A Case-based Reasoning Approach to Dynamic Few-Shot Prompting for Code Generation

Dustin Dannenhauer [1] **Zohreh Dannenhauer** [2] **Despina Christou** [1] **Kostas Hatalis** [1]

Abstract

Large language models have recently succeeded in various code generation tasks but still struggle with generating task plans for complex, real-world problems that need detailed, context-aware planning and execution. This work aims to enhance these models' accuracy in generating task plans from natural language instructions. These tasks plans, represented as python code, use custom functions to accomplish the user's request as specified in natural language. The task plans are multi-step, often include loops, and are executed in a python runtime environment. Our approach uses case-based reasoning to perform dynamic few-shot prompting to improve the large language models ability to accurately follow planning prompts. We compare the effectiveness of dynamic prompting with static three-shot and zero-shot prompting approaches finding that dynamic prompting improves the accuracy of the generated code. Additionally, we identify and discuss seven types of failures in code generation.

1. Introduction

Large language models (LLMs) have demonstrated success on a variety of tasks such as copywriting and marketing (Christou et al., 2024), natural language processing (Chowdhery et al., 2023), and code generation (Liu et al., 2024). While LLMs are proficient in generating human-quality outputs, they lack in accuracy and context-specific generation needed specifically for coding tasks. These limitations are partly due to inherent issues such as the inability to access up-to-date information on recent events (Lazaridou et al., 2022), a tendency to hallucinate facts (Maynez et al., 2020; Ji et al., 2023), and lack of long-term memory (Hatalis et al., 2024). Additionally, natural language prompts describing tasks are inherently ambiguous, leading to numerous possible interpretations and inconsistencies in the generated code.

In this work we address the problem of generating multi-step task plans represented as python source code, using base Python and custom functions provided to the LLM at runtime. Data from one function call often serves as input for subsequent function calls. Additionally, loops and if-statements are also required to fulfill the original task prompt.

To solve this problem, we introduce a case-based reasoning (CBR) approach that maintains a case-base of <problem,solution> pairs. When a user prompt is given to the system, the most similar cases are retrieved and given to the LLM as guidance to writing a new task plan. CBR is a 5 step problem solving framework: (1) Given a problem, retrieve similar past problems and their corresponding solutions, (2) Adapt a new solution based on these past pairs, (3) Review the output directly or collect user feedback, and (4) Retain the solution as a new case if successful, otherwise start over and modify the retrieve step (Watson & Marir, 1994).

CBR can be considered a lazy form of problem solving (Aha, 1997). Common statistical machine learning approaches, which are greedy, take all the training data, compute a model, and freeze that model for all future inputs; however CBR waits until it sees a new problem, retrieves the most similar past problem, solution pairs, and then adapts those past solutions for the current problem. Traditionally, the adaptation step (2) is difficult, and in this work we use the LLM to perform the adaptation. A key benefit of CBR is its ability to immediately apply new knowledge by adding a single case to the case base.

The contributions of this work are: (1) a CBR approach to dynamic prompting for code generation, (2) a classification of seven failure types in LLM code generation, and (3) an empirical evaluation of our CBR approach against zero-shot and static three-shot approaches. The paper begins with an introduction to the problem of LLMs' limitations in task planning through code generation and presents the CBR approach as a solution, along with background on generative

[1]GoCharlie.ai [2]Booz Allen Hamilton. Correspondence to: Dustin Dannenhauer <dustin@gocharlie.ai>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

AI techniques and related work. It then details our CBR dynamic prompting approach, describes the empirical setup and results, and concludes with a discussion of limitations and future work suggestions.

2. Background

Generative AI approaches to generating Python code have garnered significant attention in recent years, with several pioneering studies contributing to this field. Chen et al. (2021) introduced Codex, showcasing its capabilities in generating Python code from natural language descriptions. Feng et al. (2020) presented CodeBERT, which demonstrated the effectiveness of a pre-trained model for understanding and generating programming languages. Ahmad et al. (2021) introduced PLBART, a model specifically designed for programming language tasks. Fried et al. (2022) presented InCoder highlighting its ability to infill and synthesize code based on natural language descriptions. Nijkamp et al. (2022) introduced CodeGen focusing on multi-turn program synthesis. In this paper we specifically look at the task of generating task plans with custom functions in python that will be executed in a python runtime environment, as opposed to calling functions one at a time during conversations, like the OpenAI assistants API.[1]

LLMs increasingly integrate tool usage for enhanced capabilities and adaptability. ToolFormer (Schick et al., 2024) dynamically decides when to employ external tools during training, enabling access to APIs and databases for real-time information and task execution beyond pre-trained knowledge. However, it lacks sequential tool usage. Similarly, the Gorilla model improves LLMs' API interactions, using a retriever-aware training approach accesses comprehensive API datasets for more accurate API calls in response to natural language queries (Patil et al., 2023). WebGPT enables LLMs to access and interact with web pages for up-to-date information and accurate responses (Nakano et al., 2021). Additionally, (Smith et al., 2023) explore the synergy between human input and automated tool use in conversational AI. The primary difference between these tool use LLMs is that they are designed to call tools one step at a time (or multiple one-step calls in parallel) rather than produce python code that uses these tools sequentially.

LLMs also struggle with understanding specific project requirements like variable scopes, dependencies, and custom functions and libraries, especially in intricate programming tasks demanding deep domain understanding. Although trained on vast datasets, LLMs may lack the contextual knowledge needed for specific tasks, leading to syntactically correct but functionally incorrect code. Xu et al. (2022) highlight that existing benchmarks often do not include scenarios involving custom functions, failing to capture the complexities of real-world programming tasks involving bespoke tools and libraries.

[1] https://platform.openai.com/docs/assistants/overview

2.1. Prompt Engineering Approaches

Few-shot prompting is a prompt engineering technique to enable in-context learning for LLMs that significantly enhance their performance. This method involves providing the model with a small number of explicit examples that demonstrate the desired output format, structure, or task. By leveraging these well-crafted examples, few-shot prompting enables LLMs to perform a wide range of tasks not seen during initial pre-training or fine-tuning phases (Brown et al., 2020; Ma et al., 2024). The technique's effectiveness lies in guiding the model to generate appropriate responses by inferring required structure and style from the examples. For instance, when presented with a few sample sentences showcasing a specific writing style or format, an LLM can grasp the underlying patterns and generate new content that adheres to the same style or format.

Dynamic few-shot prompting addresses the limitations of static few-shot prompting by adapting to the specific requirements of each task, providing the most relevant examples to guide the model's output. Jie & Lu (2023) show that for general program generation that dynamic prompting using past, correct programs improved code generation and reasoning over code and math problems. However, their study didn't explore the use of custom functions within larger systems. Dynamic prompting is particularly important in scenarios involving custom tools and functions, where static examples may not adequately cover the variations and intricacies of the task. Similarly, in the conversational AI domain, the Few-shot Bot (FSB) employs prompt-based few-shot learning to handle diverse conversational skills without extensive retraining, demonstrating the versatility of few-shot techniques across various applications (Madotto et al., 2021).

Chain of Thought (CoT) prompting enhances problem-solving in large language models by modeling intermediate reasoning steps, improving performance on complex tasks like numerical reasoning (Wei et al., 2022). The Tree of Thoughts (ToT) framework advances this by allowing models to explore multiple reasoning paths and dynamically evaluate different solutions, significantly enhancing decision-making compared to CoT's linear thought process (Yao et al., 2024). Both techniques rely solely on internal model computations and do not incorporate external data during the reasoning process.

Separately, Retrieval Augmented Generation (RAG) involves embedding the original prompt and searching a database to retrieve relevant data before processing (Lewis et al., 2020), effectively augmenting the prompt with exter-

nal information, unlike CoT and ToT. Our CBR performs retrieval of cases using embedding search, similar to RAG.

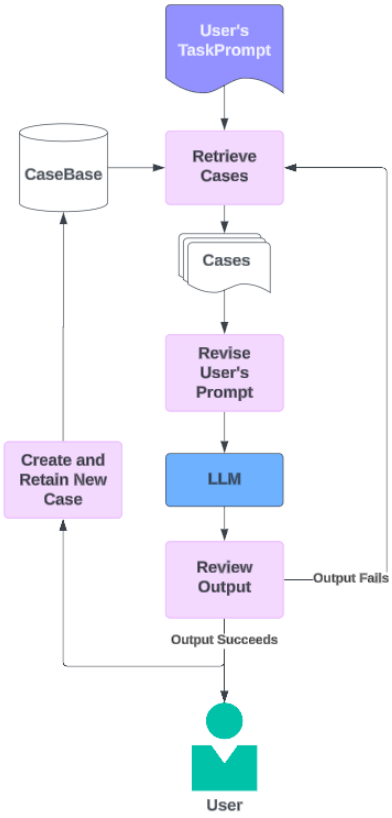

Figure 1. Case-based Reasoning with a LLM for Case Adaptation

3. CBR for Dynamic Few Shot Prompting

Figure 1 depicts our case-based reasoning dynamic few-shot prompting approach responding to user's task requests. Starting with the user's prompt, past cases are retrieved based on cosine similarity between prompt embeddings. Those cases' solutions are then appended to the user's prompt before being passed to the LLM, which generates new python code. This python code is then executed and the outputs are sent to the user. If the user provides positive feedback, this case is stored for future re-use in the case base. Positive feedback stores the case for reuse, while negative feedback temporarily affects retrieved cases' weights before a new retrieval step begins.

A case in the case base contains the following:

Task Prompt: This is a description of the task to be done, like what a user would request.

Task Plan: This is a natural language description of the steps to be carried out. This plan provides additional context for the python code.

Python Code: This is a working segment of python code that can be directly executed to achieve the task.

4. Experimental Setup

We conduct an evaluation of our dynamic few-shot prompting approach against a static version and a zero-shot version. For each version we test using GPT 4 (specifically *gpt-4*) chat completion APIs[2]. The zero-shot code generation API call is given a prompt with a description of the tools and the description of the task from the user, but no examples. For the static version, we randomly picked 3 cases from our case base (out of 41) to provide for every API call along with the user's task at the end. For the dynamic version, we use OpenAI *text-embedding-ada-002* embeddings to retrieve cases using the test prompt and the prompt portion of the case. These prompts are given in Appendix B. We set temperature=0 for all API calls.

The test set is comprised of 30 user request prompts (Appendix A) that are representative of user requests on our commercial GoCharlie.ai platform regarding web search, content creation, and image creation. The custom functions that are made available to the LLM to use in creating task plans are the following (see the prompts in Appendix B for the full definitions):

save_outputs(...) Returns a result to the user.

add_context(...) Saves content to be used by later custom function calls.

get_user_uploaded_file_data(...) Retrieve file data contents from a user uploaded file.

generate_content(...) Perform a writing task such as writing a blog or social media post.

generate_image(...) Create an AI generated image using the given prompt.

search_tool(...) Call an external search engine, such as Google.

scrape_url(...) Extract the natural english language text from a webpage.

5. Results

We manually evaluated Python code results to determine success or failure reasons. Figure 2 summarizes overall performance. Dynamic few-shot prompting surpasses static few-shot and zero-shot, though the gap between static and dynamic is smaller than expected. This may stem from our test set's limitations (Appendix A), lacking diverse prompts such as those involving user uploaded files.

[2]https://platform.openai.com/docs/api-reference/chat

Approach	FCE	MNFC	FCOO	EPDBF	MAFCO	DNFOP	FCFCO	Total Failures
Zero-Shot GPT-4	-	3.33	13.33	3.33	3.33	3.33	-	30.00
Static 3-Shot GPT-4	-	-	-	-	20.00	-	3.33	23.33
Dynamic 3-Shot GPT-4	3.33	3.33	-	-	13.33	-	-	20.00

Table 1. Percentage of failures by type. FCE: Function Call Error, MNFC: Missing Needed Function Call, FCOO: Function Calls Out of Order, EPDBF: Error Passing Data Between Functions, MAFCO: Make Assumption on Function Call Output, DNFOP: Does Not Follow Original Prompt, FCFCO: Failure to Check Function Call Output, Total Failures: Total number of failures for each approach

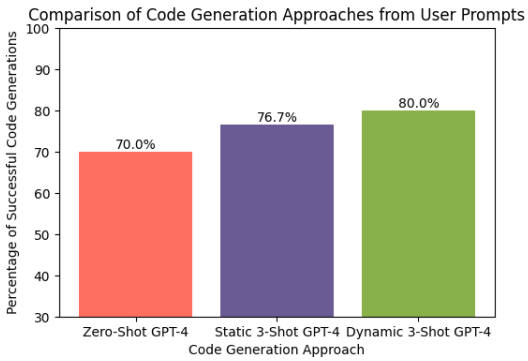

Figure 2. Percentage of Correct Code Generations

Unlike other benchmark evaluations, we do not have clear numeric measures to evaluate whether the generated code was good enough for the user. Generated code may be syntactically correct but fail to align with the user's intent or utilize tools properly. After manual evaluation, we have categorized failures into seven categories:

Function Call Error (FCE): This failure arises from improper function calls. For instance, if a function like save_outputs expects image filenames as arguments, the LLM might incorrectly pass text content instead.

Missing Needed Function Call (MNFC): This failure occurs when a required function call is absent in the generated code, such as neglecting to perform a necessary web search.

Function Calls Out of Order (FCOO): This failure happens when function calls occur in an incorrect sequence. For instance, if add_context() isn't called before another function that requires that context.

Error Passing Data Between Functions (EPDBF): This failure occurs when the output of one function isn't used for another function. For example, not using web search results when writing content, and instead only writing content from a prompt.

Makes Assumptions on Function Call Output (MAFCO): This failure happens when the LLM incorrectly assumes the format of output data from a function call. For example, assuming that the title from generate_blog() will end with a period and attempting to retrieve it by splitting on a period.

Does Not Follow Original Prompt (DNFOP): This failure happens when the generated code incorrectly assumes the intent of the prompt (possibly due to a lack of context from the user). For instance, assuming that a product description requires an AI-generated image attachment (which is incorrect).

Failure to Check Function Call Output (FCFCO): This failure arises when the generated code lacks checks on function outputs before utilization. For instance, when user-uploaded data returns a list, but the generated code fails to verify if the list is empty in cases where the user didn't upload anything.

Table 1 shows failure percentages by type and approach, revealing notable differences. Zero-shot method showed highest "Function Calls Out of Order" errors, while both static and dynamic few-shot had more "Making Assumptions on Function Call Output" errors, with dynamic few-shot showing fewer errors than static. These results indicate potential for dynamic few-shot prompting as a promising approach.

6. Conclusion

Our integration of case-based reasoning for dynamic few-shot prompting improves code generation accuracy over static three-shot and zero-shot methods, notably in minimizing seven common code generation errors. As LLMs expand into more complex domains, methods such as case base reasoning to improve prompts are increasingly valuable. However, their efficacy relies heavily on the careful curation of the case bases with human feedback to avoid misleading the LLM.

Future work should focus on extending dynamic prompting to other tasks and refining case-based reasoning for broader problem handling. Advanced case retrieval methods beyond simple embeddings retrieval are also necessary for improved accuracy. These advancements hold promise for more reliable code-generating models, enhancing automated assistance in software development and technical fields.

References

Aha, D. W. Lazy learning. In *Lazy learning*, pp. 7–10. Springer, 1997.

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Plbart: Pre-trained language model for programming language understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33: 1877–1901, 2020.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Christou, D., Hatalis, K., Staton, M. G., and Frechette, M. Chatgpt for marketers: Limitations and mitigations. *Journal of Digital & Social Media Marketing*, 11(4):307–323, 2024.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1536–1547, 2020.

Fried, D., Fu, T.-J., Andreas, J., Klein, D., and Chen, E. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Hatalis, K., Christou, D., Myers, J., Jones, S., Lambert, K., Amos-Binks, A., Dannenhauer, Z., and Dannenhauer, D. Memory matters: The need to improve long-term memory in llm-agents. *Proceedings of the AAAI Symposium Series*, 2024. URL https://api.semanticscholar.org/CorpusID:267224915.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

Jie, Z. and Lu, W. Leveraging training data in few-shot prompting for numerical reasoning. *arXiv preprint arXiv:2305.18170*, 2023.

Lazaridou, A., Gribovskaya, E., Stokowiec, W., and Grigorev, N. Internet-augmented language models through few-shot prompting for open-domain question answering. *arXiv preprint arXiv:2203.05115*, 2022.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

Liu, M., Wang, J., Lin, T., Ma, Q., Fang, Z., and Wu, Y. An empirical study of the code generation of safety-critical software using llms. *Applied Sciences*, 14(3):1046, 2024.

Ma, H., Zhang, C., Bian, Y., Liu, L., Zhang, Z., Zhao, P., Zhang, S., Fu, H., Hu, Q., and Wu, B. Fairness-guided few-shot prompting for large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Madotto, A., Lin, Z., Winata, G. I., and Fung, P. Few-shot bot: Prompt-based learning for dialogue systems. *arXiv preprint arXiv:2110.08118*, 2021.

Maynez, J., Narayan, S., Bohnet, B., and McDonald, R. On faithfulness and factuality in abstractive summarization. *arXiv preprint arXiv:2005.00661*, 2020.

Nakano, R., Hilton, J., Baldridge, J., Wu, J., Ouyang, L., Kim, C., Gilboa, E., Hesse, C., Puri, R., Robinson, S., et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Nijkamp, E., Pang, B., Xu, H., Curtright, T., Singh, M., Han, W., Kong, X., Tu, K., Wu, Y., Zhu, Y., et al. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

Smith, J., Doe, J., Brown, E., and Johnson, M. Human-loop: Tool use in conversational ai. *arXiv preprint arXiv:2301.12345*, 2023.

Watson, I. and Marir, F. Case-based reasoning: A review. *The knowledge engineering review*, 9(4):327–354, 1994.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting

elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

A. Evaluation Prompts Test Set

These prompts are representative of user requests on the GoCharlie.ai platform, and we use them as the test set in this paper to evaluate python code generation.

1. "Generate an image a cat floating in outer space"
2. "Create 2 posts for instagram linkedin twitter and facebook about my product https://www.etsy.com/listing/1503986620/magic-card-template-for-laser-engraving?"
3. "Analyze the current state of microchips in cell phones and its trends challenges and opportunities including relevant data and statistics. Provide a list of key players and a short and long-term industry forecast and explain any potential impact of current events or future developments."
4. "Offer a detailed review of Discord for Etsy shop selling DnD dice."
5. "My business is a Dry Cleaner in Eastern Market Washington DC. Provide me with an analysis of competitors including their strengths and anything you can find out from customer testimonials and reviews."
6. "I'm a realtor please create a series of social media posts for facebook about this listing https://www.zillow.com/homedetails/5322-S-Big-Lake-Rd-Wasilla-AK-99623/249629577_zpid/"
7. "Generate a list of 5 LinkedIn articles to write for bitcoin mining and choose one to make a blog about"
8. "Turn the facebook post at this https://www.facebook.com/groups/pythondev/ into a series of tweets about my community"
9. "Tell me a joke about dogs. Include a meme."
10. "Create a social media post that targets surfers and explains how our product https://windy.app/ can help them."
11. "You are an expert social media manager. I want you to create a schedule for social media posts over one month starting from July 1st. The frequency of posting will be daily. My business is called PencilsPlus and we sell premium mechanical pencils. For each post include the day it will be published a heading body text and include relevant hashtags. The tone of voice we use is friendly. For each post also include a suggestion for an image that we can use that could be found on a stock image service."
12. "Create the back story of a local pizza restaurant started by two brothers. Based off of this make a story make instagram and twitter captions pull stock images that matches the story and create a meme."
13. "Create instagram and pinterest posts and a blog post based off this link: https://www.homedepot.com/p/Artika-Imperium-9-Light-Black-and-Gold-Modern-Sputnik-Geometric-Cage-Chandelier-Light-Fixture-for-Dining-Room-or-Kitchen-CHMP-HD2BG/324526141"
14. "Create a blog post with an inspiring message that promotes our product https://www.thehumansolution.com/ergonomic-mouse.html. Focus on how the product can help people overcome a challenge or make their lives easier. Add emojis. Add a clinical tone."
15. "Analyze the website https://www.nytimes.com and provide a summary of the top 5 news stories including key details and implications."
16. "Generate an image of a robot dancing in a field of sunflowers and create a series of social media posts about the importance of finding joy in unexpected places."
17. "Research the current state of renewable energy adoption worldwide including relevant data trends and challenges. Provide a list of key players and a short and long-term industry forecast."
18. "Create a detailed review of Trello for a small business owner looking to improve their project management processes."
19. "I run a vegan bakery in Portland Oregon. Provide me with an analysis of local competitors including their strengths and weaknesses based on customer reviews and testimonials."

20. "I'm a fitness coach please create a series of Instagram posts about this product: https://www.nike.com/t/air-zoom-pegasus-39-mens-road-running-shoes-2Dc8l9"

21. "Generate a list of 5 blog post ideas related to sustainable fashion and choose one to write a detailed outline for."

22. "Turn the most recent tweet from @NASAClimate into a series of Facebook posts about the importance of addressing climate change."

23. "Tell me a joke about cats. Include a relevant GIF."

24. "Create a social media post that targets new parents and explains how our product https://www.babysense.com can help them monitor their baby's sleep."

25. "You are an expert content strategist. I want you to create a content calendar for blog posts over three months starting from September 1st. The frequency of posting will be weekly. My business is called GreenThumb and we sell organic gardening supplies. For each post include the week it will be published a title a brief description and relevant keywords. The tone of voice we use is informative and encouraging."

26. "Create the origin story of a sustainable fashion brand founded by three college friends. Based on this write a press release generate Instagram and Twitter captions find stock images that match the story and create a meme."

27. "Create Pinterest posts and a detailed product description based on this link: https://www.patagonia.com/product/mens-better-sweater-fleece-jacket/25528.html"

28. "Write a blog post with a humorous tone that showcases our product https://www.squattypotty.com. Focus on how the product can make a mundane task more enjoyable. Add relevant puns and a touch of irreverence."

29. "Create a comprehensive blog post about the benefits of practicing yoga for stress relief. Use the search engine to find relevant scientific studies and expert opinions to support your points. Generate a list of 5 simple yoga poses that can be done at home or in the office and provide step-by-step instructions for each pose. Create 2-3 images that visually demonstrate these poses and integrate them into the blog post. Additionally generate a short guided meditation script that readers can use to further reduce stress. Finally create social media posts for Instagram Facebook and Twitter to promote the blog post using relevant hashtags and compelling visuals."

30. "You are a travel blogger who has just returned from a trip to Bali Indonesia. Write a detailed blog post about your experience highlighting the best attractions restaurants and hidden gems you discovered during your visit. Use the search engine to find high-quality images of the locations you mention and integrate them throughout the post. Create a 'Top 5' list of must-see places in Bali and provide a brief description of each. Generate an infographic that showcases interesting facts and statistics about Bali's culture history and tourism industry. Finally create a series of Instagram posts and stories that give your followers a behind-the-scenes look at your trip using the images you found and relevant hashtags to increase engagement."

B. Static Prompts

These are the static prompts used in our evaluation. These prompts are appended with a test prompt when given to the LLM for chat completion.

B.1. Zero-shot static prompt

```
I will ask you to perform a task. Your job is to devise a plan to execute that
    \      task, followed by writing a Python function calling a series of tools
    \      to execute that task. You can ONLY use the tools listed and ignore
    \      tasks that cannot be solved with the given tools. Do NOT assume any
    \      other functions. You must always return a Python function.

Tools:

    - util
```

```
Description: a utilities class instance used to save context for other
   \      api tools (listed below) and to save outputs
to the user. The following functions are all within the util class.
- util.save_outputs(primary_content: str, primary_type: str,
   \      attached_files: [str]) -> None
        Description: saves the content and attached files so the user
           \      will receive it as an output; remember that images
           \      should always go in attached files, even if
           \      primary_content is empty.
        Inputs:
            primary_content (str): main text content.
            primary_type (str): the type the content is like blog, email,
               \       etc.
            attached_files (list): a list of attached files to the
               \      primary content (note: images should always be
               \      given in this list and not as the primary_content).
        Output: None

- util.add_context(content: str, context_type: str) -> None
        Description: stores the content as context for the next api tools
           \       to use. By adding context with this
        function, the next api tools that are called with already have
           \      access to the content.
        Inputs:
            content (str): content, such as relevant text, to be used as
               \      context to improve other api tools.
            context_type (str): optional, the type of content being added
               \       which must be one of the following options only:
                - 'transcriptions' for video and audio links,
                - 'search_results' for search engine results,
                - 'website' for scraped website content,
        Output: None

- util.get_user_uploaded_file_data(file_data_type: str) -> [str]
    Description: retrieve data that has been already processed from the
       \      user, given the type of data
    Inputs:
        file_data_type (str): only one of
            - 'transcriptions',
            - 'image_captions',
            - 'documents'
    Output: a list of strings of text data, one string per file

- util.generate_content(prompt: str, content_type: str) - > str
    Description: generates text content given a prompt and content type.
    Inputs:
        prompt (str): a description of the content we want to create.
        content_type (str): one of the following options only 'text', '
           \      blog', 'facebook', 'instagram', 'twitter', 'linkedin'.
    Output: a string of the generated text content.

- util.generate_image(prompt: str, image_type: str) -> str
    Description: generates an image based on a description and type of
       \      image.
```

```
            Inputs:
                prompt (str): a description of what kind of image we want.
                image_type (str): one of the following options only 'ai_generated
                    \      ', 'logo', 'meme', 'gif'.
            Output: an image filename as a string

        - util.search_tool(prompt: str, search_type: str, n: int) -> str
            Description: Google or news search based on a query prompt. It will
                \      only return links and descriptions, not images.
            Inputs:
                prompt (str): query we want to search.
                search_type (str): one of the following options 'google', 'news'.
                n (int): an optional input of the number of search results we
                    \      want.
            Output (str): text search results with the format "URL, Name, Content
                \      "

        - util.scrape_url(url: str) -> str
            Description: scrape the text from a website. This can also scrape
                \      watch youtube videos.
            Inputs:
                url (str): a valid url.
            Output: the text content from the url




Here is the user's task. Write only python code and nothing else, so that I can
    \      copy and paste it and then immediately run it (don't use markdown or
    \      any other non-python text before or after the python code, this is very
    \       important). Make sure that the python code is in a single function
    \      named 'function' that takes only the util variable.:
```

B.2. 3-shot static prompt

```
I will ask you to perform a task. Your job is to devise a plan to execute that
    \      task, followed by writing a Python function calling a series of tools
    \      to execute that task. You can ONLY use the tools listed and ignore
    \      tasks that cannot be solved with the given tools. Do NOT assume any
    \      other functions. You must always return a Python function.

Tools:

    - util
        Description: a utilities class instance used to save context for other
            \      api tools (listed below) and to save outputs
        to the user. The following functions are all within the util class.
        - util.save_outputs(primary_content: str, primary_type: str,
            \      attached_files: [str]) -> None
                Description: saves the content and attached files so the user
                    \      will receive it as an output; remember that images
                    \      should always go in attached files, even if
                    \      primary_content is empty.
                Inputs:
```

```
            primary_content (str): main text content.
            primary_type (str): the type the content is like blog, email,
               \       etc.
            attached_files (list): a list of attached files to the
               \      primary content (note: images should always be
               \      given in this list and not as the primary_content).
        Output: None

- util.add_context(content: str, context_type: str) -> None
        Description: stores the content as context for the next api tools
           \       to use. By adding context with this
        function, the next api tools that are called with already have
           \      access to the content.
        Inputs:
            content (str): content, such as relevant text, to be used as
               \      context to improve other api tools.
            context_type (str): optional, the type of content being added
               \       which must be one of the following options only:
                - 'transcriptions' for video and audio links,
                - 'search_results' for search engine results,
                - 'website' for scraped website content,
        Output: None

- util.get_user_uploaded_file_data(file_data_type: str) -> [str]
    Description: retrieve data that has been already processed from the
       \      user, given the type of data
    Inputs:
        file_data_type (str): only one of
            - 'transcriptions',
            - 'image_captions',
            - 'documents'
    Output: a list of strings of text data, one string per file

- util.generate_content(prompt: str, content_type: str) - > str
    Description: generates text content given a prompt and content type.
    Inputs:
        prompt (str): a description of the content we want to create.
        content_type (str): one of the following options only 'text', '
           \      blog', 'facebook', 'instagram', 'twitter', 'linkedin'.
    Output: a string of the generated text content.

- util.generate_image(prompt: str, image_type: str) -> str
    Description: generates an image based on a description and type of
       \      image.
    Inputs:
        prompt (str): a description of what kind of image we want.
        image_type (str): one of the following options only 'ai_generated
           \      ', 'logo', 'meme', 'gif'.
    Output: an image filename as a string

- util.search_tool(prompt: str, search_type: str, n: int) -> str
    Description: Google or news search based on a query prompt. It will
       \      only return links and descriptions, not images.
    Inputs:
```

```
                prompt (str): query we want to search.
                search_type (str): one of the following options 'google', 'news'.
                n (int): an optional input of the number of search results we
                    \      want.
            Output (str): text search results with the format "URL, Name, Content
                \      "

        - util.scrape_url(url: str) -> str
            Description: scrape the text from a website. This can also scrape
                \      watch youtube videos.
            Inputs:
                url (str): a valid url.
            Output: the text content from the url


Task: What are the top 10 sites about AI?
Plan:
1. Search for "AI" on Google and retrieve the top 10 search results.
2. Return the list of search results as the answer.
Code:
def function(util):
    prompt = "AI"
    search_type = 'search'
    n = 10
    search_results = util.search_tool(prompt=prompt, search_type=search_type, n=n
        \      )
    util.add_context(search_results)

    prompt = "List the top 10 sites on AI with brief summary."
    content_type = "text"
    answer = util.generate_content(prompt=prompt, content_type=content_type)

    util.save_outputs(primary_content=answer, primary_type='answer')

    return util
<|#####|>
Task: Search the latest news in Generative AI and summarize it in bullet points.
    \      Then, create a blog brief for me.
Plan:
1. Use the search tool to find the latest news in Generative AI.
2. Analyze the text content and generate bullet points summarizing the news.
3. Save the bullet points as outputs.
4. Generate a blog brief based on the bullet points.
5. Save the blog brief as an output.
Code:
def function(util):
    prompt = 'latest news in Generative AI'
    search_type = 'news'
    n = 5
    search_results = util.search_tool(prompt=prompt, search_type=search_type, n=n
        \      )
    util.add_context(search_results)

    prompt = "Summarize the news in Generative AI in bullet points. Cite sources
```

```
       \      ."
    content_type = "Text"
    bullet_points = util.generate_content(prompt=prompt, content_type=
       \      content_type)
    util.add_context(bullet_points)

    util.save_outputs(primary_content=bullet_points, primary_type='bullet_points
       \      ')

    prompt = "From the bullet point News in Generative AI, write a blog breif."
    content_type = 'blog_brief'
    blog_brief = util.generate_content(prompt=prompt, content_type=content_type)

    util.save_outputs(primary_content=blog_brief, primary_type='blog_brief')

    return util
<|#####|>
Task: Make two tweets based off of this video https://www.youtube.com/watch?v=
   \      jEeWiTGMWCw
Plan:
1. Scrape the youtube video url to get the transcription.
2. Save the transcription text as context.
3. Write the first tweet.
4. Save the first tweet.
5. Write the second tweet.
6. Save the second tweet.
Code:
def function(util):
    url = "https://www.youtube.com/watch?v=jEeWiTGMWCw"
    transcribed_text = util.scrape_url(url=url)
    util.add_context(transcribed_text)

    prompt = 'Write a tweet based on the video transcription'
    content_type = 'twitter'

    tweet1 = util.generate_content(prompt=prompt, content_type=content_type)
    util.save_outputs(primary_content=tweet1, primary_type='tweet')

    tweet2 = util.generate_content(prompt=prompt, content_type=content_type)
    util.save_outputs(primary_content=tweet2, primary_type='tweet')

    return util
<|#####|>
Task: Offer a detailed review of Discord for Etsy shop selling DnD dice.
Plan:
1.
```