Pairing Analogy-Augmented Generation with Procedural Memory for Procedural Q&A

Anonymous ACL submission

Abstract

While LLMs in the RAG paradigm have shown remarkable performance on a variety of tasks, they still under-perform on unseen domains, especially on complex tasks like procedural question answering. In this work, we introduce a novel formalism and structure for manipulating text-based procedures. Based on this formalism, we further present a novel dataset called LCStep, scraped from the LangChain Python docs. Moreover, we extend the traditional RAG system to propose a novel system called Analogy Augmented Generation (AAG), that draws inspiration from human analogical reasoning and ability to assimilate past experiences to solve unseen problems. The proposed method uses a frozen language model with a custom procedure memory store to adapt to specialized knowledge. We demonstrate that AAG outperforms few-shot and RAG baselines on LCStep, RecipeNLG, and the CHAMP datasets under a pairwise LLM-based evaluation.

1 Introduction

Procedural knowledge, or "knowledge-how", is a form of knowledge involving the ability to *do* something, as opposed to prepositional knowledge, which is the awareness of a fact ("knowledge-that") (Pavese, 2022). Philosophers debate whether procedural knowledge is always reducible to prepositional knowledge, but the functional distinction is regardless significant in the study of LLM-based agents, which are expected to not only understand facts about their environment but also *plan* and *reason* about actions in that environment.

To that end, efforts have been made both theoretically (Georgeff and Lansky, 1986) and empirically (Huang et al., 2022) to enable information systems that explicitly handle procedural knowledge, but the recent success of LLMs on a wide variety of tasks has led to a new interest in LLM agents which are expected to plan, reason, and act within a world of some kind, whether in the real world (Wang et al., 2024b) or in a virtual environment (Huang et al., 2022; Wang et al., 2024a; Tan et al., 2024).

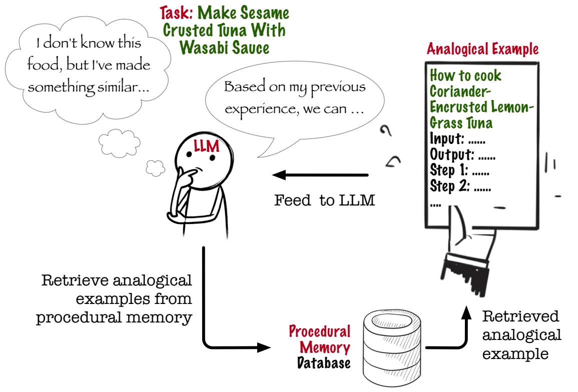

Task: Make Sesame Crusted Tuna With Wasabi Sauce
I don't know this food, but I've made something similar...
Based on my previous experience, we can ...
LLM
Analogical Example
How to cook Coriander-Encrusted Lemon-Grass Tuna
Input: ......
Output: ......
Step 1: ......
Step 2: ......
....
Feed to LLM
Retrieve analogical examples from procedural memory
Procedural Memory Database
Retrieved analogical example

Figure 1: An illustration of using the AAG system to generate a recipe for an unseen food.

Unfortunately, one way LLMs consistently struggle is in underrepresented or specialized domains not well-represented in training data, or domains which require up-to-date knowledge (Kandpal et al., 2023). In these circumstances, rather than report a lack of knowledge LLMs tend to 'hallucinate' responses that are incoherent or factually incorrect (Zhang et al., 2023). A popular approach to mitigate these limitations is Retrieval-Augmented Generation (RAG) (Lewis et al., 2021), which integrates text retrieved from an external set of documents into the prompt to the language model. This approach has been shown empirically to increase the accuracy of LLM responses across a wide range of tasks (Gao et al., 2024).

While LLMs in a RAG paradigm have made drastic improvements on factual knowledge tasks, procedural knowledge applications have been relatively less explored. Several recent works have explored procedural tasks in the domain of planning where the purpose is to create a step-by-step plan to achieve a goal (Brahman et al., 2023; Huang et al., 2022). However, these works focus on leveraging model's internal knowledge or limited external knowledge by better prompt engineering.

Inspired by the analogical reasoning capability of human cognition (Parsons and Davies, 2022), we propose Analogy-Augmented Generation (AAG), a novel system built upon the traditional RAG paradigm, for the use-case of procedural question answering. This AAG pipeline retrieves from a specialized procedural memory during a multi-step process that generates and refines a candidate procedure that accomplishes the input task (see Figure 1). AAG especially shines on tasks outside the training domain of the LLM. We expect AAG to surpass the performance of traditional RAG on knowledge-intensive procedural datasets like CHAMP (Mao et al., 2024) and our self-curated Langchain dataset. Our contributions in this paper are the following:

- We provide a formalism for text-based procedural knowledge that fits easily into existing information retrieval systems while also providing a minimal structure to be leveraged by specialized procedure storage in the future.
- We collect a dataset of procedures for accomplishing tasks with the Langchain Python library (Chase and contributors, 2022), and name this dataset LCStep. Since Langchain was introduced in October 2022, it does not feature in the training data of many LLMs, making LCStep a good test of a system's ability to handle procedural knowledge outside the distribution familiar to the LLM.
- Based on our procedural knowledge formalism, we devise a new RAG system called analogy-augmented generation (AAG), which leverages a store of existing procedural knowledge to solve procedural question answering tasks. We evaluate this system on LCStep, plus two other datasets (CHAMP (Mao et al., 2024) and RecipeNLG (Bień et al., 2020)) which we adapt for the procedural knowledge task. We show that leveraging this procedural knowledge improves the generated procedure steps according to LLM-based pairwise comparisons.

2 Related Work

Built upon the traditional RAG paradigm (Lewis et al., 2021), AAG differs from the conventional RAG in three key aspects: analogical reasoning, the iterative refinement workflow (section 3) and application in knowledge-intensive procedural QA. An extensive survey covering the status-quo of RAG research can be found in (Gao et al., 2024).

Standard Prompting Methods: Zero-shot prompting (Chang et al., 2024), few-shot prompting (Brown et al., 2020), and RAG (Lewis et al., 2021) are the most common LLM-based approaches to natural language generation. Few-shot prompting requires a fixed set of labeled examples, and RAG requires some supporting set of documents, although these documents may not be of the same form as the final text being generated by the system, e.g. when augmenting with a set of general-knowledge Wikipedia articles to answer factual questions.

Analogical Reasoning in LLMs: Many papers attempt to elicit reasoning in LLMs by various prompting methods (Zhou et al., 2024; Wang et al., 2023). Specific to analogical reasoning, early attempts show by case study that LLMs can do analogical reasoning just like humans (Webb et al., 2023). Recent works explore analogy generation and analogy reasoning with knowledge graphs on LLMs (Yuan et al., 2023; Bhavya et al., 2023, 2022). These works are applied to prepositional knowledge, and rely on large-scale external knowledge bases to store entity relationships to perform analogical reasoning. More closely related to our work are Yu et al. (2023) and Yasunaga et al. (2024), in which they prompt the model to propose analogous problems, whose solutions (also created by the model) are then related by analogy to the original problem in order to inform the final solution. In our work, we retrieve real solutions to analogous real problems from memory, which benefits performance especially in knowledge-intensive domains.

Iterative Pipeline: Numerous recent works based on the RAG paradigm undertake an iterative workflow to generate the response to an input query. Asai et al. (2023) train a separate LM to predict reflection tokens like 'retrieve', 'relevant', 'irrelevant', etc. within the response, and these tokens determine subsequent control flow to accordingly generate or refine subsequent chunks of the response. On the contrary, (Shao et al., 2023) intersperse Generation-Augmented Retrieval (GAR) and RAG to sequentially expand the search query in the memory and obtain relevant information to plug into the final response. Our work comes close to (Shao et al., 2023) where we expand the search query to retrieve relevant info from the procedural memory at the granularity of procedural steps. To facilitate this finer search capabilities, we have organized our procedural memory as objects with a schema that explicitly contains the inputs, goal and the steps.

3 Our Method

We describe the details of our Analogy-Augmented (AAG) system in the sections below. Our system extends the traditional RAG system (Lewis et al., 2021) and introduces three novel modules to perform better on the task of procedural question-answering: 1) **procedural memory store**, a novel representation of memory, designed specifically for procedures; 2) **query rewriting and summarization**: rewrites the input query as a set of questions, answers to which can be procured through analogous procedures in the memory; and 3) **iterative refinement with self-critic**: an LLM-based critic evaluates the candidate response and suggests edits, which are then performed by the 'LLM edit performer' module. A schematic view of our system can be seen in Figure 2. We now describe each of these novel modules in more detail, followed by an end-to-end description of the overall control flow. Please refer to the appendix for detailed prompts used by each module.

3.1 Procedural Memory

We define a procedure as $(x, y, (s_1, \ldots, s_k))$ where:

- x is an input string,
- y is an output string, and
- $(s_1, \ldots, s_k)$ is a sequence of ordered steps, each of which is a string.

The task of procedural generation then is to generate $(s_1, \ldots, s_k)$ given x and y.

This definition is simpler than other conceptions of procedures that use directed acyclic graphs (Georgeff and Lansky, 1986), depending on the text of the steps to define any non-linear control flow. It also allows procedures to be composable: two procedures $(x, y, (s_1, \ldots, s_k)), (y, z, (t_1, \ldots, t_\ell))$ can be composed into a procedure $(x, z, (s_1, \ldots, s_k, t_1, \ldots, t_\ell))$ that produces result z given input x.

This loose formalism allows systems to easily benefit from text embeddings as they see fit, while also relying on their sequential structure. In the custom procedure store for AAG, this is done by representing the procedure in a natural text format appropriate to the type of procedure when creating embeddings, while otherwise maintaining the input, output, and steps as Python `str`, `str`, and `list[str]` types respectively when these procedures are retrieved, generated, etc.

3.2 Query Rewriting

The input to the AAG system is comprised of the goal to be achieved (the procedure output) and the resources we have access to (the procedure input). Using this input query directly to search the memory can yield insufficient supporting information when the necessary information doesn't use words similar to words in the query. Humans solve this problem by linking and assimilating knowledge from relevant past concepts and experiences (Piaget, 1968; Vosniadou and Ortony, 1989), composing a solution by recombining components of past tasks. Most recently, Yasunaga et al. (2024) demonstrated the merits of analogical prompting where the LLM is asked to recall relevant analogous problems to the problem in hand and then leverage them to solve the given problem.

Inspired by these works, we rewrite the input query to the system as a set of questions required to prepare an appropriate procedure for achieving the specified goal. We prompt the LLM to first generate a high-level outline of the procedure that will lead to the specified goal, like in HyDE (Gao et al., 2022), and then based on these high-level steps suggest relevant questions which can be answered with relevant procedures in the memory. We extend the query rewriting prompt used by Ma et al. (2023) for our case and generate a maximum of $N = 4$ queries for a given input query. For example, given the input query: 'create a custom LLM agent that can interact with users in a conversational manner using a specific language style using an LLM, Serp API', the following 4 queries are generated by the query re-writer:

- How to train a language model for a specific language style?
- Best practices for integrating a language model into a conversational interface?
- How to use the Serp API to extract relevant information from search engine results?
- Strategies for optimizing performance of a custom LLM agent in conversational interactions?

3.3 Summarization

Each of the queries received from the query rewriting module is individually used to retrieve from the procedure memory. The challenge, however, is

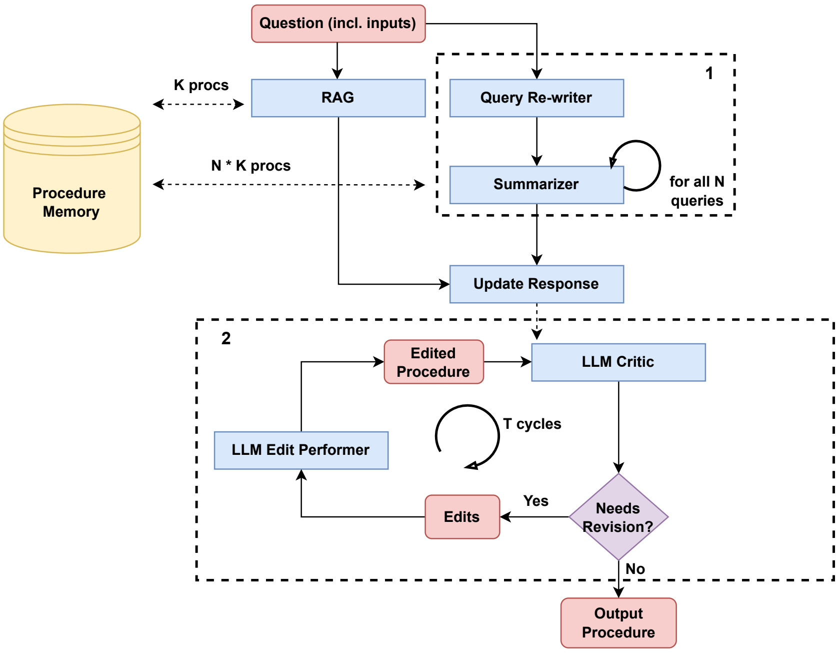


Figure 2: Schematic view of the AAG Architecture. The input and the intermediate outputs are enclosed in red rounded rectangles whereas all the modules involving an LLM have been enclosed in blue rectangles. The boxes 1 and 2 denote our architectural contributions, where 1 shows the query re-writing and the summarization step and the 2 shows the iterative refinement with self-critic step. The procedural memory store holds objects structured as procedures (see sec 3.1). The RAG module and the summarizer module interact with the memory store to retrieve K and N * K procedures relevant to the input query and all the re-written queries (one at a time) respectively.

that the retrieved procedures often contain the required information hidden between irrelevant steps.

Thus, motivated by this observation, the summarization module prompts the LLM with the question and the retrieved procedures, asking it to generate a summary of the retrieved procedures as an answer to the question. This module abstracts out the unnecessary details and includes only the useful information from all the retrieved procedures, thus overcoming the issue of duplicates in retrieved procedures for different queries as well. Each question along with its summary is stacked together as "Q: <Question> [newline] A: <Summary>" to build up high quality context which is used by both the Update Response and the Edit performer module.

3.4 Iterative Refinement with Self-Critic

LLMs have been show to act as powerful self-critics for their own outputs, evaluating the outputs across several dimensions and suggesting feedback crucial to improving their output (Madaan et al., 2023; Saunders et al., 2022). We augment the AAG system with an iterative refinement procedure guided by the same LLM acting as a critic for the output it has generated. As seen in Figure 2, LLM critic takes the output of the Update Response module, evaluates if any edits are required, and suggests them as a bulleted list. In case no edits are required, the critic appends the string *'NO UPDATE REQUIRED'* to its response. The suggested edits are then performed by the LLM with a prompt including the list of edits, context from the summarizer, and the current version of the output procedure steps. This edited version is then re-evaluated by the LLM critic and the cycle continues for a maximum of $T = 3$ cycles. The output at the end of the iterative refinement is the final answer of the AAG system to the input query.

3.5 End-to-End Workflow

On a very high level, the AAG system takes as input a query by the user and outputs a bulleted list of steps that will lead the user to the desired goal. Using the terminologies defined in Section 3.1, the input query is of the format "*output* using *input*", where the user specifies the goal and the resources available to use to reach the goal. Assuming we have a procedural memory instantiated with the procedure objects as outlined in Sections 3.1 and 4.3, the detailed workflow of the AAG system is as follows (refer to Figure 2):

1. The RAG module interacts with the procedural memory using the user input query and retrieves the top $K = 3$ relevant procedures (highest cosine similarity in the embedding space) from the memory.
2. The query re-writer module takes input the user input query, prepares a high-level mind map to solve the query and re-writes into $N = 4$ queries about information required at different steps to arrive at the solution.
3. The summarizer module interacts with the procedural memory and generates a summary answer to each re-written query through their retrieved procedures from the memory.
4. The update response module performs the first update to the RAG response using the context provided by the generated summaries and passes the updated response to the critic.
5. The LLM critic decides if edits are required and suggests them in a bulleted list.
6. The LLM performs edits to its own response using the edits suggested by the critic (same LLM) and the context from the summaries.
7. The edited procedure is then re-evaluated for edits. Repeat step 5 for a maximum of $T = 3$ cycles. The self-refined version is then returned as the output of the system.

4 Experiments

To demonstrate the effectiveness of analogy-augmented generation (AAG) for generating procedures, we apply it to three procedural knowledge datasets. Here the task is to generate the list of steps, given the input text and output text.

RecipeNLG is a corpus of more than 2 million food recipes collected from the internet (Bień et al., 2020). Under our procedural knowledge framework, we construct a procedure by using the recipe title as output, the ingredients as input, and the directions as the procedural steps. For our experiments, we select a random subset of 10,000 recipes from this dataset. From these we select 2,000 test examples and 1,000 validation examples.

LCStep dataset is a self-curated dataset where the goal is the procedure output. The dataset was prepared by scraping the LangChain python docs and filtering out 180 tutorials and guides. For the 180 tutorials/guides, we prompted GPT-4 to extract a list of high-level steps necessary to accomplish the goal and rate those extracted procedures using a list of criteria. Those not matching the criteria were revised both manually and through GPT-4 before the final human filtering (more details in the appendix). We sorted the 276 procedures in LCStep by increasing length and selected the final (longest) 56 examples as the test set, with the preceding 27 examples as a validation set. We selected the longest examples for the test set because we want to observe AAG's ability to leverage simpler procedural knowledge when proposing steps for more complicated tasks.

CHAMP is a dataset of 270 competition-level math problems annotated with concepts, hints, and, step-by-step solutions (Mao et al., 2024). For the procedure generation task, we use the problem statement as the output, the hints and problem category as input, and the step-by-step solutions as the procedure steps. We also append "The answer is *answer*" as a final step in the procedure. We randomly shuffle the problems and choose 54 test examples and 27 validation examples. Random shuffling is important to ensure diversity so that each data split contains problems of all types.

The remaining examples in each of the dataset form the procedural memory for that respective dataset.

4.1 Baselines

We compare our system against widely-used natural language generation approaches that involve prompting an LLM. For our experiments, we use `gpt-3.5-turbo-0125` from OpenAI as the language model for all approaches.

Zero-Shot: We prompt the language model to generate the steps for a procedure, given the procedure's input and output.

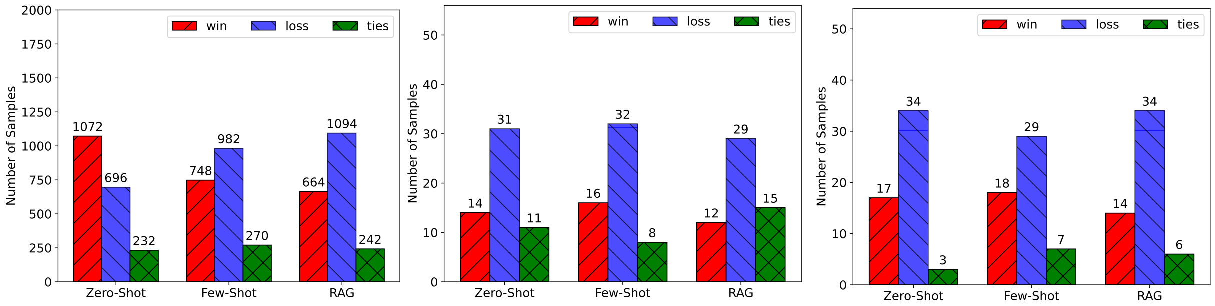
Few-Shot: We sample $k = 3$ random procedures from the training set, and include those in the prompt after the task instructions, which are the same as in the zero-shot case.

RAG: We retrieve $k = 3$ procedures based on the provided input and output, and then use those 3 procedures in the prompt like the few-shot case.

4.2 Evaluation Methodology

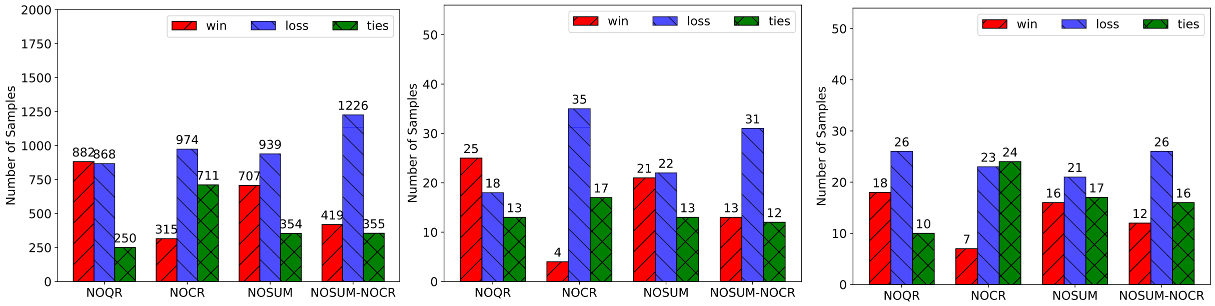
For all our plots, we perform a pairwise evaluation between the proposed AAG system and the concerned method using the LLM. The LLM is prompted to evaluate the two procedures on

- their ability to accomplish the specified goal,
- the clarity and flow of the steps and the level of detail, and



RecipeNLG LCStep CHAMP

Figure 3: The three plots show the pairwise evaluation results (see section 4.2) of the three baselines against AAG. The red bar with forward slash pattern represents the 'wins' i.e. when the concerned method was chosen over AAG, the blue bar with backward slash pattern represents the 'loss', i.e. when the AAG system was chosen over the concerned method and the green bar with cross pattern is when the pairwise evaluator equally likes both the systems' responses. Higher length of the 'loss' bar than the 'win' bar for each method in each dataset highlights that AAG is chosen over its counterpart on all the datasets in a pairwise scenario.



RecipeNLG LCStep CHAMP

Figure 4: The three plots show the pairwise evaluation results (see section 4.2) of the four ablation methods against AAG. The red bar with forward slash pattern represents the 'wins' i.e. when the concerned method was chosen over AAG, the blue bar with backward slash pattern represents the 'loss', i.e. when the AAG system was chosen over the concerned method and the green bar with cross pattern is when the pairwise evaluator equally likes both the systems' responses. Higher length of the 'loss' bar than the 'win' bar for each method in each dataset highlights that AAG is preferred over its ablation counterparts on all the datasets in a pairwise scenario.

- using only the resources specified in user input.

The prompt example can be found in the appendix. Note that the ground-truth steps are not given as input in the prompt, to prevent biasing the LLM towards only matching the keywords in the generated and ground-truth procedure steps. To further control for randomness and de-bias the LLM evaluation towards ordering of procedures in the prompt, we run a total of 10 prompt requests for each sample, with 5 requests having AAG procedure placed first and other 5 having the other method placed first. Moreover, the LLM is set to a temperature of 0.7 and each of the 10 calls is on a different seed to the LLM, but the 10 seeds remain fixed when evaluating all samples. Finally, the result for a sample is calculated using majority voting amongst the results of 10 LLM calls.

In the bar plots in figures 3 and 4, the *'win'* bar denotes the number of samples where the concerned method was chosen over AAG, the *'loss'* bar denotes where AAG was preferred, and the *'ties'* bar represents the number of samples where the LLM selects both the AAG and the other method equally in the 10 LLM calls. It is imperative to note that we report pairwise comparison of methods with respect to AAG, therefore the numbers of one method cannot be directly compared with the numbers of another method. A tall bar for any method means that the LLM is more *confident* that the one system is better than the other in the pair considered, not that the one system is better as compared to other systems not in the pair with shorter bar lengths.

4.3 System Design Choices

The AAG system has no trainable parameters and simply relies on frozen LLMs. The temperature of the OpenAI language model is kept to the default value of 0.7 for all the results and the evaluations. Lack of any training makes the AAG system flexible to be used with even the commercial black-box LLMs like GPT-4 etc. We use the *all-mpnet-base-v2* model from sentence-transformers (Reimers and Gurevych, 2019) as the embedding model for the objects in the procedural memory Weaviate store, creating dense 768 dimensional embeddings for each procedure object. All the three data fields of a procedure (input, output, and steps) are embedded together in one embedding.

4.4 Results

Figure 3 shows the result of pairwise comparison of AAG with each of the three baselines discussed in section 4.1, i.e. Zero-Shot, Few-Shot and RAG. The three bars for each method describe the 'win', 'loss' and 'ties' as explained in section 4.2. Higher value of 'loss' bar for a given method and dataset means that AAG was preferred over that method on that dataset. From figure 3, it can be clearly observed that the AAG outperforms its counterpart for all methods on all datasets by a significant margin, highlighting the efficacy of our method.

However, on the RecipeNLG dataset, it can be seen that the Zero-Shot method performs better than the AAG system. This can be attributed to the fact that the LLM has already seen recipes and information related to them in its training dataset. Due to this, it performs best when no context is provided since any form of context is going to introduce more noise for the LLM. Although AAG performs worse than Zero-Shot on RecipeNLG, it still surpasses the performance of Few-Shot and RAG which emphasizes that AAG generates a better context from the procedural memory and subsequently outputs a better response to the user query. For LCStep and CHAMP datasets, which are relatively unseen and hard for the LLM, AAG outperforms the Zero-Shot method by considerable margin, thereby bolstering our claim that AAG outshines on tasks outside the domain of the LLM. Refer to figures 5 and 6 for a qualitative comparison of the generated procedure.

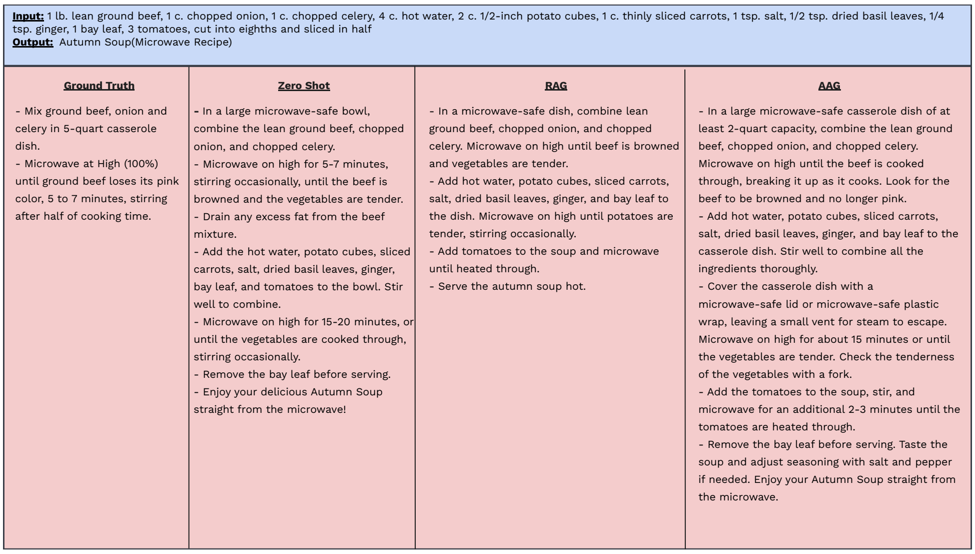
4.5 Ablation Study

We perform a rigorous ablation study of our proposed AAG architecture. We experiment with removing the three architectural contributions in all permutations, i.e. the query re-writer, summarizer and the self-critic guided iterative refinement. We summarize each of the ablation method below:

- **AAG-NOSUM:** This represents the AAG system but without the summarizer. Instead of creating summary answers for each re-written query, we simply concatenate all the retrieved procedures for all the queries, removing the duplicates and form one big context with all the procedures. This context is used in place of the summary based context in the AAG pipeline, keeping everything else same.
- **AAG-NOCR:** This represents the AAG system but without the self-critic guided iterative refinement. Every other pipeline component still remains. The output of the Update Response module is considered as the final output of the system.
- **AAG-NOSUM-NOCR:** This represents the AAG system but without both summarizer and the self-critic guided iterative refinement. The Update Response module receives the context as in AAG-NOSUM and its output is considered as the final output of the system.
- **AAG-NOQR:** This represents the AAG system but without the query re-writing module. Since there are no re-written queries, there is no summarizer and Update Response module in this method. The response from RAG is directly fed into self-critic based iterative refinement, whose output is considered as system's final output.

Results: Figure 4 demonstrates the results for the pairwise comparison between AAG and each of the ablation methods above on all the three datasets. From the figure, it can be clearly observed that AAG performs better than AAG-NOCR with a large margin on all the three datasets, highlighting the importance of using the self-critic based iterative refinement for better performance. Moreover, AAG-NOSUM bars prove that removing the summarizer module from the AAG pipeline hampers the performance, thus performing worse than the AAG. These findings, along with the bars for AAG-NOSUM-NOCR, further strengthen the claim that the combination of summarizer and the critic is crucial to high performance of the AAG system. In absence of any of these components, there is a reduction in performance.

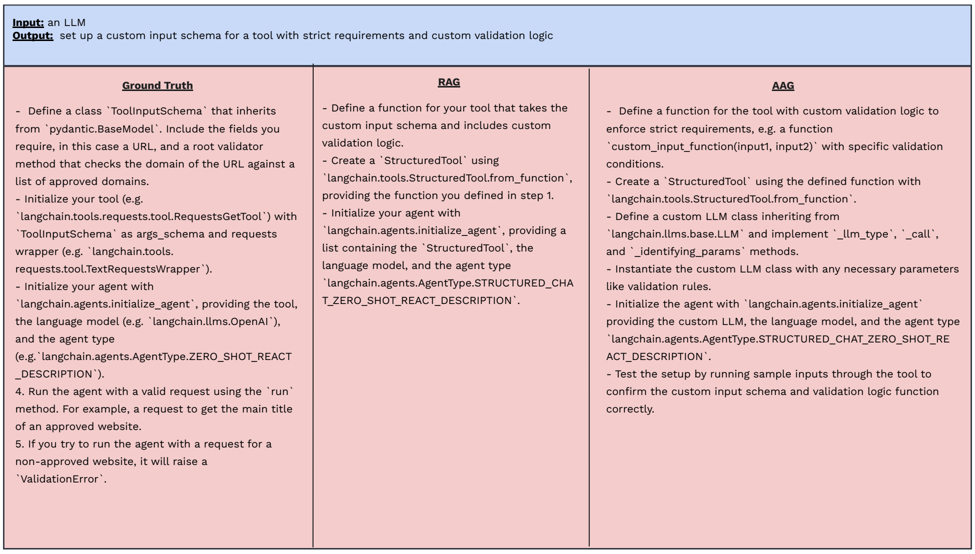
However, the case of AAG-NOQR is interesting as it performs at par with AAG on RecipeNLG and LCStep datasets. Since the numbers for these experiments in the figure 4 are very close, the conclusion is ambiguous. On the other hand, the addition of the query re-writing module provides significant advantage to AAG on the CHAMP dataset.



Input: 1 lb. lean ground beef, 1 c. chopped onion, 1 c. chopped celery, 4 c. hot water, 2 c. 1/2-inch potato cubes, 1 c. thinly sliced carrots, 1 tsp. salt, 1/2 tsp. dried basil leaves, 1/4 tsp. ginger, 1 bay leaf, 3 tomatoes, cut into eighths and sliced in half
Output: Autumn Soup(Microwave Recipe)

Ground Truth	**Zero Shot**	**RAG**	**AAG**
- Mix ground beef, onion and celery in 5-quart casserole dish. - Microwave at High (100%) until ground beef loses its pink color, 5 to 7 minutes, stirring after half of cooking time.	- In a large microwave-safe bowl, combine the lean ground beef, chopped onion, and chopped celery. - Microwave on high for 5-7 minutes, stirring occasionally, until the beef is browned and the vegetables are tender. - Drain any excess fat from the beef mixture. - Add the hot water, potato cubes, sliced carrots, salt, dried basil leaves, ginger, bay leaf, and tomatoes to the bowl. Stir well to combine. - Microwave on high for 15-20 minutes, or until the vegetables are cooked through, stirring occasionally. - Remove the bay leaf before serving. - Enjoy your delicious Autumn Soup straight from the microwave!	- In a microwave-safe dish, combine lean ground beef, chopped onion, and chopped celery. Microwave on high until beef is browned and vegetables are tender. - Add hot water, potato cubes, sliced carrots, salt, dried basil leaves, ginger, and bay leaf to the dish. Microwave on high until potatoes are tender, stirring occasionally. - Add tomatoes to the soup and microwave until heated through. - Serve the autumn soup hot.	- In a large microwave-safe casserole dish of at least 2-quart capacity, combine the lean ground beef, chopped onion, and chopped celery. Microwave on high until the beef is cooked through, breaking it up as it cooks. Look for the beef to be browned and no longer pink. - Add hot water, potato cubes, sliced carrots, salt, dried basil leaves, ginger, and bay leaf to the casserole dish. Stir well to combine all the ingredients thoroughly. - Cover the casserole dish with a microwave-safe lid or microwave-safe plastic wrap, leaving a small vent for steam to escape. Microwave on high for about 15 minutes or until the vegetables are tender. Check the tenderness of the vegetables with a fork. - Add the tomatoes to the soup, stir, and microwave for an additional 2-3 minutes until the tomatoes are heated through. - Remove the bay leaf before serving. Taste the soup and adjust seasoning with salt and pepper if needed. Enjoy your Autumn Soup straight from the microwave.

Figure 5: Qualitative example showing comparison between the ground-truth steps and the steps generated by Zero Shot, RAG and the AAG system on the RecipeNLG dataset. The AAG procedure clearly includes more details on how to perform each step and is easier to follow, for example in checking the tenderness of vegetables and how to serve. The baselines on the other hand suggest steps but on a coarser granularity.



Input: an LLM
Output: set up a custom input schema for a tool with strict requirements and custom validation logic

Ground Truth	**RAG**	**AAG**
- Define a class `ToolInputSchema` that inherits from `pydantic.BaseModel`. Include the fields you require, in this case a URL, and a root validator method that checks the domain of the URL against a list of approved domains. - Initialize your tool (e.g. `langchain.tools.requests.tool.RequestsGetTool`) with `ToolInputSchema` as args_schema and requests wrapper (e.g. `langchain.tools. requests.tool.TextRequestsWrapper`). - Initialize your agent with `langchain.agents.initialize_agent`, providing the tool, the language model (e.g. `langchain.llms.OpenAI`), and the agent type (e.g.`langchain.agents.AgentType.ZERO_SHOT_REACT _DESCRIPTION`). 4. Run the agent with a valid request using the `run` method. For example, a request to get the main title of an approved website. 5. If you try to run the agent with a request for a non-approved website, it will raise a `ValidationError`.	- Define a function for your tool that takes the custom input schema and includes custom validation logic. - Create a `StructuredTool` using `langchain.tools.StructuredTool.from_function`, providing the function you defined in step 1. - Initialize your agent with `langchain.agents.initialize_agent`, providing a list containing the `StructuredTool`, the language model, and the agent type `langchain.agents.AgentType.STRUCTURED_CHA T_ZERO_SHOT_REACT_DESCRIPTION`.	- Define a function for the tool with custom validation logic to enforce strict requirements, e.g. a function `custom_input_function(input1, input2)` with specific validation conditions. - Create a `StructuredTool` using the defined function with `langchain.tools.StructuredTool.from_function`. - Define a custom LLM class inheriting from `langchain.llms.base.LLM` and implement `_llm_type`, `_call`, and `_identifying_params` methods. - Instantiate the custom LLM class with any necessary parameters like validation rules. - Initialize the agent with `langchain.agents.initialize_agent` providing the custom LLM, the language model, and the agent type `langchain.agents.AgentType.STRUCTURED_CHAT_ZERO_SHOT_RE ACT_DESCRIPTION`. - Test the setup by running sample inputs through the tool to confirm the custom input schema and validation logic function correctly.

Figure 6: Qualitative example showing comparison between the ground-truth steps and the steps generated by RAG and the AAG system on the LCStep dataset. The AAG procedure includes more details like function names and signatures and good practices like testing the built systems, in contrast to the RAG system.

5 Conclusion

We have introduced a simple framework for operating on procedural knowledge, created a novel LLM system that leverages this framework, and demonstrated its increased performance over standard RAG, especially when generating procedures in domains unfamiliar to the LLM. The case where AAG was shown to harm performance over the baseline was for RecipeNLG, a domain extremely familiar to general-purpose LLMs like GPT-3.5. The system performed especially well on the LCStep dataset we collected, demonstrating that AAG can better augment a frozen LLM with the knowledge needed to answer questions on unseen domains. These results show that structured procedural memory, query rewriting, response summarization, and iterative critique/refinement all bolster the performance of our system on procedure generation tasks. Furthermore, these results suggest that our theoretical framework provides a basis for more effectively solving high-level planning tasks in varied domains.

Limitations and Future Work

This framework deliberately ignores procedure structure that is non-linear, meaning that a non-linear procedure must express its non-linearity in the text of the steps. This is a reasonable assumption for many tasks performed by humans, but not for computer algorithms.

Our AAG method as presented here does not yet leverage all of the structure afforded by our procedure definition. In future work we plan to more tightly couple our system with the formal structure we have proposed, by leveraging the composable nature of our procedure definition to perform finer-grained search and reasoning over sub-procedures.

Ethical Considerations

The use of automated systems to generate and execute plans can and will result in unintended consequences, whether humans are in the loop or on the loop (Leins and Kaspersen, 2021). In addition, automating actions generally done by humans (here planning and reasoning) has the social effect of anthropomorphizing these systems, leading to a gradual shift in accountability from human stakeholders onto the systems themselves (Weidinger et al., 2021).

The domains which we have applied our system to in this work do not constitute immediate risks. For LCStep, the generated steps cannot easily be converted into code, although this could become increasingly easy as planning-based code generation continues to improve (Jiang et al., 2024). Mathematical problem solving poses no real-world risks.

Thanks to Google's AI Overviews feature, there has already been at least one instance of a generative AI system producing a food recipe that would be harmful if followed (Robison, 2024).[1] While it might be obvious to most that it's a bad idea to add glue to food, it's clear that automated recipe generation could result in worse than a botched meal. No matter how many safeguards are added, systems that generate content ought to appropriately inform users of the risks of relying on the content without verification. We hope that the general public continues to be made aware of humorous mistakes like this so that people learn to approach AI content with healthy skepticism.

---

[1] See `https://web.archive.org/web/20240222195427/https://stayhealthyhacks.com/can-you-freeze-off-a-mole/` for another humorous example of incorrect food-related content generated by AI.

References

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *Preprint*, arXiv:2310.11511.

Bhavya Bhavya, Jinjun Xiong, and Chengxiang Zhai. 2022. Analogy generation by prompting large language models: A case study of instructgpt. *arXiv preprint arXiv:2210.04186*.

Bhavya Bhavya, Jinjun Xiong, and Chengxiang Zhai. 2023. Cam: A large language model-based creative analogy mining framework. In *Proceedings of the ACM Web Conference 2023*, pages 3903–3914.

Michał Bień, Michał Gilski, Martyna Maciejewska, Wojciech Taisner, Dawid Wisniewski, and Agnieszka Lawrynowicz. 2020. RecipeNLG: A cooking recipes dataset for semi-structured text generation. In *Proceedings of the 13th International Conference on Natural Language Generation*. Association for Computational Linguistics.

Faeze Brahman, Chandra Bhagavatula, Valentina Pyatkin, Jena D. Hwang, Xiang Lorraine Li, Hirona J. Arai, Soumya Sanyal, Keisuke Sakaguchi, Xiang Ren, and Yejin Choi. 2023. Plasma: Making small language models better procedural knowledge models for (counterfactual) planning. *Preprint*, arXiv:2305.19472.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. 2024. Efficient prompting methods for large language models: A survey. *arXiv preprint arXiv: 2404.01077*.

Harrison Chase and contributors. 2022. Langchain. Version 0.0.249.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise zero-shot dense retrieval without relevance labels. *Preprint*, arXiv:2212.10496.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2024. Retrieval-augmented generation for large language models: A survey. *Preprint*, arXiv:2312.10997.

M.P. Georgeff and A.L. Lansky. 1986. Procedural knowledge. *Proceedings of the IEEE*, 74(10):1383–1398.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR.

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2023. Large language models struggle to learn long-tail knowledge. *Preprint*, arXiv:2211.08411.

Kobi Leins and Anja Kaspersen. 2021. Seven myths of using the term “human on the loop”: “Just what do you think you are doing, Dave?”. *Carnegie Council for Ethics in International Affairs*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *Preprint*, arXiv:2305.14283.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Yujun Mao, Yoon Kim, and Yilun Zhou. 2024. Champ: A competition-level dataset for fine-grained analyses of llms’ mathematical reasoning capabilities. *Preprint*, arXiv:2401.06961.

John-Dennis Parsons and Jim Davies. 2022. The neural correlates of analogy component processes. *Cognitive Science*, 46(3):e13116.

Carlotta Pavese. 2022. Knowledge How. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*, Fall 2022 edition. Metaphysics Research Lab, Stanford University.

Jean Piaget. 1968. *Genetic Epistemology*. Columbia University Press.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Kylie Robison. 2024. Google AI Overview: Hallucinations and Glue in Pizza. *The Verge*.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. *Preprint*, arXiv:2206.05802.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Weihao Tan, Ziluo Ding, Wentao Zhang, Boyu Li, Bohan Zhou, Junpeng Yue, Haochong Xia, Jiechuan Jiang, Longtao Zheng, Xinrun Xu, Yifei Bi, Pengjie Gu, Xinrun Wang, Börje F. Karlsson, Bo An, and Zongqing Lu. 2024. Towards general computer control: A multimodal agent for red dead redemption ii as a case study. *arXiv preprint arXiv: 2403.03186*.

Stella Vosniadou and Andrew Ortony. 1989. *Similarity and analogical reasoning*. Cambridge University Press.

Boshi Wang, Xiang Yue, and Huan Sun. 2023. Can ChatGPT defend its belief in truth? evaluating LLM reasoning via debate. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11865–11881, Singapore. Association for Computational Linguistics.

Lei Wang, Chen Ma, Xin Feng, et al. 2024a. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18:186345.

Ruoyu Wang, Zhipeng Yang, Zinan Zhao, Xinyan Tong, Zhi Hong, and Kun Qian. 2024b. Llm-based robot task planning with exceptional handling for general purpose service robots. *arXiv preprint arXiv: 2405.15646*.

Taylor Webb, Keith J Holyoak, and Hongjing Lu. 2023. Emergent analogical reasoning in large language models. *Nature Human Behaviour*, 7(9):1526–1541.

Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, Zac Kenton, Sasha Brown, Will Hawkins, Tom Stepleton, Courtney Biles, Abeba Birhane, Julia

Haas, Laura Rimell, Lisa Anne Hendricks, William Isaac, Sean Legassick, Geoffrey Irving, and Iason Gabriel. 2021. Ethical and social risks of harm from language models. *arXiv preprint arXiv: 2112.04359*.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H. Chi, and Denny Zhou. 2024. Large language models as analogical reasoners. *Preprint*, arXiv:2310.01714.

Junchi Yu, Ran He, and Rex Ying. 2023. Thought propagation: An analogical approach to complex reasoning with large language models. *arXiv preprint arXiv: 2310.03965*.

Siyu Yuan, Jiangjie Chen, Changzhi Sun, Jiaqing Liang, Yanghua Xiao, and Deqing Yang. 2023. Analogykb: Unlocking analogical reasoning of language models with a million-scale knowledge base. *Preprint*, arXiv:2305.05994.

Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2023. Siren's song in the ai ocean: A survey on hallucination in large language models. *Preprint*, arXiv:2309.01219.

Pei Zhou, Jay Pujara, Xiang Ren, Xinyun Chen, Heng-Tze Cheng, Quoc V. Le, Ed H. Chi, Denny Zhou, Swaroop Mishra, and Huaixiu Steven Zheng. 2024. Self-discover: Large language models self-compose reasoning structures. *arXiv preprint arXiv: 2402.03620*.

A Appendix

A.1 LCStep Dataset Creation

LCStep contains three sets of documents: API reference, conceptual documentation, and procedures. In this paper, we only used the procedures, but we plan to leverage the supporting material in future systems. See Figure 7 for a diagram of the process of generating the LCStep data.

As Langchain was unstable and changing quickly at the time we collected the data, we made sure to capture the state of the code and documentation at version 0.0.249.

A.1.1 API Reference

We generate the API reference material from the source files in the LangChain GitHub repository using Sphinx. These files contain descriptions of all APIs in the Python package, including call signatures and argument descriptions. These files do not contain any usage examples or high-level explanation.

A.1.2 Conceptual Documentation and Procedures

We collected these resources by scraping the Langchain Python docs when v0.0.249 was the version live on the website. We manually filter out topic pages and stubs, leaving 228 documents. We then manually classified these into around 30 documents of conceptual documentation, and around 180 documents containing tutorials/guides.

For the 180 tutorials/guides, we prompted GPT-4 (see Listing **??**) to extract a list of high-level steps necessary to accomplish the goal. We then prompted GPT-4 (see Listing 2) to rate those extracted procedures using a list of criteria. We found that this caught many mistakes where GPT-4 did not follow all the stated instructions. In those cases, we had the model revise the steps to meet the requirements, and then we manually checked the revised versions.

Listing 1: prompt for extracting procedures

```
You are helping convert tutorials
for a Python library into step-by-
step procedures. The tutorials
describe how to perform various
tasks with code and examples that
use relevant APIs in the library.
Your task is to simplify a tutorial
into two parts: a description of the
 specific goal, and a minimal set of
 the necessary instructions. Write "
Goal: {description}", and use
ordered numbering for the steps.

The goal should be a complete and
specific task rather than a general
approach, e.g. "integrate a
calculator into a chat model"
instead of "integrate custom
functions with OpenAI functions
agent". You usually should not refer
 to specific APIs in the goal
description; focus on describing
functionality rather than a
particular approach.

Each instruction is a logically
distinct action, and explicitly
refers to the library methods used
for that action. Remove all
unnecessary details, code,
parameters, and example usage; keep
only the information strictly
necessary to accomplish the
procedure. A side note with
auxiliary info may be provided after
 the steps, but this note must not
contain information relevant to
accomplishing the described goal.

Do not mention importing the
required modules as a separate step.
 Instead, always use the full path
```

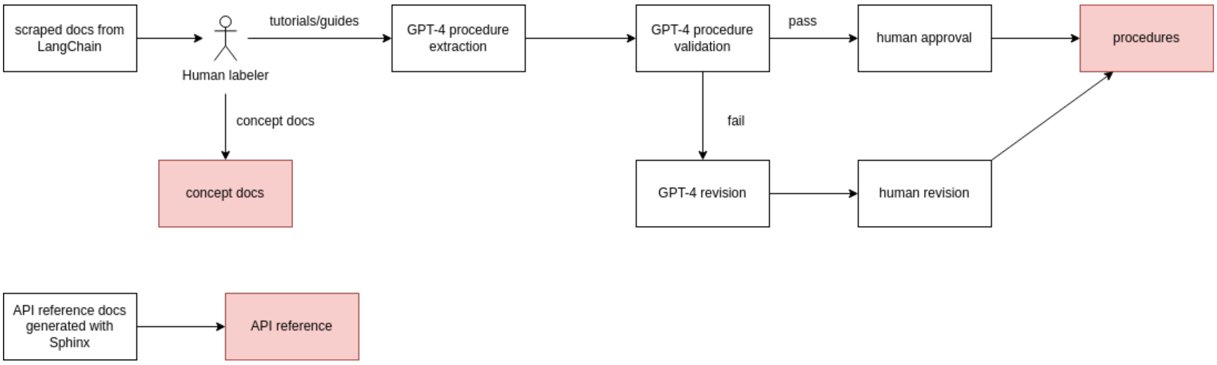


Figure 7: The workflow used to generate the LCStep dataset.

```
    to APIs (e.g. `langchain.chat_models
    .ChatOpenAI`) when referring to them
     in the steps. Do not mention
    specific API parameters unless they'
    re important to understanding how
    the procedure fits together, e.g.
    don't include things like the
    verbosity or the model temperature
    unless those matter for
    accomplishing the stated goal.

    If the raw text actually contains
    more than one logically distinct
    procedure, you may then output
    multiple sets of goals and
    instructions for each different
    procedure (delimited by NEW
    PROCEDURE), but this should only be
    done if the goals are fundamentally
    different. Don't create separate
    procedures if one procedure could be
     expressed in a side note. These
    separate procedures must not make
    reference to one another, as in ”
    follow steps 2-4 from the previous
    procedure”.
```

Listing 2: prompt for checking procedures

```
You are judging the output of another
language model on a particular task. The
 original instructions are provided
below:

BEGIN INSTRUCTIONS
{instructions from Listing 1}
END INSTRUCTIONS

Below is an example of how you should
judge the text. Please list whether the
provided text passes or fails each
requirement in the list:

- [PASS] format is as expected
- [PASS] the goal is a specific task
rather than general approach
- [PASS] the resources line includes
only the required external APIs or
models that must be accessible for the
steps to work
- [PASS] the steps fully achieve the
task described in the goal
- [PASS] does not unnecessarily refer to
 APIs in the goal description
- [PASS] instructions are logically
distinct
- [FAIL] no unnecessary details, code,
or parameters like verbosity or
temperature, etc. (INSERT REASON FOR
FAIL)
- [PASS] side note is relevant if
present
- [PASS] all API references use the full
 path
- [PASS] if there are multiple
procedures, they're describing two
independent processes, not two sub
processes that succeed one another
- [PASS] if there are multiple
procedures, they're describing two
different goals, not the same process
with a simple variation

If any requirement fails, correct the
procedure by rewriting it in a manner
that passes all checks. If either of the
 last two checks fail, join the
procedures together and specify any
necessary variation in the instructions
or a side note.
```

A.2 Full Example of AAG with Prompts

Here we'll take an example from the LCStep test set. See the attached code for the prompts adapted to the other datasets. The output is `set up a custom input schema for a tool with strict requirements and custom validation logic,` and the input is an `LLM`. The reference steps are the following:

Listing 3: reference steps for this example

```
    1. Define a class `ToolInputSchema`
    that inherits from `pydantic.
    BaseModel`. Include the fields you
    require, in this case a URL, and a
    root validator method that checks
    the domain of the URL against a list
     of approved domains.
    2. Initialize your tool (e.g. `
    langchain.tools.requests.tool.
    RequestsGetTool`) with the `
    ToolInputSchema` as the args_schema
    and a requests wrapper (e.g. `
    langchain.tools.requests.tool.
    TextRequestsWrapper`).
    3. Initialize your agent with `
    langchain.agents.initialize_agent`,
    providing the tool, the language
    model (e.g. `langchain.llms.OpenAI`)
    , and the agent type (e.g. `
    langchain.agents.AgentType.
    ZERO_SHOT_REACT_DESCRIPTION`).
    4. Run the agent with a valid
    request using the `run` method on
    your agent. For example, a request
    to get the main title of an approved
     website.
    5. If you try to run the agent with
    a request for a non-approved website
    , it will raise a `ValidationError`.
```

Given the input and output strings, the first prompt generates the queries:

Listing 4: query rewriting prompt

```
Please output high-level steps to
complete the task below.

Then, given this high-level solution,
think carefully step by step and provide
 4 search engine queries for knowledge
that you need to refine the solution to
the question.

The output should be 'steps:' followed
by a bulleted list with elements
starting with '- ', and then 'queries:'
followed by another bulleted list.

---

I want to create set up a custom input
schema for a tool with strict
requirements and custom validation logic
 using these resources: an LLM.
```

The output is the following:

Listing 5: query rewriting output

```
  steps:
  - Understand the requirements and
  constraints for the tool's input
  schema.
  - Identify the necessary fields and
  validation rules based on the LLM.
  - Implement the custom input schema
  with the strict requirements and
  validation logic.
  - Test the custom input schema to
  ensure it functions correctly.

  queries:
  - How to define custom input schema in
   [tool name]?
  - Best practices for implementing
  custom validation logic in [tool name
  ]?
  - How to integrate an LLM into custom
  input schema?
  - Common pitfalls to avoid when
  setting up a custom input schema in [
  tool name]?
```

Each of those queries is then used to retrieve 3 procedures from the procedure store. Those are listed here:

Listing 6: retrieved procedure titles for each query

```
- How to define custom input schema in [
tool name]?
  - set up a tool that requires multiple
   inputs for an agent
  - build a structured tool chat agent
  capable of using multi-input tools and
   handling memory
  - set up a tool that requires a single
   string input for an agent
- Best practices for implementing custom
 validation logic in [tool name]?
  - create custom tools for a LangChain
  agent and handle tool errors
  - create a prompt template with
  variable validation
  - set up a tool that requires multiple
   inputs for an agent
- How to integrate an LLM into custom
input schema?
  - set up a tool that requires multiple
   inputs for an agent
  - create a custom LLM that returns the
   first N characters of the input
  - Create a custom MRKL agent with a
  custom LLMChain that takes multiple
  inputs, capable of answering queries
  in a specific language
- Common pitfalls to avoid when setting
up a custom input schema in [tool name]?
  - build a structured tool chat agent
  capable of using multi-input tools and
   handling memory
  - set up a tool that requires multiple
   inputs for an agent
  - create custom tools for a LangChain
  agent and handle tool errors
```

Then another prompt summarizes the knowledge. For the first generated query (`How to define custom input schema in [tool name]?`) that looked like this:

Listing 7: summarization prompt

```
[Instruction]
You are a human expert whose job is to
summarise the retrieved information
below to answer the question. Please
include the information only from the
provided knowledge and make sure that
the summary is complete, short and
concise. Avoid introductory and closing
lines at the start and end of your
response. Don't directly refer to the
titles in the provided knowledge when
generating the summary.

```

```
---

[BEGIN QUESTION]
How to define custom input schema in [
tool name]?
[END QUESTION]

[BEGIN INFORMATION]
DOCUMENTATION 'set up a tool that
requires multiple inputs for an agent'
using an LLM:

1. Define a function for your tool that
takes multiple inputs.
2. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined.
3. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the `StructuredTool`,
the language model, and the agent type `
langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.

DOCUMENTATION 'build a structured tool
chat agent capable of using multi-input
tools and handling memory' using an LLM,
 PlayWrightBrowserToolkit:

1. Initialize a PlayWrightBrowserToolkit
 using `langchain.agents.agent_toolkits.
PlayWrightBrowserToolkit.from_browser`
with an asynchronous browser created
using `langchain.tools.playwright.utils.
create_async_playwright_browser`.
2. Get the tools from the browser
toolkit using the `get_tools` method.
3. Initialize your language model using
`langchain.chat_models.ChatOpenAI`.
4. Initialize your agent using `
langchain.agents.initialize_agent`,
providing the tools, the language model,
 and the agent type `langchain.agents.
AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.
5. Execute the `arun` method on your
agent with a string as the input.

DOCUMENTATION 'set up a tool that
requires a single string input for an
agent' using an LLM:

1. Define a function for your tool that
takes a single string input and parses
it into multiple inputs for the actual
operation.
2. Create a `Tool` using `langchain.
agents.Tool`, providing the tool name,
the function you defined, and a
description.
3. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the `Tool`, the language
 model, and the agent type `langchain.
agents.AgentType.
ZERO_SHOT_REACT_DESCRIPTION`.
[END INFORMATION]
```

Then the exact prompt from our RAG implementation is used to create an initial candidate set of steps:

Listing 8: RAG prompt

```
Please generate high-level steps to
accomplish the specified goal using the
LangChain Python library. Don't include
code, extraneous commentary, or examples
, but do refer to the specific LangChain
 APIs (or other APIs) used in each step.
 Don't produce any text other than the
list of steps. Use any of the provided
reference documentation to answer the
question. Think carefully about your
steps and enclose any steps you are
uncertain about in the format like '[[ <
step> ]]'

---

DOCUMENTATION 'set up a tool that
requires multiple inputs for an agent'
using an LLM:

1. Define a function for your tool that
takes multiple inputs.
2. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined.
3. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the `StructuredTool`,
the language model, and the agent type `
langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.

DOCUMENTATION 'set up a tool that
requires a single string input for an
agent' using an LLM:

1. Define a function for your tool that
takes a single string input and parses
it into multiple inputs for the actual
operation.
2. Create a `Tool` using `langchain.
agents.Tool`, providing the tool name,
the function you defined, and a
description.
3. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the `Tool`, the language
 model, and the agent type `langchain.
agents.AgentType.
ZERO_SHOT_REACT_DESCRIPTION`.

DOCUMENTATION 'create a custom LLM that
returns the first N characters of the
input' using LangChain's base LLM class:

1. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
2. Implement the `_llm_type` property in
 the custom class to return a string
that identifies the type of LLM.
3. Implement the `_call` method in the
custom class to accept a string prompt
```

```
and optional stop words, and return the
first N characters of the prompt.
4. Implement the `_identifying_params`
property in the custom class to return a
 dictionary with "n" as the key and the
number of characters to return as the
value.
5. Instantiate the custom LLM class,
providing the number of characters to
return as an argument.

Please generate a list of instructions
to accomplish 'set up a custom input
schema for a tool with strict
requirements and custom validation logic
' using the procedures above. Create and
 use these resources in your response:
an LLM.
```

The resulting candidate is listed below:

Listing 9: RAG result

```
1. DOCUMENTATION 'set up a custom input
schema for a tool with strict
requirements and custom validation logic
' using an LLM:
2.
3. Define a function for your tool that
takes the desired multiple inputs and
includes custom validation logic to
enforce strict requirements.
4. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined.
5. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
6. Implement the `_llm_type` property in
 the custom class to return a string
that identifies the type of LLM.
7. Implement the `_call` method in the
custom class to accept the structured
inputs from the tool function and apply
custom validation logic.
8. Implement the `_identifying_params`
property in the custom class to return a
 dictionary with the required input keys
 and their corresponding validation
criteria.
9. Instantiate the custom LLM class,
providing any necessary parameters such
as validation rules.
10. Initialize your agent with `
langchain.agents.initialize_agent`,
providing a list containing the custom
LLM, the language model, and the agent
type `langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.
```

Now we insert one of the generated summaries and ask the LLM to update the candidate according to the new knowledge:

Listing 10: update prompt

```
Please update the provided high-level
steps to accomplish the specified goal
using the LangChain Python library.
Focus more on improving the uncertain
steps enclosed in '[[]]'. Don't include
code, extraneous commentary, or examples
, but do refer to the specific LangChain
 APIs (or other APIs) used in each step.
 Don't produce any text other than the
list of steps. Use any of the provided
reference answers to relevant questions
on the steps to achieve the specified
goal.

---

[BEGIN KNOWLEDGE]
Q: How to define custom input schema in
[tool name]?
A: To define a custom input schema in [
tool name], you can follow the steps
outlined in the documentation. For a
tool that requires multiple inputs,
define a function that takes multiple
inputs, create a `StructuredTool` using
`langchain.tools.StructuredTool.
from_function`, and initialize the agent
 with the `StructuredTool`, the language
 model, and the agent type `langchain.
agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`. If the tool requires a single string
input, define a function that parses the
 string into multiple inputs, create a `
Tool` using `langchain.agents.Tool`, and
 initialize the agent with the `Tool`,
the language model, and the agent type `
langchain.agents.AgentType.
ZERO_SHOT_REACT_DESCRIPTION`.

Q: Best practices for implementing
custom validation logic in [tool name]?
A: To implement custom validation logic
in [tool name], you can follow these
best practices:

1. Define a function that takes a single
 string as input and returns a string as
 output for the tool.
2. Create a new tool using `langchain.
tools.Tool.from_function`, specifying
the function, a unique name, and a
description.
3. Handle tool errors by defining a
function that takes a `ToolException` as
 a parameter and returns a string, then
set the `handle_tool_error` parameter of
 your tool to this error handling
function.
4. Initialize your agent with the
necessary tools, language model, and
agent type according to the specific
requirements.

Q: How to integrate an LLM into custom
input schema?
A: To integrate a custom LLM into a
custom input schema, you can follow
these steps:

1. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
2. Implement properties and methods in
```

```
the custom class to handle input and
return the desired output.
3. Instantiate the custom LLM class with
 the necessary parameters.
4. Create a `StructuredTool` using the
defined function.
5. Initialize your agent with the custom
 `StructuredTool`, the language model,
and the appropriate agent type.

Q: Common pitfalls to avoid when setting
 up a custom input schema in [tool name
]?
A: When setting up a custom input schema
 in [tool name], it is important to
avoid common pitfalls such as:

- Ensuring proper initialization of
PlayWrightBrowserToolkit and language
model for structured chat agents.
- Defining functions correctly for tools
 that require multiple inputs.
- Handling tool errors by defining error
 handling functions for tools created
using SerpAPIWrapper and OpenAI.
[END KNOWLEDGE]

[BEGIN STEPS]
1. DOCUMENTATION 'set up a custom input
schema for a tool with strict
requirements and custom validation logic
' using an LLM:
2.
3. Define a function for your tool that
takes the desired multiple inputs and
includes custom validation logic to
enforce strict requirements.
4. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined.
5. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
6. Implement the `_llm_type` property in
 the custom class to return a string
that identifies the type of LLM.
7. Implement the `_call` method in the
custom class to accept the structured
inputs from the tool function and apply
custom validation logic.
8. Implement the `_identifying_params`
property in the custom class to return a
 dictionary with the required input keys
 and their corresponding validation
criteria.
9. Instantiate the custom LLM class,
providing any necessary parameters such
as validation rules.
10. Initialize your agent with `
langchain.agents.initialize_agent`,
providing a list containing the custom
LLM, the language model, and the agent
type `langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.
[END STEPS]

Please update the list of steps to
accomplish 'set up a custom input schema
 for a tool with strict requirements and
 custom validation logic' using the
knowledge above. Create and use these
resources in your response: an LLM.
Please output only the updated steps.
Your response should start with '1.'.
The final response should not contain
direct references to the knowledge above
.
```

Once that update is received, it is validated by the following prompt:

Listing 11: validation prompt

```
[INSTRUCTION]
You are a human critic whose job is to
validate the provided procedure, propose
 the changes to be made and evaluate if
the steps lead to the mentioned user
goal or not. You should also assess if
the quality of the steps can be improved
 by modifying the flow of the steps or
adding more details to make it more
clear and doable.

Furthermore, it is very important for
the procedure to use all the mentioned
input resources. Carefully judge if the
procedure uses all the resources and
point out in your response if it misses
something.

You should always suggest only your
edits in a bulleted list. If there are
no edits to be made, please only respond
 'NO UPDATE REQUIRED'. You are required
to strictly follow the mentioned output
format.

---

[USER GOAL]
set up a custom input schema for a tool
with strict requirements and custom
validation logic

[INPUT RESOURCES]
an LLM

[BEGIN PROCEDURE]
1. Define a function for your tool that
takes the desired multiple inputs and
includes custom validation logic to
enforce strict requirements.
2. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined.
3. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
4. Implement the `_llm_type` property in
 the custom class to return a string
that identifies the type of LLM.
5. Implement the `_call` method in the
custom class to accept the structured
inputs from the tool function and apply
custom validation logic.
6. Implement the `_identifying_params`
property in the custom class to return a
 dictionary with the required input keys
 and their corresponding validation
criteria.
7. Instantiate the custom LLM class,
providing any necessary parameters such
as validation rules.
8. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the custom LLM, the
language model, and the agent type `
langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.
[END PROCEDURE]
```

The validator's response was:

Listing 12: validation response

~~~
**Edits:**
- The procedure should include more
specific details and examples to clarify
 the steps for setting up a custom input
 schema.
- Provide examples or templates for
defining the function with custom
validation logic and implementing the
custom LLM class.
- Add a step to test the setup to ensure
 the custom input schema is working as
expected.
- It would be beneficial to include a
brief explanation of each step to
enhance understanding.

Here is the revised procedure:

[BEGIN PROCEDURE]
1. Define a function for your tool that
takes the desired multiple inputs and
includes custom validation logic to
enforce strict requirements. For example
:
 ```python
 def custom_input_function(input1,
 input2):
     # Custom validation logic
     implementation
     if input1 > 0 and isinstance(input2
     , str):
         return True
     else:
         return False
 ~~~
2. Create a `StructuredTool` using `
langchain.tools.StructuredTool.
from_function`, providing the function
you defined in step 1.
3. Define a custom LLM class that
inherits from `langchain.llms.base.LLM`.
 For example:
 ```python
 class CustomLLM(langchain.llms.base.LLM
 ):
     def __init__(self, model_path):
         super().__init__(model_path)
 ```
4. Implement the `_llm_type` property in
 the custom class to return a string
that identifies the type of LLM.
5. Implement the `_call` method in the
custom class to accept the structured
inputs from the tool function and apply
custom validation logic.
6. Implement the `_identifying_params`
property in the custom class to return a
 dictionary with the required input keys
 and their corresponding validation
criteria.
7. Instantiate the custom LLM class,
providing any necessary parameters such
as validation rules.
8. Initialize your agent with `langchain
.agents.initialize_agent`, providing a
list containing the custom LLM, the
language model, and the agent type `
langchain.agents.AgentType.
STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION
`.
9. Test the setup by running sample
inputs through the tool to ensure that
the custom input schema and validation
logic are working as expected.
[END PROCEDURE]
```

This cycle continues until all generated summaries are used.
```