Self-Enhancing Programming-Driven Reasoning for Visual Question Answering

Anonymous ACL submission

Abstract

In Visual Question Answering (VQA) tasks, program-driven reasoning methods have made significant progress by transforming solutions into executable code. However, existing approaches often fall short due to their reliance on a single code generation iteration, lacking the flexibility to handle unforeseen errors. To address this limitation, we propose the Self-Enhancing Programming-driven Reasoning framework for VQA (Seper). Seper combines a code generator and evaluator to decompose questions into multistep instructions, dynamically generating Python code based on input. The code evaluator performs both forward and backward evaluations, triggering an iterative code regeneration process that enables continuous optimization. Additionally, we introduce prompt tuning to improve the quality of the generated code. Our experiments on the GQA and OK-VQA datasets demonstrate Seper's superior performance, highlighting its potential to advance VQA programming methods. Code: https://anonymous.4open.science/r/Seper-5540/

1 Introduction

The pursuit of artificial general intelligence (AGI) has led to the development of large-scale visual models, which often rely on task-specific datasets, such as object grounding, visual question answering, or image segmentation. These models struggle with complex, real-world tasks that require multi-step reasoning. For example, answering "What was the color of the first horse that crossed the finish line?" involves identifying the finish line, recognizing the first horse, and determining its color. Such intricate tasks challenge the effectiveness and scalability of end-to-end models.

Early approaches, such as Modular Visual Question Answering(Andreas et al., 2016), decompose tasks into atomic units but require extensive supervision, limiting their adaptability across domains. The emergence of large language models has driven interest in automatic module integration, giving rise to program-driven reasoning methods (Schick et al., 2023; Surói s et al., 2023; Subramanian et al., 2023; Gupta and Kembhavi, 2022), which tackle complex problems through step-by-step code solutions. By addressing subproblems via defined APIs, these methods eliminate the training costs of traditional modular approaches and provide enhanced interpretability.

Existing programming-driven reasoning methods often lack iterative and reflective processes, typically performing only a single iteration of code generation and execution. These methods fail to address unforeseen issues, such as syntax errors or erroneous API results, which can affect the final outputs. While human programmers can debug code manually, this process is time-consuming and lacks real-time efficiency. Additionally, large language models (LLMs) usually require specific prompts to generate code, yet they struggle with complex visual language tasks due to difficulties in comprehending detailed task requirements, often resulting in suboptimal code generation.

Self-iterative models have emerged as a powerful approach in AI, particularly for code generation tasks that demand high precision. These models rely on unit tests to detect errors and use compiler outputs to inform corrections. By iteratively refining the generated code based on test results, they can effectively handle complex programming challenges, ensuring both correctness and optimization.

When applying the self-iterative framework to visual programming, several key challenges must be addressed during its implementation:

- Without unit tests, how can we effectively detect errors in the generated code?
- How can we revise the generated code based on the detected errors?

To address these limitations, we propose the Self-Enhancing Programming-Driven Reasoning (Seper) framework for Visual Question Answering (VQA), as illustrated in Figure 2. The Seper framework goes beyond traditional visual programming approaches by integrating multiple code evaluation components that automatically assess the generated code and produce detailed reports for further improvement.

To address these limitations, we propose the **S**elf-**e**nhancing **p**rogramming-Driven **R**easoning (Seper) framework for Visual Question Answering (VQA), as illustrated in Figure 2. The Seper framework goes beyond traditional visual programming approaches by integrating multiple code evaluation components that automatically assess the generated code and produce detailed reports for further improvement.

Our main contributions include:

- We introduce the first iterative framework for Visual Question Answering that integrates both code generation and evaluation, marking a significant advancement in the field.
- We propose a supervised prompt tuning method for automatically adjusting prompt templates used by the code generator and reviewers during code evaluation.
- Experimental results on real benchmarks demonstrate the superior performance of our approach.

2 Related Work

The early method for the Visual Question Answering is the **Modular Visual Question Answering** (Modular VQA), which refers to a framework used in Visual Question Answering (VQA) tasks that decomposes the question-answering process into distinct, modular components (Andreas et al., 2016). For example, Promptcap(Hu et al., 2023) is the first to leverage the text reasoning capabilities of large language models in conjunction with generated image captions to address visual questions. Similarly, Prophet(Shao et al., 2023) employs a large language model to aggregate judgments from multiple VQA models, improving the overall accuracy of answers by combining insights from different systems.

In contrast to the methods described above, recent research has introduced **Visual Programming** frameworks that uses the incontext learning ability of large language models to generate python-like modular programs, which are then executed to get both the solution and a comprehensive and interpretable rationale. Each line of the generated program may invoke one of several off-the-shelf computer vision models, image processing subroutines, or python functions to produce intermediate outputs that may be consumed by subsequent parts of the program. This allows for the seamless integration of new components, enhancing the system's versatility and adaptability to a broad range of VQA tasks. Several studies have explored visual programming for VQA (Surís et al., 2023; Subramanian et al., 2023; Gupta and Kembhavi, 2022; Khan et al., 2024; Dou et al., 2024), but these approaches typically generate code in a single pass, without iterative refinement. In contrast, our method continuously evaluates and revises the generated code, enabling ongoing optimization through repeated iterations, which ensures improved performance and adaptability.

However, the code generated by visual programming may not always run as expected. Even if execution is possible, issues such as incorrect interface calls or parameter mismatches can arise, leading to inaccurate results. **Code refinement** refers to the process of improving, optimizing, or cleaning up code while preserving its overall functionality. Statistical and learning-based techniques for code refinement have a long history in both programming languages and machine learning, typically focused on repairing human-written code (Long and Rinard, 2016; Bader et al., 2019; Le Goues et al., 2021). Self-Edit (Zhang et al., 2023) explores self-repair with natural language feedback on apps. Similarly, T-eval (Chen et al.) assesses Codex's ability to self-repair across a variety of tasks.

However, the aforementioned work is specifically designed for text-based tasks and relies on correction through unit testing (Olausson et al., 2023), which requires knowing the correct output in advance—an approach that is difficult to apply to Visual Question Answering (VQA) due to the challenge of labeling correct outputs. To address this issue, our framework incorporates a self-enhancing mechanism that autonomously identifies problems in the generated code and revises it accordingly.

3 Error Analysis Of Existing Methods

To examine the influence of code generation quality and pre-trained model capabilities on programming-driven reasoning, we analyzed 200 randomly selected error cases from Viper (Surís et al., 2023) using the GQA (Hudson and Manning, 2019) test-dev dataset. Figure 1 presents the evaluation results, categorizing the failure reasons into three main groups:

1. **Code Error:** These errors arise during code execution and include syntax issues (e.g., missing semicolons or mismatched parentheses) and runtime problems (e.g., invalid input, memory overflow, or division by zero), which can lead to program termination.

2. **API Limitation:** This refers to situations where API calls, particularly those involving pre-trained models, fail to produce expected results. This category includes: **Image-Text-Contrastive (ITC) models**, which use models like CLIP (Radford et al., 2021) and XVLm (Zeng et al., 2021) which assess image attributes and are affected by object names and properties; **Visual Question Answering (VQA) models**, where slight variations in question phrasing can impact outcomes despite identical semantics; and **Location (LOC) models**, such as GroundingDINO (Liu et al., 2023), which detect and crop objects in images, with object name sensitivity influencing detection accuracy.

3. **Logic Error:** Can be classified into answer logic errors and potential logic errors.

    - **Answer Inaccuracies:** Return values misaligned with question format (e.g., yes/no for multiple-choice).
    - **Potential Logic Errors:** Reflect mismatches between the question-solving process and code logic, often arising with more complex images than anticipated, such as missing target objects, as illustrated in Figure 5.

The insights from Figure 1 reveal that Viper mainly faces API limitations and logic errors, with code errors being rare due to GPT's robustness. VQA models notably contribute to API limitations, highlighting the importance of question representation in visual question answering.

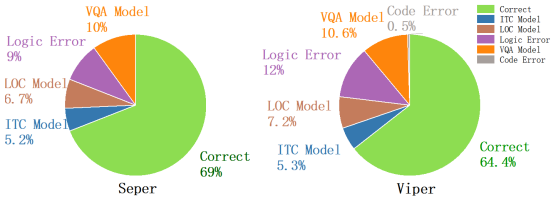

Figure 1: Error analysis results.

4 Methodology

The Seper framework combines code generation, execution, and evaluation. Given an image and question, the code generator breaks down the task, generates code, and passes it to the Python compiler. The code is evaluated in two stages: forward evaluation checks input, and backward evaluation traces errors. Language Model-based reviewers detect issues, and if errors are found, a review report is generated to refine the code iteratively until it passes or the iteration limit is reached.

4.1 Code Generation

As shown in Figure 2(a), the code generation process consists of two stages using a Code Large Language Model (CodeLLM): rationale generation and Python code generation. To enhance reasoning interpretability and soundness, CodeLLM first generates rationales that break down the task into subtasks, guiding the subsequent code generation. These subtasks are then embedded as comments in the code to improve readability.

A predefined prompt, as detailed in Listing 1, guides the CodeLLM in generating accurate Python code. This prompt combines the question $\hat{q}$, API specifications, examples, and a review report, providing a comprehensive input to steer the code generation process.

Listing 1: Prompt for code generation

```
# Here are the APIs available for
    calling
${API specifications}
# Learn from the following examples
${Related examples}
# The question
${Question}
# The review result from previous
    iteration
${Rationale}
# Generate python code to answer the
    question, with variable "image" as
    input
```

Prompt Composition: To answer a visual question using an image, the goal is to leverage a Large

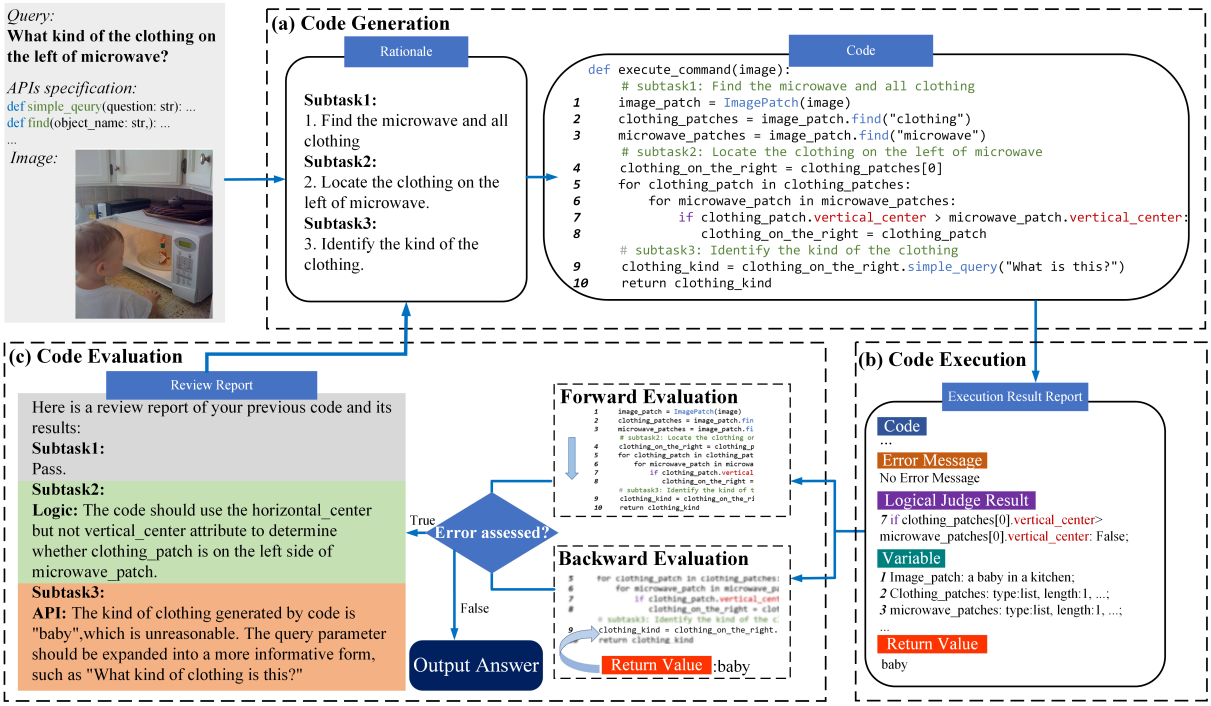

Figure 2: Our proposed Seper framework.

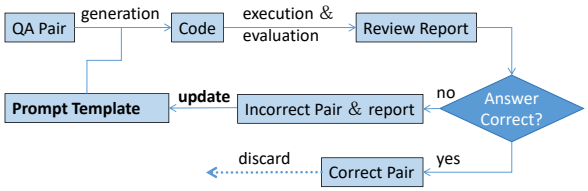

Figure 3: Supervised Prompt Tuning framework.

Language Model to generate Python code by enhancing the question representation with three key elements:

1. **API Specifications:** We use the `ImagePatch` API specification and text query $\hat{q}$ as input prompts for the program generation model (Surís et al., 2023). While following the original implementation, our approach differs in categorizing the APIs into VQA, Location, and Other types. Detailed API specifications are in Appendix B.

2. **Related examples**: The process of example selection is crucial. We follow the setup of mainstream methods by using manually constructed examples as context in experiments.

3. **Rationale**: Including the three subtasks decomposed by the LLM, as well as the possible review report from the review process detailing code execution results and error corrections, is vital for the code generator. CodeLLM uses this feedback to iteratively refine and regenerate code.

4.2 Code Execution

Upon code generation, we employ the Python compiler to execute the code and gather all necessary variables and intermediate data for thorough review. Ultimately, a comprehensive execution result report is generated, as illustrated in Figure 2(b).

Execute Code: Upon code generation, snippets will be injected to preserve variables, which will be stored in a Python dictionary for later analysis.

Generate Execution Result Report: Multiple execution result reports will be generated in structured format, each covering five key components comprehensively: (1) **Source code**: The Python code generated by the code generator; (2) **Error Message**: Returns detailed error information (location, type, stack trace) for the first encountered error, or "No error" if none detected; (3) **Return Value**: The final text-based return value or an empty string in case of encountering errors; (4) **Runtime variables**: Variables are categorized as ImagePatch objects with BLIP-2 descriptions, basic strings, and collections with text-converted elements and lengths; (5) **Logical judgement result**: Extracted conditional statements trace execution flow through loops and branches, using transformed variables to

indicate decisions.

4.3 Code Evaluation

The code evaluation process, as shown in Figure 2(c), involves extracting the relevant execution results for each sub-task and conducting evaluations in two sub-stages: forward and backward.

- **Forward Evaluation:** Verifying whether each line of code ascertain its successful execution, followed by validating whether the output of each line conforms to the expected results.
- **Backward Evaluation:** Assessment of code output against intended objectives, followed by retrospective analysis of potential errors and intermediate results.

Unlike traditional code generation tasks (Long and Rinard, 2016; Bader et al., 2019; Le Goues et al., 2021), which can be validated using test cases, our approach faces the challenge of missing test cases, leading to uncertain and unverifiable outcomes. To address this, we utilize heuristic-based reviews focused on the three components outlined in the previous section: **code error checking** to identify syntax and runtime issues, **API checking** to verify API calls and their results, and **code logic checking** to detect logical errors causing incorrect outputs. Different Language Model-based reviewers evaluate each component, following a two-step process using the prompts detailed in Appendix C:

- **Result assessment:** This process identifies errors by analyzing execution result reports, examining explicit and implicit errors from various sources. Each report is reviewed by different reviewers who provide detailed responses and a true or false answer. A majority vote then determines the final acceptance of the response.
- **Review generation:** Reviewers compile assessment results to identify error types and underlying causes, offering specific suggestions for code improvement. The goal is to provide a clear and thorough analysis to guide code modifications and enhance quality.

Finally, we merge the reports of each sub-task into a comprehensive report and provide feedback to the code generator.

4.3.1 Forward Evaluation.

The forward evaluation stage verifies that each line of code executes correctly and produces the expected output. It focuses on local issues, such as runtime errors, syntax errors, and the proper functioning of API calls.

Code Error Reviewer

Code error reviewer assesses the basic errors in the code and generate reviews as follows:

Result assessment: Leveraging the error information and location captured during the code execution, we directly utilize the error message in the execution result report. If no error message is present, the review generation step is skipped.

Review generation: In the prompt for the Code Error Reviewer, we first provide definitions for syntax errors and runtime errors. Subsequently, we instruct the reviewer to address syntax errors directly and to add exception handling for runtime errors by generating new code for the erroneous part.

API Reviewer

The API reviewer's main objective is to systematically identify and evaluate errors in API calls, consolidating these results into a comprehensive review. Given that VQA-type and location-type APIs exhibit the most errors, the reviewer employs specialized strategies tailored to these API types.

Result Assessment: Each API is treated as a black box without disclosing intermediate results. The output variable's name indicates the expected output type. For example, "clothing_kind" in line 9 suggests the answer should be a clothing type. Results are assessed by checking alignment with the expected type. This involves two stages: extracting key details from the execution report (e.g., objects, parameters, and output values) and guiding the reviewer to evaluate acceptability by prompt template. For vqa APIs, reviewers assess the plausibility and clarity of the answer. For location-type APIs, reviewers confirm the presence of the object as per the image caption and its relevance to the original question.

Review Generation: Inspired by the Exploration-Exploitation Dilemma in reinforcement learning, we design two instruction sets for guiding the API reviewer in generating comments. The first encourages CodeLLM to adjust parameters of existing API calls, such as using more specific parameters if a "simple_query(.)" output is vague. The second promotes exploration by suggesting alternative

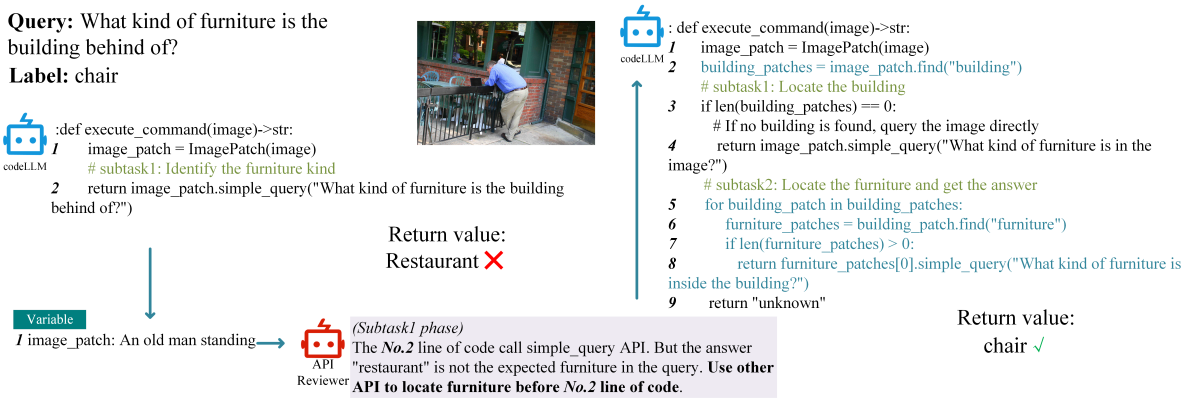

Figure 4: Example of VQA type API limiation.

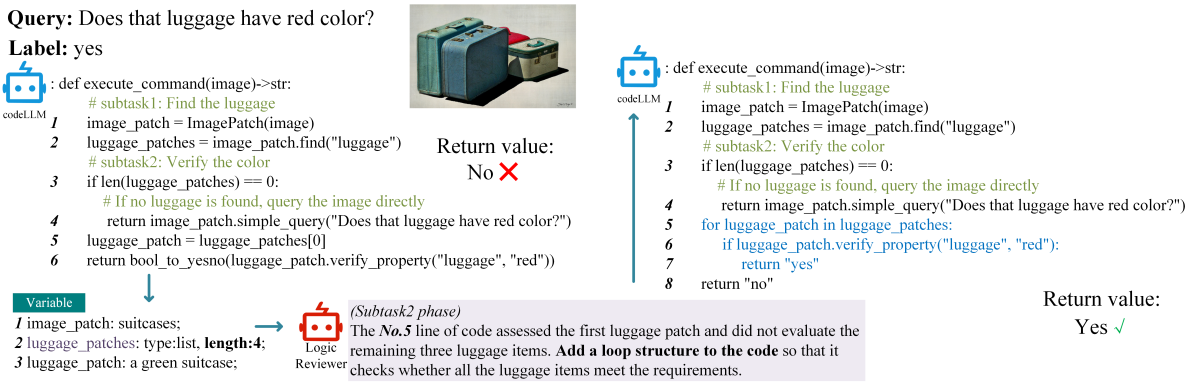

Figure 5: Example of lacking structure type logical error.

API calls. For example, as shown in Figure 4, when vqa-type APIs alone are insufficient, Seper prompts the code generator to combine them with loc-type APIs.

4.3.2 Backward Evaluation.

In neural networks, backpropagation calculates final error and updates gradients in preceding layers. Similarly, we propose a backward evaluation mechanism to trace final errors up the source code, using a logic reviewer to identify and analyze error origins.

Logic Reviewer

As discussed in section 3, logic errors can be categorized into answer errors and potential errors.

Result Assessment: Unlike APIs, which are treated as black boxes, a logic reviewer can access all code lines and runtime variables, enabling a thorough evaluation of the code's logic. The assessment is conducted from two perspectives: 1) **Answer Inaccuracies:** These errors are similar to those in Visual Question Answering (VQA) systems, such as misinterpretations of queries like "Is he to the right or left of the mirror?". Reviewers assess whether the response aligns with the intended question. 2) **Potential Errors:** These errors stem from issues in key algorithmic constructs, including sequential, branching, and looping structures. Such errors may arise from missing structures (e.g., absent loops) or inaccuracies within existing ones (e.g., incorrect branch conditions). Reviewers identify these issues by analyzing the variables and source code to ensure the logic is correctly implemented.

Review Generation: Reviewers analyze the code to identify logic errors and provide actionable feedback for improvement. The evaluation is conducted from two perspectives: 1) **Answer Inaccuracies:** For answer errors, reviewers trace the return value back to the first incorrect statement to identify the root cause, which may involve an API or logical error. This process helps pinpoint the exact issue and informs further analysis. 2) **Potential Errors:** For potential errors, such as incorrect spatial properties in left-right judgment (as shown in Figure 5), reviewers assess the logic structure as a whole. They evaluate whether the algorithm's constructs are correctly implemented and aligned with the

intended behavior, suggesting modifications if necessary. Finally, reviewers output "pass" if no errors are found; otherwise, the aggregated report is sent to CodeLLM for further iteration.

4.4 Supervised Prompt Tuning

To improve the performance of CodeLLM, we propose a supervised prompt tuning method to optimize prompt templates for code generation. As shown in Figure 3, the process begins by running Seper on a training dataset using initial prompt templates to generate and execute code. We construct a sample pool, initially populated with all training samples. After evaluating the results, we select representative examples, especially those that frequently cause errors, for prompt refinement. In an iterative process, error-prone samples are retained in the pool, while correct samples are gradually discarded, with only a small subset kept. These examples help the reviewer model identify failure patterns and refine the prompt templates. Details are provided in Appendix F.2.

5 Experiment

5.1 Experimental Setup

Benchmarks. We evaluate our method on the GQA (Hudson and Manning, 2019) and OK-VQA (Marino et al., 2019) datasets. GQA focuses on multi-hop questions using human-annotated scene graphs for complex compositional queries, while OK-VQA assesses answering image-related questions requiring external knowledge.

Implementation. We utilized ChatGPT(OpenAI, 2023) to serve as both code generator and three reviewers, each with distinct sessions to ensure independence. Choosing ChatGPT is to maintain consistency with the mainstream visual programming methods that have predominantly used ChatGPT, and also because, as previous research(Chen et al., 2023). We employ BLIP-2(Li et al., 2023), GroundingDINO(Liu et al., 2023), and XVLM(Zeng et al., 2021) as the pre-trained models underpinning our API. Detailed parameters are provided in the Appendix A.

5.2 Results

Table 1 presents results on the GQA test-dev and OK-VQA validation datasets, showing that Seper outperforms all other zero-shot methods. Despite utilizing large foundational models, programming-based methods like ViperGPT and CodeVQA are limited by their single-pass approach, resulting in lower accuracy. Seper, with its iterative framework for code generation and evaluation, achieves the highest accuracy of 51.3% on GQA and 54.1% on OK-VQA, outperforming the next best methods, VisProg, by 0.8% and 1.5%, respectively. This demonstrates the advantage of Seper's multi-stage review process in enhancing performance.

Table 1: Result on the GQA and OK-VQA,*indicates an evaluation on a stratified sample of the test set, which may not be directly comparable to results on the full test set.

Model	GQA Acc.(%)	OK-VQA Acc.(%)
Supervised		
VisRep	59.2	-
LXMERT	60.0	-
NSM	63.0	-
CRF	72.1	-
KAT	-	54.4
RA-VQA	-	54.5
REVIVE	-	58.0
PromptCap	-	58.8
Zero-Shot		
BLIP-2	44.7	45.9
Flamingo	-	50.6
ViperGPT	48.1	51.9
CodeVQA	49.0	53.5
VisProg*	50.5	52.6
Seper(ours)	**51.3**	**54.1**

Figure 6 compares the accuracy of the baseline and Seper across various question types in GQA. With the integration of the Code Evaluation module and prompt tuning, Seper outperforms the baseline in several categories. Notably, the accuracy for "positionChoose" questions increased dramatically from 0.37 to 0.95, highlighting the effectiveness of the Code Evaluation module in handling complex positional relationships, as demonstrated in line 7 of Figure 2(a). Although some question types saw a slight decrease in accuracy compared to the baseline, a detailed analysis of these variations can be found in Appendix F.3.

5.3 Ablations

Table 2 presents an ablation study on the GQA dataset, evaluating the impact of various components on model performance. The baseline model achieves 48.1% accuracy. Incorporating code error

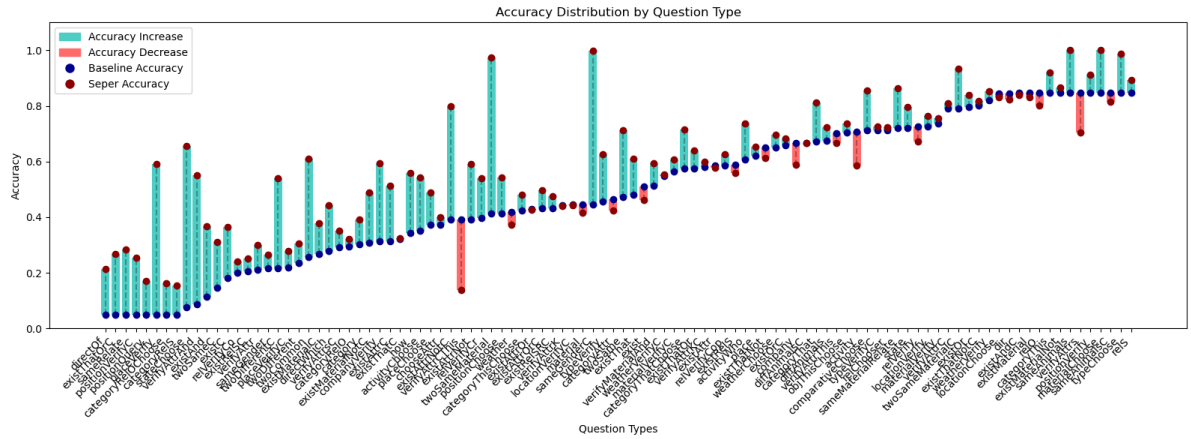

Figure 6: Accuracy distribution among different question types in GQA.

Table 2: Ablation results on the GQA and OK-VQA. ✓indicates the component is included in the configuration.

Reviewers				
Code Error	API	Logic	P-Tuning	Acc(%)
Baseline				48.1 (+0.0)
Two-component Configuration				
✓	✓			48.7 (+0.6)
✓		✓		49.3 (+1.2)
✓			✓	48.8 (+0.7)
	✓	✓		49.1 (+1.0)
	✓		✓	48.6 (+0.5)
		✓	✓	**49.8 (+1.7)**
Three-component Configuration				
✓	✓	✓		50.4 (+2.3)
✓	✓		✓	49.5 (+1.4)
✓		✓	✓	50.0 (+1.9)
	✓	✓	✓	**50.9 (+2.8)**
Full Configuration				
✓	✓	✓	✓	**51.3 (+3.2)**

and API reviewers improves accuracy to 48.7%, while adding logic reviewers boosts it further to 49.3%. Combining code error and logic reviewers results in 49.8% accuracy. The configuration with all three components—code error, API, and logic reviewers—achieves 50.4%, and adding prompt tuning further increases accuracy to 50.9%. The full configuration, including all components and P-Tuning, reaches the highest accuracy of 51.3%, representing a 3.2% improvement over the baseline. These results underscore the significance of multi-level reviews and prompt tuning in enhancing VQA model performance, with each additional component contributing to more refined and accurate outcomes.

Error Analysis. For the GQA dataset, we manually inspected 100 samples to analyze error sources. As shown in Figure 7, our method outperforms Viper by eliminating basic code errors, significantly reducing logical errors, and effectively identifying API-related issues in models.

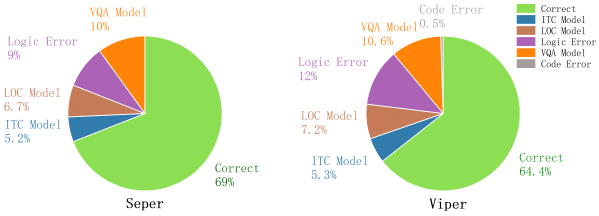

Figure 7: Error analysis of Seper and Viper.

6 Conclusion

We introduced the Seper framework, an iterative approach to visual question answering that integrates multi-level reviews and innovative evaluation stages with distinct reviewers. We also proposed a supervised prompt template tuning method to automatically optimize prompts for code reviewers. Our approach significantly outperforms existing programming-driven reasoning methods, addressing API limitations and enhancing logic, while showcasing the potential of LLMs as effective code reviewers. However, the reliance on a closed-source ChatGPT for reviews has increased costs, and we did not fully address the API limitations or enhance it to resolve associated issues. The logic review process also lacks the insights necessary for generating improved solutions. Future work will focus on reducing costs by leveraging open-source language models.

7 Limitations

In this study, our methodology did not show a significant advantage over existing programming-driven reasoning approaches. Additionally, we acknowledge that the limitations of the API were not fully addressed, nor was the API enhanced to effectively tackle the underlying issues. As a result, the logic review was not robust enough to provide deeper insights, which are essential for generating more refined code solutions. Furthermore, the reliance on a closed-source ChatGPT for multiple review iterations led to relatively high costs, a factor that should be considered in future research.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2016. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 39–48.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, et al. 2023. T-eval: Evaluating the tool utilization capability step by step. *arXiv preprint arXiv:2312.14033*.

Zi-Yi Dou, Cheng-Fu Yang, Xueqing Wu, Kai-Wei Chang, and Nanyun Peng. 2024. Re-rest: Reflection-reinforced self-training for language agents. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15394–15411.

Tanmay Gupta and Aniruddha Kembhavi. 2022. Visual programming: Compositional visual reasoning without training.

Yushi Hu, Hang Hua, Zhengyuan Yang, Weijia Shi, Noah A Smith, and Jiebo Luo. 2023. Promptcap: Prompt-guided image captioning for vqa with gpt-3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2963–2975.

Drew A Hudson and Christopher D Manning. 2019. Gqa: A new dataset for real-world visual reasoning and compositional question answering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6700–6709.

Zaid Khan, Vijay Kumar BG, Samuel Schulter, Yun Fu, and Manmohan Chandraker. 2024. Self-training large language models for improved visual program synthesis with visual reinforcement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14344–14353.

Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic program repair. *IEEE Software*, 38(4):22–27.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. 2023. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*.

Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–312.

Kenneth Marino, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. 2019. Ok-vqa: A visual question answering benchmark requiring external knowledge. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 3195–3204.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation. *arXiv preprint arXiv:2306.09896*.

OpenAI. 2023. Chatgpt (march 14 version) [large language model]. https://chat.openai.com.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools.

Zhenwei Shao, Zhou Yu, Meng Wang, and Jun Yu. 2023. Prompting large language models with answer heuristics for knowledge-based visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14974–14983.

Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. 2023. Modular visual question answering via code generation.

Dóidac Surós, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning.

Yan Zeng, Xinsong Zhang, and Hang Li. 2021. Multi-grained vision language pre-training: Aligning texts with visual concepts. *arXiv preprint arXiv:2111.08276*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. *arXiv preprint arXiv:2305.04087*.

A Implemetntion Details

We set the temperature parameter of codellm to 0, the temperature parameter of the reviewer to 0.25, and the rest of the parameters remain as usual. Besides, we capped the maximum number of iterations at 3 to prevent infinite function iterations. For each dataset, we employed 16 in-context examples. The pre-trained model and parameter settings we use are shown below:

GroundingDINO. We use the implementation from the official GitHub repository. The threshold of box is 0.30 and the threshold of text is 0.25.

XVLM. We used the official implementation, specifically the version finetuned for retrieval on MSCOCO. In addition to this, we set the similarity threshold to 0.5. Meaning if the similarity between the image and the text is more than 0.5, the two are considered to match.

BLIP-2. We used the Flan-T5 XXL version from Huggingface. We set the beams in the generation parameter to 5.

B API Listings

We provide different API definitions for different datasets.

B.1 API for GQA

```
from PIL import Image
from vision_functions import find_in_image, simple_qa, verify_property, best_text_match

def bool_to_yesno(bool_answer: bool)->str:
    return "yes" if bool_answer else "no"


class ImagePatch:
    """A Python class containing a crop of an image centered around a particular object, as well as relevant information.
    Attributes
    ----------
    cropped_image : array_like
        An array-like of the cropped image taken from the original image.
    left : int
        An int describing the position of the left border of the crop's bounding box in the original image.
    lower : int
        An int describing the position of the bottom border of the crop's bounding box in the original image.
    right : int
        An int describing the position of the right border of the crop's bounding box in the original image.
    upper : int
        An int describing the position of the top border of the crop's bounding box in the original image.

    Methods
    -------
    find(object_name: str)->List[ImagePatch]
        Returns a list of new ImagePatch objects containing crops of the image centered around any objects found in the image matching the object_name.
    simple_query(question: str=None)->str
        Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
    exists(object_name: str)->bool
        Returns True if the object specified by object_name is found in the image, and False otherwise.
    verify_property(property: str)->bool
        Returns True if the property is met, and False otherwise.
    best_text_match(string1: str, string2: str)->str
        Returns the string that best matches the image.
    crop(left: int, lower: int, right: int, upper: int)->ImagePatch
        Returns a new ImagePatch object containing a crop of the image at the given coordinates.
    """

    def __init__(self, image, left: int=None, lower: int=None, right: int=None, upper: int=None):
        """Initializes an ImagePatch object by cropping the image at the given coordinates and stores the coordinates as attributes.
        If no coordinates are provided, the image is left unmodified, and the coordinates are set to the dimensions of the image.
        Parameters
        -------
        image : array_like
            An array-like of the original image.
        left : int
            An int describing the position of the left border of the crop's bounding box in the original image.
        lower : int
            An int describing the position of the bottom border of the crop's bounding box in the original image.
        right : int
            An int describing the position of the right border of the crop's bounding box in the original image.
        upper : int
```

```
            An int describing the position of the top border of the crop's bounding box in the original image.

        """
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2] # width
            self.upper = image.shape[1] # height
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.upper = upper
            self.right = right
            self.lower = lower

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

        self.horizontal_center = (self.left + self.right) / 2
        self.vertical_center = (self.lower + self.upper) / 2

    def find(self, object_name: str)->List["ImagePatch"]:
        """Returns a new ImagePatch object containing the crop of the image centered around the object specified by object_name.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        """
        return find_in_image(self.cropped_image, object_name)

    def simple_query(self, question: str=None)->str:
        """Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------

        >>> # Which kind of animal is not eating?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     animal_patches = image_patch.find("animal")
        >>>     for animal_patch in animal_patches:
        >>>         if not animal_patch.verify_property("animal", "eating"):
        >>>             return animal_patch.simple_query("What kind of animal is eating?") # crop would include eating so keep it in the query
        >>>     # If no animal is not eating, query the image directly
        >>>     return image_patch.simple_query("Which kind of animal is not eating?")

        >>> # What is in front of the horse?
        >>> # contains a relation (around, next to, on, near, on top of, in front of, behind, etc), so ask directly
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
        """
        return simple_qa(self.cropped_image, question)

    def exists(self, object_name: str)->bool:
        """Returns True if the object specified by object_name is found in the image, and False otherwise.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.

        Examples
        -------
        >>> # Are there both cakes and gummy bears in the photo?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     is_cake = image_patch.exists("cake")
        >>>     is_gummy_bear = image_patch.exists("gummy bear")
        >>>     return bool_to_yesno(is_cake and is_gummy_bear)
        """
        return len(self.find(object_name)) > 0

    def verify_property(self, object_name: str, property: str)->bool:
        """Returns True if the object possesses the property, and False otherwise.
```

```
        Differs from 'exists' in that it presupposes the existence of the object specified by object_name, instead checking whether the object possesses the property.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        property : str
            A string describing the property to be checked.

        Examples
        -------
        >>> # Do the letters have blue color?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     letters_patches = image_patch.find("letters")
        >>>     # Question assumes only one letter patch
        >>>     if len(letters_patches) == 0:
        >>>         # If no letters are found, query the image directly
        >>>         return image_patch.simple_query("Do the letters have blue color?")
        >>>     return bool_to_yesno(letters_patches[0].verify_property("letters", "blue"))
        """
        return verify_property(self.cropped_image, object_name, property)

    def best_text_match(self, option_list: List[str]) -> str:
        """Returns the string that best matches the image.
        Parameters
        -------
        option_list : str
            A list with the names of the different options
        prefix : str
            A string with the prefixes to append to the options

        Examples
        -------
        >>> # Is the cap gold or white?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     cap_patches = image_patch.find("cap")
        >>>     # Question assumes one cap patch
        >>>     if len(cap_patches) == 0:
        >>>         # If no cap is found, query the image directly
        >>>         return image_patch.simple_query("Is the cap gold or white?")
        >>>     return cap_patches[0].best_text_match(["gold", "white"])
        """
        return best_text_match(self.cropped_image, option_list)

    def crop(self, left: int, lower: int, right: int, upper: int)->"ImagePatch":
        """Returns a new ImagePatch cropped from the current ImagePatch.
        Parameters
        -------
        left : int
            The leftmost pixel of the cropped image.
        lower : int
            The lowest pixel of the cropped image.
        right : int
            The rightmost pixel of the cropped image.
        upper : int
            The uppermost pixel of the cropped image.
        -------
        """
        return ImagePatch(self.cropped_image, left, lower, right, upper)

    def overlaps_with(self, left, lower, right, upper):
        """Returns True if a crop with the given coordinates overlaps with this one,
        else False.
        Parameters
        ----------
        left : int
            the left border of the crop to be checked
        lower : int
            the lower border of the crop to be checked
        right : int
            the right border of the crop to be checked
        upper : int
            the upper border of the crop to be checked

        Returns
        -------
        bool
            True if a crop with the given coordinates overlaps with this one, else False
        """
        return self.left <= right and self.right >= left and self.lower <= upper and self.upper >= lower
```

Listing 2: API for GQA.

B.2 API for OK-VQA

```
from PIL import Image
from vision_functions import obtain_query_response_from_image
from nlp_functions import llm_query, process_guesses

def llm_query(question: str)->str:
    '''Answers a text question using GPT-3. The input question is always a formatted string with a variable in it.

    Parameters
    ----------
    question: str
        the text question to ask. Must not contain any reference to 'the image' or 'the photo', etc.
    '''
    return llm_query(question)

def process_guesses(question: str, guesses: List[str])->str:
    '''Processes a list of guesses for an answer to a question and returns the best answer.'''
    return process_guesses(question, guesses)

class ImagePatch:
    def __init__(self, image, left: int=None, lower: int=None, right: int=None, upper: int=None):
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2]
            self.upper = image.shape[1]
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.lower = lower
            self.right = right
            self.upper = upper

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

    def simple_query(self, query: str):
        """Answer basic queries about the image patch.
        Parameters


        ----------
        query: str
            the simple query about the image patch in the form of a question

        Returns
        -------
        str
            a guess for the answer to the question
        """
        answer = obtain_query_response_from_image(self.cropped_image, query)
        return answer
```

Listing 3: API for OK-VQA.

C Prompt For Reviewer

C.1 Prompt for vqa type APIs

Prompt for result asscessment

```
prompt = f'''You are a reviewer, your task is to work hand in hand with another AI to solve a question about a certain picture,
Your task is to check whether the answer meets the following two criteria and give a judge result with reasoning process:
1.Is it possible that this answer is the correct answer? Output Sure, possible, or impossible.
2.Is it this answer clear enough?Output yes or no.


Examples as follows:
expected_result: material_type
object: airplane
query: What is the material?
answer: plastic
judge:
- Is it possible that this answer is the correct answer?:Possible. Plastic is not a common material for airplanes,unless it is a toy airplane.
- Is it this answer clear enough?:Yes.It is clear enough.

expected_result:skateboard_kind
object:skateboard
query: What is the this?
answer:kid
judge:
- Is it possible that this answer is the correct answer?: Impossible.Kid is not a kind of skateboard.
- Is it this answer clear enough?:Yes.It is clear enough.

expected_result: horse_kind
object:horse.
query:What kind of horse is it?
answer:horse.
judge:
- Is it possible that this answer is the correct answer?:Sure.
- Is it this answer clear enough?:No.It is vague.

expected_result: person
object:clothing
query:Who will love this suit?
answer:No one.
judge:
```

```
- Is it possible that this answer is the correct answer?:Impossible. The answer should be occupation, gender, and everything else that is used to describe people.
- Is it this answer clear enough?:No.It is vague.


(Examples finished)

expected_result:{expected_result}
object:{object}
query:{query}
answer:{model_output}
judge:
'''
```

Listing 4: Prompt for vqa type APIs reviewer on resualt asscessment phase.

Prompt for review generation

```
prompt = f'''
You are a reviewer. You are reviewing a code execution flow, and there is an error in an API call statement. Another large language model has already identified the error and its cause. Please provide suggestions for modification.

Here are some suggestion for you to provide:
1.Replace the query parameter of original function. Use a more informative question.
2.Use other function to answer the original visual question.
For example,if there are options given or implied in the question,you can use best_text_match with giving options.


Some of the api's provided to you may be limited by the model resulting in less than expected results, these api's include:
find(object_name): different names for the same object may lead to different results.
simple_query(question): for different questions with the same semantics, may lead to different results.
verify_property(object_name,property): for different object_name or property with same semantics, may lead to different results.
best_text_match(option_list): Returns the string that best matches the image.


Examples as follows:
(Example 1)
API statement: skateboard_kind = skateboard.simple_query('What is this?')
Actual Result: Kid.
Judge from last step: Kid is not a kind of skateboard.
Review: The original query parameter 'What is this?' is too simple. The code should use a more informative information.

(Example 2)
API statement: horse_kind = horse_patch.simple_query("What kind of horse is it?")
Actual Result: horse
Judge from last step: The answer is too vague.
Review: The original query parameter 'What kind of horse is it?' is specific enough. But the answer is still too vague. Try to use "best_text_match" and giving some horse kind as options.

(Example finished)

API statement:{code}
Actual Result:{result}
Judge from last step:{judge}
Review:
'''
```

Listing 5: Prompt for vqa type APIs reviewer on review generation phase.

C.2 Prompt for location type API

Prompt for result asscessment

```
prompt = f'''You are an expert in textual reasoning, and you are good at reasoning from textual information to get answers to questions.
Now here's a image with an AI-generated description of that image(not 100% correct), and the query designed for that image.
You need to speculate on the answer to another question based on the original query and the description.


Here are some examples:
(Example1)
ImageCaption: The image depict an aux and bar.
OriginalQuery:The color of the top of the aux is green or blue?
Question1: Does foo exists in the image?
Answer: Although the existence of foo is not mentioned in the caption, its presence is implied in the query, because if foo does not exist, the question becomes unanswerable.Yes,it exists.


(Example2)
ImageCaption: There is an bar in the left of the foo.
OriginalQuery: The color of the aux of foo is green or blue?
```

```
Question: Does aux exists in the image?
Answer: If aux does not exist, there will be no answers to the query. Althought it does not mention in the caption,the aux exists.



(Example3)
ImageCaption: There are an foo and an aux in the image.
OriginalQuery: Can you see bar or aux in the image?
Question: Does bar exists in the image?
Answer: No,it does not exist.Based on the image caption and there is no affirmation of its existence in the query.So bar does not exist in the image.

(Example finished)
ImageCaption:{(caption)}
OriginalQuery:{(query)}
Question: Based on the the query "{(query)}" and image caption,Does "{(object)}" exists in the image?

Remeber:
- Learn from previous examples.
- The information in the query more important than the information in the caption,because caption is generated by ai that not 100 percent correct.
Output:
{
    'existence':'Sure' or 'Impossible',
}
'''
```

Listing 6: Prompt for location type APIs reviewer on resualt asscessment phase.

Prompt for review generation

```
prompt = f'''
You are a reviewer. You are reviewing a code execution flow, and there is an error in an API call statement. Another large language model has already identified the error and its cause. Please provide suggestions for modification.

Here are some suggestion for you to provide:
1.Replace the object parameter in function. Try use more other synonym object name replace the original object name.
2.Use other function as error handler while you can not find tagret object.

Some of the api's provided to you may be limited by the model resulting in less than expected results, these api's include:
find(object_name): different names for the same object may lead to different results.
simple_query(question): for different questions with the same semantics, may lead to different results.
verify_property(object_name,property): for different object_name or property with same semantics, may lead to different results.
best_text_match(option_list): Returns the string that best matches the image.

Previous code fail to find the {object} which should be in the image based on the {query}. Try to give some suggestions.

Examples as follows:
(Example1)
Query: What do the vase and the paper have in common?
Object should be found: paper
Review: Try use more other synonym object name replace 'paper' in the paper_patches = image_patch.find("paper") for increase chance to find paper in the image.

(Example2)
Query: How are the vehicles to the right of the walking person that is walking on the sidewalk called?
Object should be found: vehicles
Review: Explore more parameters.Replace the vehicle with 'car' or other words of the same semantics.

(Example3)
Query: What is the color of the carpet?
Object should be found: carpet
Review: Add simple_query function for error handler.

(Examples finished)
Query:{query}
Object should be found:{object}
Review
'''
```

Listing 7: Prompt for location type APIs reviewer on review generation phase.

C.3 Prompt For Logic Reviewer

Prompt for result asscessment of answer error

```
f'''
Suppose you are a checker, and you are working on a visual question task with another AI, who will give his answer to the visual question,
and your task is to determine whether the answer conforms to the form of the question
Output 'Accept' or 'Refuse' with your thinking process.
Learn from examples.
Examples as follows:
(Example1)
Visual Question:What kind of horse is it?
```

```
AI Answer:horse.
Judge: Refuse.The answer is too vague.

(Example2)
Visual Question:Who will love this suit?
AI Answer:No one.
Judge: Refuse. The answer should be occupation, gender, and everything else that is used to describe people.
(Example finished)

Remeber:

- The AI answer does not need to be extremely precise, but rather within a certain range. For example, the answer to the question Where is here? can be "street" or "parking lot", rather than a specific location.
- Do not give 'correct answer' of the question. Becuase you can not give correct answer for an unseen image.

Visual Question:{{(question)}}
AI Answer:{{(answer)}}
Judge:
'''
```

Listing 8: Prompt for logic reviewer on result asscessment phase of answer error.

Prompt for review generation of answer error

```
Prompt = f'''
I want you to serve as a senior code engineer. I will provide you with the code,including its intermediate states and return values.
You should be able to trace back from the return values to identify the root cause statements that lead to these return values.

Examples as follows:
(Example1)
code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    # Question assumes only one person patch
    if len(person_patches) == 0:
        # If no person is found, query the image directly
        return image_patch.simple_query("What does the person to the left of the helmet hold?")
    person_patch = person_patches[0]
    helmet_patches = image_patch.find("helmet")
    # Question assumes only one helmet patch
    if len(helmet_patches) == 0:
        return "no"
    for helmet_patch in helmet_patches:
        if helmet_patch.horizontal_center > person_patch.horizontal_center:
            return helmet_patch.simple_query("What does the person to the left of the helmet hold?")
    return "no"
intermediate_result:
    'image_patch': 'a baseball game',
    'person_patches': 'lens:4',
    'len(person_patches)': '4',
    'if len(person_patches) == 0?': 'False',
    'person_patch': 'a baseball player holding a bat',
    'helmet_patches': 'lens:3',
    'len(helmet_patches)': '3',
    'if len(helmet_patches) == 0?': 'False',
    'helmet_patch.horizontal_center': '333.5',
    'person_patch.horizontal_center': '230.0',
    'if helmet_patch.horizontal_center > person_patch.horizontal_center?': 'True'
return value: right

Return value traceability analysis The root line is the "helmet_patch.simple_query("What does the person to the left of the helmet hold?")" line.The reason is because horizontal_center of one of the helmet_patch bigger than that of person_patch.

(Example2)
# Which color is the device the monitor is to the right of?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    # Question assumes only one monitor patch
    if len(monitor_patches) == 0:
        # If no monitor is found, query the image directly
        return image_patch.simple_query("Which color is the device the monitor is to the right of?")
    for monitor_patch in monitor_patches:
        device_patches = image_patch.find("device")
        for device_patch in device_patches:
            if device_patch.horizontal_center < monitor_patch.horizontal_cent er:
                return device_patch.simple_query("Which color is the device?")
    return "unknown"

intermediate_result:
    "image_patch": "a computer monitor and a laptop computer on a desk",
      "monitor_patches":"lens:2",
      "len(monitor_patches)":"2",
      "if len(monitor_patches) == 0?":"False",
```

```
      "device_patches":"lens:4",
      "device_patch.horizontal_center":"285.0",
      "monitor_patch.horizontal_center":"319.0",
      "if device_patch.horizontal_center < monitor_patch.horizontal_center?": "True"
return value: right

Return value traceability analysis: The root line is the "return device_patch.simple_query("Which color is the device?")" line.The reason is
because horizontal_center of one of the device_patch smaller than that of monitor_patch.
(Examples finished)

The first code is:{code}
The intermediate result:{inter_res}
return_value:{return_value}
Return value traceability analysis:
'''
```

Listing 9: Prompt for logic reviewer on review generation phase of answer error.

Prompt for result asscessment and review generation phase of potential error

```
Prompt = f'''
I want you to serve as a senior code engineer. I will provide you with the code,including its intermediate states and return values.
First.Summarize the content of the intermediate variables. The term 'image_patch' represents the content of the image; the number of patches indicates how many of a certain object are present in the image.
Second.Analyse from questions code and intermediate results. Check if there are errors in the loops and selection structures in this code, or if there is a lack of loops and selection structures.

Examples as follows:
(Example1)
code:
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    person_patches = image_patch.find("person")
    # Question assumes only one person patch
    if len(person_patches) == 0:
        # If no person is found, query the image directly
        return image_patch.simple_query("What does the person to the left of the helmet hold?")
    person_patch = person_patches[0]
    helmet_patches = image_patch.find("helmet")
    # Question assumes only one helmet patch
    if len(helmet_patches) == 0:
        return "no"
    for helmet_patch in helmet_patches:
        if helmet_patch.horizontal_center > person_patch.horizontal_center:
            return helmet_patch.simple_query("What does the person to the left of the helmet hold?")
    return "no"
intermediate_result:
    'image_patch': 'a baseball game',
    'person_patches': 'lens:4',
    'len(person_patches)': '4',
    'if len(person_patches) == 0?': 'False',
    'person_patch': 'a baseball player holding a bat',
    'helmet_patches': 'lens:3',
    'len(helmet_patches)': '3',
    'if len(helmet_patches) == 0?': 'False',
    'helmet_patch.horizontal_center': '333.5',
    'person_patch.horizontal_center': '230.0',
    'if helmet_patch.horizontal_center > person_patch.horizontal_center?': 'True'
return value: right


Summary of intermediate variables: The image shows a baseball game,there are 4 persons in the image and 3 helmet.
Possible problems: The code assume that there are only one person patch,but actually there are 4 persons. The code is missing loop structure for handling of different situations when the number of people is greater than 1.


(Example2)
# Which color is the device the monitor is to the right of?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    monitor_patches = image_patch.find("monitor")
    # Question assumes only one monitor patch
    if len(monitor_patches) == 0:
        # If no monitor is found, query the image directly
        return image_patch.simple_query("Which color is the device the monitor is to the right of?")
    for monitor_patch in monitor_patches:
        device_patches = image_patch.find("device")
        for device_patch in device_patches:
            if device_patch.horizontal_center < monitor_patch.horizontal_cent er:
                return device_patch.
```

```
    simple_query("Which color is the
    device?")
    return "unknown"

intermediate_result:
    "image_patch": "a computer monitor
    and a laptop computer on a desk",
      "monitor_patches":"lens:2",
      "len(monitor_patches)":"2",
      "if len(monitor_patches) == 0?":"
    False",
      "device_patches":"lens:4",
      "device_patch.horizontal_center
    ":"285.0",
      "monitor_patch.horizontal_center":
     "319.0",
      "if device_patch.horizontal_center
     < monitor_patch.horizontal_center
    ?": "True"
return value: right


Summary of intermediate variables: The
    image shows a computer monitor and a
     laptop computer on a desk,There are
     2 monitor and 4 devices.
Possible problems: Although there are
    multiple Monitors in the problem,
    for the device, the judgment
    condition is already satisfied at
    the first device, and the answer is
    returned without obtaining the
    colors of other devices that meet
    the condition later. The code is
    lack of loop structure.

(Example3)
# In which part of the photo is the
    bridge, the bottom or the top?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    bridge_patches = image_patch.find("
    bridge")
    # Question assumes only one bridge
    patch
    if len(bridge_patches) == 0:
        # If no bridge is found, query
    the image directly
        return image_patch.simple_query
    ("In which part of the photo is the
    bridge, the bottom or the top?")
    if bridge_patches[0].vertical_center
     > image_patch.vertical_center:
        return "bottom"
    else:
        return "top"

intermediate_result:
image_patch:A bridge cross the river.
bridge_patches:lens:1
if bridge_patches[0].vertical_center >
    image_patch.vertical_center? True
...

Summary of intermediate variables: The
    image shows 1 bridge cross the river
    .

Possible problems: The judgment of up
    and down in the code is reversed.
    The patch with a larger vertical
    center should be placed above, and
    similarly, the patch with a larger
    horizontal center should be placed
    on the right side of the image. The
    code snippet uses incorrect
    attributes in the selection
    structure.

(Examples finished)

Your return format should be as follows:
Summary of intermediate variables:
    Included length of list and text
    description.
Possible problems: The possible problems
     hide in code based on query and
    summary of intermediate variables.

The code is:{code}
The intermediate result:{inter_res}
return_value:{return_value}
Summary of intermediate variables:
'''
```

Listing 10: Prompt for result asscessment and review generation phase of potential error.

C.4 Prompt For Code Error Reviewer

```
f'''Error in execution of statement {
    code}:{error}.
Here are 2 error types of code execution
    :
Syntax Error: Syntax errors are the most
     basic type of error, typically
    occurring when the written code does
     not adhere to the syntax rules of
    the programming language. For
    example, missing semicolons,
    mismatched parentheses, or spelling
    errors in variable names.
Runtime Error: Runtime errors occur
    during program execution and are
    often caused by invalid input data,
    memory overflow, or division by zero
    . Such errors can potentially lead
    to program crashes.

What is the type of {error}? If  i t s  a
     syntax error, please provide the
    corrected code directly. If  i t s  a
     runtime error, add exception
    handling statements at the
    corresponding location in the
    original code.

fixed_code:'''
```

Listing 11: Prompt for code error reviewer on review generation phase.

D Prompt For Rationale Generation

```
f'''You are currently assisting another
    artificial intelligence in
    completing a Visual Question
    Answering (VQA) task. You need to
    break down the solution process for
```

```
    the following question into 1-5 sub-
    steps, allowing the AI to invoke
    tools for each sub-step and
    ultimately arrive at an answer.
    Example as follows:
(Example1)
Query: Where is this?
1. Identify prominent landmarks or
    features in the image that can help
    determine the location.
2. Analyze the surroundings of the
    landmarks or features to gather
    contextual information that could
    indicate the location.
3. Based on  contextual information,
    infer the most likely location
    depicted in the image.

(Example2)
Query: What kind of car is near the pond
    ?
1. Find the pond in the image.
2. Determine which cars are near the
    pond based on their positions
    relative to the pond's location.
3. Identify the type or model of the car
    .
(Examples finished)

Query:{query}
'''
```

Listing 12: Prompt for rationale generation in code generation satge.

E Prompt For Supervised Prompt Tuning

```
f'''As a professional prompt engineer
    specializing in prompt optimization
    based on review feedback, I'll help
    you create a template for analyzing
    reports and optimizing prompts:

As an experienced Prompt Optimization
    Specialist, please carefully analyze
     the following Review Report and
    optimize the Prompt Template
    accordingly.

Review Report:
{Review Report}

Original Prompt Template:
{Prompt Template}

Please follow these steps for analysis
    and optimization:

Analyze key issues and improvement
    suggestions mentioned in the Review
    Report
Based on this feedback, evaluate the
    original prompt in the following
    aspects:
Clarity and specificity of instructions
Accuracy of role definition
Output format standardization
Reasonableness of constraints
Helpfulness of examples
Propose specific optimization
    suggestions
Generate an optimized version of the
    prompt
Optimization focus:

Maintain the core functionality of the
    prompt
Enhance instruction executability
Improve output consistency
Supplement necessary context
Perfect constraints
Improve example explanations
Please provide:

Problem analysis and optimization
    approach
List of recommended modifications
Complete optimized prompt
I will optimize according to the
    original prompt's core intent while
    addressing issues raised in the
    review.'''
```

Listing 13: Prompt for supervised prompt tuning.

F Additional Results

F.1 GQA Validation Result

We also evluate Seper on GQA Validation set, which contains 2000 balanced question typed samples. As shown in 4, our method is two percent better than current visual programming method.

F.2 Prompt Tuning Experiment

We selected the GQA training dataset to tune the prompt templates in Seper. During each tuning iteration, we categorize the samples into four types: 1) **Invalid**: Both the previous and current prompt templates result in incorrect answers for these samples. 2) **Valid**: The previous prompt templates result in incorrect answers, but the current iteration leads to correct answers. 3) **Toxic**: The previous prompt templates result in correct answers, but the current iteration leads to incorrect answers. 4) **Other**: Both the previous and current prompt templates result in correct answers.

As shown in Table 3, the prompt tuning process significantly reduces the number of toxic samples, with the 5-th iteration leading to a modest overall increase in accuracy from 55.3% to 57.2%. Consequently, we used the prompt templates in the 5-th iteration for subsequent experiments.

F.3 Accuracy Analysis across Question Types

8 As shown in Figure x, we categorized the samples in GQA according to the variation in accuracy between Seper and the baseline. The five categories are: 0 to 50% increase, 50 to 100% increase, over

Table 3: Accuracy results on the GQA dataset across iterations in supervised prompt tuning.

Iteration	Other	Valid	Invalid	Toxic	Acc
0	-	-	-	-	55.3
1	44.2	10.9	33.8	11.1	55.1
3	44.8	10.7	34.0	10.5	55.5
5	46.2	11.0	33.7	**9.1**	**57.2**
10	46.0	10.8	33.9	9.3	56.8

Table 4: Result On GQA Validation.

	Accuracy(%)
ViperGPT	55. 2
CodeVQA	55. 3
Seper	57. 8

100% increase, 0 to 10% decrease, and 10 to 100% decrease. Nearly 70% of the samples achieved accuracy improvements, with 17.6% of the question categories seeing over 100% increases in accuracy, demonstrating the effectiveness of the Seper framework. However, 30% of the samples still saw accuracy declines, and around 3.9% of the question categories had accuracy drops of over 10%, which include the following three categories:

Type	Baseline (%)	Seper (%)
activity	70.71	58.49 (-17.27%)
verifyAttrCThis	39.12	13.83 (-64.65%)
verifyAttrs	84.74	70.50 (-16.80%)

Table 5: Accuracy Reduction for Problematic Categories

Through sampling the examples for these three question categories, we found that these types of questions are more prone to **overlooking global information** issues.

```
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    return image_patch.simple_query("
    What is the man doing?")
```

Listing 14: Code generated by Viper.

For example, for the 'activity' category question "What is the man doing", Viper's generated code is as shown in Listing 14. Seper's generated code is as shown in Listing 15. The difference is that Seper generated more detailed steps - it first locates the specific person, and then judges their action. However, this also led to a lack of global

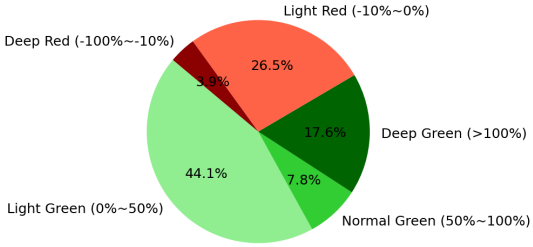

Figure 8: Accuracy Variation between Seper and Baseline in GQA Samples.

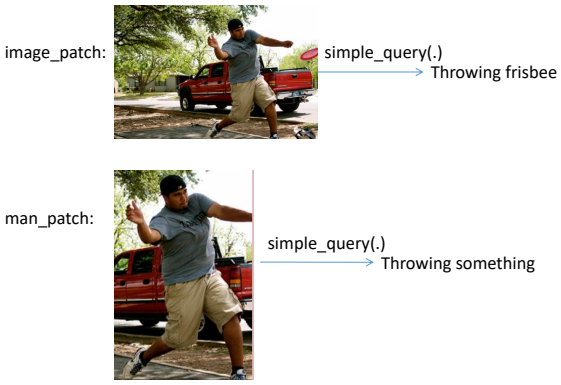

Figure 9: Accuracy Variation between Seper and Baseline in GQA Samples.

information in the judgment, as shown in Figure 9. In Seper's case, since it was unable to perceive the existence of the Frisbee, it was unable to answer the question correctly.

We considered emphasizing the role of global information in the review process, such as using an image-captioning model to generate descriptions of images and enforcing these descriptions as reference information for reviewers during their evaluation. However, we found that this approach could also introduce potential biases due to inaccuracies or distortions in the captions generated by the image-captioning model, thereby negatively impacting the review process. As a result, we ultimately decided to retain our current approach. In future work, we will explore more effective ways to incorporate global information into the review process while minimizing hallucinations or errors.

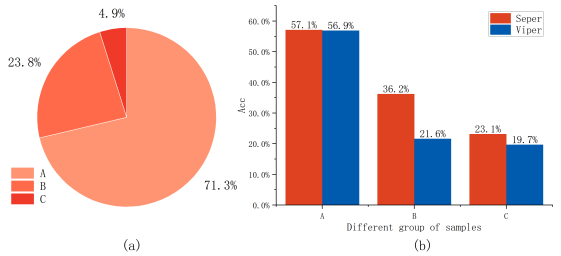

Figure 10: (a) Proportion of samples with different iteration counts for Seper in GQA, we divided samples with different iteration counts into three groups: A, B, and C. These three groups underwent 1, 2, and 3 iterations respectively. (b) Accuracy of Seper and Viper in groups A, B, and C samples.

```python
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    # subtask1: find the man
    man_patch = image_patch.find("man")
    # subtask2: Identify the man's activity
    return man_patch.simple_query("What is the man doing?")
```

Listing 15: Code generated by Seper.

F.4 Group Analysis

Based on Seper's iteration counts, we categorized the GQA validation samples into three groups: A, B, and C, corresponding to 1, 2, and 3 iterations, respectively. Viper, in contrast, can be regarded as having executed only a single iteration across all sample types.

Figure 10(a) illustrates the distribution of samples by iteration count, with approximately one-fourth undergoing a second iteration by Seper, and only about 5% necessitating a third. Figure 10(b) contrasts Seper and ViperGPT's performance, showing a declining accuracy trend across the groups. This trend highlights significant difficulty variation, with Group A being the easiest and Group C the most challenging. In Group A, where only one iteration was required, the self-correcting framework shows minimal deviation from ViperGPT. However, in the more complex Group B, the self-correcting framework surpasses ViperGPT by approximately 15% in accuracy. Even in Group C, the most difficult, Seper demonstrates a clear advantage, indicating that Seper's ability to address complex problems is substantially enhanced by self-correcting techniques.

As shown in Table 6, we conducted a comparative analysis of the average time consumption per sample between Seper and Viper. The results indicate that our method requires an average of 3.12 seconds per sample, while Viper completes the task in 2.37 seconds, representing a 31.6% increase in processing time. This elevated time consumption is primarily attributed to the multiple rounds of interaction with the Large Language Model required for code generation in our approach. In future work, we aim to optimize these temporal costs through various strategies, such as minimizing prompt length or deploying local LLM implementations.

Metric	Viper	Seper
Avg Time	2.37s	3.12s
Iteration Distribution	100% × 1	71% × 1 24% × 2 5% × 3

Table 6: Time Consume Comparison Between Viper and Seper.