.4 Implement Basic JavaScript Alert for Error Handling:**

* **Action: Use try-catch blocks to handle any errors that may arise from API calls, or any other javascript code. Display the error to the user by using alert().**
* **Testing: Purposely introduce errors in your API request, and verify that you are getting a message in your front-end application, also verify that no errors are being displayed in your browser's console.**
* **Documentation: Document how you are managing errors using comments.**

**Relationships: task6, task5**

**Best Practices:**

* **Use a clear and consistent coding style.**
* **Always use descriptive names for your variables and functions.**

**Outputs:**

* **Working script.js**
* **JavaScript code that displays data from the back end**
* **Error alerts**

**VIII. Task 8: Testing & Refinement - Scenario & Data**

**Task ID: task8**

**Goal: Thoroughly test all components and identify areas for refinement.**

**Estimated Time: 2 Weeks**

**Description: This task requires you to test all components of your application for realistic and functional performance. You should manually test every functionality, and identify and fix any issues you may find.**

**Subtasks:**

**8.1 Generate Multiple Scenarios and Review for Realism:**

* **Action: Use the implemented UI to generate multiple scenarios across different data domains and write down your opinions on their realism and usability.**
* **Testing: Use at least 1 scenario for every domain, check that they are realistic, different from each other, and that they are usable for practice.**
* **Documentation: Record all tests and results in the Lessons Learned ClickUp document, also mention the prompt you have used, and if there are any issues, or limitations, that you have found in the created text.**

**8.2 Inspect Datasets for Correctness and Presence of Data Quality Issues:**

* **Action: Download every generated dataset and check their structure, their types, and if they contain the data quality issues (missing values, outliers and formatting errors) you have implemented.**
* **Testing: Inspect all generated datasets using different tools (such as python libraries or spreadsheet programs) and create a log of all data types, number of columns, number of rows and the issues you have found in the data.**
* **Documentation: Log the type of data and if all of your data quality requirements are being followed in your Lessons Learned document.**

**8.3 Verify that the Data Aligns with the Generated Scenario:**

* **Action: Use several examples to see if the generated scenarios and datasets are aligned, both in content and structure. For example if you have generated a dataset about finances, the scenario should be of a financial nature.**
* **Testing: Use at least one scenario of every domain, to check that the dataset and the scenario are properly connected.**
* **Documentation: Document all of your findings in the Lessons Learned document, and explain how you tested and verified the connection between scenario and dataset.**

**8.4 Test the "Generate Scenario" Button and CSV Download Link:**

* **Action: Test that the button is working correctly and every time you click it, it retrieves new information. Check that the download link allows for a download, and it is correct for every different scenario you have generated.**
* **Testing: Test both with all the combinations of domains, by clicking them several times.**
* **Documentation: Make a list of tests that you have performed and document your results in your Lessons Learned document.**

**8.5 Test Error Handling for Different Scenarios:**

* **Action: Induce a series of errors, and use a combination of invalid input, invalid parameters, and broken API keys to simulate a bad use case, and force the application to trigger different errors, and see if they are handled correctly.**
* **Testing: Use different methods to induce errors, and verify that the errors are being properly displayed to the user in the UI, and that they are also being stored in your logs.**
* **Documentation: Create a clear list of all tests that you have performed, what was the outcome, and what did you change to fix them in your code base.**

**Relationships: task7, task5**

**Best Practices:**

* **Always record your thoughts and findings in a Lessons Learned log.**
* **Never move from one test to the other until you have clearly documented and fixed all issues that you may find.**

**Outputs:**

* **Fully tested app.**
* **Refined code and prompt outputs.**
* **Bug fixes.**

**IX. Task 9: Documentation & Initial Tracking Implementation**

**Task ID: task9**

**Goal: Ensure the codebase is well-documented and implement basic logging for monitoring and debugging.**

**Estimated Time: 1 Week**

**Description: This task will get your code ready for other people to read it, and will create the logs necessary for better debugging in the future.**

**Subtasks:**

**9.1 Add Comments to Python Code:**

* **Action: Read all the python files you have created, and add clear and concise comments to every function, class, loop, line of code, and method that you have used.**
* **Testing: Verify that if you (or any other developer) read the code for the first time, they would be able to understand your reasoning behind every section.**
* **Documentation: Your code comments should be descriptive, and should be clear for anybody reading your code.**

**9.2 Create README.md File:**

* **Action: Create a README.md markdown file in the root of the project, to present a summary of your code.**
* **Testing: Verify that you can view the README.md file correctly and that it is displayed properly.**
* **Documentation: The file should contain all of these elements: a project overview, clear setup instructions, guidelines on how to use the application, a license for your code, and a list of all project files and folders, and their purpose.**

**9.3 Implement Basic Logging Using Python's logging Module:**

* **Action: Create a new file for your logging implementation, and add to your app.py code to import and use that implementation. Use different levels (info, error, etc) to log all important aspects of your code (such as the successful generation of a scenario, or an error that happened).**
* **Testing: Check that the logging is working by reviewing your app.log file and if all the information that you wanted to log has been saved there.**
* **Documentation: Document all steps for implementing logging in your "Error Handling Strategy" document. Explain the type and purpose of each level you are using, also document the code that uses logging by adding code comments.**

**Relationships: task8**

**Best Practices:**

* **Use descriptive messages for each comment, explaining the purpose of the code.**
* **Write a complete, clear, and simple to understand README.md.**
* **Use different log levels to indicate what is being logged.**

**Outputs:**

* **Fully commented codebase.**
* **Working README.md.**
* **Basic logging functionality.**