Documentation Retrieval Improves Planning Language Generation

Anonymous ACL submission

Abstract

Certain strong LLMs have shown promise for zero-shot formal planning by generating planning languages like PDDL. Yet, performance of most open-source models under 100B parameters has been reported to be close to zero due to the low-resource nature of these languages. We significantly improve their performance via a series of lightweight pipelines that integrates documentation retrieval with modular code generation and error refinement. With models like Llama-4-Maverick, our best pipeline improves plan correctness from 0% to over 80% on the common BlocksWorld domain. However, while syntactic errors are substantially reduced, semantic errors persist in more challenging domains, revealing fundamental limitations in current models' reasoning capabilities.[1]

1 Introduction

Using large language models (LLMs) for planning has garnered significant attention, with two main paradigms as shown in Figure 1. First, the LLM-as-Planner approach (Kambhampati et al., 2024; Valmeekam et al., 2023; Stechly et al., 2025; Majumder et al., 2023) relies on the reasoning ability of LLMs to directly generate action plans based on descriptions of the environment. In contrast, the LLM-as-Formalizer (Tang et al., 2024; Guo et al., 2024; Zhang et al., 2024) approach leverages the code generation capability of LLMs to represent the environment in some planning language, which is then passed to a formal solver to derive a plan. Leading to better interpretability and verifiability of the plans, the latter approach has recently gained considerable attention, with Planning Domain Definition Language (PDDL) as one of the predominant formal languages for LLM planning (see the Appendix A for an example of PDDL).

While LLMs have been shown to somewhat able to generate PDDL, their performance has proven unsatisfactory in realistic and rigorous evaluations (Zuo et al., 2025). Even state-of-the-art coding LLMs have shown close-to-zero performance as PDDL formalizers on planning benchmarks especially when the model size is less than 100 billion parameters (Huang and Zhang, 2025), while an array of code generation techniques struggle to improve performance (Kagitha et al., 2025). Moreover, training data for low-resource and domain-specific languages like PDDL is extremely limited, making generation even more challenging (Tarassow, 2023; Joel et al., 2024). Existing attempts of improvement such as fine-tuning (Cassano et al., 2023; McKenna et al., 2025; Giagnorio et al., 2025) and translation from high-resource languages (Liu et al., 2024) require supervised PDDL data that barely exists. In contrast, retrieval of library documentation (Zhou et al., 2023; Dutta et al., 2024) has proven effective for high-resource languages.

We find that simply providing the documentation to LLMs does not help low-resource PDDL generation. However, we present some novel methods that generate PDDL either modularly or with error refinement, while only retrieving the most relevant documentation. These methods enable a "0 to 1" breakthrough of PDDL generation performance for models like Llama-4-Scout and Llama-4-Marverick on domains like BlocksWorld, improving correctness from 0% to 50%. Moreover, we

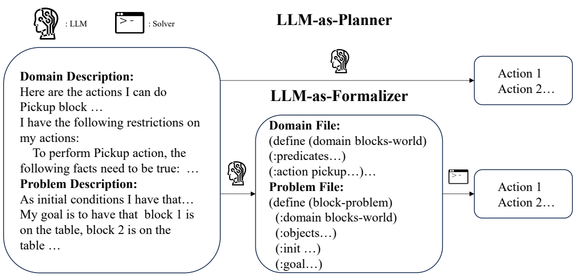

Figure 1: A simplified illustration of LLM-as-Planner and LLM-as-Formalizer on the BlocksWorld domain.

[1]Our code and data are attached with the submission.

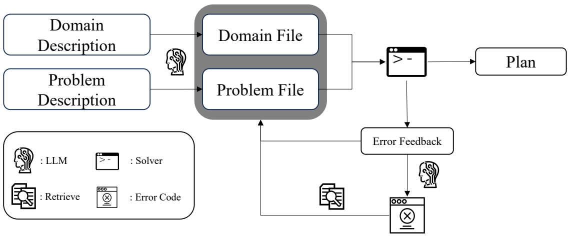

Figure 2: Overview of one of our pipeline that retrieve documents based on error codes located by LLM, and finally using them as hints to correct the code.

verify the intuition that documentation significantly reduces syntax errors, but has limited effect on semantic errors. We also present interesting findings that LLMs are more reliant on documentation initially than during error refinement, different models vary in their ability to leverage documentation effectively and that examples are more effective than descriptions in the documentation.

2 Methodology

We conduct experiments in text-based simulated planning environments. Each planning problem in the dataset is accompanied by a *domain description* (DD) outlining the environment, and a *problem description* (PD) specifying the task objective.

We begin with the most basic setting, referred to as **Base**, where a LLM zero-shot generates PDDL code. Given the DD and PD as input, the LLM produces a Domain File (DF) and a Problem File (PF):

$$\texttt{DF},\ \texttt{PF} = \mathsf{LLM}(\texttt{DD},\ \texttt{PD})$$

Building upon this, we leverage the PDDL documentation (Doc) during generation. We consider two approaches, **Once w/ Whole Doc** where the model is given an entire Doc before generating the entire PDDL, and **Modular w/ Specific Doc** where the model incrementally generates PDDL code guided by relevant parts of the Doc. Here, we break down the DF structure into types, predicates, actions, etc. and DF structure into initial and goal states. We partition the Doc accordingly.

$$\texttt{DF},\ \texttt{PF} = \mathsf{LLM}(\texttt{DD},\ \texttt{PD},\ \texttt{Doc})$$

Next, we optionally perform up to three rounds of iterative error correction. We first use a PDDL solver to obtain error feedback:

$$\texttt{Err\_Feedback} = \mathsf{Solver}(\texttt{DF},\ \texttt{PF})$$

Without the Doc, the standard **Refinement w/o Doc** directly input the error feedback back to the LLM to re-generate the PDDL:

$$\texttt{DF},\ \texttt{PF} = \mathsf{LLM}(\texttt{DF},\ \texttt{PF},\ \texttt{Err\_Feedback})$$

With the Doc, we attempt to retrieve a specific, helpful part that pertains to the particular error. Using the feedback directly as the query is referred to as **Refinement w/ Feedback-Retrieved Doc**. Otherwise, we may prompt an LLM to localize the code that caused the error based on the feedback, referred to as **Refinement w/ Code-Retrieved Doc**.

$$\texttt{Err\_Code} = \mathsf{LLM}(\texttt{Err\_Feedback})$$

In either case, we then retrieve the most relevant documentation snippet using the BM25 (Robertson et al., 2009) retrieval algorithm:

$$\texttt{Rel\_Doc} = \mathsf{BM25}(\texttt{Err\_Feedback}|\texttt{Err\_Code})$$

Finally, the LLM corrects the code using the retrieved Doc, the Error_Feedback, and the localized Error_Code if any.

$$\begin{aligned}\texttt{DF},\ \texttt{PF} = \mathsf{LLM}(&\texttt{DF},\ \texttt{PF},\ \texttt{Err\_Feedback},\\ &[\texttt{Err\_Code}],\ \texttt{Rel\_Doc})\end{aligned}$$

The full prompts and the pseudocode are provided in Appendix D, and C.

While we only consider PDDL as the planning language in this work following cited works, we also have explored the feasibility of using Satisfiability Modulo Theories (SMT) solvers—specifically Z3, a general-purpose solver for constraint satisfaction planning problems. Following Hao et al. (2025), our evaluation shows that Z3 exhibits suboptimal performance when handling complex planning tasks and is thus not discussed further (see details in Appendix B).

3 Evaluation

Dataset To conduct experiments in a text-based simulation environment, we use the dataset from (Huang and Zhang, 2025). Included are three simulated planning domains, BlocksWorld, Logistics, Barman from the International Planning Competition (IPC, 1998), with increasing action space and reported difficulty. We also consider Mystery BlocksWorld (Valmeekam et al., 2023) where all keywords are perturbed to combat LLM memorization. Each instance comes with domain and problem descriptions and ground-truth PDDL domain and problem files that are used to validate a predicted plan. Each domain has 100 tasks of varying problem complexity and description naturalness. We use the heavily templated descriptions which are also the easiest due to the reported close-to-zero

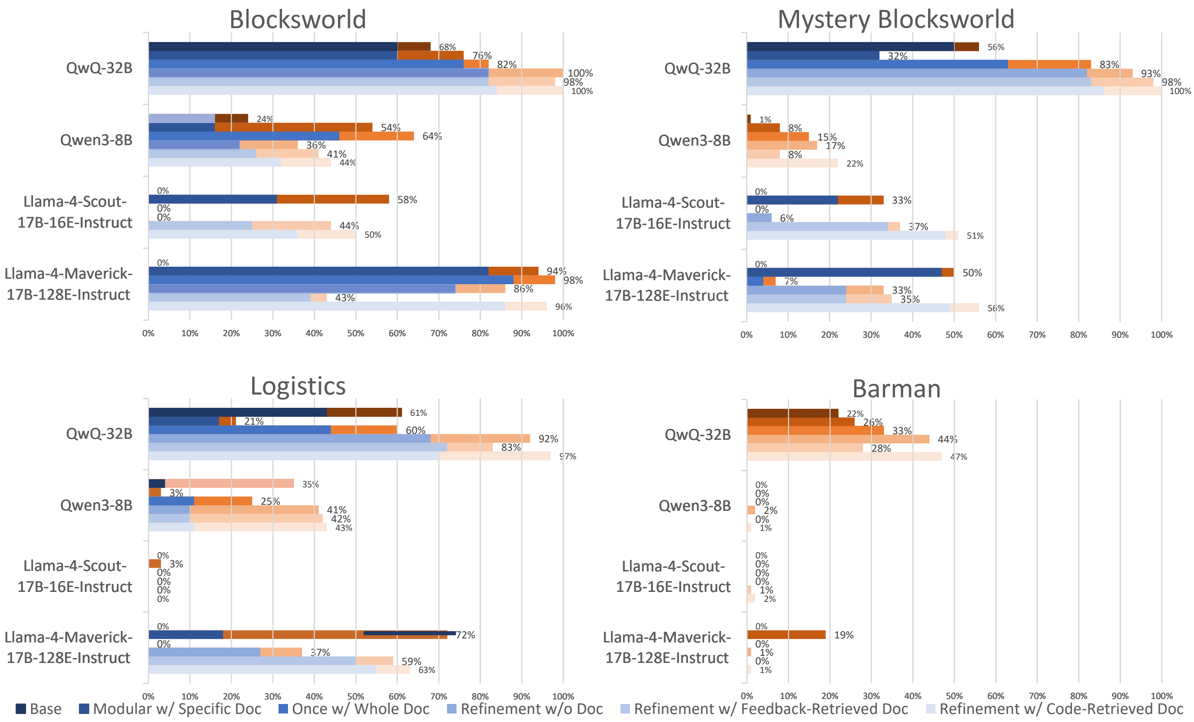

Figure 3: Syntactic accuracy (orange) and semantic accuracy (blue) on various planning domains.

performance of LLMs with less than 100B parameters that we focus on. We crawl, process and use the Planning Wiki[2] as the source of documentation of the PDDL language.

Metrics We follow Kagitha et al. (2025) and use syntactic and semantic accuracy to assess the DF and PF generated by an LLM. Syntactic accuracy is the percentage of problems where no syntax error are returned by the planning solver. Semantic accuracy is the percentage of problems where a plan is not only found but also correct. We use the dual-bfws-ffparser planner Muise (2016) to solve for the plan and the VAL4 (Howey et al., 2004) to validate the plan against the gold DF and PF.

Model We conduct experiments on six open-source models, ranging from 8B to 32B parameters: Llama-4-Maverick-17B-128E-Instruct, Llama-4-Scout-17B-16E-Instruct[3], QwQ-32B, Qwen3-8B[4]. We follow most cited previous works and only consider zero-shot prompting.

4 Results

We present the following key conclusions based on the results shown in the Figure 3.

Documentation brings a 0 to 1 breakthrough. On BlocksWorld, most LLMs under the Base setting perform close to zero accuracy, as observed in previous work. However, when equipped with appropriate documentation, they demonstrate a dramatic increase in their ability to generate valid PDDL. While the improvement depends on the LLM, Llama-4-Maverick sees a dramatic improvement of syntactic accuracy from 0% to over 90% and semantic accuracy of 0% to over 80% with the help of documentation but regardless of error refinement. Other originally zero-performing models such as Llama-4-Scout see an improvement of 50% for syntactic and 30% for semantic accuracy. On more challenging domains, absolute performance for all LLMs are thwarted, while documentation still greatly improves syntactic accuracy for many models. Overall, models that previously failed entirely begin to become functional as planning formalizers.

Specific docs significantly reduces syntax errors. Documentation proves effective in reducing syntax errors during both initial PDDL generation (*Modular w/ Specific Doc*) and subsequent error-correction (*Refinement w/ Code-Retrieved Doc*). This effect is especially evident in the case of Llama-4-Scout, which fails to generate any valid PDDL originally regardless of whether error cor-

[2] https://planning.wiki/guide/whatis/pddl

[3] https://github.com/meta-llama/llama-models/tree/main/models/llama4

[4] https://github.com/QwenLM/Qwen3

rection is applied. Only when supported by relevant docs can it successfully generate valid PDDL, much of which leading to correct plans. Notably, using feedback to retrieve doc does not lead to consistent or significant performance gains, as the retrieved documents often fail to accurately correspond to the actual errors. This highlights that retrieval based on error codes is more effective in improving the accuracy of documentation retrieval.

Docs cannot reliably reduce semantic errors. During error correction, Llama-4-Maverick shows a 3% improvement in syntax accuracy on the Logistic dataset under the *Refinement w/ Code-Retrieved Doc* setting compared to the *Refinement w/o Doc* setting. However, its semantic accuracy decreases by 1%. This is because generating valid PDDL not only requires syntactic correctness but also an accurate representation of the environment. Otherwise, the resulting plan may fall into a loop, fail to reach the goal due to insufficient executable actions, or be unnecessarily complex. Achieving this depends heavily on the reasoning capabilities and world modeling abilities of the LLM, and simply providing documentation is not sufficient to enhance such reasoning.

LLMs exhibit varying sensitivity to documentation across different phases of the code generation process. Our results reveal that documentation exerts a stronger influence during the initial code generation phase compared to the subsequent error refinement phase. Specifically, in the *Formalize* phase—corresponding to the initial generation of PDDL—providing specific documentation significantly improves syntax accuracy, reaching up to 72% for modular models with targeted documentation. In contrast, the benefits of documentation during the later *Refinement* phase are substantially smaller. This suggests that models rely more on documentation cues when initially producing structured code, whereas later refinements depend more on internal representations and the code previously generated.

LLMs that are better at generating PDDL can make more effective use of documentation. Since QwQ-32B and Qwen3-8B outperform LLaMA-4 models in the *Base* setting, we consider them more proficient at PDDL generation. Compared to the *Base* and *Modular w/ Specific Doc* settings, these PDDL-proficient models (QwQ-32B and Qwen3-8B) perform better under the *Once w/ Whole Doc* setting. In contrast, the less proficient LLaMA-4 model does not outperform *Modular w/ Specific Doc* under the same condition. This suggests that for models less capable of generating PDDL, modular generation is more effective, as they tend to become overwhelmed when processing large amounts of documentation.

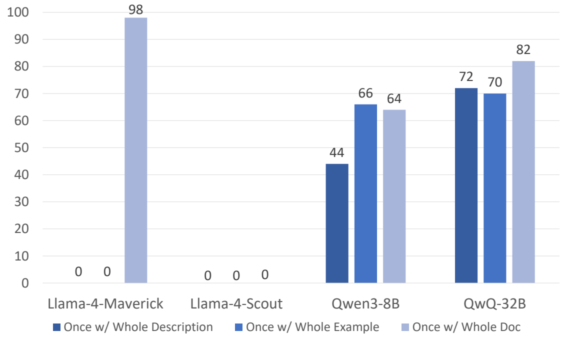

Figure 4: Syntactic accuracy of different models under various document conditions on BlocksWorld. Once w/ whole example refers to all the examples in the doc, and Once w/ whole description refers to all the textual descriptions in the doc.

Using examples to convey knowledge is more effective than using descriptions. Figure 4 presents the performance of different types of documentation in the LLM-as-Formalizer setting. Among all types, *Once w/ whole doc* yields the best results. Notably, for Llama-4-Maverick, performance is 0% when provided with only examples or only descriptions, but nearly 100% when given the entire documentation. Comparing *Once w/ whole example* and *Once w/ whole description*, we observe that examples consistently outperform descriptions. This suggests that examples are easier for LLMs to comprehend and are more useful for correcting syntax errors. Furthermore, even for models with inherently strong PDDL generation capabilities, such as QwQ-32B, the use of documentation still leads to a noticeable improvement in performance.

5 Conclusion

Our experiments clearly demonstrate that incorporating documentation to the process greatly improves generation of low-resource formal languages like PDDL. We show that for models less skilled at generating PDDL, documentation is only useful when paired with techniques like modular generation or error refinement. For more capable models, documentation accuracy matters more. Despite the clear gain, models still struggle when their size is small and when the domain is complex, which future work should strive to address.

6 Limitations

While our proposed pipelines significantly improve the syntactic and, to a lesser extent, semantic accuracy of PDDL generation in low-resource settings, several limitations remain. First, our methods rely on well-structured documentation and domain descriptions; performance may degrade in noisy or under-specified environments. Moreover, documentation itself may contain outdated, incomplete, or inaccurate information, which can mislead the model during generation. Second, although documentation helps reduce syntax errors, semantic correctness still heavily depends on the model's internal reasoning capabilities, which are limited for smaller LLMs. Lastly, our evaluation is confined to a few benchmark domains; generalization to more diverse or real-world planning scenarios remains to be verified.

The datasets we use are all under the MIT License.

References

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Carolyn Jane Anderson, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2023. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8:677 – 708.

Avik Dutta, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. 2024. RAR: Retrieval-augmented retrieval for code generation in low resource languages. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 21506–21515, Miami, Florida, USA. Association for Computational Linguistics.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing code generation for low-resource languages: No silver bullet. *ArXiv*, abs/2501.19085.

Weihang Guo, Zachary Kingston, and Lydia E. Kavraki. 2024. Castl: Constraints as specifications through llm translation for long-horizon task and motion planning. *Preprint*, arXiv:2410.22225.

Yilun Hao, Yang Zhang, and Chuchu Fan. 2025. Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. *Preprint*, arXiv:2410.12112.

R. Howey, D. Long, and M. Fox. 2004. Val: automatic plan validation, continuous effects and mixed initiative planning using pddl. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301.

Cassie Huang and Li Zhang. 2025. On the limit of language models as planning formalizers. *Preprint*, arXiv:2412.09879.

IPC. 1998. International planning competition. `https://www.icaps-conference.org/competitions`.

Sathvik Joel, Jie Jw Wu, and Fatemeh H. Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages.

Prabhu Prakash Kagitha, Andrew Zhu, and Li Zhang. 2025. Addressing the challenges of planning language generation. *Preprint*, arXiv:2505.14763.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Saldyt, and Anil Murthy. 2024. Llms can't plan, but can help planning in llm-modulo frameworks. *Preprint*, arXiv:2402.01817.

Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. 2024. Synthesizing programmatic reinforcement learning policies with large language model guided search. *ArXiv*, abs/2405.16450.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. 2023. Clin: A continually learning language agent for rapid task adaptation and generalization. *Preprint*, arXiv:2310.10134.

Nick McKenna, Xinnuo Xu, Jack Williams, Nick Wilson, Benjamin Van Durme, and Christian Poelitz. 2025. Synthetic function demonstrations improve generation in low-resource programming languages. *ArXiv*, abs/2503.18760.

Christian Muise. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*.

Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2025. Chain of thoughtlessness? an analysis of cot in planning. *Preprint*, arXiv:2405.04776.

Hao Tang, Darren Key, and Kevin Ellis. 2024. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *Preprint*, arXiv:2402.12275.

Artur Tarassow. 2023. The potential of llms for coding with low-resource and domain-specific programming languages. *ArXiv*, abs/2307.13018.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Preprint*, arXiv:2206.10498.

Li Zhang, Peter Jansen, Tianyi Zhang, Peter Clark, Chris Callison-Burch, and Niket Tandon. 2024. PDDLEGO: Iterative planning in textual environments. In *Proceedings of the 13th Joint Conference on Lexical and Computational Semantics (*SEM 2024)*, pages 212–221, Mexico City, Mexico. Association for Computational Linguistics.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. *Preprint*, arXiv:2207.05987.

Max Zuo, Francisco Piedrahita Velez, Xiaochen Li, Michael L. Littman, and Stephen H. Bach. 2025. Planetarium: A rigorous benchmark for translating text to structured planning languages. *Preprint*, arXiv:2407.03321.

A Data and PDDL Examples

Figure 5 and 6 is an example of the dataset Heavily_Templated_BlocksWorld-100 from (Huang and Zhang, 2025).

B Z3 Result

We followed the (Hao et al., 2025) by using Formulator to define all possible variables in the environment and generate their instantiation information before producing the Z3 code. However, we did not adopt their iterative error correction method. In their experiments, Formulator improved the results on the BlocksWorld domain from 0.2 to 96.2.

We conducted experiments on our dataset using GPT-4o as the LLM, but the results were 0. The distribution of error causes is shown in the Table 1. Goal unsatisfied means that the final output plan cannot solve the problem correctly. We analyzed the cause of this error. We printed the state of each time slice and found that as long as any condition in the goal state is met, the planning will stop. When we tried to let LLM correct this error, it only caused more syntax errors, and never corrected the error. This is likely because our dataset is more complex—theirs only involved 4 blocks, whereas ours often includes more than 10 blocks.

Since even the simplest BlocksWorld dataset yielded a score of 0 after following the (Hao et al., 2025) approach, we did not apply our pipeline to Z3 and instead reported the findings in the appendix.

C Pseudocode of Refinement w/ Code-Retrieved Doc

Algorithm 1 shows the Pseudocode of Refinement w/ Code-Retrieved Doc.

	Heavily BlocksWorld	
Model	syntax error	goal unsatisfied
gpt-4o	16/100	84/100

Table 1: Z3 Result

D Prompt

Figure 9 10 11 12 and 13 is the Prompt of all our methods. Refinement w/ Feedback-Retrieved Doc and Refinement w/ Code-Retrieved Doc use the same prompt but different retrieved docs.

```
I am playing with a set of blocks. Here are the actions I can do

   Pickup block
   Unstack block from another block
   Putdown block
   Stack block on another block

I have the following restrictions on my actions:
    To perform Pickup action, the following facts need to be true: clear block, block on table,
       arm-empty.
    Once Pickup action is performed the following facts will be true: holding block.
    Once Pickup action is performed the following facts will be false:  clear block, block on
       table, arm-empty.
    To perform Putdown action, the following facts need to be true: holding block.
    Once Putdown action is performed the following facts will be true: clear block, block on
       table, arm-empty.
    Once Putdown action is performed the following facts will be false: holding block.
    To perform Stack action, the following needs to be true: clear block2, holding block1.
    Once Stack action is performed the following will be true: arm-empty, clear block1, block1
       on block2.
    Once Stack action is performed the following will be false: clear block2, holding block1.
    To perform Unstack action, the following needs to be true: block1 on block2, clear block1,
       arm-empty.
    Once Unstack action is performed the following will be true: holding block1, clear block2.
    Once Unstack action is performed the following will be false:, block1 on block2, clear
       block1, arm-empty.
```

Figure 5: DD for the BlocksWorld domain

```
As initial conditions I have that, block 1 is clear, block 2 is clear, block 3 is clear, block
       4 is clear, arm-empty, block 1 is on the table, block 2 is on the table, block 3
is on the table, and block 4 is on the table.
My goal is to have that block 1 is on the table, block 2 is on the table, block 3 is on the
       table, and block 4 is on the table.
```

Figure 6: PD for the BlocksWorld domain

```
(define (domain blocks-world)
  (:requirements :strips :typing)
  (:predicates (clear ?x - block)
               (on ?x ?y - block)
               (ontable ?x - block)
               (holding ?x - block)
               (arm-empty))

  (:action pickup
    :parameters (?b - block)
    :precondition (and (clear ?b) (ontable ?b) (arm-empty))
    :effect (and (holding ?b) (not (clear ?b)) (not (ontable ?b)) (not (arm-empty))))

  (:action unstack
    :parameters (?b1 ?b2 - block)
    :precondition (and (on ?b1 ?b2) (clear ?b1) (arm-empty))
    :effect (and (holding ?b1) (clear ?b2) (not (on ?b1 ?b2)) (not (clear ?b1))
       (not (arm-empty))))

  (:action stack
    :parameters (?b1 ?b2 - block)
    :precondition (and (clear ?b2) (holding ?b1))
    :effect (and (arm-empty) (clear ?b1) (on ?b1 ?b2) (not (clear ?b2))
       (not (holding ?b1))))

  (:action putdown
    :parameters (?b - block)
    :precondition (holding ?b)
    :effect (and (ontable ?b) (clear ?b) (arm-empty) (not (holding ?b))))
)
```

Figure 7: DF for the BlocksWorld domain

```
(define (problem block_problem)
  (:domain block-stacking)
  (:objects block1 block2 block3 block4 - block)
  (:init
    (clear block1)
    (on block1 block2)
    (clear block3)
    (on block3 block4)
    (on_table block2)
    (on_table block4)
    (arm_empty)
  )
  (:goal (and
    (on_table block1)
    (on_table block2)
    (on_table block3)
    (on_table block4)
  ))
)
```

Figure 8: PF for the BlocksWorld domain

Base Prompt

You are a PDDL expert. Here is a game we are playing.
{domain_description}
{problem_description}
Write the domain and problem files in minimal PDDL.

Figure 9: Base Prompt

modular w/ specific doc Prompt

You are a PDDL expert.
Is next step needed here: yes
next step: domain definition
next domain file:
(define (domain construction)
)

Is next step needed here: yes
next step: requirements
next domain file:
(define (domain game)
  (:requirements :strips :adl :typing)
)

[More steps here]
…

follow the example above, please do not add any other word, generate only the next one step for the domain file.
Here is a game we are playing.
{domain_description}
{problem_description}

Figure 10: modular w/ specific doc Prompt

Once w/ Whole Doc

Knowledge:
{doc}

are a PDDL expert. Here is a game we are playing.
{domain_description}
{problem_description}
Write the domain and problem files in minimal PDDL.

Figure 11: Once w/ Whole Doc Prompt

Refinement w/o Doc

Wrong_domain_file:
{previous_domain_file}
Wrong_problem_file:
{previous_problem_file}
error feedback:{result}
Instruction: I provided a wrong set of PDDL files, you need according to the error feedback, give me the corrected domain_file and problem_file. You must make changes.

Figure 12: Refinement w/o Doc Prompt

Refinement w/ Retrieved Doc

Knowledge:{doc}

Wrong_domain_file:
{previous_domain_file}
wrong_problem_file:
{previous_problem_file}
Wrong PDDL:
{query}

error: {result}

Instruction: I provided a wrong PDDL files and the documentation for the errors, you need according to the documentation, give me the corrected domain_file. You must make changes, and give me a logical reason for why you change like that. Do not add any other word.

Figure 13: Refinement w/ Retrieved Doc Prompt

Algorithm 1 Retrieval-Augmented PDDL Generation with Iterative Correction

Require: Domain Description (DD), Problem Description (PD)
Ensure: Valid Domain File (DF) and Problem File (PF)

$\langle DF, PF \rangle \leftarrow$ LLM(DD, PD)
while true **do**
  $feedback \leftarrow$ Solver(DF, PF)
  if feedback indicates success **then**
    return $\langle DF, PF \rangle$
  end if
  $e\_type \leftarrow$ Parse_Error_Type$(feedback)$
  if e_type == `syntax_error` **and** $feedback.file$ == `DF` **then**
    $e\_code \leftarrow$ LLM$(feedback)$
    $doc \leftarrow$ Retrieve(e_code)
    $\langle DF, PF \rangle \leftarrow$ LLM$(DF, PF, e_code, feedback, doc)$
  else if e_type == `syntax_error` **and** $feedback.file$ == `PF` **then**
    $\langle DF, PF \rangle \leftarrow$ LLM$(DF, PF, feedback)$
  else if e_type == `semantic_error` **then**
    $\langle DF, PF \rangle \leftarrow$ LLM$(DF, PF, feedback)$
  else
    raise `UnknownErrorType`
  end if
end while