VDebugger: Harnessing Execution Feedback for Debugging Visual Programs

Xueqing Wu, Zongyu Lin, Songyan Zhao, Te-Lin Wu, Pan Lu, Nanyun Peng, Kai-Wei Chang
University of California, Los-Angeles
`{xueqing.wu,linzongy21,songyan,telinwu,pan.lu,violetpeng,kwchang}@cs.ucla.edu`
`https://shirley-wu.github.io/vdebugger/index.html`

Abstract

Visual programs are executable code generated by large language models to address visual reasoning problems. They decompose complex questions into multiple reasoning steps and invoke specialized models for each step to solve the problems. However, these programs are prone to logic errors, with our preliminary evaluation showing that 58% of the total errors are caused by program logic errors. Debugging complex visual programs remains a major bottleneck for visual reasoning. To address this, we introduce **VDebugger**, a novel critic-refiner framework trained to localize and debug visual programs by tracking execution step by step. VDebugger identifies and corrects program errors leveraging detailed execution feedback, improving interpretability and accuracy. The training data is generated through an automated pipeline that injects errors into correct visual programs using a novel mask-best decoding technique. Evaluations on six datasets demonstrate VDebugger's effectiveness, showing performance improvements of up to 3.2% in downstream task accuracy. Further studies show VDebugger's ability to generalize to unseen tasks, bringing a notable improvement of 2.3% on the unseen COVR task. Code, data and models are made publicly available at `https://github.com/shirley-wu/vdebugger/`.

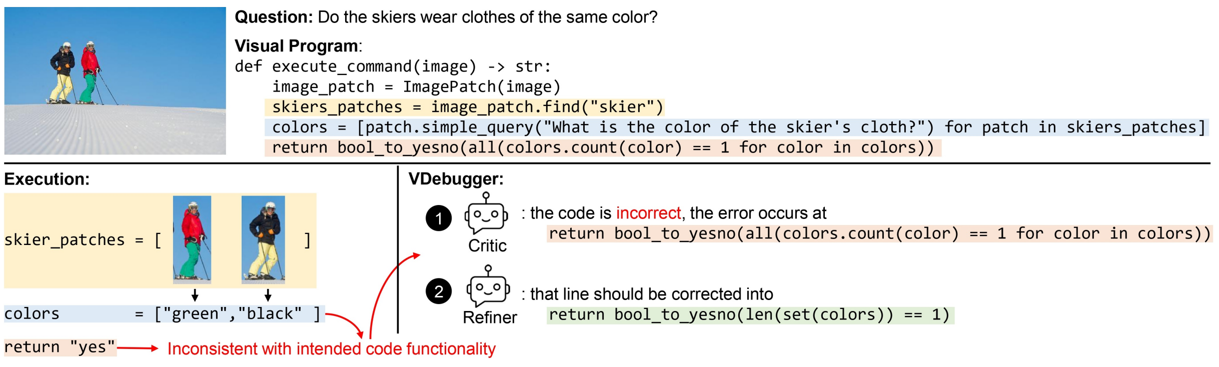
1 Introduction

Complex visual reasoning is a crucial yet challenging problem that often requires compositionally synthesizing multiple reasoning steps before drawing the final conclusion. For example, to answer the visual question in Figure 1: *"Do the skiers wear jackets of the same color?"*, one must identify all skiers, determine the colors of their jackets, and assess whether the colors are the same. End-to-end vision-language models (VLMs) excel at individual tasks such as object detection (Li et al., 2022b) and visual instruction following (Liu et al., 2023). However, they struggle to generalize to complex tasks requiring compositional reasoning and inherently lack interpretability (Surís et al., 2023; Yüksekgönül et al., 2023; Kamath et al., 2023, 2024).

To devise a more interpretable and generalizable reasoning process, a recent approach leverages the code generation capabilities of large language models (LLMs) to generate "*visual programs*" (Surís et al., 2023; Gupta and Kembhavi, 2023). As shown in Figure 1, the visual program decomposes a complex question into a sequence of programmatically executable steps. During execution, the visual program invokes foundational specialist models to perform visual perception and synthesize the results of each reasoning step into the final answer. The inherent compositionality of programs allows this approach to perform compositional reasoning while ensuring generalization and interpretability.

Nonetheless, program errors become a bottleneck for this approach, accounting for 58% of total errors as shown in our evaluation. Following the advancement of LLM self-refinement in general-domain code generation (Chen et al., 2023) and LLM agents (Madaan et al., 2023; Shinn et al., 2023), recent work leverages zero-shot prompting of LLMs to debug visual programs based on some given feedback (Stanic et al., 2024; Gao et al., 2023). However, their feedback typically focuses on limited aspects such as compilation errors. Furthermore, the zero-shot prompting technique is less effective for self-critique and self-correction of programs, especially for smaller LLMs, as shown in recent work (Luo et al., 2023; Tian et al., 2024; Lan et al., 2024; Jiang et al., 2024).

In this work, we propose **VDebugger**, a tool trained to debug visual programs by tracking their execution step by step. As shown in Figure 1, VDebugger takes as input the execution states at each step, including the code being executed and the resulting change of variable values. Based on such information, the **critic** identifies fine-grain program


Question: Do the skiers wear clothes of the same color?
Visual Program:
Execution:
VDebugger:
1
Critic
: the code is incorrect, the error occurs at
2
Refiner
: that line should be corrected into
Inconsistent with intended code functionality

```
def execute_command(image) -> str:
    image_patch = ImagePatch(image)
    skiers_patches = image_patch.find("skier")
    colors = [patch.simple_query("What is the color of the skier's cloth?") for patch in skiers_patches]
    return bool_to_yesno(all(colors.count(color) == 1 for color in colors))
```

```
skier_patches = [                ]
colors        = ["green","black" ]
return "yes"
```

```
return bool_to_yesno(all(colors.count(color) == 1 for color in colors))
```

```
return bool_to_yesno(len(set(colors)) == 1)
```

Figure 1: **Overview of visual programming and VDebugger. Above**: the *visual program* invokes APIs to answer the input question. Each involved API (e.g. `find`) is implemented with a specialized foundation VLM (e.g. object detection model). **Below:** VDebugger debugs the visual program by inspecting the execution process. In this example, the `colors` variable represents the colors of all skier's jackets and contains two values, but the return value `"yes"` suggests that all skiers wear jackets of the same color. Catching this discrepancy, the critique identifies that the last line of the program is incorrect, and the refiner rewrites that line into the correct code.

errors down to the line, and the **refiner** rewrites the error-inducing line to correct the program.

To train the VDebugger, we devise an automated pipeline to collect training data at scale. For visual question answering task, speficially, we prompt an LLM to generate visual programs for the input questions from existing datasets. The programs whose execution results match the ground truth answers are taken as correct programs. In order to create incorrect programs, we inject errors by resampling parts of these originally correct programs and thereby generating modifications that affect the execution results. The VDebugger thus learns to identify and correct visual program errors utilizing these automatically curated positive and negative program pairs. In particular, we propose a *mask-best sampling* algorithm that increases the success rate of error injection by up to 10 times compared to greedy decoding. Eventually, we generate a total of $47.7k$ program pairs for VDebugger training.

We evaluate VDebugger on a total of 6 datasets covering various forms of visual question answering (Hudson and Manning, 2019; Acharya et al., 2019; Suhr et al., 2019) and visual grounding (Yu et al., 2016). Based on both CodeLlama-7B and CodeLlama-13B, VDebugger consistently improves the performance by up to 3.2% accuracy. VDebugger can also be employed to debug visual programs generated by proprietary code generation models such as GPT-3.5 and brings notable gains of up to 4.9% accuracy. By jointly training VDebugger on all six datasets with different task forms, VDebugger demonstrates generalization capability capable of handling unseen tasks such as question answering based on variable number of images (Bogin et al., 2021).

In summary, our contributions are three-folds: (1) We propose VDebugger, a novel framework for debugging visual programs capable of reasoning over execution process and performing explainable debugging; (2) We develop a pipeline to automatically generate large-scale training datasets including $47.7k$ program pairs; (3) Our VDebugger trained on top of 7B and 13B LLMs achieves significant improvements across 6 datasets and can generalize to unseen scenarios.

2 Related Work

Visual reasoning. The large-scale pre-training of VLMs has demonstrated significant success (Radford et al., 2021). When fine-tuned, these models can effectively adapt to specific tasks such as instruction following (Liu et al., 2023; Bai et al., 2023), visual question answering (Li et al., 2022a, 2023), and object detection (Li et al., 2022b). Despite their impressive performance on these individual tasks, VLMs still struggle with compositional reasoning that requires composing multiple reasoning steps (Hudson and Manning, 2019; Suhr et al., 2019; Bogin et al., 2021). Visual programming addresses this problem (Surís et al., 2023; Gupta and Kembhavi, 2023) by generating executable programs that decompose the question into multiple reasoning steps and invoke specialized VLMs for each step. However, the program errors in the generated code become a bottleneck of this approach.

Self-debugging and self-refinement. We present a comprehensive comparison between this work

	Method	Feedback			Training
		Error	Unit-test	Execution states	
LLM self-refinement	SelfRefine (Madaan et al., 2023)	N/A	N/A	✗	✗
	Reflexion (Shinn et al., 2023)	N/A	N/A	✗	✗
	Refiner (Paul et al., 2024)	N/A	N/A	Per step	✓
General-domain code debugging	SelfDebug (Chen et al., 2023)	✓	✓	✗	✗
	Jiang et al. (2024)	✓	✓	✗	✓
	LDB (Zhong et al., 2024)	✓	✓	Per block	✗
Visual program debugging	Stanic et al. (2024)	✓	N/A	✗	✗
	Define (Gao et al., 2023)	✓	N/A	Per block	✗
	VDebugger (Ours)	✓	N/A	Per step	✓

Table 1: **Comparison against existing work.** The distinction of our VDebugger against existing work are mainly two-folds: (1) we utilize a more fine-grained feedback information of step-wise execution states, and (2) we automatically collect large-scale training data for model training.

and existing work for self-debugging and self-refinement in Table 1. Existing techniques have explored LLM self-refinement for reasoning, decision making, and language generation tasks (Madaan et al., 2023; Shinn et al., 2023). These work largely relies on self-generated feedback, which is less effective especially for code-related tasks (Huang et al., 2023). Paul et al. (2024) tracks the intermediate states step-by-step during mathematical reasoning, which is shown to be more beneficial. For general-domain code generation, self-debugging can leverage more reliable feedback information such as execution error and pass/fail results of unit-tests (Chen et al., 2023). Zhong et al. (2024) further divides a program into multiple code blocks and takes the execution states before and after each block as feedback. However, visual programs do not have unit-tests available. Existing work for debugging visual programs either use execution error (Stanic et al., 2024) or block-wise execution states (Gao et al., 2023) as feedback, which may not be fine-grained enough to cover all potential errors. Our feedback is more informative by tracking execution states step-by-step. Another trend of recent work is to generate synthetic data for training self-debugging models, which is particularly helpful for smaller LLMs (Paul et al., 2024; Jiang et al., 2024). We follow this trend to collect large-scale training sets for training VDebugger.

3 VDebugger Framework

VDebugger consists of two components, a **critic** and a **refiner**. The debugging process is illustrated in Alg. 1. Starting with an initial program P_0 and its execution feedback, the critic model C detects and localizes potential errors. Subsequently, the refiner R corrects these identified errors. This iterative process continues until the critic model C deems the program satisfactory.[1]

Algorithm 1 VDebugger algorithm

```
Require: Critic $C$, refiner $R$, score threshold $th$, max step $T$,
    initial program $P_0$
 1: $P = P_0$
 2: for $i = 1$ to $T$ do
 3:     $fb = \text{EXECUTE}(P)$                    ▷ Collect feedback
 4:     $score, loc = C(P, fb)$   ▷ Identify and localize error
 5:     if $score > th$ then                        ▷ Correct program
 6:         return $P$
 7:     end if
 8:     $P_{new} = R(P, fb, loc)$                   ▷ Refine program
 9:     $P = P_{new}$
10: end for
11: return $P$
```

Execution feedback. In contrast to previous approaches that focus on execution errors and block-wise execution states (Stanic et al., 2024; Gao et al., 2023), our objective is to develop a general and comprehensive feedback mechanism that can cover a wider range of errors. Drawing inspiration from the stepping debugging strategy[2] of human programmers, we track the execution process step by step and document each executed program line, the resulting changes in the intermediate variables, and any errors encountered during execution. This feedback information is fed to VDebugger in text format as in Figure 8 in the Appendix.

Critic. Critic C jointly detects and localizes the error in the program. Formally, given the input program P and its feedback information collected through execution (denoted as $\text{EXECUTE}(P)$),

$$score, loc = C(P, \text{EXECUTE}(P)),$$

where $score$ represents how likely the program P is correct. P is considered correct when $score$ ex-

[1]While this is a general framework applicable to various programming languages, this work focuses on Python.

[2]https://en.wikipedia.org/wiki/Stepping_(debugging)

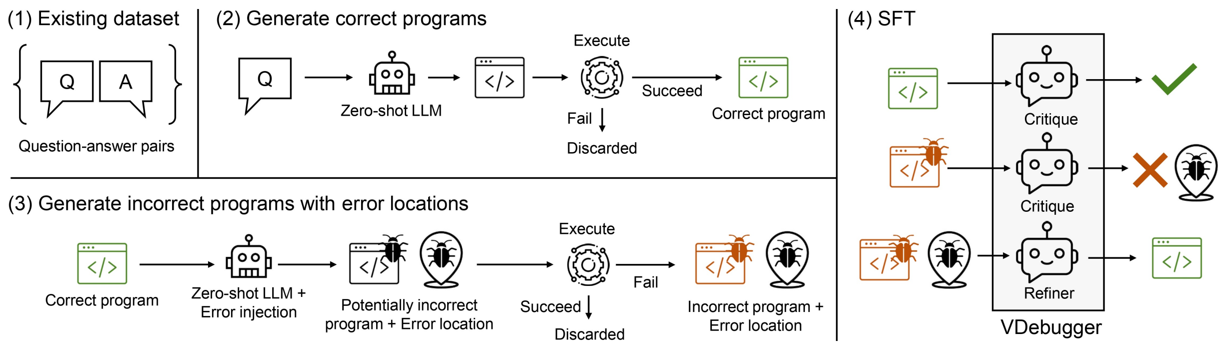

(1) Existing dataset
Q
A
Question-answer pairs
(2) Generate correct programs
Q
Zero-shot LLM
Execute
Succeed
Fail
Discarded
Correct program
(3) Generate incorrect programs with error locations
Correct program
Zero-shot LLM + Error injection
Potentially incorrect program + Error location
Execute
Succeed
Fail
Discarded
Incorrect program + Error location
(4) SFT
Critique
Critique
Refiner
VDebugger

Figure 2: **Training data collection pipeline.** Given an existing dataset of question-anwswer pairs, we prompt LLM to generate correct programs, inject error to generate incorrect programs, and use the paired data for SFT training.

ceeds a threshold th (0.5 in this work). The critic classify the program P into either correct or incorrect. Concretely, C first generates a correctness token chosen from $\{t_{\checkmark}, t_{\text{\textit{x}}}\}$ representing whether the program is correct or incorrect, so the probability assigned to token $t_{\checkmark}$ can serve as $score$. If the token $t_{\text{\textit{x}}}$ is generated, C further generates the error location loc. Here, the location loc is a span within program P defined by its start and end positions, which can be a word, a line, multiple lines, or any continuous segment.

Refiner. Conditioned on the error location loc, refiner R rewrites location loc to fix the program. Formally,

$$P_{new} = R(P, \textsc{Execute}(P), loc),$$

where the output program P_{new} only differs with the input program P at location loc.

4 Training of VDebugger

With the critic-refiner framework introduced above, now we design an automated pipeline to collect training data tailored for our framework. Our goal is to obtain tuples $\{(P_{\checkmark}, P_{\text{\textit{x}}}, loc)\}$ where in each tuple, the correct program $P_{\checkmark}$ and incorrect one $P_{\text{\textit{x}}}$ only differ at location loc. As in Figure 2, our pipeline consists of two steps: (1) generating correct programs, and (2) generating incorrect programs with error locations.

Correct program generation. Given pairs of questions and ground truth answers from existing datasets, we prompt LLM to generate an initial pool of visual programs denoted as $\mathcal{P}^{(0)}$. The subset of programs whose execution results match the ground truth labels (denoted as $\mathcal{P}^{(0)}_{\checkmark}$) will be kept for the next step, while the rest of the programs (denoted as $\mathcal{P}^{(0)}_{\text{\textit{x}}}$) will be discarded.

Incorrect program generation. For each correct program $P \in \mathcal{P}^{(0)}_{\checkmark}$, we obtain a potentially incorrect program P' by resampling part of the program P at a random location loc. We then execute program P' and select those whose execution results do not match ground truth labels, denoted as $\mathcal{P}^{(1)}_{\text{\textit{x}}}$.

Concretely, we first parse the correct program P into a abstract syntax tree and randomly sample a subtree as location loc. We then mask out the selected location and prompt LLM to recover the masked content. To more effectively inject errors to the location, we propose a **mask-best sampling** strategy. At each decoding step, given the probability distribution p predicted by LLM, we mask out the token i^* with highest probability and only sample from the tail distribution $p^{(tail)}$:

$$\begin{aligned} i^* &= \arg\max_i p_i \\ p_i^{(tail)} &= \begin{cases} 0 & i = i^* \\ p_i / (1 - p_{i^*}) & i \neq i^*. \end{cases} \end{aligned}$$

To ensure output quality, we only apply mask-best sampling to tokens with low confidence, determined as follows:

$$p_{i^*} - p_{i^{(2)}} < th, \; i^{(2)} = \arg\max_{i \neq i^*} p_i,$$

and we apply mask-best sampling to at most N tokens. The threshold th is set as 0.9 in this work. The formal algorithm is in Alg. 2. As shown in Table 2, mask-best dramatically increases the rate at which an error is successfully injected by up to 10 times (from 3.7% to 38.9%). We manually analyze and categorize 200 errors injected into GQA dataset. As shown in Figure 3, greedy sampling generates a large number of superficial errors referencing variables before their creation. In contrast, mask-best sampling produces a broader range of more complex and diverse errors.

	$\lvert\mathcal{P}_{✗}^{(0)}\rvert$	$\lvert\mathcal{P}_{✓}^{(0)}\rvert$	Greedy		Mask-best		Final
			$\lvert\mathcal{P}_{✗}^{(1)}\rvert$	Error Rate	$\lvert\mathcal{P}_{✗}^{(1)}\rvert$	Error Rate	
GQA	21,874	18,126	3,927	21.7%	7,758	42.8%	11,188
TallyQA	17,310	22,690	842	3.7%	8,843	38.9%	9,593
NLVRv2	19,415	25,085	3,803	15.2%	10,263	40.9%	13,948
RefCOCO	21,013	18,987	4,710	24.8%	8,635	45.5%	12,949

Table 2: **Statistics of collected training data.** We report the number of correct and incorrect programs in the initial pool (denoted as $\lvert\mathcal{P}_{✗}^{(0)}\rvert$ and $\lvert\mathcal{P}_{✓}^{(0)}\rvert$), the number of incorrect programs generated via greedy decoding and mask-best decoding (denoted as $\lvert\mathcal{P}_{✗}^{(1)}\rvert$), and the rate at which an error is successfully injected computed as $\lvert\mathcal{P}_{✗}^{(1)}\rvert / \lvert\mathcal{P}_{✓}^{(0)}\rvert$ (denoted as Error Rate). In total, we collect 47,678 paired training data.

Algorithm 2 Mask-best sampling

```
Require: LLM, prompt, confidence threshold $th$, maximum
    numbers for mask-best sampling $N$, maximum number
    of tokens $T$
 1: $P = []$                                  ▷ Empty string for sampling
 2: $n = 0$                                   ▷ Mask-best sampling counter
 3: for $i = 1$ to $T$ do
 4:     $p = \text{LLM}(\text{prompt}, P)$
 5:     if $n < N$ and $p_{i^*} - p_{i^{(2)}} < th$ then
 6:         $p = p^{(tail)}$                   ▷ Sample from the tail
 7:         $n = n + 1$
 8:     end if
 9:     $P = [P; \text{SAMPLE}(p)]$
10:     if EOS is sampled then
11:         break
12:     end if
13: end for
14: return $P$
```

Training. Pairing programs from $\mathcal{P}_{✓}^{(0)}$ and $\mathcal{P}_{✗}^{(1)}$, we obtain a training set $\{(P_{✓}, P_{✗}, loc)\}$ for training the critic C and refiner R. Our training objectives are as follows:

$$\mathcal{L}_C = \sum \mathcal{L}\left(t_{✗}, loc \mid P_{✗}, \text{EXECUTE}(P_{✗})\right) + \mathcal{L}\left(t_{✓} \mid P_{✓}, \text{EXECUTE}(P_{✓})\right)),$$

$$\mathcal{L}_R = \sum \mathcal{L}(P_{✓} \mid P_{✗}, \text{EXECUTE}(P_{✗}), loc).$$

where $\mathcal{L}$ represents autoregressive language modeling objective. However, training C only on programs with injected errors limits its ability to detect errors in naturally generated programs due to the distribution shift. Leveraging the large pool of incorrect programs $\mathcal{P}_{✗}^{(0)}$ generated in the first step, we introduce an additional objective to address the distribution shift:

$$\mathcal{L}_{C'} = \sum_{P_{✓} \in \mathcal{P}_{✓}^{(0)}} \mathcal{L}(t_{✓} \mid P_{✓}, \text{EXECUTE}(P_{✓})) + \sum_{P_{✗} \in \mathcal{P}_{✗}^{(0)}} \mathcal{L}(t_{✗} \mid P_{✗}, \text{EXECUTE}(P_{✗})),$$

and the final training objective for C is $\mathcal{L}_{C,final} = \mathcal{L}_C + \mathcal{L}_{C'}$. The mixed objective enables C to detect and localize errors in naturally generated programs without requiring error location annotations for these programs.

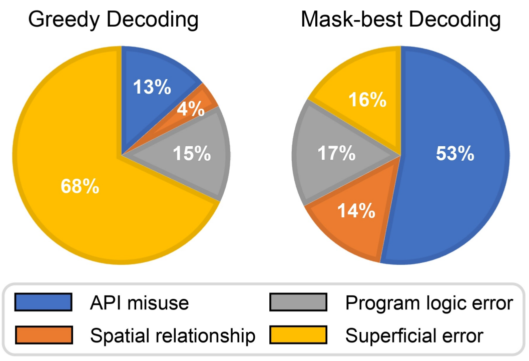


Figure 3: Categorization of synthetic errors generated by greedy decoding and mask-best decoding respectively.

5 Experiments

In this section, we aim to: (1) evaluate the effectiveness of VDebugger by comparing against existing self-debugging methods; (2) analyze the benefits brought by each individual component; and (3) demonstrate its generalization capability by debugging programs generated by other LLMs and by evaluating on unseen tasks.

Dataset. We experiment on three forms of tasks including 6 datasets: (1) *Visual question answering with one image*, including GQA dataset (Hudson and Manning, 2019) targeting compositional question answering and TallyQA dataset (Acharya et al., 2019) targeting counting; (1) *Visual question answering with multiple images*, including NLVRv2 dataset (Suhr et al., 2019) where each question is accompanied by two images; (3) *Visual grounding* including three variants of RefCOCO dataset (Yu et al., 2016): the original RefCOCO dataset, RefCOCO+ that disallows location descriptions, and RefCOCOg that involves longer and more complex text descriptions. We report accuracy for question answering tasks and IoU for visual grounding tasks.

	GQA	TallyQA	NLVRv2	RefCOCO	RefCOCO+	RefCOCOg	Mean
Base VLM	44.7	29.9	55.4	55.0	52.2	50.1	47.9
CodeLlama-7B as code generator							
No Debugging (Surís et al., 2023)	43.1	45.4	59.7	56.2	51.6	53.9	51.7
SelfDebug (Chen et al., 2023)	42.7	44.4	61.4	55.9	51.4	52.3	51.4
LDB (Zhong et al., 2024)	41.8	39.4	56.4	50.4	51.8	52.2	48.7
VDebugger (Ours)	**46.3** (+3.2)	**46.4** (+1.0)	61.4 (+1.7)	**58.7** (+2.5)	**52.3** (+0.7)	**56.3** (+2.4)	**53.6** (+1.9)
VDebugger w/ Gen (Ours)	46.0 (+2.9)	46.3 (+0.9)	**61.8** (+2.1)	58.3 (+2.1)	51.8 (+0.2)	55.9 (+2.0)	53.3 (+1.6)
CodeLlama-13B as code generator							
No Debugging (Surís et al., 2023)	45.4	47.7	64.8	56.7	54.7	55.1	54.1
SelfDebug (Chen et al., 2023)	41.6	42.8	62.5	41.2	39.2	46.2	45.6
LDB (Zhong et al., 2024)	42.7	37.4	57.1	51.0	51.1	50.8	48.3
VDebugger (Ours)	**48.1** (+2.7)	48.3 (+0.6)	65.0 (+0.2)	58.1 (+1.4)	**55.5** (+0.8)	**58.3** (+3.2)	55.5 (+1.4)
VDebugger w/ Gen (Ours)	**48.1** (+2.7)	**48.6** (+0.9)	**65.8** (+1.0)	**58.5** (+1.0)	55.0 (+0.3)	58.2 (+3.1)	**55.7** (+1.6)

Table 3: **Main results.** We report accuracy for GQA, TallyQA, NLVRv2, and IoU for RefCOCO datasets. We compare the performance of two debugging baselines and our VDebugger (highlighted in the table). Here, VDebugger w/ Gen denotes the generalist model trained on all datasets. For comparison, we also report the performance of the base VLMs.

For training data collection, we generate 4 training sets for the GQA, TallyQA, NLVRv2 and RefCOCO datasets respectively. We use CodeLlama-7B-Python (Rozière et al., 2024) to generate the initial program pool $\mathcal{P}^{(0)}$, and use CodeLlama-7B-Instruct to generate incorrect programs $\mathcal{P}^{(1)}_{\text{\ding{55}}}$ with both greedy decoding and mask-best sampling. We collect 9$\sim$14k training data for each dataset and in total 47.7k data. Detailed statistics are in Table 2.

Evaluated models. We use ViperGPT (Surís et al., 2023) as our base visual program generator before any debugging. We train VDebugger on each dataset based on CodeLlama-7B-Python and CodeLlama-13B-Python. We further train a generalized variant on the mix of all datasets denoted as VDebugger w/ Gen. During inference, we use a maximum iteration step of $T = 3$ unless otherwise noted. We compare our method against two code debugging methods: SelfDebug (Chen et al., 2023) and LDB (Zhong et al., 2024). SelfDebug debugs the program based on unit-test feedback. Since visual programs do not have unit tests available, we replace it with our execution feedback. LDB uses execution states per program block to iteratively rewrite each block, making it more expensive than our strategy. Both SelfDebug and LDB relies on zero-shot prompting without any training.

5.1 Results

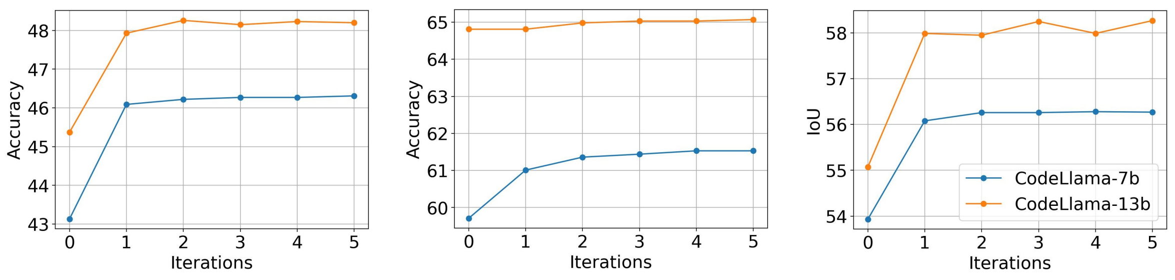
Table 3 shows our main results on all six datasets. Both SelfDebug and LDB slightly hurt the performance, likely due to the limited self-debugging capability of small LLMs as noted by recent studies (Luo et al., 2023; Tian et al., 2024; Lan et al., 2024; Jiang et al., 2024). The challenge is exacerbated by the absense of visual programs during the pre-training stage of LLMs, highlighting the necessity of training debugging models for visual programs. In contrast, our VDebugger consistently improves the performance in every dataset, achieving improvements of up to 3.2% accuracy.

Ablation study. We investigate the contribution of each component as shown in Table 4. Specifically, we aim to: (1) assess the individual contributions of critic and refiner components, and (2) evaluate the benefits of execution feedback. We report the critic's binary accuracy in predicting overall program correctness, as well as the percentage of incorrect programs successfully fixed by refiner, denoted as refiner success rate. The critic demonstrates consistently strong performance, with high binary accuracy ranging from 67% to 80% across different datasets. Our manual evaluation of 59 examples from GQA shows the predicted error-inducing errors are correct in 74% of the cases. However, the refiner success rate is less reliable, varying dramatically from 10% to 57% across datasets. When enhanced with execution feedback, the critic achieves more performance gains while the benefits to refiner performance are minimal. When reflected in the final performance on the downstream tasks, execution feedback consistently brings benefits on all datasets. In general, VDebugger can reliably perform self-critique utilizing execution feedback, and the remaining challenges mainly lie in correcting the program after the errors are identified.

Performance by iteration. VDebugger can per-

	Critic Acc.		Refiner SR		Task Performance		
Dataset	w/o FB	w/ FB	w/o FB	w/ FB	No Debug	Ours w/o FB	Ours w/ FB
GQA	69.6	**73.9** (+4.3)	**47.7**	**47.7** (+0.0)	43.1	45.7 (+2.6)	**46.3** (+3.2)
TallyQA	56.3	**72.1** (+15.8)	**10.7**	10.3 (-0.4)	45.4	45.9 (+0.5)	**46.4** (+1.0)
NLVRv2	66.1	**67.3** (+1.2)	10.8	**15.1** (+4.3)	59.7	60.4 (+0.7)	**61.4** (+1.7)
RefCOCO	71.9	**80.9** (+9.0)	26.7	**27.2** (+0.5)	56.2	58.3 (+2.1)	**58.7** (+2.5)
RefCOCO+	70.4	**77.4** (+7.0)	56.3	**57.1** (+0.8)	51.6	51.3 (-0.3)	**53.9** (+2.3)
RefCOCOg	64.5	**73.1** (+8.6)	32.2	**33.1** (+0.9)	53.9	55.7 (+1.8)	**56.3** (+2.4)
Mean	66.5	74.1 (+7.7)	30.7	31.8 (+1.0)	51.6	52.9 (+1.2)	53.8 (+2.2)

Table 4: **Ablation study.** We report the critic accuracy (Acc.), the refiner success rate (SR), and final task performance on downstream tasks. For each component, we report the performance either without or with execution feedback (denoted as w/o FB and w/ FB). We also report downstream task performance before any debugging. Results are evaluated on 7B-level VDebugger models.



(a) Performance on GQA. (b) Performance on NLVR. (c) Performance on RefCOCOg.

Figure 4: Performance on GQA, NLVRv2 and RefCOCOg datasets by the number of debugging iterations.

	GQA	TallyQA	NLVRv2	RefCOCO	RefCOCO+	RefCOCOg	Mean
CodeLlama-70b as code generator							
No Debugging	46.1	45.2	68.1	54.0	49.6	53.3	52.7
VDebugger 7b	48.8 (+2.7)	47.9 (+2.7)	**68.9** (+0.8)	**57.4** (+3.4)	**50.9** (+1.3)	**56.7** (+3.4)	**55.1** (+2.4)
VDebugger 13b	**48.9** (+2.8)	**48.5** (+3.3)	68.6 (+0.5)	56.9 (+2.9)	**50.9** (+1.3)	56.4 (+3.1)	55.0 (+2.3)
DeepSeek-Coder-33B as code generator							
No Debugging	47.3	46.2	**67.9**	59.6	51.8	55.4	54.7
VDebugger 7b	**48.9** (+1.6)	**46.9** (+0.7)	67.8 (-0.1)	**60.5** (+0.9)	52.4 (+0.6)	57.1 (+1.7)	**55.6** (+1.9)
VDebugger 13b	**48.9** (+1.6)	46.2 (+0.0)	67.4 (-0.5)	**60.5** (+0.9)	**52.8** (+1.0)	**57.7** (+2.3)	**55.6** (+1.9)
GPT-3.5 as code generator							
No Debugging	45.9	43.6	63.9	60.6	54.0	57.5	54.3
VDebugger 7b	49.3 (+3.4)	**46.4** (+2.8)	**68.8** (+4.9)	61.2 (+0.6)	**54.7** (+0.7)	**58.8** (+1.3)	**56.5** (+2.2)
VDebugger 13b	**49.6** (+3.7)	**46.4** (+2.8)	68.7 (+4.8)	**61.3** (+0.7)	54.1 (+0.6)	**58.8** (+1.3)	**56.5** (+2.2)

Table 5: VDebugger can debug visual programs generated by larger LLMs, including CodeLlama-70b, DeepSeek-Coder-33B and GPT-3.5.

form iterative debugging until the critic determines the program as correct. Figure 4 demonstrates the performance curve by the number of iterations on three representative datasets for the three task forms, GQA, NLVRv2, and RefCOCOg. We find that most performance gains occur in the first one or two iterations, after which performance plateaus and may slightly decline. Qualitative analysis shows that more iterations are beneficial for complex problems, where the initial debugging attempt often fails, so VDebugger need to iteratively refines the program in a trial-and-error manner. An example is shown in Figure 10 in the Appendix.

Generalization to other code generators. While VDebugger is trained on programs generated by CodeLlama models, it can be employed to debug programs generated by LLMs with larger number of parameters. As shown in Table 5, we experi-

	RSVG	COVR
Base VLM	18.1	41.2
CodeLlama-7b as code generator		
No Debugging	17.9	41.5
VDebugger	**18.7** (+0.8)	**43.8** (+2.3)
CodeLlama-13b as code generator		
No Debugging	18.3	46.9
VDebugger	**18.8** (+0.5)	**47.9** (+1.0)

Table 6: VDebugger can generalize to unseen tasks, including visual grounding for remote sensing images (RSVG) and visual question answering over variable number of images (COVR). We report IoU for RSVG and accuracy for COVR.

ment with two open LLMs, CodeLlama-70b and DeepSeek-Coder-33B (Guo et al., 2024), and the proprietary LLM GPT-3.5. Despite these models being up to ten times larger than our base models and achieving higher performance without any debugging, VDebugger's debugging process still consistently brings improvements, demonstrating its generalization capability. Thus, employing zero-shot large-scale LLMs debugged by a small VDebugger can be a good strategy to enhance performance at a reasonable cost.

Generalization to unseen tasks. We evaluate the generalist variant VDebugger w/ Gen, which is trained on all six datasets, on two unseen datasets: (1) RSVG (Zhan et al., 2023), a visual grounding dataset for remote sensing images, a challenging task due to the dense objects and complex spatial relationships in remote sensing images; and (2) COVR (Chen et al., 2022), a novel task form requiring the model to answer questions based on a variable number of images. Table 6 shows that VDebugger consistently improves performance on both datasets, demonstrating its ability to generalize to unseen domains and task formulations.

Data quality. To verify the quality of automatically generated data, we manually examine 100 programs from each training set. We evaluate the proportions of incorrect programs, or "false positives", among the programs considered correct. Most datasets have relatively low false positive ratio: 16% for GQA, 13% for TallyQA, and 19% for RefCOCO. Due to the answer format, including free-form strings, numbers and bounding boxes, an exact match in the final answer ensures the program has a high probability to be correct. On the other hand, NLVRv2 dataset has a higher false

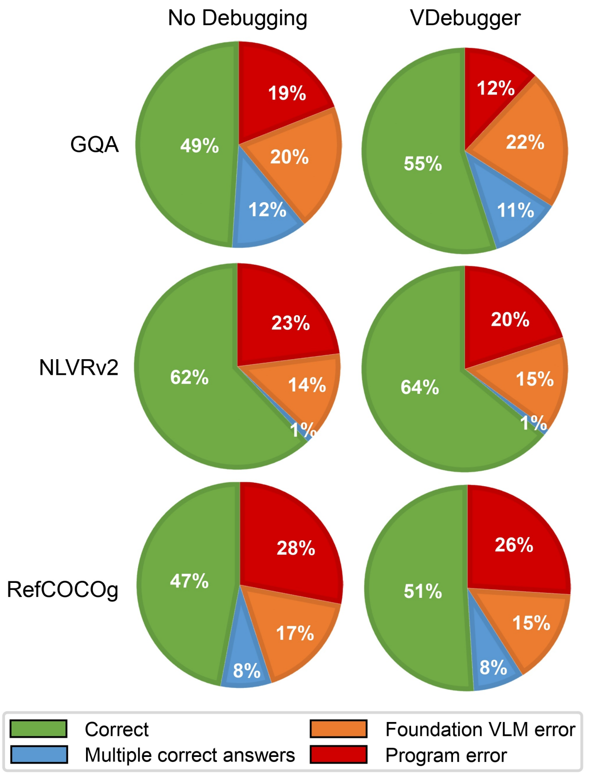
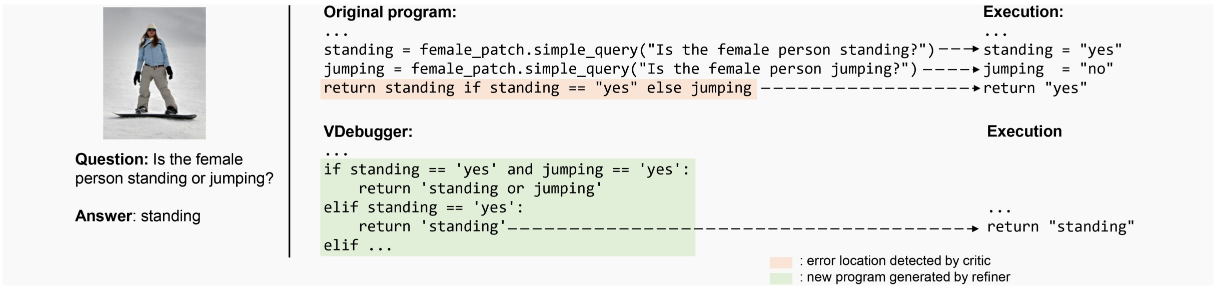


Figure 5: **Sources of errors on GQA, NLVRv2 and RefCOCOg datasets.** We categorize the predictions into four categories: correct, multiple correct answers (where the prediction is correct but does not match the ground truth annotation), foundation VLM errors, and program errors.

positive ratio (40%) due to its binary label format. However, its effect can be mitigated by training on multiple datasets, as shown by the generalist VDebugger outperforming the specialist VDebugger on NLVRv2 dataset. While another concern over data quality is the program optimality, we observe that visual programs tend to have straightforward code structures, and thus have limited potential for algorithmic optimization. For example, 68% of the programs do not contain loop structures.

Qualitative analysis. We analyze the sources of errors by examining 100 examples from each of the three datasets: GQA, NLVRv2, and RefCOCOg. As shown in Figure 5, program errors significantly affect the end performance, accounting for 49% to 62% of total errors varying by dataset. VDebugger consistently reduces program errors on all datasets, especially on GQA. An example of VDebugger fixing program error is in Figure 6. Interestingly, we observe that VDebugger can also help recover from foundation VLM errors especially on RefCOCOg dataset. While errors incurred by foundation VLMs remain a crucial bottleneck for visual programs, VDebugger can invoke foundation VLMs in an al-



Question: Is the female person standing or jumping?

Answer: standing

Original program:

```
...
standing = female_patch.simple_query("Is the female person standing?")
jumping = female_patch.simple_query("Is the female person jumping?")
return standing if standing == "yes" else jumping
```

Execution:

```
...
standing = "yes"
jumping  = "no"
return "yes"
```

VDebugger:

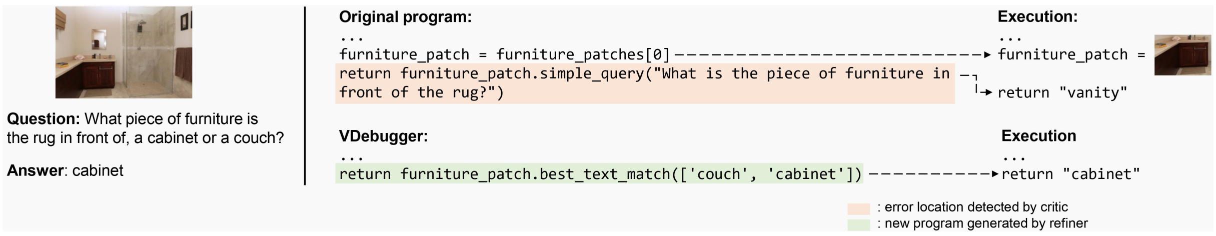
```
...
if standing == 'yes' and jumping == 'yes':
    return 'standing or jumping'
elif standing == 'yes':
    return 'standing'
elif ...
```

Execution

```
...
return "standing"
```

🟥 : error location detected by critic
🟩 : new program generated by refiner

Figure 6: **Example where VDebugger fixes program error.**



Question: What piece of furniture is the rug in front of, a cabinet or a couch?

Answer: cabinet

Original program:

```
...
furniture_patch = furniture_patches[0]
return furniture_patch.simple_query("What is the piece of furniture in
front of the rug?")
```

Execution:

```
...
furniture_patch =
return "vanity"
```

VDebugger:

```
...
return furniture_patch.best_text_match(['couch', 'cabinet'])
```

Execution

```
...
return "cabinet"
```

🟥 : error location detected by critic
🟩 : new program generated by refiner

Figure 7: **Example where VDebugger recovers from foundation model error.** The question answering model yields incorrect answer "vanity" in the original program. By detecting this error, VDebugger invokes the foundation VLMs in an alternative way and thus obtains the correct answer.

	Cost (s/it)	Compared with No Debugging	Compared with last iteration
T = 0	3.08	-	-
T = 1	4.44	+ 44%	+44%
T = 2	4.72	+ 53%	+6%
T = 3	4.81	+ 56%	+2%

Component	Cost (s/it)
Initial program generation	1.42
Program execution	1.66
Critic inference	0.09
Refiner inference	0.08

Table 7: **Computational cost of VDebugger**, measured by seconds per item (s/it). **Left:** Computational cost by iteration step T. $T = 0$ represents the no debugging baseline. **Right:** Breakdown of the computational cost of each component.

ternative way to avoid the identified errors. An example is shown in Figure 7.

Computational complexity. To measure the additional computational overhead brought by the critic-refiner framework and the iterative process, we measure the computational cost by iteration step T as well as a breakdown of the cost of each component in the framework. The detailed statistics are reported in Table 7. While VDebugger moderately increases the computational cost by 56% compared to the no debugging baseline, the major computational overhead arises from program execution, rather than the inference of critic and refiner models. Additionally, increasing the number of debugging iterations only marginally increases the inference cost, since most debugging is addressed within the first round.

6 Conclusion

VDebugger is a critic-refiner framework fine-tuned to detect, localize, and correct errors in visual programs leveraging fine-grained execution feedback. The training data is collected through an automated pipeline that first generates correct programs and then effectively injects errors using mask-best sampling. Experiments on six datasets demonstrate that VDebugger consistently brings improvements, and further studies verifies VDebugger's generalization to unseen tasks. A future direction is to allow the visual program debugger to access visual information in addition to relying on textual information, and to jointly train it with foundation VLMs.

7 Limitations

We hereby discuss the potential limitations of our work:

(1) In this work, our critic model can provide basic explanations of identified errors by predicting errors locations. However, human programmers may benefit from more detailed explanations in natural language. The automatic collection of such text-rich description is very challenging. Therefore, obtaining expert annotations would be a valu-

able though costly future step to enhance the interpretability of the debugging process.

(2) Our work mainly focuses on established tasks such as visual question answering and visual grounding. While these tasks demonstrate the effectiveness of our framework, real-world applications often require systems to interact dynamically with humans, respond to open-ended questions, and perform on-demand reasoning. Although our current work does not directly address these complex, real-world scenarios, we believe our method is generic framework that can be adapted for such applications. Exploring the application of our self-debugging method to more in-the-wild and diverse scenarios is an exciting direction for future research.

(3) Following prior work (Gupta and Kembhavi, 2023; Surís et al., 2023), our method utilizes a text-only language model (LLM) to generate visual programs, which may introduce limitations to its capabilities. Incorporating visual information and/or jointly training the debugger with foundational VLMs could be a valuable direction for future research, potentially further enhancing its self-critic capabilities.

Acknowledgement

This research is based upon work supported by CISCO, U.S. DARPA ECOLE Program No. #HR00112390060, and OFFICE OF NAVAL RESEARCH Award #N00014-23-1-2780. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

Manoj Acharya, Kushal Kafle, and Christopher Kanan. 2019. Tallyqa: Answering complex counting questions. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 8076–8084. AAAI Press.

Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. 2023. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond. *arXiv preprint arXiv:2308.12966*.

Ben Bogin, Shivanshu Gupta, Matt Gardner, and Jonathan Berant. 2021. COVR: A test-bed for visually grounded compositional generalization with real images. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9824–9846, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jingqiang Chen, Chaoxiang Cai, Xiaorui Jiang, and Kejia Chen. 2022. Comparative graph-based summarization of scientific papers guided by comparative citations. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 5978–5988, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Minghe Gao, Juncheng Li, Hao Fei, Liang Pang, Wei Ji, Guoming Wang, Wenqiao Zhang, Siliang Tang, and Yueting Zhuang. 2023. De-fine: Decomposing and refining visual programs with auto-feedback. *CoRR*, abs/2311.12890.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Tanmay Gupta, Amita Kamath, Aniruddha Kembhavi, and Derek Hoiem. 2022. Towards general purpose vision systems: An end-to-end task-agnostic vision-language architecture. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16399–16409.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 14953–14962. IEEE.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *CoRR*, abs/2310.01798.

Drew A. Hudson and Christopher D. Manning. 2019. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 6700–6709. Computer Vision Foundation / IEEE.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024. Training llms to better self-debug and explain code. *Preprint*, arXiv:2405.18649.

Amita Kamath, Jack Hessel, and Kai-Wei Chang. 2023. What's" up" with vision-language models? investigating their struggle with spatial reasoning. *arXiv preprint arXiv:2310.19785*.

Amita Kamath, Cheng-Yu Hsieh, Kai-Wei Chang, and Ranjay Krishna. 2024. The hard positive truth about vision-language compositionality. *arXiv preprint arXiv:2409.17958*.

Tian Lan, Wenwei Zhang, Chen Xu, Heyan Huang, Dahua Lin, Kai Chen, and Xian-ling Mao. 2024. Criticbench: Evaluating large language models as critic. *arXiv preprint arXiv:2402.13764*.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven C. H. Hoi. 2023. BLIP-2: bootstrapping language-image pre-training with frozen image encoders and large language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 19730–19742. PMLR.

Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. 2022a. BLIP: bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 12888–12900. PMLR.

Liunian Harold Li, Pengchuan Zhang, Haotian Zhang, Jianwei Yang, Chunyuan Li, Yiwu Zhong, Lijuan Wang, Lu Yuan, Lei Zhang, Jenq-Neng Hwang, Kai-Wei Chang, and Jianfeng Gao. 2022b. Grounded language-image pre-training. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10955–10965. IEEE.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. In *NeurIPS*.

Liangchen Luo, Zi Lin, Yinxiao Liu, Lei Shu, Yun Zhu, Jingbo Shang, and Lei Meng. 2023. Critique ability of large language models. *Preprint*, arXiv:2310.04815.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. 2024. REFINER: Reasoning feedback on intermediate representations. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1100–1126, St. Julian's, Malta. Association for Computational Linguistics.

Ram Rachum, Alex Hall, Iori Yanokura, et al. 2019. Pysnooper: Never use print for debugging again.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Aleksandar Stanic, Sergi Caelles, and Michael Tschannen. 2024. Towards truly zero-shot compositional visual reasoning with llms as programmers. *CoRR*, abs/2401.01974.

Alane Suhr, Stephanie Zhou, Ally Zhang, Iris Zhang, Huajun Bai, and Yoav Artzi. 2019. A corpus for reasoning about natural language grounded in photographs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6418–6428, Florence, Italy. Association for Computational Linguistics.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *IEEE/CVF International Conference on Computer Vision, ICCV 2023, Paris, France, October 1-6, 2023*, pages 11854–11864. IEEE.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Zhiyuan Liu, and Maosong Sun. 2024. Debugbench: Evaluating debugging capability of large language models. *Preprint*, arXiv:2401.04621.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Licheng Yu, Patrick Poirson, Shan Yang, Alexander C. Berg, and Tamara L. Berg. 2016. Modeling context in referring expressions. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II*, volume 9906 of *Lecture Notes in Computer Science*, pages 69–85. Springer.

Mert Yüksekgönül, Federico Bianchi, Pratyusha Kalluri, Dan Jurafsky, and James Zou. 2023. When and why vision-language models behave like bags-of-words, and what to do about it? In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Yang Zhan, Zhitong Xiong, and Yuan Yuan. 2023. Rsvg: Exploring data and models for visual grounding on remote sensing data. *IEEE Transactions on Geoscience and Remote Sensing*, 61:1–13.

Lily Zhong, Zilong Wang, and Jingbo Shang. 2024. LDB: A large language model debugger via verifying runtime execution step-by-step. *CoRR*, abs/2402.16906.

A Artifacts

This work involves the following artifacts:

Datasets: GQA (Hudson and Manning, 2019) distributed under CC-BY-4.0 license, TallyQA (Acharya et al., 2019) distributed under Apache-2.0 license license, NLVRv2 (Suhr et al., 2019) distributed under CC-BY-4.0 license, RefCOCO (Yu et al., 2016) (including RefCOCO, RefCOCO+ and RefCOCOg variants) distributed under Apache-2.0 license, RSVG (Zhan et al., 2023) without license specified, and COVR (Bogin et al., 2021) distributed under MIT license.

Software: We use transformers (Wolf et al., 2020) and deepspeed (`https://github.com/microsoft/DeepSpeed`) for model training, both distributed under Apache-2.0 license. We collect execution feedback of visual programs using `pysnooper` (Rachum et al., 2019) distributed under MIT license.

Models: We use CodeLlama (Rozière et al., 2024) distributed under Llama's own license[3] and DeepSeek-Coder (Guo et al., 2024) distributed under MIT License.

This work creates the following artifacts:

Datasets: We collect training data for our VDebugger based on GQA (Hudson and Manning, 2019), TallyQA (Acharya et al., 2019), NLVRv2 (Suhr et al., 2019) and RefCOCO (Yu et al., 2016) datasets. Detailed statistics are in Table 2.

Software: The code for training and inference of VDebugger and training data collection.

Models: The VDebugger 7B and 13B models, trained on each individual dataset as well as the generalist model trained on all datasets.

In summary, all the artifacts involved permit research use. Our use is consistent with their intended use. We plan to release our software, datasets and models with license Apache-2.0 license, which is compatible with the original access conditions. All our artifacts are limited to English and do not cover multilingual scenarios.

B Implementation Details of VDebugger

Since VDebugger is implemented based on LLMs, we need to effectively represent execution feedback $\texttt{Execute}(P)$ and error location loc with text. The execution feedback is tracked and formatted via `pysnooper` (Rachum et al., 2019). An example is shown in Figure 8: the feedback representation covers the final return value, each code line being executed, their resulted change in intermediate variable values, and execution errors if any. To represent a local span loc with text, instead of directly generating the starting and ending location, we represent it by repeating the original program and wrapping location loc with some special tokens. An example is shown in 9.

C Experimental Details

Base VLM: We use the same set of base VLMs as in Surís et al. (2023). To report the performance of base VLMs, we use the question answering model

[3] `https://github.com/meta-llama/llama/blob/main/LICENSE`

Question: What item of furniture is not large?

Program:

```
def execute_command(image) -> str:
    image_patch = ImagePatch(image)
    image_patch = best_image_match(list_patches=[ImagePatch(image)], content=['item'], return_index=True)
    return image_patch.simple_query('What item of furniture is not large?')
```

Execution feedback:

```
-> None

call        1 def execute_command(image) -> str:
line        2     image_patch = ImagePatch(image)
New var:....... image_patch = ImagePatch(left=0, right=500, upper=375, lower=0, height=375, width=500, horizontal_center=250.0, vertical_center=187.5)
line        3     image_patch = best_image_match(list_patches=[ImagePatch(image)], content=['item'], return_index=True)
Modified var:.. image_patch = 0
line        4     return image_patch.simple_query('What item of furniture is not large?')
exception   4     return image_patch.simple_query('What item of furniture is not large?')
Exception:..... AttributeError: 'int' object has no attribute 'simple_query'
Call ended by exception
```

Figure 8: **Text representation of feedback information.** The feedback incorporates the final return value, each code line being executed, their resulted change in intermediate variable values, and execution errors if any.

loc: return image_patch.simple_query('What item of furniture is not large?')

Representation:

```
def execute_command(image) -> str:
    image_patch = ImagePatch(image)
    image_patch = best_image_match(list_patches=[ImagePatch(image)], content=['item'], return_index=True)
    <BUG>return image_patch.simple_query('What item of furniture is not large?')<BUG/>
```

Figure 9: **Text representation of location** loc**.** In this example, special tokens `<BUG>` and `<BUG/>` wraps the location of interests.

BLIP-2 (Li et al., 2023) for visual question answering tasks, and the object detection model GLIP (Li et al., 2022b) for visual grounding tasks. Since BLIP-2 can only take one image as input, we concatenate all images into one when handling multiple images, such as in the NLVRv2 and COVR datasets.

VDebugger: For fine-tuning VDebugger, we use CodeLlama-7B-Python and CodeLlama-13B-Python as the base model. We truncate the context length into within 1024 tokens. We use a total batch size of 128 sentences per batch (including), a learning rate of $2e-5$, a linear scheduler for learning rate, and a warmup ratio of 0.03. We train the CodeLlama-7B-Python for 3 epochs and CodeLlama-13B-Python for 1 epoch on all datasets. With 4 A6000 GPU, the training of refiner takes ∼4 hours and the training of critic takes ∼12 hours. In inference, we use greedy decoding with 256 as the maximum number of tokens.

Evaluation: We evaluate the models on the `testdev` split of GQA, the Test-Complex split of TallyQA, the `test1` split of NLVRv2, the `testA` split by UNC of RefCOCO and RefCOCO+, and the standard test set split by UMD for RefCOCOg. We report accuracy for GQA, TallyQA and NLVRv2, and IoU for RefCOCO, RefCOCO+ and RefCOCOg. For accuracy, following the setting of Surís et al. (2023), we first preprocess the answer produced by our method by removing stopwords and then use exact matching.

	BLIP	**InstructBLIP**
End-to-end VLMs	43.1	48.3
Visual programming	45.4	48.4
VDebugger	**48.1**	**51.1**

Table 8: Performance of end-to-end VLMs, vanilla visual programming approach (Surís et al., 2023) without debugging, and our VDebugger evaluated on GQA dataset. We experiment with BLIP (Li et al., 2022a) following Surís et al. (2023) as well as the more powerful VLM InstructBLIP (Gupta et al., 2022).

D Visual Programming v.s. End-to-End VLMs

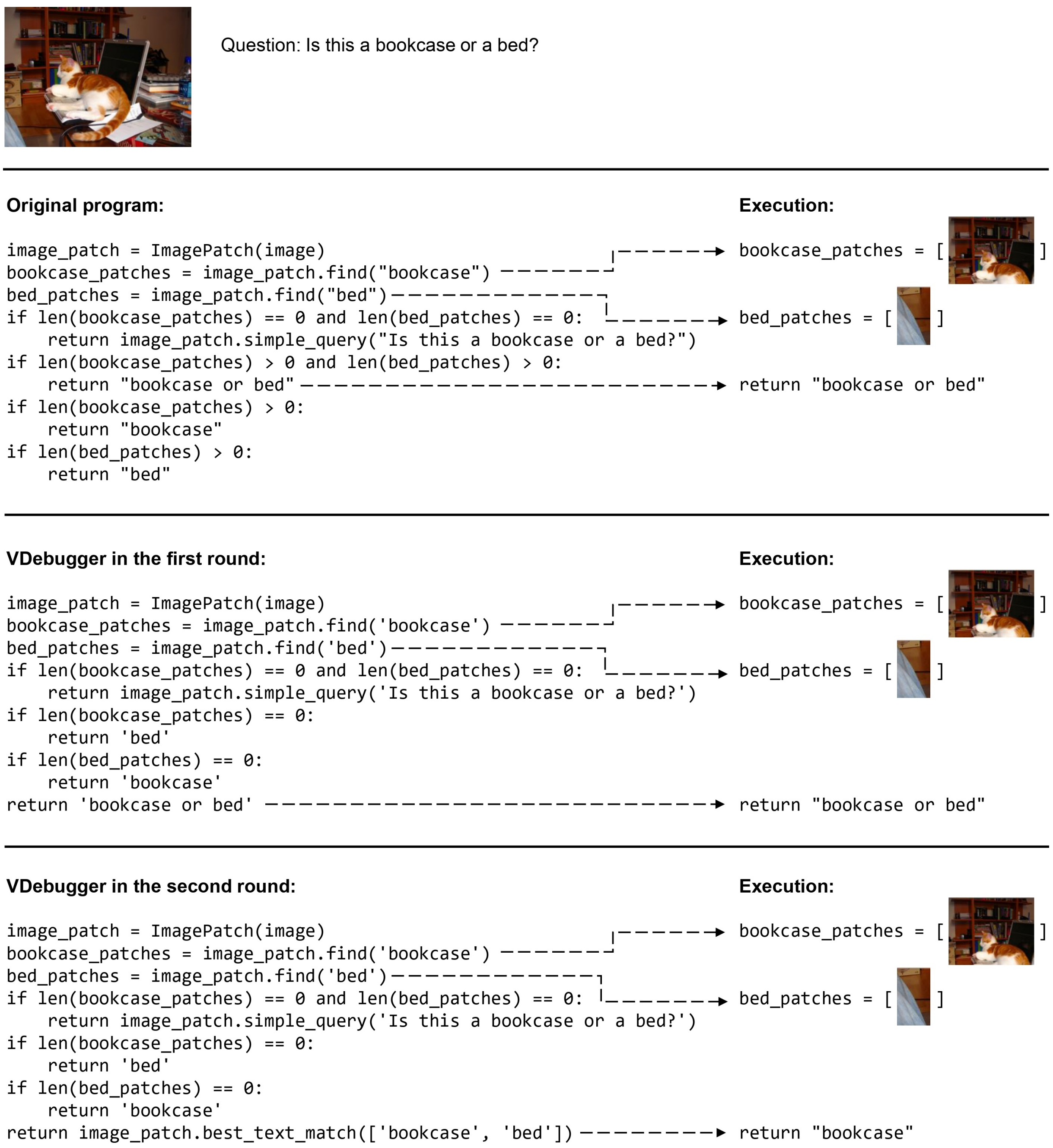
Visual programming and end-to-end VLMs are two different approaches to visual reasoning. Visual programming invokes multiple foundation VLMs through code, while end-to-end VLMs directly take an image as input and generate texts as output. Despite their seemingly different methodologies, visual programming is a complementary technique that can be combined with end-to-end VLMs to offer additional benefits. Firstly, visual programming can integrate with more powerful VLMs to further enhance performance as shown in Table 8. Secondly, despite the rapid development of end-to-end VLMs, they still have difficulty reasoning with compositional concepts such as counting and spatial relationship. Visual programming offer benefits in tasks like such as compositional reasoning, counting, and enhancing interpretability.

E Qualitative Examples

Figure 10 shows an example where more iterations of VDebugger bring benefits.

F Prompts

Figure 11 shows the prompt we use for generating incorrect programs.



Question: Is this a bookcase or a bed?

Original program:

```
image_patch = ImagePatch(image)
bookcase_patches = image_patch.find("bookcase")
bed_patches = image_patch.find("bed")
if len(bookcase_patches) == 0 and len(bed_patches) == 0:
    return image_patch.simple_query("Is this a bookcase or a bed?")
if len(bookcase_patches) > 0 and len(bed_patches) > 0:
    return "bookcase or bed"
if len(bookcase_patches) > 0:
    return "bookcase"
if len(bed_patches) > 0:
    return "bed"
```

Execution:

```
bookcase_patches = [ ]
bed_patches = [ ]
return "bookcase or bed"
```

VDebugger in the first round:

```
image_patch = ImagePatch(image)
bookcase_patches = image_patch.find('bookcase')
bed_patches = image_patch.find('bed')
if len(bookcase_patches) == 0 and len(bed_patches) == 0:
    return image_patch.simple_query('Is this a bookcase or a bed?')
if len(bookcase_patches) == 0:
    return 'bed'
if len(bed_patches) == 0:
    return 'bookcase'
return 'bookcase or bed'
```

Execution:

```
bookcase_patches = [ ]
bed_patches = [ ]
return "bookcase or bed"
```

VDebugger in the second round:

```
image_patch = ImagePatch(image)
bookcase_patches = image_patch.find('bookcase')
bed_patches = image_patch.find('bed')
if len(bookcase_patches) == 0 and len(bed_patches) == 0:
    return image_patch.simple_query('Is this a bookcase or a bed?')
if len(bookcase_patches) == 0:
    return 'bed'
if len(bed_patches) == 0:
    return 'bookcase'
return image_patch.best_text_match(['bookcase', 'bed'])
```

Execution:

```
bookcase_patches = [ ]
bed_patches = [ ]
return "bookcase"
```

Figure 10: **Example where more iterations of VDebugger bring benefits.** The original program results in incorrect answer "bookcase or bed" because the object detection model incorrectly identifies a bed. VDebugger detects the error through the unreasonable return value and attempts the first round of debugging. Although the program structure is significantly changed in this round, the execution still leads to the incorrect answer due to the same issue. In the second round, VDebugger successfully resolves the problem.

~~~
[INST] I am writing code to handle visual question answering tasks by calling computer vision APIs. Some content from the
code is masked (represented as "<MASKED>". Please recover the original code.
My code:
```python
# {QUESTION}
{CODE}
```

Your code should be wrapped in ```python and ```. The code should be exactly the same as my code, except recovering the
masked content.

—

Below are the available APIs and some example usages:

```python
{API_DEFINITION}
```[/INST] Here's the original code with the `<MASKED>` section replaced:
```python
# {QUESTION}
{PROGRAM_SIGNATURE}
~~~

Figure 11: Prompt for generating incorrect program. Here, the blue texts are the prompt, the orange text are the fixed prefix for model generation, and {QUESTION}, {CODE}, {API_DEFINITION}, and {PROGRAM_SIGNATURE} are placeholders to be filled in during actual generation.