Lemur Astrologer Coding

Goal-Oriented Multi-Turn (MT) Coding

Task Specifications

Table of Contents

❗ *Important Announcement:*

1. ***NEW* - Common gaming-related prompts are no longer allowed ("Build a tictactoe game", "Build a tetris game"). *Here is a list of prompts* that are commonly-seen and will be SBQ.**

2. **Read this doc for more info!Common Prompts**
    a. **There's an additional consideration for the Accuracy Dimension: Input Validation. Read the changes to align your work.**
    b. **Prompts must be labeled accurately by difficulty + sub-category**
    c. **Code must be fully tested - bad code will result in SBQ + removal from project**
    d. **All code in rewritten responses or preferred responses are REQUIRED to have sufficient code comments!**
    e. **For rewrites only: Paragraphs describing processes in multiple sentences should be converted to bullet points (or numbered lists, if the process is ordered).**

Project Overview

The goal of this project is to train AI models to handle a variety of coding-specific tasks. You'll do this by having a multi-turn ("MT") conversation with the model to guide it to fulfilling a specific goal. We call this "Goal-Oriented Multi-Turn." You'll start by having a specific goal in mind and prompting the model to try to get as close to the goal as possible.

Your work will identify strengths and weaknesses in the model, help identify where/when the model gets something wrong, and help improve the model's reasoning skills with each prompt and rewritten response. In a nutshell, the **main** objective is to craft realistic prompts that cause the model to fail.

IMPORTANT!

If you've already worked on this project, please review these **Recent Changes**:

- 10/31/24 - NEW Rating Criteria: Input Validation.
- 10/31/24 - Changes to rewrite guidelines - now required at every turn
- 10/31/24 - Presentation failures cannot be the only mistake/issue
- Stylistic Guidance & Changes when Rewriting Responses
- Execution Instructions

Task Overview

Here is a high-level overview of the task:

Step 1: Goal Setting

- Establish a clear and achievable goal tailored to a specific coding task and for a target programming language

Step 2: Prompt Writing & Tagging

- Craft an effective prompt and properly tag its task category and difficulty level

Step 3: Response Evaluation

- Ensure your prompt results in either one or both responses to your prompt failing, and then rating each model responses for accuracy, efficiency, and adherence to instructions

Step 4: Execution

- Test the code in each of the model responses by uploading a screenshot of your code's output, list any setup and run commands, and show the input and actual result when executed

Step 5: Ranking Responses

- Compare and rank the two responses, determining which one best aligns with the task criteria

Step 6: Rewriting the Preferred Response

- Rewrite the *preferred* response to ensure it fully achieves the goal of the task

Step 7: Continuing the Conversation

- Prompt the model again until reaching the minimum turn requirement to guide the model closer to the goal

---

Task Specifications

Step 1: Goal Setting

In this step, you'll **create a goal that matches the specified task category, target programming language, and difficulty level**.

Goals should

- Be specific, clear, and easy to understand, leading the model to fulfilling the goal
    - Example Goal: "Build a pong game"
- **If you're unfamiliar with the task category or target programming language, it's best to skip the task and select another one - you will set your preferences in the courses.**

After writing your goal, select the goal type and sub-category. There are **2 GOAL TYPES,** with sub-categories as follows:

1. **Error Correction**

- **Fixing Model Errors**
    - Identifies and corrects specific errors in model responses. Rectifies inaccuracies, bugs, or other mistakes in code or text.
- **Handling Ambiguous Requests**
    - Trains the model to interpret and clarify unclear instructions. Encourages reasonable assumptions or requests for clarification to fulfill tasks properly.

2. **Multi-Step**
    - **Building Progressively**
        - Guides model through a series of steps to achieve a complex task. Encourages structured and logical problem-solving through each step.
    - **Deep Dive**
        - Explores a concept in greater depth. Encourages nuanced responses and deeper understanding.
    - **Refining Requests**
        - Refines model's responses with more precise requests. Adjusts and narrows prompts to improve quality and accuracy.

Step 2: Prompt Writing & Tagging

In this step, you will write a prompt that **aligns with the chosen category** and **difficulty level, guiding the model toward achieving the task's goal.** Additionally, you will **label the prompt** to clearly indicate the selected **category** and **difficulty level**. If the prompt you write covers more than one category, the **main request** must be of the **chosen category**.

1. **Prompt Creation:**
    - Write the initial prompt based on the task category and programming language.
    - Ensure each prompt is clear, specific, and challenging to the model.
    - **Design the prompts to reflect the set difficulty level.**
    - Use only English for all prompts.
    - Avoid low-effort prompts, such as single-sentence prompts or incredibly generic prompts without any constraints.

Special Notes:

- Prompts specifically not allowed for this project include "counting" prompts (asking the model to do some kind of counting operation like "generate N of something", "do something in N steps", "give me X amount of lines", "have this on line number x" - the customer does not want these as they are already aware of them, so this will be rejected if you submit a counting related prompt). **This includes referencing line numbers in the prompt, such as "debug the function on line 40".**

- Avoid asking the model to interpret scenarios or formulas from topics such as math, physics, etc - this is a **Coding project** and the model failures/deviations should be failures on coding logic and understanding. **Failures/deviations that rely on this approach will be SBQ'd.** It's alright if a task includes formulas, as long as the model isn't expected to interpret them and penalized for not doing so correctly.
- Prompts should not be related to common games (i.e. **Build a tic-tac-toe game, Build a tetris game, see this sheet for more examples**), these tasks are no longer being accepted and will be rejected.

2. **Prompt Tagging:**
   - Label your prompt with the appropriate task category (make sure to pay close attention for subsequent turns - this is a common error!)

IMPORTANT: Simplified for Educational Purposes - Example of common sub-category error:

Task Category: Code Generation/Synthesis
Turn 1 Prompt: "Make a 2d minigolf game using JS"
Turn 1 Category: Text to Code *(makes sense)*
Turn 2 Prompt: "Additionally add a stopwatch to the top right corner"
Turn 2 Category: Text to Code *(❌ bad, it should be **Text to Code Edits**, because we're asking for edits to existing code)*

   - Next, tag the difficulty level (your prompt should match the difficulty, if you think it doesn't then tag it differently!)

3. **Enhancing Testability:**
   - **Recommend code that runs on Replit (or any convenient IDE of your choice)**: This tool simplifies testing by setting up 80%+ of coding environments.
   - **Add setup instructions**: In subsequent prompts, ask the model to include setup steps for testing the code (e.g., using `pip` for Python or `npm` for Node.js).
   - **Limit dependencies**: Avoid prompts that require external dependencies like local databases or APIs, unless you can provide them and they are easily accessible for a reviewer - otherwise, you may be penalized.
   - **Use a staged approach**: For complex tasks like app development, break up the prompts into steps (e.g., pull requests) for easier testing. You should **not** have a laundry list of requirements in your prompts.

4. **Final Considerations:**
   - Make your prompts **realistic** and diverse.
   - The goal is to simulate real-world tasks that are both complex and testable, pushing the model's capabilities while maintaining clarity.

A Prompt Error we commonly see on this project:

- Not thoroughly testing code generated by the models in response to your prompt.
  - **Solution:** Test each part of the code to ensure it runs correctly and meets your requirements.

- Your prompt is not actually failing the model in either response! This will result in your task being rejected - you must ensure your prompt fails **at least one of the model responses!**
- Your prompt must include all of the requirements you expect and/or desire, if one of the model responses does something that is not to your preference (i.e. it solves something iteratively instead of recursively), you cannot penalize the model for that unless you specifically required in your prompt that the solution follow a specific approach.

Tips for Avoiding this Error:

- **Write Clear and Testable Prompts**
  - Create prompts that result in outputs you can easily test and review.
- **Use Tools Like Replit**
  - Ask the model to generate code that runs directly on platforms like Replit.
  - Replit supports many libraries and can automatically set up testing environments.
- **Minimize External Dependencies**
  - Avoid requiring libraries or databases that are difficult to set up unless you provide clear instructions or alternative solutions.

⚠️You must review the Prompt Examples (<- click here) below to get a good sense of what is good and bad for this project. ⚠️

Task Categories

Instruction Type	**Category Type**	**Description**	**Code required in prompt?**
Generation/Synthesis	• Code completion • Text to code • Text-to-SQL	Generate immediately executable code from natural language text description or existing code snippet, infilling, etc.	No
Editing/Rewriting	• Code summarization/compression • Text to Code edits • Code translation • Code refactoring	Make changes or adjustments to existing code to meet new requirements or conditions, such as altering functionality or updating or enhancing features.	Yes (code must work properly)
Debugging	• Debugging and troubleshooting • Testing • Security Review	Identify and correct errors in existing code, such as debugging, resolving syntax errors, and fixing logical mistakes.	Yes (code must contain a bug)
Documentation	• Codebase documentation • Comment generation • Commit text generation • API documentation • Create example usages of this function • Document this function	Generating or updating documentation related to code to help developers understand the code, its functionality, and its usage	Yes (code must work properly)
Review/Critique	• Code review • Log analysis (text -> text) • Quality assurance	Code review & best practices is the process of reviewing and improving code quality, security, and maintainability by applying best practices, standards, and guidelines, and ensuring compliance with coding standards and regulations	Yes (code must work properly)

Code Ecosystem	• IDEs or development workflows • CLI (command line interface) • Version control	Interacting with coding environments, such as IDEs, version control systems, or build tools, to automate tasks, integrate code, or manage development workflows	No

Difficulty Level

Here are the difficulty levels that you may encounter:
Please note that all prompts must be original and copying anything that already exists online on sources such as Leetcode and HackerRank etc. are strictly prohibited and will result in automatic removal from the project. The goal of having human contributors writing prompts is that we want to challenge the models in ways that the models can't already learn from the internet!

	Medium (Undergrad)	**Hard** (Graduate)	**Challenger** (SME)
Knowledge Required	Limited domain/algorithmics knowledge or implementation context (e.g., architecture, libraries, pre-existing code)	Knowledge of standard algorithms and data structures, common libraries and concepts or additional code context may be required to achieve an optimal solution	Expert domain knowledge or information on the specific application or deployment scenario, including substantial specific API/code context
Prompt Ambiguity	There is little ambiguity in the question (in the case of underspecification, good default behaviors are easy to come up with or not essential) and limited complexity of specifications (in instructions)	Medium ambiguity in the prompt (e.g., needs to come up with reasonable ad-hoc data representation or class structure without explicit guidance), multiple requirements should be satisfied, or multiple bugs should be found	Finding good solutions needs non-trivial design decisions regarding data structures, algorithms or code architecture/design patterns
Solution Complexity	The solution is easy to explain (e.g., code doesn't need comments to be understood) and to test for/debug (limited corner cases)	Involves corner cases that should be dealt with separately; an explanation of the solution requires some abstraction or decomposition of the problem into a few subproblems	Finding a solution requires solving several non-trivial subproblems or finding non-trivial bugs; a problem involves tricky corner cases, and explaining the solution to a non-expert requires adding context
Prompt Example	*I have a folder that contains MIDI covers of songs that I created myself. Here is what the song metadata looks like:* *[* *{ "songData": {* *"title": "",* *"artist": "",* *"album": "",* *"duration": "",* *},* *"midiFilePath": "",* *}* *]* *I want a React app that displays my MIDI music library with a play button next to each song. Clicking the play*	*A requirement dictated by management is that files are NOT allowed to use import statements that are erroneous, i.e. a package is imported but none of the methods are utilized in the file it is included inside of. The reasoning being that this can be confusing, wastes space, and is just sloppy programming. We have a quite large Python codebase and it would be unfeasible for someone to manually comb through all the files and check for this condition. Please help write a script in Python which accepts a root folder as an input and then recursively searches through all files and directories, checking the import statements used within each file and ensuring that they are not erroneous. Any files found should be recorded in a text file for manual review.*	*I work for an oil company in the Midwest, and we are drilling in some newly acquired fields that supposedly contain oil and other gas deposits five layers deep into the Earth. However, it has been brought to our attention recently that there are ore and precious metal deposits where we planned on drilling. This is problematic because our equipment for drilling for oil and gas will be damaged if it comes into contact with the precious metals. Furthermore, the precious metals will be destroyed in the process. We are trying to create a system where we can analyze the depth of where we plan to drill and determine where the metals are placed in the holes amongst the oil and gas so we can plan how we will drill without damaging anything. If we can do this programmatically, we hope to implement it with our equipment so they can drill*

	button should play the MIDI file. Use the midi-player-js library. Create some visualization based on the MIDI sequence while a song is playing.		*automatically by analyzing the soil at each depth. How can we generate an artificial dataset and implement this system in Python, TensorFlow?*

❗ Prompt Examples (Highly recommended)

Please review the examples here in the: Prompt Examples which is in the appendix of these instructions

For more examples, please reference the **prompt example bank.**

Step 3: Evaluating Responses & Continuing the Task

In this step, you will assess the model's responses based on specific criteria and provide follow-up prompts to improve its output.

3a. Producing at least one bad response

Every step should involve **at least one bad model response resulting from the prompt you provide.** If both responses are good, please write a more challenging prompt and try again.

How to quickly identify a good vs bad response: If there are any areas in the rating dimensions where a response has **at least one issue in the P0 or P1 level, see below**, we can count that response as "bad."

- If a response only has issues in the presentation criteria (P2) that is **not** sufficient enough to be rated as "bad"!

3b. Rating the Responses based on each dimension

For each turn, after specifying how the two models did overall, each individual response should be rated on performance according to the following dimensions along with a brief explanation.

Quick snapshot of the dimensions (more detailed table below) and the level of priority (meaning these dimensions are most important vs less important):

Level of priority should be used to **determine which response is better with regards to your rating for the turn.** For example, if R1 has an instruction following issue while R2 has efficiency issues, R2 would be the better response.

- Instruction Following: The response answers all requests in the prompt

    - This checks if the model understood **all** of the requests of the prompt and is addressing each one.
    - Make sure to check the entire implementation. If the model *tries* to address the request but *fails to*, it is an **accuracy** issue.
    - **Priority Level:** P0 - Highest Importance
- **Accuracy**: All claims and code in the response is accurate and fully correct
    - You will need to Google the claims made by the model or execute code to check this
    - **Priority Level:** P0 - Highest Importance
    - **Sub category - Input Validation/Error Handling**: The response covers all meaningful input edge cases and handles them correctly.
        - The response validates all inputs, handling invalid data.
        - **Priority Level**:P0 - Highest Importance
- ~~**Optimality and Efficiency**~~**Optimality and Efficiency**: The response presents the most optimal and efficient solutions
    - The response is using common practices and standards
    - **Priority Level:** P1 - Second Highest Importance
- **Up-to-Date**: The response uses only the most recent APIs, functions, or libraries available.
    - APIs, functions, or libraries used aren't causing compilation or runtime errors due to deprecation.
    - **Priority Level:** P1 - Second Highest Importance
- **Presentation**: The response format follows the Style and Presentation guidelines
    - It should follow the presentation rubric, such as:
        - Enough comments in the code.
        - A professional tone (no pleasantries / fluff).
        - An answer that is concise, without repetitive statements.
        - Explanations that use bullet points.
    - **Priority Level:** P2 - Third Highest Importance

Each dimension rating should include a brief explanation as to why the given rating was chosen (e.g., why "Major Issues" was selected for Accuracy). The explanations/justifications you write for dimension ratings should be **specific** and **clear**. If there are any bugs, or there are issues, specify **what the issues are** and **what the causes of the issues are**.

Avoid generalizations like “No Issue”, “N/A”, “This has an error”, etc. If there are no issues, give a very brief explanation as to why.

Examples of Bad Justifications:

- “There is a syntax error”: ❌ This is only half of the story - you should briefly explain what’s causing the syntax error.
- “The response doesn’t satisfy all of the prompt requirements”: ❌ This is very vague. If you find that the response doesn’t satisfy all of the requirements, you

should be specifying which requirements it doesn't satisfy and why it doesn't satisfy them.

Response Rating Rubric

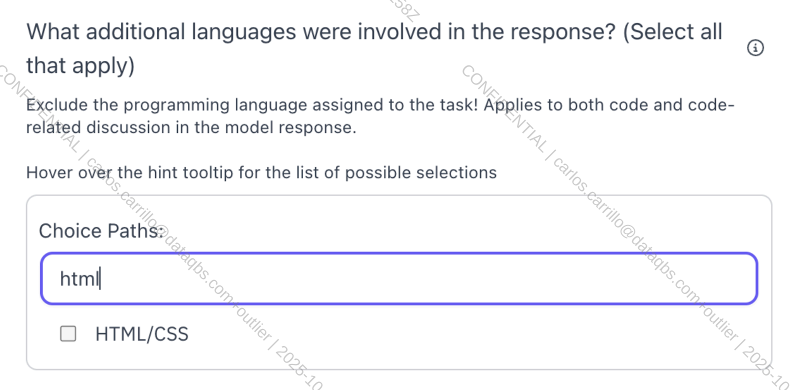
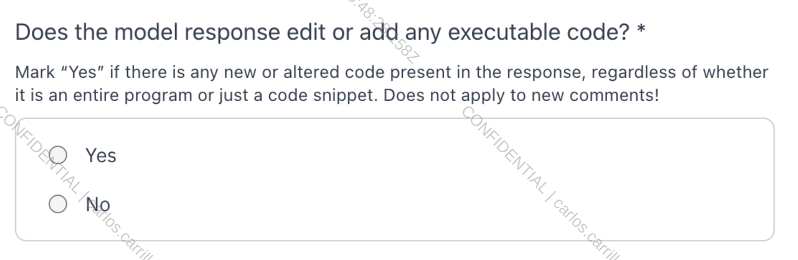
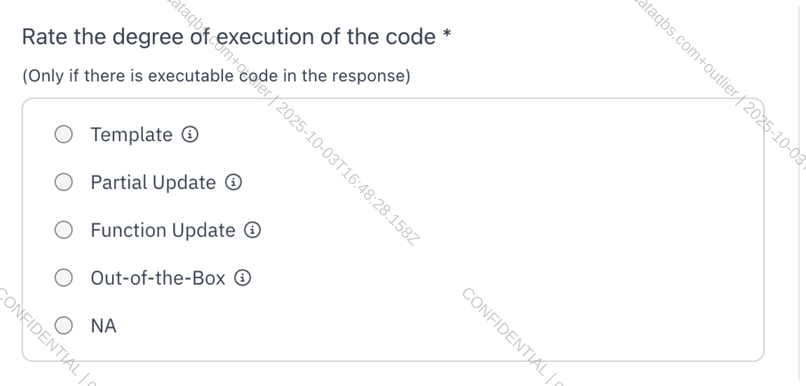
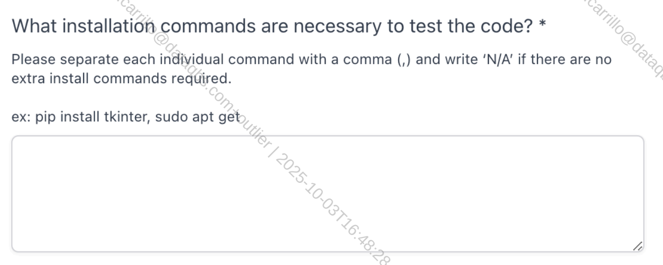
Dimensions	NA (0)	No Issue (1)	Minor Issue (2)	Major Issue (3)
Instruction following	***This dimension cannot be NA.***	The response meets the main request and all constraints, showing a strong understanding of the prompt, even if there are minor implementation errors. It handles any ambiguities well and stays within the specified requirements.	The response fulfills the primary request but does not entirely adhere to all the constraints. The response could have better handled the ambiguity of the prompt. **Common errors:** - Fails some but not ALL constraints	The response fails to fulfill the primary request OR fulfills the primary request but does not adhere to *any* constraints. **Common errors:** - Fails to do the primary request
Accuracy **and** **[NEW] Input Validation / Error Handling**	Can only be NA if the response contains no code or factual claims, and does not rely on prior context.	**The code runs error-free**, produces the correct output, and follows best practices. All text and comments are accurate, and the response is contextually appropriate with any previous errors fixed. – All meaningful edge cases are covered. The response includes thorough validation for expected inputs and error handling for invalid data, ensuring robustness and resilience to common input errors. Example: The function `calculate_discount(price, discount_percentage)` validates all inputs. It checks that `price` and `discount_percentage` are positive numbers, ensures `discount_percentage` does not exceed 100%, and returns a clear error message if values are out of expected ranges or of incorrect types (e.g., strings).	**The code runs** but has **minor** warnings or **low-risk** security issues. The content is mostly accurate, but some statements are unclear or make unproven claims. Previous errors remain but don't affect the current response. – Some edge cases are covered. The response handles most expected inputs but misses certain edge cases, which could lead to potential errors or exceptions under specific conditions. The function `calculate_discount(price, discount_percentage)` includes basic validation, such as checking that `price` and `discount_percentage` are positive numbers. However, it lacks checks for certain edge cases, such as ensuring `discount_percentage` does not exceed 100% or verifying that inputs are numeric.	**The code doesn't run** due to logic errors, produces incorrect output, or has major security flaws. The response includes false claims, lacks context, and previous errors were not fixed, making the issues worse. – No edge cases are covered. The response lacks validation for all inputs, making it vulnerable to errors when faced with unexpected or invalid data inputs. The function `calculate_discount(price, discount_percentage)` performs no validation on its inputs, assuming `price` and `discount_percentage` are always valid and within expected ranges. This could cause runtime errors or incorrect results if given invalid inputs, such as negative numbers, `discount_percentage` over 100%, or non-numeric types, making the function unreliable.
Optimality and Efficiency	Can only be NA if the response contains no code using functions or statements aside from the assignment	The code is well-optimized, handles edge cases, and follows standard best practices. If top performance isn't required, it still performs efficiently without adding unnecessary complexity.	The code performs well but could use minor optimizations. It generally follows best practices but may not scale for large datasets.	The code exhibits severe performance and efficiency issues. The code does not adhere to common practices and standards.

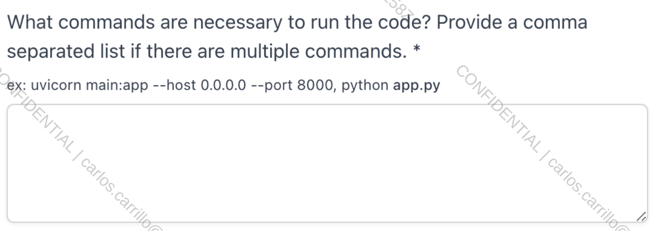
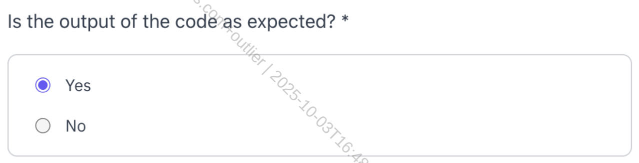
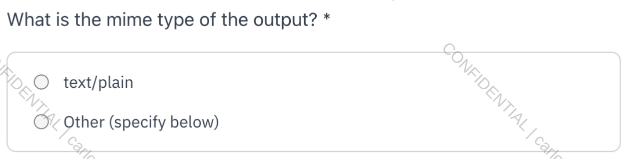
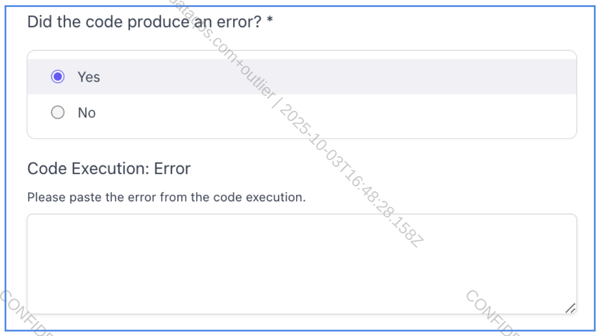
Dimensions	**NA (0)**	**No Issue (1)**	**Minor Issue (2)**	**Major Issue (3)**
Presentation **WHEN REWRITING, YOU MUST FIX ALL PRESENTATION ISSUES**	***This dimension cannot be NA.***	**The code is well-documented, with clear comments and explanations for any modifications. Code included in the prompt that did not originally have comments should have comments if included in the response.** The response is concise, well-organized, and uses readable variable and function names. Complex processes are broken down with bullets, and Markdown is correctly formatted with clear hierarchies. Formatting is neat, with triple backticks for code blocks, and proper use of bold and italics for emphasis. White space and line breaks improve readability, and tables are correctly aligned. Functions are modular and follow standard patterns, such as using `if __name__ == "__main__":` blocks for structure. There are no redundant solutions provided for the same problem.	**The documentation is generally clear but could use more detail**. There are minor language errors that don't affect readability, and formatting could be improved for clarity. Variable and function names are understandable, but some structural changes—like adding bullets or logical sections—would help. Functions are present but may need more modularity, and some explanations are missing, making the code harder to follow in parts. **Common Errors** - Uses backticks inconsistently - Uses camelCase and snake_case inconsistently	**The documentation is missing or inadequate, or lacking code comments entirely, making the code hard to understand**. The response is poorly formatted and lacks structure, with unclear variable and function names. The logic is disorganized, and there are no explanations for key decisions, making it difficult to follow, integrate, or reuse. Programming language tags are also missing.
Up-to-Date	**NA (0)**	**Up-to-Date**	**Out-of-Date**	
	The code does not call on any libraries or functions.	The code uses the most fresh API, libraries, or functions available to solve problems efficiently. The code uses a maintained library or function which is an older version that still works (even if it is less efficient).	The code uses a deprecated API, library or function, causing a runtime or compile-time error.	

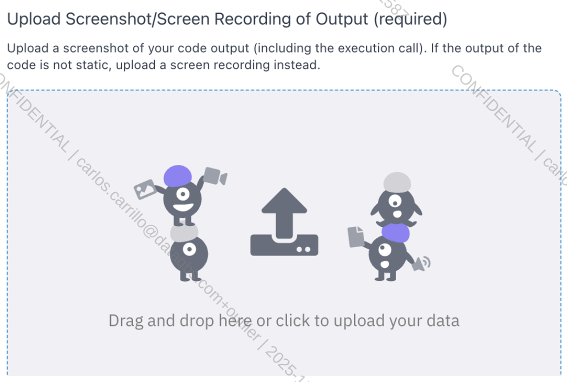
Step 4: Execution (YOU MUST TEST YOUR CODE!)

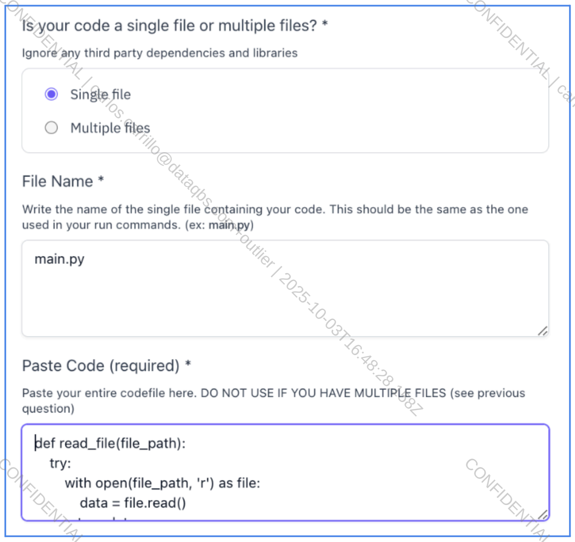
For any response that changes the executable code, you will also be required to execute the code in the response. Note: *This doesn't apply to responses that have only added/modified code comments.*
Below are the steps that you will encounter for running code!

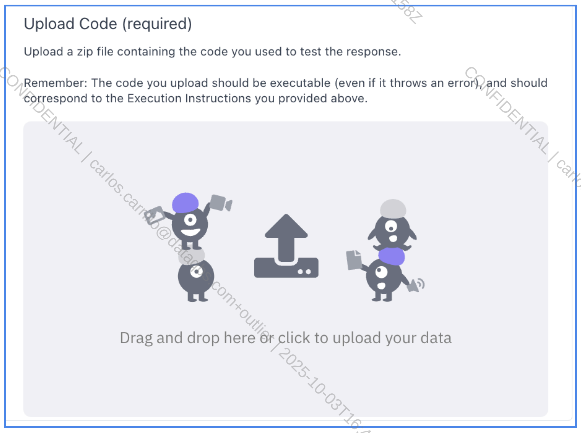
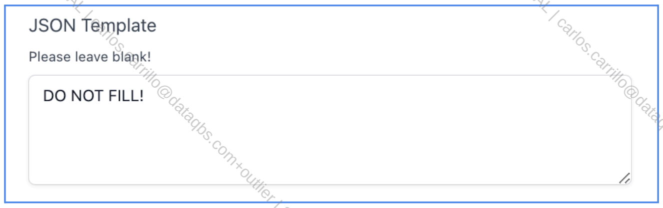
Execution Guidelines

Programming Language: ● List the additional programming languages used in the code. This should include any programming language used in the response that isn't the task's assigned language. ● Start typing in the given space to see a list of possible languages.	What additional languages were involved in the response? (Select all that apply) Exclude the programming language assigned to the task! Applies to both code and code-related discussion in the model response. Hover over the hint tooltip for the list of possible selections Choice Paths: html ☐ HTML/CSS
Executable Code: ● If the model altered or generated any executable code click "yes". In this case you need to respond to the following prompts. ● Otherwise, select "no" which ends this section.	Does the model response edit or add any executable code? * Mark "Yes" if there is any new or altered code present in the response, regardless of whether it is an entire program or just a code snippet. Does not apply to new comments! ○ Yes ○ No
Degree of Execution: ● Template - A skeleton of the code with comments as to how it can be filled in. ● Partial Update - Response updates the first half of the code, or the second half of the code with a comment. ● Function Update - Response updates a single function ● Out-of -the-Box - Response code should run as written (ignoring whether it contains bugs) ● NA	Rate the degree of execution of the code * (Only if there is executable code in the response) ○ Template ○ Partial Update ○ Function Update ○ Out-of-the-Box ○ NA
Installation Commands: ● List any commands needed to set up the environment or install dependencies (e.g., pip install pandas numpy). ● Separate each individual command with a comma and a space (,) and use 'N/A' if there are no extra install commands required.	What installation commands are necessary to test the code? * Please separate each individual command with a comma (,) and write 'N/A' if there are no extra install commands required. ex: pip install tkinter, sudo apt get

Run Commands: • Provide any commands needed to execute the code (e.g., python3 main.py). • Use a comma-separated list if multiple commands are required. • **This should never be blank or N/A**	 What commands are necessary to run the code? Provide a comma separated list if there are multiple commands. * ex: uvicorn main:app --host 0.0.0.0 --port 8000, python **app.py**
Output Expectation: • Indicate whether the output matches expectations (choose yes or no). • Select "no" if the code has bugs, unexpected results, or incorrect behavior.	 Is the output of the code as expected? * [✓ Yes] No
Output MIME Type: • Specify the MIME type of the output (e.g., text/plain, application/json, etc.). • A complete list of MIME types can be found [here]().	 What is the mime type of the output? * text/plain Other (specify below)
Record any errors: • **If code runs without error, indicate 'no'.** • **Otherwise, click 'yes' and record any errors in the space provided.**	 Did the code produce an error? * [✓ Yes] No Code Execution: Error Please paste the error from the code execution.

Upload Screenshot/Screen Recording of Output • If the output is static (e.g., simple print statements), upload a screenshot. • If the output involves dynamic behavior (e.g., games, real-time updates, animations), provide a screen recording showing the code's execution and output. • If there is an error that happens in certain step or after some specific steps, provide a screen recording on how to get the same error instead of a screenshot.	 Upload Screenshot/Screen Recording of Output (required) Upload a screenshot of your code output (including the execution call). If the output of the code is not static, upload a screen recording instead. Drag and drop here or click to upload your data

Upload Files
• If your code is a single file, paste the entire code in the textbox.
• If your code is made up of multiple files, upload the complete set as a **zip** file.
  ○ The code should be executable, even if it throws errors during the run.
	 Is your code a single file or multiple files? *
	Ignore any third party dependencies and libraries
	[✓ Single file] Multiple files
	File Name *
	Write the name of the single file containing your code. This should be the same as the one used in your run commands. (ex: main.py)
	main.py
	Paste Code (required) *
	Paste your entire codefile here. DO NOT USE IF YOU HAVE MULTIPLE FILES (see previous question)
	`def read_file(file_path):`
	`    try:`
	`        with open(file_path, 'r') as file:`
	`            data = file.read()`

	 Upload Code (required) Upload a zip file containing the code you used to test the response. Remember: The code you upload should be executable (even if it throws an error), and should correspond to the Execution Instructions you provided above. Drag and drop here or click to upload your data
Output: ● Include the output or error message generated when the code is run. ● This should show the actual result of the code execution.	
Output Expected: ● Indicate whether the output matches expectations (choose yes or no). ● Select “no” if the code has bugs, unexpected results, or incorrect behavior.	
Output MIME Type: ● Specify the MIME type of the output (e.g., text/plain, application/json, etc.). ● A complete list of MIME types can be found **here.**	
JSON template ● **Leave this field blank!**	 JSON Template Please leave blank! DO NOT FILL!

Step 5: Ranking Responses

In this step you will compare the two model responses and rate which one is better on a scale of 1-8, indicating whether Response 1 or Response 2 performed better.

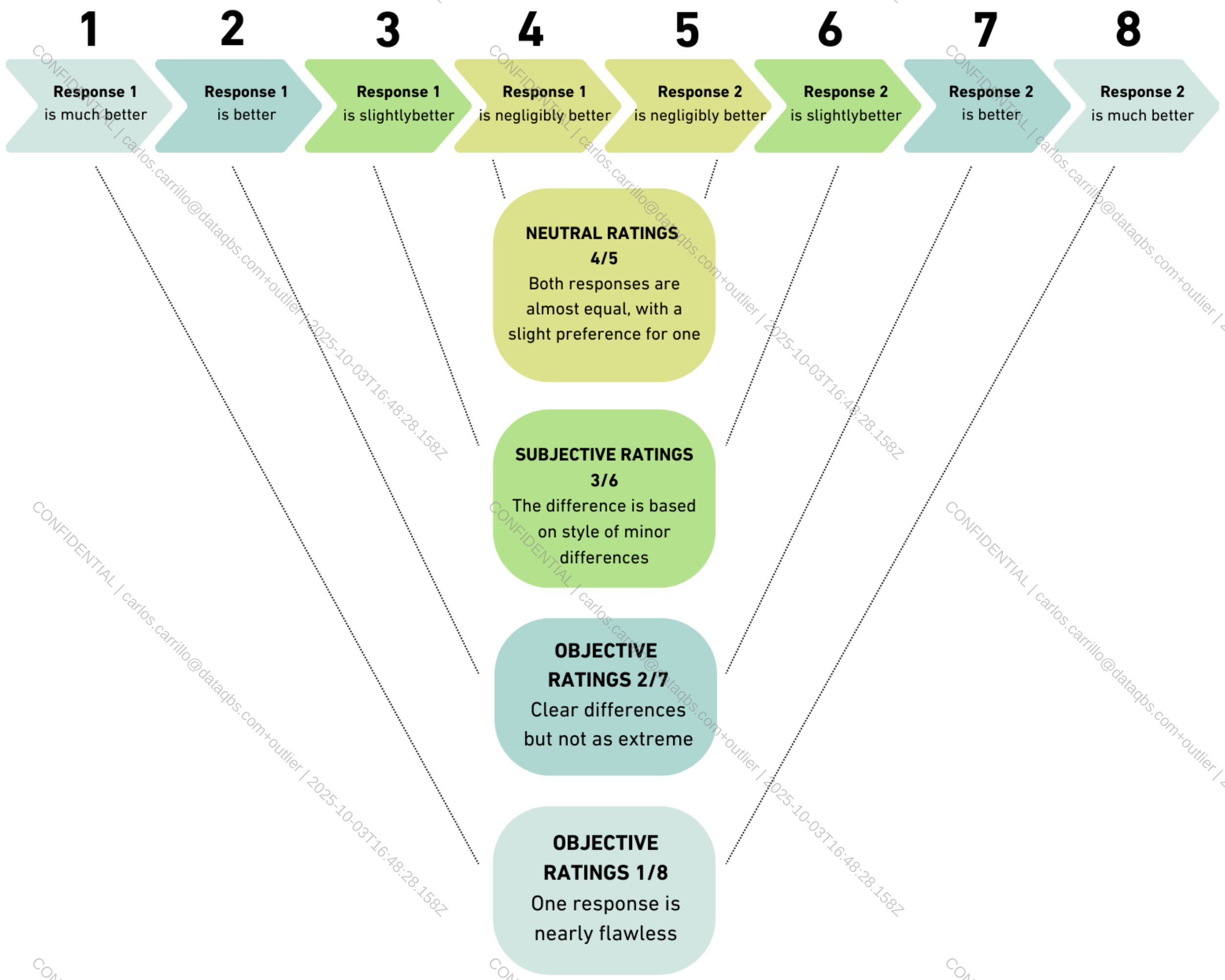
This should match how you rated the models on each dimension (i.e. if Response 2 performed worse on every dimension, then the score should be a 1).

Ranking Justification Guidelines:

- **Base your ranking on the rating dimensions in the rubric, listed by importance.**
- **Provide helpful and factual feedback** without unnecessary comments or conversation.
- **Clear Documentation**: Use docstrings for code explanations and modify code directly in your response, avoiding extra copy-pasting.
- **Readability and Structure**: Ensure code is easy to read, with a focus on structured and modular design (e.g., use functions and `if __name__ == "__main__":` blocks).
- **Use bullet points or numbered lists** for clear and organized explanations.
- Choose the response that is **easier to understand and integrate** with minimal user effort.

Rating Scale

Scores from 1-8 will demonstrate either a strong or neutral preference between the two responses



1
2
3
4
5
6
7
8
Response 1 is much better
Response 1 is better
Response 1 is slightlybetter
Response 1 is negligibly better
Response 2 is negligibly better
Response 2 is slightlybetter
Response 2 is better
Response 2 is much better
NEUTRAL RATINGS 4/5 Both responses are almost equal, with a slight preference for one
SUBJECTIVE RATINGS 3/6 The difference is based on style of minor differences
OBJECTIVE RATINGS 2/7 Clear differences but not as extreme
OBJECTIVE RATINGS 1/8 One response is nearly flawless

Use the priorities mentioned here to determine which response is better based on the severity of the issues.

Step 6: Rewriting Each Preferred Response

In this step, once you have selected the better response from each turn, you will be required to **always** rewrite the **preferred** response to achieve the goal of the prompt and fix any issues that were identified.

The only scenario where a rewrite would **not** be necessary is if one of the model responses is completely **perfect**, meeting all the requirements and specifications of the prompt. In this case, indicate that a *response rewrite is not needed*. **This is very rare and requires a thorough justification, in most cases, the response will require a rewrite.**

Specification List

1. The re-written response should **fully achieve the most recent prompt.**
2. The re-written response should **address any secondary objectives implied by the prompt.**
3. The re-written response should **correct all errors (major or minor)**
4. The re-written response should **be coherent and logically connected to the prior conversation.**
5. The re-written response should **fulfill all the dimensions for "No Issues" in the rating rubrics.**
6. The re-written response should **follow the formatting specifications.**
7. The code of the re-written response should **contain enough comments.**
8. **The code of the re-written response MUST run properly and be optimal and efficient. Failure to properly test code results in removal from the project.**

Formatting/Presentation Requirements

- Key terms should be highlighted in bold, whereas titles, articles, etc. are italicized.
- Remove pleasantries such as "Sure," "Certainly," "I can help with that," etc.
- Make responses more concise, remove all fluff and unnecessary phrases (i.e. “*Welcome to the world of VS Code*!”)
- Tone should be straightforward/professional.
- Code should be well-commented
- Test outputs include a comment with the expected response.
- Explanations should use bullet points.
    - Rewritten response replaces any paragraphs (especially those with at least 3 points) with bullet points instead
- All repetitive phrasing/wording must be removed.

🟡 Here are examples of **formatting/presentation requirements**

Make sure to keep track of the changes that you made as you’ll have to write them out in the next step.

Stylistic Guidance & Changes for Rewrites

Stylistic Change	Description	Bad Example	Better Example

Descriptive Code Comments	Provide comments that explain the code’s logic and intent rather than stating the obvious.	Python
		`# Loop through the list`
		`for item in items:`
		`    process(item)`
			Python
			`# Iterate over the list of items to apply a discount to each`
			`for item in items:`
			`    apply_discount(item)`

Concise Natural Language Explanations	Avoid repetitive phrasing (e.g., “I added,” “We created”). Combine related actions into single sentences to improve clarity.	*Repetitive phrasing:* "I added a guessedWords array to keep track of words that have already been guessed." "I updated the checkSecretWord function to check if an input word has already been guessed." "I added a checkIfAllGuessed function."	- A guessedWords array was added to track guessed words. - The checkSecretWord function displays an alert if a word has been guessed. - The checkIfAllGuessed function ensures all secret words are handled and manages animations."
Use Lists or Bullet Points for Clarity	Use bullet points or numbered lists when describing more than three actions or steps.	*Not itemized:* "First, we import libraries like pandas, numpy, and sklearn. Then, we create a dataset, convert categorical variables, split data, and train a model."	**Itemized:** 1. Import libraries: pandas, numpy, sklearn. 2. Create a dummy dataset with 5 attributes and 1 target variable. 3. 1-hot coded categorical variables.. 4. Split data into training and testing sets. 5. Train a Random Forest model.
Consistent Formatting	Maintain consistent formatting when referring to libraries, methods, or code snippets.	*Inconsistent formatting:* Highlighting one keyword but not the other. “The code uses **matplotlib** and **tkinter**.”	*Consistent formatting:* Highlighting **both** keywords for clarity and consistency. “The code uses **matplotlib** and **tkinter**.”
Use Lists for Multiple Steps	Organize steps or instructions using numbered lists to clearly present each step.	"Check the input, validate user data against the database, and store data in the database."	**Use a list for clarity:** 1. Check the input for errors. 2. Validate user data against the database. 3. Store the data in the database.

Step 7: Continuing the Conversation

Multi-Turn Conversation Guidelines:

For goal-oriented multi-turn tasks, structure the conversation to build logically toward the final goal:

- **Build Progressively**
  - Each prompt should add value and guide the model closer to completing the task.
- **Encourage Depthng**
  - As the conversation advances, focus on enhancing the model’s understanding and pushing for more detailed responses.
- **Evaluate and Iterate**
  - Assess each response, rewrite as needed, and ensure every turn contributes meaningfully to the final solution.

If the required number of turns has been met, you can end the task, even if the goal hasn't been fully achieved. This approach maintains focus and prevents unnecessary iterations.

PRO TIPS!

- For Subsequent Turns, you may choose any category or difficulty but they must be aligned with your prompt
- **THE OVERALL IDEA of this project is to test the model's ability to recall context from prior turns**
    - Subsequent Prompts **should not be disconnected from previous prompts**
    - It's easier to go back and modify the goal, if the model is doing well in recalling and building on previous context
    - You don't need to "fully complete" a goal, as it is simply an alignment tool
        - **EXAMPLE:** If the goal was to build a game, having parts of that game built is fine

Appendix

Grading Rubrics

Prompt Evaluation Rubric

This is the rubric we use to measure the quality of your prompts.

Criteria	**1-2 (Fail)**	**3 (Okay)**	**4-5 (Good/ Perfect)**	**Additional Notes**
Prompt Instruction Type	• The initial prompt does not follow the specified instruction type. • Subsequent prompts are not labeled correctly.	N/A	• The prompt follows the specified instruction type. • Subsequent prompts are labeled correctly.	• Instruction types include: • Generation/Synthesis • Editing/Rewriting • Debugging • Documentation • Review/Critique • Code Coessence
Prompt Difficulty	• The difficulty does not match the specified level.	N/A	• The prompt reflects the specified difficulty level.	• Difficulty levels include: • Medium (Undergrad) • Hard (Graduate) • Challenger (SME)
Prompt Continuity	• The prompt does not logically follow the conversation. • It fails to achieve the conversation goals.	• The prompt may have slight shifts but still furthers the conversation.	• The prompt logically continues the conversation. • Topic shifts are allowed if they help achieve the conversation	• Applies if not the first turn.

			goals.	
Clarity & Specificity	• The prompt is vague and unclear. • Critical details are missing.	• The prompt is mostly clear but may be interpreted in multiple ways.	• The prompt is clear and specific. • No assumptions are needed to answer.	• The prompt should be in English.
Feasibility	• The request is impractical or impossible for an AI in a single response. • Conflicting instructions are given.	• The request is feasible but requires effort or compromises.	• The request is fully actionable and realistic. • No conflicting instructions.	• Requests should not contain conflicting or impractical elements. Example: Asking for a complex algorithm in one step.

Response Grading Rubric

This is the rubric we use to grade your response ratings.

Field	1-2 (Fail)	3 (Okay)	4-5 (Good/ Perfect)
Model Responses Rating	• **Major Rating Disagreement**:(e.g., marked "Good" vs. "Bad") and poor justification.	N/A	Rating aligns with expectations.
Prompt Adherence and Instruction following	**Major Rating Disagreement**: The prompt adherence or instruction following is incorrect (e.g., marked "N/A" instead of "Major Issues"). **Minor Rating Disagreement (2+)**: Minor disagreements occur on 2 or more occasions.	**Minor Rating Disagreement (1)**: Minor variance on 1 occasion, and justification does not fully support this variance.	Rating is correct, or "N/A" is appropriately used when the response cannot be assessed.
Correctness and Accuracy **[input validation]**	**Major Rating Disagreement**: Significant issues with correctness (e.g., marked "Major Issues") or minor disagreements on 2+ occasions. No edge cases are covered. The response lacks validation for all inputs, making it vulnerable to errors when faced with unexpected or invalid data inputs.	**Minor Rating Disagreement (1)**: Slight variance on 1 occasion, justification lacks support. Some edge cases are covered. The response handles most expected inputs but misses certain edge cases, which could lead to potential errors or exceptions under specific conditions.	Rating matches expectations, or "N/A" is used when applicable. All meaningful edge cases are covered. The response includes thorough validation for expected inputs and error handling for invalid data, ensuring robustness and resilience to common input errors.
Performance, Optimality, and Efficiency	**Egregious Rating Disagreement**: Issues with performance were missed (e.g., not marking "Major Issues"), with flawed justification.	**Major Issues**: Minor performance issues were missed, but the justification somewhat supports the rating.	No major issues missed in performance.
Readability, Documentation, and Presentation	**Egregious Rating Disagreement**: Significant documentation or readability issues were not marked.	**Major Issues**: Some readability or presentation issues were missed, but the justification is somewhat valid.	No major readability or documentation issues were missed.

Up-to-Date	**Egregious Rating Disagreement**: Incorrectly marked as "N/A" or "Up-to-Date" when it should have been "Out-of-Date."	N/A	Rating is accurate or correctly identifies outdated information.
MT Conversation Quality	● Task with multiple turns does not consider holistic, conversation-level factors. ● Conversation is incoherent or turns don't follow logically. ● Task doesn't require multi-turn structure (MT) or lacks context from prior turns.	● Task is somewhat long-winded but follows logically, considering conversational factors. ● Conversation accomplishes its goal but could be completed in one turn.	● Task fulfills the main goal of the conversation. ● Turns follow logically and handle topic shifts effectively. ● Each turn builds on the prior, maintaining consistency and context. ● New turns incorporate older turns for elaboration and coherence. ● Multi-turn context is crucial for generating appropriate responses.

Response Rewrite Rubric (if applicable)

This is the rubric we use to measure the quality of your rewritten responses

Field	1-2 (Fail)	3 (Okay)	4-5 (Good/ Perfect)	Additional Notes
Accuracy	• **Major Factual Errors**: Response has 1+ major factual errors or misleading points. • **Minor Factual Errors**: Response has 2+ minor factual errors.	• Contains only 1 minor factual error or misleading statement.	• No factual errors or misleading statements.	• A major error involves incorrect/misleading data central to the request. • A minor error is near the subject matter but doesn't affect the main point.
Instruction Following / Response Fulfillment	• **Main Goal Miss**: Does not achieve the main goal or make progress in the conversation. • **Explicit Instruction Miss**: Misses 1+ key instructions. • **Not Fulfilled**: Fails to answer the question.	• All explicit instructions are followed. • **Secondary Objectives Miss**: Some secondary objectives not addressed. • **Subjective Instruction Miss**: Subjectively misses some parts.	• Fully achieves the main goal or makes clear progress. • Follows all instructions and fully answers the question.	• Rule of thumb: for word count, ±10% is acceptable. • Example: If a question asks for a historical event year, even if implied, the year should be provided.
Unnecessary Greetings / Pleasantries	• Contains greetings/pleasantries such as "Sure, I'd love to help."	N/A	• No unnecessary greetings or pleasantries.	• Only flag unnecessary phrases at the beginning or end of the response. • Phrases like "Here is..." are not considered pleasantries.
Depth / Nuance	• **Little to No Detail**: Response is superficial and lacks meaningful depth	• Has enough detail but may need more depth or nuance.	• Balanced, insightful, and focused without going overboard.	• Too much detail can lead to confusion. Be clear about what points are most important.

	or insight. • **Excessive Detail**: Overly complex and obscures key points.	• **Too Much Detail**: Slightly too much, but doesn't obscure key points.		
[Rewrite/SxS] Clearly Worse Than Model Response	• **Worse Than Original**: Clearly performs worse than the original across rubric categories. • **Worse Than Side by Side Model**: Performs worse overall.	• Performs similarly to the original or side-by-side comparison.	• Performs better overall than the original or side-by-side comparison.	• Don't penalize for minimal/no changes if the original response was acceptable. • Compare against state-of-the-art models.

Prompt Examples

Prompt Category	**Good Examples** (specific)	**Bad Examples** (vague)
Code generation	I am a software engineer analyzing our CDN service performance. I want to write a Bash script to extract the related data from the cache statistics report, count the hit ratio, and filter out the cache items' ID with a hit ratio of less than 0.85. 1. The cache statistics report is a CSV file. Its header row contains the Item ID, Name, Viewer Location, Time, Request Count, Hit Count, Miss Count, and Error Count. And the Item ID is unique. 2. The input to the script is the name of the report. 3. Before calculating, first check whether the count is valid. If any Request Count or Hit Count is missing or Hit Count is greater than Request Count, the script will log the missing or wrong data row and skip it. The hit ratio formula is Hit Count divided by Request Count. The hit ratio should be rounded to two decimal places. 4. The output file is a text file containing unique item IDs and their hit ratio, sorted by ratio in descending order.	**[Asking for a Sudoku solver is too generic of a prompt, and it lacks complexity.]** Hello, I want you to create an automatic sudoku solver in Bash that will take a .txt file that includes 81 numbers from the top row to the bottom with empty cells replaced by zeros in the first line.

Code debugging	As a friend group, we really love to play DnD and other roleplaying games. For the upcoming weekend, I wanted to surprise my friends by developing a multiplayer text-based adventure game with C++. In this game players can explore a fantasy world, interact with characters, and solve puzzles. The game uses a complex system of pointers and dynamic memory allocation to manage player actions, game state, and world events. However, there are several bugs in the code that leads me to crashes, memory leaks, and incorrect game states. I cannot test the game because it outputs nothing. Could you help me to identify the bugs in the code and fix them?
	
	C/C++
	`#include <iostream>`
	`#include <string>`
	`#include <map>`
	
	`using namespace std;`
	
	`class GameEvent {`
	`public:`
	`    string description;`
	`    bool completed;`
	
	`    GameEvent(string desc) : description(desc), completed(false) {}`
		[This is bad because it is too vague for a debugging prompt, being "concerned" over code is not enough to warrant a debugging prompt, a debugging prompt points to a very specific issue like an error or crash; additionally the prompt veers off into being a code editing prompt in the second part which is bad]
		
		Hello, I'm currently working on a JavaScript project which is aimed to actively monitor system performance statistics, as well as network activity and performance. It will generate logs every hour and properly handle data so that memory doesn't become an issue. Currently though, I'm suspicious of the log file handling capabilities of the code, and the asynchronous os.cpuUsage has me concerned. Please look over my code and confirm or deny if these concerns are warranted, make any changes you feel would improve the code.
		
		Also, I intend to include an active alert system for critical performance issues, as well as a text-based report which will compare logs and display a comparison for the user. Please implement these.
		
		Also keep in mind this is a multi-person project, please do not include OS specific libraries and functions.

```
};

class Character {
public:
    string name;
    int health;

    Character(string charName) : name(charName), health(100) {}

    void takeDamage(int amount) {
        health -= amount;
        if (health < 0) health = 0;
    }
};

class Player : public Character {
public:
    vector<GameEvent*> events;

    Player(string name) : Character(name) {}

    void addEvent(GameEvent* event) {
        events.push_back(event);
    }

    void completeEvent(string eventDescription) {
        for (auto& event : events) {
            if (event->description == eventDescription && !event->completed) {
                event->completed = true;
                cout << name << " completed: " << event->description << endl;
                return;
            }
        }
        cout << "Event not found or already completed." << endl;
    }
};

class Game {
private:
    map<string, Character*> characters;
    Player* currentPlayer;

public:
    Game() : currentPlayer(nullptr) {}

    ~Game() {
        for (auto& pair : characters) {
            delete pair.second;
        }
    }
```

JavaScript

```
// Import the os-utils library
const os = require('os-utils');
const fs = require('fs');

// Function to monitor network performance
function monitorNetworkPerformance() {
    let networkData = {
        downloadSpeed: 0,
        uploadSpeed: 0,
        latency: 0,
        packetLoss: 0
    };

    // Use the navigator.connection API to get network
performance data
    if (navigator.connection) {
        networkData.downloadSpeed =
navigator.connection.downlink; // in Mbps
        networkData.uploadSpeed =
navigator.connection.uplink; // in Mbps
        networkData.latency = navigator.connection.rtt;
// in ms
        networkData.packetLoss =
navigator.connection.effectiveType; // effective network
type
    } else {
        console.error("Network Information API not
supported by this browser.");
    }

    return networkData;
}

// Function to monitor system performance
function monitorSystemPerformance() {
    let systemData = {
        cpuUsage: 0,
        memoryUsage: 0
    };

    // Use os-utils to get system performance data
    os.cpuUsage(function(v) {
        systemData.cpuUsage = v * 100; // in %
    });

    systemData.memoryUsage = (1 - os.freememPercentage())
* 100; // in %
```

```cpp
    void addCharacter(string name) {
        characters[name] = new Character(name);
    }

    void setCurrentPlayer(string name) {
        if (characters.find(name) != characters.end()) {
            currentPlayer = static_cast<Player*>(characters[name]);
        } else {
            cout << "Character not found." << endl;
        }
    }

    void displayStatus() {
        for (auto& pair : characters) {
            cout << "Character: " << pair.first << ", Health: " << pair.second->health <<
endl;
        }
    }

    void simulateEvent(string description) {
        if (currentPlayer == nullptr) {
            cout << "No player set." << endl;
            return;
        }

        GameEvent* newEvent = new GameEvent(description);
        currentPlayer->addEvent(newEvent);
        cout << "New event added: " << description << endl;
    }

    void resolveConflict(int damage) {
        if (currentPlayer != nullptr) {
            currentPlayer->takeDamage(damage);
            cout << currentPlayer->name << " took " << damage << " damage!" << endl;
        }
    }
};

int main() {
    Game myGame;

    myGame.addCharacter("Alice");
    myGame.addCharacter("Bob");

    myGame.setCurrentPlayer("Alice");
    myGame.simulateEvent("Find the lost treasure");
    myGame.resolveConflict(20);

    myGame.setCurrentPlayer("Bob");
```

```javascript
    return systemData;
}

// Function to log data
function logData() {
    const networkData = monitorNetworkPerformance();
    const systemData = monitorSystemPerformance();

    const logEntry = {
        timestamp: new Date().toISOString(),
        networkData: networkData,
        systemData: systemData
    };

    fs.appendFile('performance_logs.json',
JSON.stringify(logEntry) + '\n', (err) => {
        if (err) {
            console.error('Error writing to log file:',
err);
        } else {
            console.log('Log entry recorded:', logEntry);
        }
    });
}

// Function to start monitoring
function startMonitoring() {
    setInterval(logData, 3600000); // Log data every hour
}

// Start monitoring
startMonitoring();
```

	`    myGame.simulateEvent("Save the village");`
	
	`    myGame.displayStatus();`
	
	`    return 0;`
	`}`	

Code review	As part of our cybersecurity module, I'm tasked with creating a Bash script that implements a basic file encryption and decryption service. The service should allow users to encrypt files using OpenSSL and decrypt them using a password. This is my initial implementation:
	
	Unset
	`#!/bin/bash`
	
	`# Simple file encryption and decryption service using OpenSSL`
	
	`encrypt_file() {`
	`    local file_to_encrypt=$1`
	`    echo "Encrypting $file_to_encrypt..."`
	`    openssl enc -aes-256-cbc -salt -in "$file_to_encrypt" -out "${file_to_encrypt}.enc" -k "$2"`
	`    echo "File encrypted: ${file_to_encrypt}.enc"`
	`}`
	
	`decrypt_file() {`
	`    local encrypted_file=$1`
	`    echo "Decrypting $encrypted_file..."`
	`    openssl enc -aes-256-cbc -d -in "$encrypted_file" -out "${encrypted_file%.enc}" -k "$2"`
	`    echo "File decrypted: ${encrypted_file%.enc}"`
	`}`
	
	`# Check if user wants to encrypt or decrypt`
	`if [[ $1 == "encrypt" && -f $2 ]]; then`
	`    encrypt_file "$2" "$3"`
	`elif [[ $1 == "decrypt" && -f $2 ]]; then`
	`    decrypt_file "$2" "$3"`
	`else`
	`    echo "Usage: $0 [encrypt|decrypt] [filename] [password]"`
	`fi`
	
	However, I have concerns about some potential security risks. For example, the current implementation accepts the encryption password in plaintext as a command-line argument, which can expose sensitive information. I also want to ensure the script handles errors more gracefully and is efficient for large file sizes. Please help review the code and suggest improvements to make it more secure and scalable. How can I avoid exposing the password and improve the script's error handling and efficiency for large files?
		[This is bad because it's too vague!]
		
		I'm working on the "Living the Social Life" web page. Please review the HTML and CSS code below and do the following:
		
		1. Enhance the code structure and organization.
		2. Update the HTML code to use appropriate HTML5 semantic elements.
		
		Here is the HTML code:
		
		Unset
		`html placeholder`
		
		Here is the CSS code:
		
		Unset
		`css placeholder`

Code Editing/Writing (Modification)

I wrote HTML and JavaScript codes. I aimed to print a car on the screen and move it with the "WASD" keys. However, the car doesn't look like a car. There's only a red rectangle, and I expect you to change it to a more realistic car. The point of view should be from above, so I expect to see the car from the upper perspective. I should be able to see the 4 tires, the windscreen, and the front lamps. Also, add a feature that allows the car to explode when the user presses the "space" button, and a "game over" message should be seen on the screen. There's no need to add any buttons or functionalities in the "game over" screen. Here is my code:
HTML:

Unset
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Move Car with WASD Keys</title>
  <style>
    body {
      margin: 0;
      overflow: hidden;
      background-color: lightgray;
    }
    #gameArea {
      width: 100vw;
      height: 100vh;
      position: relative;
    }
    #car {
      width: 50px;
      height: 100px;
      background-color: red;
      position: absolute;
      top: 50%;
      left: 50%;
      transform: translate(-50%, -50%);
    }
  </style>
</head>
<body>
  <div id="gameArea">
    <div id="car"></div>
  </div>

  <script src="script.js"></script>
</body>
</html>
```

JavaScript:

[Prompt would have been better if contributor had given the model an `index.html` and/or some more specific instructions about how the contributor planned to setup and execute the code. Additionally, this is a strange thing to ask for; the contributor essentially having the model build a back end on the front end. In terms of prompts, this is a rather unrealistic/unfeasible scenario for a model to address.]

I am working on upgrading the Task Manager class in JavaScript into a sophisticated task management application. The following enhancements are required: first, implement a User model to manage multiple users, allowing each user to have their own set of tasks. Incorporate a method for adding users, ensuring tasks are associated with the specific user. Next, modify the Task model to include a tags property that allows tasks to be categorized by multiple tags. The addTask method should ensure validation for the task. Include methods for filtering tasks based on priority and sorting tasks by due date. Finally, local file storage for persistent task management should be implemented to save and load tasks automatically from the file. It should not duplicate users and tasks. Users should be loaded into the users' list, and tasks should be loaded into the tasks list by default. Here is the code:

JavaScript
```
// Task Model
class Task {
    constructor(id, title, completed = false) {
        this.id = id;
        this.title = title;
        this.completed = completed;
    }
}

// Task Manager Class
class TaskManager {
    constructor() {
        this.tasks = [];
    }

    addTask(task) {
        if (!task.id || !task.title) {
            throw new Error('Invalid task. Ensure it has
an id and title.');
        }
        this.tasks.push(task);
    }

    removeTask(taskId) {
        const index = this.tasks.findIndex(task =>
task.id === taskId);
```

JavaScript

```javascript
window.onload = function() {
  const car = document.getElementById('car');
  let carX = window.innerWidth / 2 - 25;
  let carY = window.innerHeight / 2 - 50;
  const speed = 10;

  car.style.left = `${carX}px`;
  car.style.top = `${carY}px`;

  document.addEventListener('keydown', (event) => {
    switch (event.key.toLowerCase()) {
      case 'w':
        carY -= speed;
        break;
      case 's':
        carY += speed;
        break;
      case 'a':
        carX -= speed;
        break;
      case 'd':
        carX += speed;
        break;
    }


    carX = Math.max(0, Math.min(carX, window.innerWidth - 50));
    carY = Math.max(0, Math.min(carY, window.innerHeight - 100));


    car.style.left = `${carX}px`;
    car.style.top = `${carY}px`;
  });
};
```

```javascript
        if (index === -1) throw new Error('Task ID not
found.');
        this.tasks.splice(index, 1);
    }

    getTasks() {
        return this.tasks;
    }
}

// Sample Usage
const taskManager = new TaskManager();
const task1 = new Task(1, "Learn JavaScript");
const task2 = new Task(2, "Refactor Task Manager");

taskManager.addTask(task1);
taskManager.addTask(task2);
console.log(taskManager.getTasks());
```

Code Ecosystem	I am a solutions architect at a tech company and I need an auto-scaling mechanism for a Python-based web application running in Docker containers, deployed across a hybrid cloud infrastructure (Microsoft Azure, AWS, and GCP). The system should scale based on real-time CPU load. The auto-scaling mechanism should use a custom load monitoring tool that will run within the containers and report the CPU load directly to a central monitoring system. When the CPU load exceeds a certain threshold, the system should automatically scale across all three cloud providers. Explain how this monitoring tool should be designed and how the communication between containers running in different cloud environments would take place even when affected by network inconsistencies and varying latencies while maintaining high availability and load balancing and considering data consistency, fault tolerance, and potential for cross-cloud issues. Explain each step.	**[This prompt is too vague and the ask and end goal is unclear]** Create a Python CLI tool that generates structured content using user input. Use Jinja2 for templating. The tool must insert inputs into a predefined template and save the result as a file locally.