PropTest: Automatic Property Testing for Improved Visual Programming

Jaywon Koo[♣] Ziyan Yang[♣] Paola Cascante-Bonilla[◇]
Baishakhi Ray[♡] Vicente Ordonez[♣]
[♣]Rice University [◇] University of Maryland [♡]Columbia University
{jk125, zy47, vicenteor}@rice.edu
pcascant@umd.edu, rayb@cs.columbia.edu

Abstract

Visual Programming has recently emerged as an alternative to end-to-end black-box visual reasoning models. This type of method leverages Large Language Models (LLMs) to generate the source code for an executable computer program that solves a given problem. This strategy has the advantage of offering an interpretable reasoning path and does not require finetuning a model with task-specific data. We propose PropTest, a general strategy that improves visual programming by further using an LLM to generate code that tests for visual properties in an initial round of proposed solutions. Our method generates tests for data-type consistency, output syntax, and semantic properties. PropTest achieves comparable results to state-of-the-art methods while using publicly available LLMs. This is demonstrated across different benchmarks on visual question answering and referring expression comprehension. Particularly, PropTest improves ViperGPT by obtaining 46.1% accuracy (+6.0%) on GQA using Llama3-8B and 59.5% (+8.1%) on RefCOCO+ using CodeLlama-34B.

1 Introduction

Visual reasoning tasks often require multi-hop reasoning that goes beyond surface-level observations. This type or reasoning typically involves complex multi-step processes, external knowledge, or understanding of compositional relationships between objects or entities. End-to-end vision and language models based on deep neural networks trained with huge amounts of data are used to tackle these tasks (Li et al., 2023; Alayrac et al., 2022; Yu et al., 2022; Driess et al., 2023; Li et al., 2022a; Wang et al., 2023). However, these methods often fail at multi-hop compositional reasoning as they aim to solve a wide array of reasoning tasks in a single forward pass. Recent work has proposed Visual Programming as a principled way to tackle visual reasoning (Gao et al., 2023; Surís et al., 2023; Gupta and Kembhavi, 2023; Subramanian et al., 2023). These techniques work by leveraging a Large Language Model (LLM) to generate the logic of a program in the form of its source code that can be used to solve the problem. These methods can combine various tools in complex ways and offer interpretability and the opportunity to diagnose failures in their predicted logic.

Visual programming methods that rely on code generation and program execution to solve a task still rely on end-to-end pre-trained Vision Language Models (VLMs) either as tools that can be invoked by the program or as a *fallback* option when the generated code contains syntax or runtime errors. In other words, if the generated code contains errors, then a default end-to-end VLM is invoked. For these methods to be effective, the generated source code should produce solutions that lead to correct results on average more often than their *fallback* VLM. However, there are still many instances where a generated source code contains no syntax or runtime errors, but the logic of the program produces results that contain incorrect logic to solve the problem. Some of these are easier to spot, such as instances where the code returns the wrong data type, or the wrong type of answer for the given problem (e.g. answering with a location when the question is about a quantity). We posit that code testing and assertion error checking which are established practices in software development, should also help these types of methods in guiding them toward better solutions.

We introduce PropTest, a visual programming framework that generates automatic property test cases to guide code generation and identify logic that is likely to contain errors. Fig. 1 showcases a motivating example for our proposed method. PropTest first generates property test cases using an LLM which probes for data type inconsistencies, syntactic errors, and semantic properties of the results. For instance, in the showcased question

Findings of the Association for Computational Linguistics: EMNLP 2024, pages 8241–8256

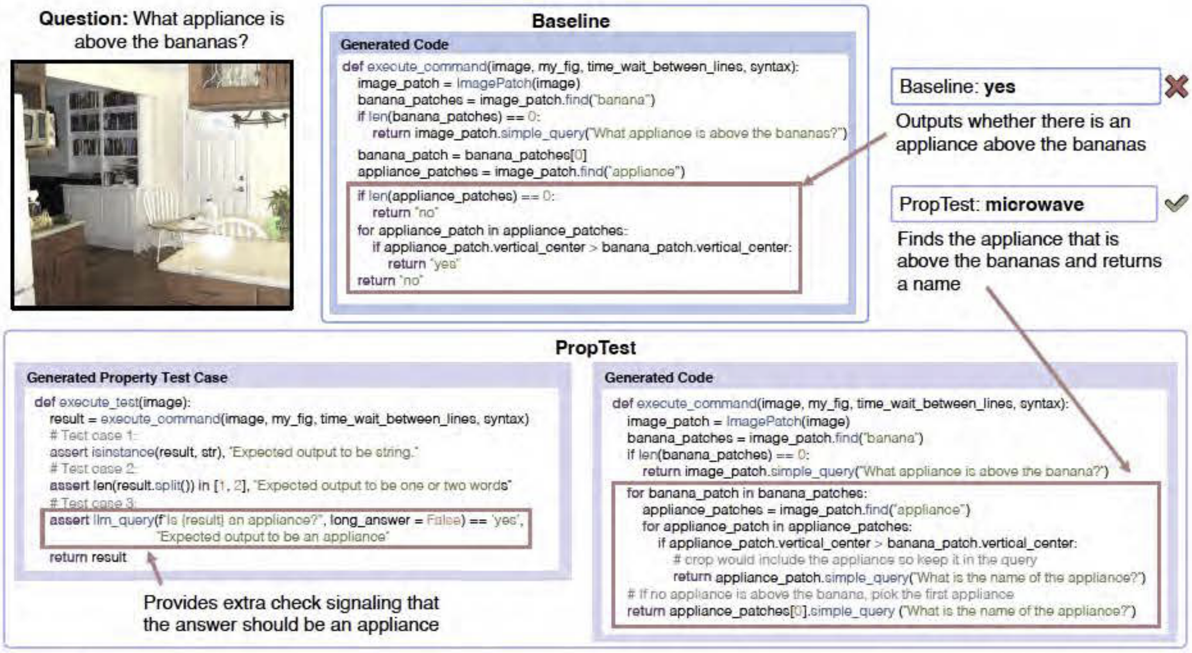


Figure 1: Visual programming methods generate code for a program to solve a vision-and-language task such as VQA. PropTest improves on these methods by automatically generating testing code that probes for several output properties. This is used as additional information when generating code and checking the correctness of the output solutions. As a baseline we use ViperGPT under CodeLlama-7B for this example.

What appliance is above the bananas?, the generated test code anticipates that the answer should be a Python `string` data type, that it should be limited to one or two words, and that the output should be a type of *appliance*. We find that this type of tests consistently help the LLM generate code for the program that is less likely to contain errors.

PropTest can filter out incorrect outputs resulting from errors in logic or failures in dependent modules and redirect these cases when appropriate to the *fallback* VLM. Moreover, PropTest provides additional information about failure cases and in characterizing the type of errors. Additionally, previous visual programming methods rely on closed-source models, making it hard to reproduce results due to continuous version updates, deprecation of older models (e.g., Codex), and usage costs (Gupta and Kembhavi, 2023; Surís et al., 2023; Subramanian et al., 2023). Our main experiments rely exclusively on public models, such as CODELLAMA (Roziere et al., 2023) and LLAMA3 (AI@Meta, 2024), which we expect to serve as stable baselines for future work on this area. We evaluate PropTest on three different tasks: Compositional visual question answering (GQA (Hudson and Manning, 2019)), External knowledge-dependent image question answering (A-OKVQA (Schwenk et al., 2022)), and Visual grounding (RefCOCO and RefCOCO+ (Yu et al., 2016)). Our experiments show that property tests significantly enhance performance across these benchmarks. We also analyze detailed errors from a software engineering perspective (assertion, runtime, and syntax).

Our contributions can be summarized as follows:

- We propose PropTest, a novel framework that uses automatic property test case generation for detecting logic, syntax, and runtime errors, which are used to guide code generation.
- PropTest improves interpretability when errors occur, bridging the gap between LLMs and VLMs on code generation.
- Our proposed method obtains superior results on four benchmarks compared to a baseline model conditioned on four different publicly available LLMs and one proprietary LLM.

2 Method

We introduce PropTest, a framework for leveraging property test code generation. A commonly recommended practice in software development is to write tests first and then write the code for the logic of the program so that it passes the tests. This is the responsible programmer approach to software development. We emulate this approach in PropTest by first generating testing code and then generating

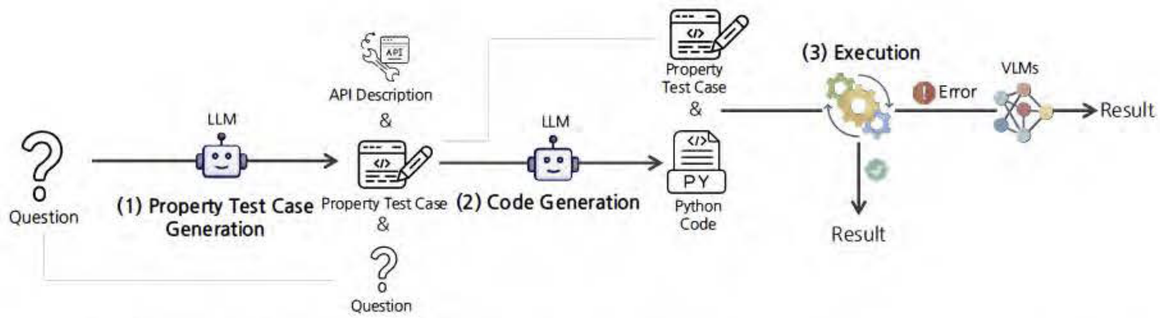


Figure 2: An overview of PropTest. Given an image and a question, the goal is to generate Python code that can be executed to get an answer. PropTest first calls an LLM to generate test cases based on the inferred properties of the answer. Then, the generated test cases are used to improve the quality of Python code.

code to solve the task conditioned on the testing code. Fig. 2 shows an overview of our method.

Let us consider a question such as *What kind of toy is the boy playing with?*, we can easily infer that the answer should be a type of *toy*. We utilize this insight to provide information to the code generation model, narrowing down the search space rather than only relying on single-step prompt optimization. Additionally, generating property test cases is generally simpler than generating code because test cases are shorter and more straightforward. Creating an easier test case first sets a baseline to generate more complex code. Property test cases guide the code generation process and increase the likelihood of generating accurate and effective code solutions.

Our framework first generates property test cases using an LLM by providing a problem statement as a prompt, e.g., a question, or a referring expression. The source code for these generated tests is then added to the prompt of the LLM, along with the original problem statement and detailed API documentation of the available tools or modules. We employ the same API and tools used in ViperGPT (Surís et al., 2023), which also relies on generic functions from the Python programming language. The code generation model then outputs the code solution that addresses the problem statement and returns a plausible result.

We concatenate the generated property test case and the code solution and apply an execution engine where we also provide the visual input. There can be a syntax or runtime error inside the generated main code. An assertion error will occur if the code output does not pass any of the property test cases. If execution proceeds without errors, including syntax, runtime, or assertion errors, the result is returned, and the process concludes. In the event of an error, we default to a task-specific *fallback* VLM and return.

3 Property Test Case Generation

The purpose of using a property tests is to verify whether a generated code works as expected and guide an LLM to generate better code that meets basic properties. The design of property test cases varies based on the data type of the answer due to the different tools (APIs) available for each type. In this section, we explain in detail the design process for prompts used to generate property tests for visual question answering tasks, where the task answer is text (section 3.1) and for visual grounding tasks, where the task answer is an image with bounding boxes (section 3.2).

3.1 Property Tests for Visual Question Answering

Visual question answering tasks contain queries that require multi-hop reasoning or external knowledge. To solve these tasks, we propose two property test case generation strategies along with corresponding in-context prompts to guide the LLM toward the generation of property tests with similar logic. We include our prompts in Appendix A.3.

Basic Property Test Case Generation. This type of test only relies on basic Python functions without using external APIs or tools. As shown in Fig. 3a, this approach is effective when the question mentions several candidates. Furthermore, this strategy can be applied to yes-or-no questions, where it checks the type of the property.

Advanced Property Test Case Generation. For this type of test cases, we also allow the use of tools through an API specification, specifically the use

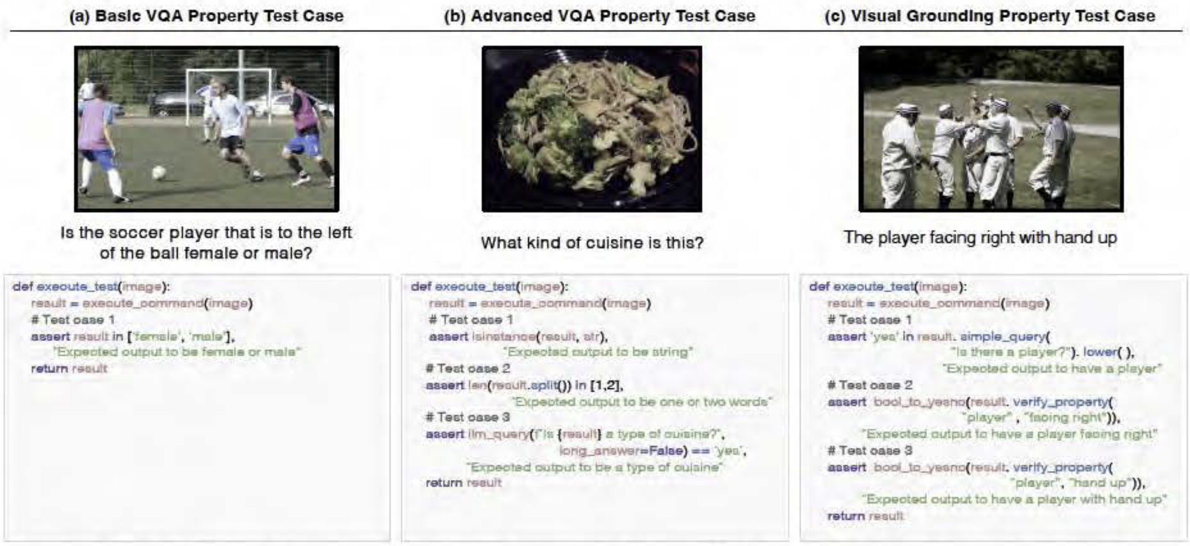


Figure 3: Three different examples of property test cases generated for visual question answering and for visual grounding. The `execute_command()` is the generic name of the generated program code routine and `result` is the output from executing it.

of an LLM that can check the output result through various properties. Particularly, our generated test code can use an `llm_query()` function to construct more advanced assertion statements. Fig. 3b shows an example where given the question *What kind of cuisine is this?*, the first test case checks the return data type, which should be a Python `string`. Then a second assertion checks that the output is just one or two words in length. The third test case checks the semantic property of the returned result. Knowing that the expected answer should be a type of *cuisine*, we use LLM queries in the test case to verify whether the result correctly identifies a *cuisine* type. This effectively narrows the expected result space for the code generation model, helping it produce more accurate solutions.

3.2 Property Tests for Visual Grounding

Visual grounding tasks require returning a bounding box in an image that corresponds to an input text query. To construct property test cases for such tasks, we utilize a set of tools that take images as inputs. Particularly, our test code can use functions such as `simple_query()`, `verify_property()`, and `bool_to_yesno()`. The `simple_query()` function is used to answer straightforward questions about the image, `verify_property()` checks whether an object has a given attribute as a property, and `bool_to_yesno()` converts boolean values into "yes" or "no" responses. As shown in Fig. 3c, given the input referring expression *the player facing right with hand up*, our test case begins by confirming if a player is inside the result bounding box. It then proceeds to verify, in sequence, whether the identified player is facing *right* with *hand up*, thus checking whether the given output is likely to reflect the given query.

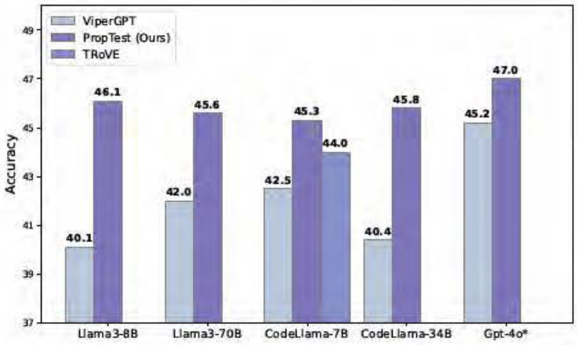
4 Experiments

We introduce the experimental setup (section 4.1), and results on different LLMs (section 4.2)

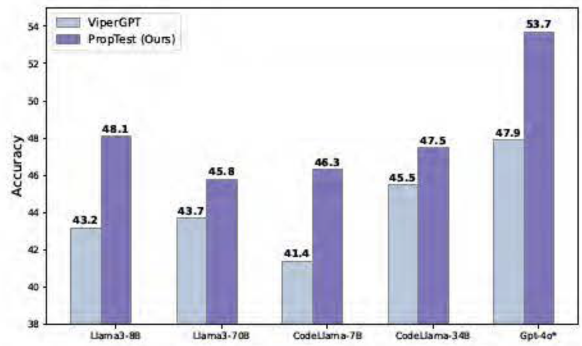
4.1 Experimental Setup

Tasks and Metrics. We validate PropTest on the Visual Question Answering (VQA) and Visual Grounding tasks. For VQA, we evaluate on GQA (Hudson and Manning, 2019), and A-OKVQA (Schwenk et al., 2022), which contain complex multi-hop questions that require compositional reasoning. We use exact matching accuracy as our metric for GQA, where answers must correspond to a single ground truth answer. We use soft accuracy (SAcc) (Antol et al., 2015) for A-OKVQA. For Visual Grounding, we use standard benchmarks, including testA split on RefCOCO and RefCOCO+ (Yu et al., 2016). The evaluation metric is the intersection over union (IoU) score.

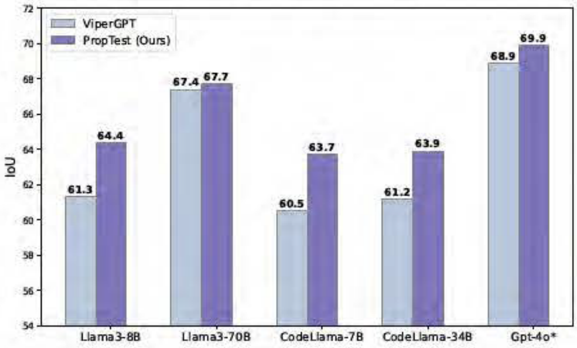
Model Comparison. Similar to prior work, for VQA we use BLIP-2 (Li et al., 2023) as our *fallback* VLM, and GLIP (Li et al., 2022a) for Visual Grounding. The tools and API specifications for PropTest are consistent with those employed by ViperGPT (Surís et al., 2023), ensuring a standardized basis for comparison. Therefore, for our exper-



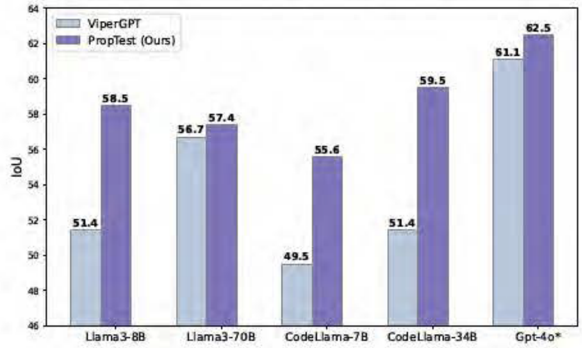
(a) Results on GQA using different LLMs



(b) Results on A-OKVQA using different LLMs



(c) Results on RefCOCO using different LLMs



(d) Results on RefCOCO+ using different LLMs

Figure 4: Comparison of our method against visual programming methods with different LLMs across two tasks, four benchmarks. We report Accuracy on two visual question answering benchmarks, and IoU on two visual grounding benchmarks. GPT-4o* results are only tested on 500 subsamples.

imental comparisons, we compare PropTest with other code generation models - ViperGPT (Surís et al., 2023), and end-to-end models including BLIP-2 (Li et al., 2023) and GLIP (Li et al., 2022a). The only other publicly available neuro-symbolic method is the concurrent work from Wang et al. (2024), which uses CODELLAMA-7B.

Implementation Details. We implement PropTest using the open-source LLMs including CODELLAMA (7B, 34B) (Roziere et al., 2023) and LLAMA3 (8B, 70B) (AI@Meta, 2024) for code generation. The specific implementation details are described in Appendix A.

4.2 Results

Quantitative Results. One common concern with previous work is that evaluations performed with API-based black-box models (e.g. GPT-3.5, GPT-4) are hard to reproduce and track as there are many different upgrades on these models. They can also be discontinued (e.g. Codex), making past work non-reproducible. Our main experiments are conducted using CODELLAMA and LLAMA3, which are publicly available and free to use for research purposes. As part of our work, we will also release an API-free implementation of ViperGPT. Additionally, we evaluate PropTest using GPT-4o to contextualize our work. We limit our evaluation to 500 randomly sampled subsets for each data split, specifically for GPT-4o.

Our main results are shown in Fig. 4. Overall, PropTest shows improvements over ViperGPT in all settings. The model that provides the most gain varies by dataset, smaller models such as CodeLlama-7B and Llama3-8B tend to benefit more with PropTest (e.g., +6.0% on GQA with Llama3-8B, +4.9% on A-OKVQA with both LLMs and +7.1% on RefCOCO+ with Llama3-8B) but even larger models also show gains, including GPT-4o. Notably, CodeLlama-34B outperforms or shows greater improvement over ViperGPT compared to Llama3-70B across all datasets. This is due to CodeLlama-34B's training with code, making it superior in code generation despite its smaller size relative to Llama3-70B. We also noticed that GPT-4o shows the best results on all datasets.

Moreover, PropTest outperforms the *fallback* VLMs we rely on, while also providing enhanced interpretability in all settings. The *fallback* VLM

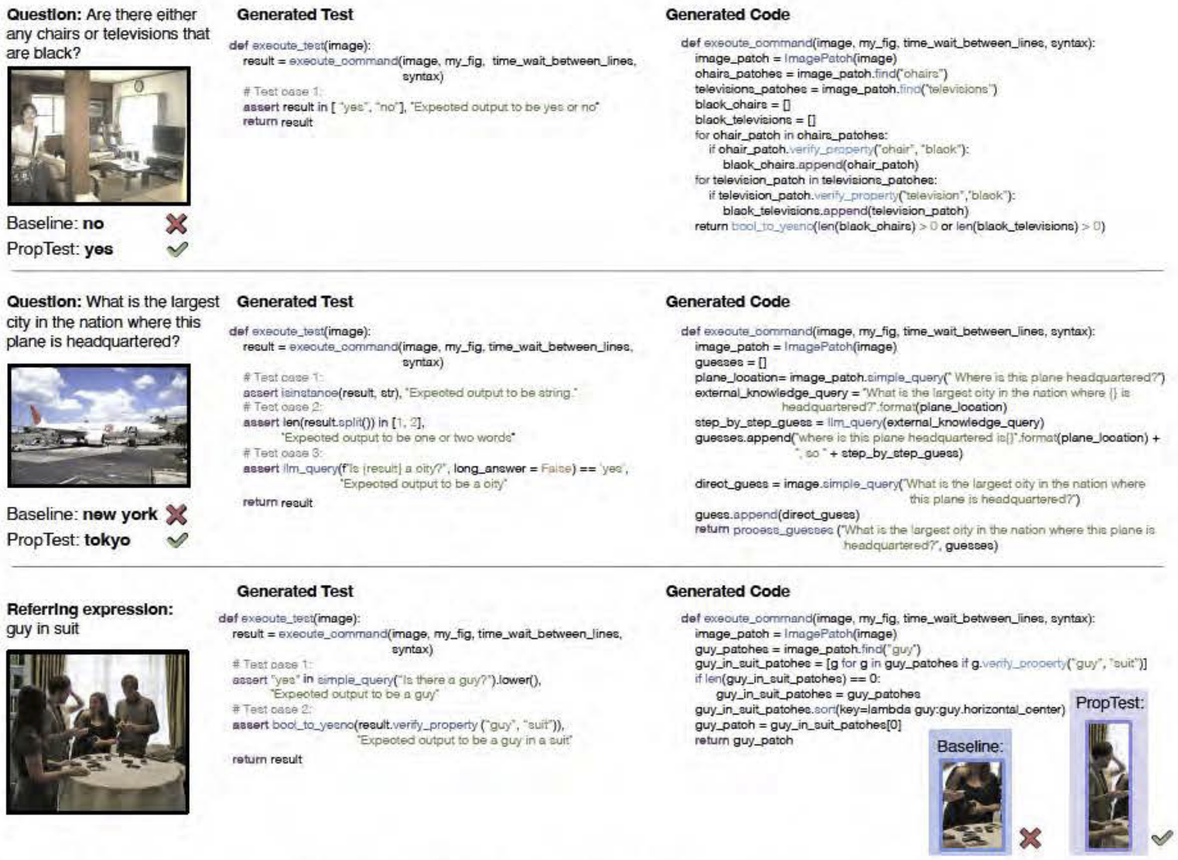


Figure 5: Example results on GQA, A-OKVQA and RefCOCO. We show cases where PropTest succeeds but the baseline ViperGPT fails. Input questions and answers are shown on the left, generated property test cases in the middle, and code on the right.

results are 42.4%[1] on GQA, 45.1% on A-OKVQA, 55.0% on RefCOCO, and 52.2% on RefCOCO+. While ViperGPT sometimes underperforms compared to VLMs depending on the LLMs, PropTest remains robust, performing well on all models, including smaller ones.

We did not compare our models to previous visual programming methods that use closed API-based LLMs (Yuan et al., 2024; Subramanian et al., 2023; Chen et al., 2023b), as it would be unfair or unfeasible due to the different or deprecated LLMs used in those models.

Qualitative Results. Fig. 5 shows representative examples of the types of property tests that get generated and output programs. By leveraging property test cases, PropTest generates a code with correct logic and results on cases that fail to return a correct answer due to logical errors on ViperGPT. In addition, we illustrate cases with logical errors that produce assertion errors in Appendix C. By checking on logical errors, PropTest provides extra interpretability on the reason for failure. More qualitative results are shown in Appendix B.

5 Error Analysis & Discussion

In this section, we first focus on the question: *What types of errors does the code generation model produce?* We analyze the errors in the generated code from ViperGPT and PropTest across datasets, categorizing them into three basic Python errors: Assertion, Runtime, and Syntax errors. We report results using Llama3-8B in Table 1.

We first note that code generation models produce more errors in visual grounding tasks than in VQA tasks. This is because visual grounding involves stricter assertions in test cases, leading to a higher frequency of assertion errors. In visual grounding, all test cases check the result `image_patch` for specific properties, and errors occur when objects or properties are missing. In contrast, VQA often involves simpler yes-or-no checks, where incorrect results might still pass the test. Furthermore, RefCOCO+ has a higher overall error rate compared to RefCOCO due to its com-

[1]Result under the same setting as ViperGPT, differing from the original work (Li et al., 2023)

Dataset	Method	# Errors	Assert.	Runt.	Syntax
GQA	ViperGPT	411 (3.3%)	-	322	89
	PropTest	1264 (10.0%)	1001	227	36
A-OKVQA	ViperGPT	11 (1.0%)	-	9	2
	PropTest	174 (15.2%)	169	3	2
RefCOCO	ViperGPT	281 (5.0%)	-	240	41
	PropTest	871 (15.4%)	617	241	13
RefCOCO+	ViperGPT	435 (7.6%)	-	386	49
	PropTest	1132 (19.8%)	875	250	7

Table 1: Error Analysis on ViperGPT (Surís et al., 2023) and PropTest across benchmarks using Llama3-8B including runtime and syntax errors.

plex queries. The simpler queries in RefCOCO make PropTest generate test cases that accurately identify the target object, resulting in fewer errors. Detailed analysis with examples is in Appendix C.

We also find that due to additional assertion errors, PropTest has higher overall errors compared to ViperGPT. Nevertheless, PropTest notably reduces runtime and syntax errors on three datasets (e.g., 322 $\rightarrow$ 227 runtime, 89 $\rightarrow$ 39 syntax errors in GQA). This reduction indicates that the inclusion of property test cases enhances code generation quality in the aspects of runtime and syntax errors. However, the increase in assertion errors, leading to a rise in total errors, implies that PropTest relies more on the *fallback* model. This raises the question: *Does the performance gain of PropTest come from an increased dependence on VLMs?*

To address this, we compare the performance of ViperGPT and PropTest without using the *fallback* model for error handling, as shown in Table 2. We evaluate "w/o fallback models" in Table 2 over all cases and count the case as wrong whenever an error occurs (e.g., assertion, runtime, and syntax error) or the answer is incorrect. When the case fails the property test, it will generate an assertion error, and we count it as wrong. Across all datasets, PropTest either outperforms or performs on par with ViperGPT, demonstrating that the performance gain is from improved code quality rather than increased reliance on VLMs.

Now, we move on to another question: *How does running a test case during execution help when there is an error?* To address this, we compare PropTest with an approach where property test cases are only provided for code generation but are not executed to catch errors ("w/o running test" in Table 2). Our findings show that running test cases

	w/o VLMs as fallback		w/ VLMs as fallback	
Dataset	ViperGPT	PropTest	PropTest *w/o running tests*	PropTest
GQA	39.1	43.8	45.8	46.1
A-OKVQA	42.8	42.8	47.3	48.1
RefCOCO	60.1	61.6	63.8	64.4
RefCOCO+	50.2	55.8	58.1	58.5

Table 2: Ablation study on the reliance on Visual Language Models (VLMs) for error handling in generated code and the impact of executing test cases.

ViperGPT	Incorrect		Correct	
PropTest	Correct	Incorrect	Correct	Incorrect
GQA	86 (11.30%)	303 (39.82%)	297 (39.03%)	75 (9.86%)
A-OKVQA	53 (6.74%)	356 (45.29%)	358 (45.55%)	19 (2.42%)
RefCOCO	278 (43.99%)	154 (24.37%)	159 (25.16%)	41 (6.49%)
RefCOCO+	119 (18.25%)	169 (25.92%)	316 (48.47%)	48 (7.36%)

Table 3: Accuracy comparison of PropTest and ViperGPT (Surís et al., 2023) when both models generate outputs with correct types using Llama3-8B. We show the counts and percentages of each correct/incorrect combination.

in the presence of errors increases accuracy, indicating that our generated property test cases are effective at detecting incorrect code (e.g., +0.8 in A-OKVQA).

Furthermore, we ask another question: *Does the PropTest improve the quality of the code in cases where the baseline also generates correct output types?* To tackle this, we compare the results where both the ViperGPT (Surís et al., 2023) and PropTest produced correct output types. To extract the samples where both the ViperGPT and PropTest produced correct output types, we run generated property tests on the outputs of the ViperGPT. We sampled 1000 subsets from each benchmark and gathered the samples where the output of the code solution passed the property tests in both ViperGPT and PropTest. We used the code solutions by Llama3-8B. Since A-OKVQA uses soft accuracy as a metric, we assume the output is correct when the soft accuracy is larger than 0.5. We consider the result to be correct if the IoU exceeds a threshold of 0.7 for RefCOCO and RefCOCO+. The results shown in Table 3 indicate that PropTest consistently outperforms ViperGPT. Across all benchmarks, there are more cases where PropTest produces correct answers while ViperGPT is incorrect, compared to the reverse scenario. Particularly, in GQA, among the cases where both models produced correct out-

Method	Acc.	# Errors	Assert.	Runt.	Syntax
Basic VQA	45.6	732 (5.8%)	469	232	31
Advanced VQA	46.1	1264 (10%)	1001	227	36

Table 4: Error analysis on GQA dataset using basic and advanced property tests using Llama3-8B, including runtime and syntax errors. APIs are used for the Advanced VQA property test cases, where only basic Python functions are used in Basic VQA.

put types, PropTest provided correct answers while the ViperGPT was incorrect in 11.30% of the cases. Conversely, ViperGPT was correct while PropTest was incorrect in 9.86% of the cases. These results demonstrate that even with information about the type (properties), property tests lead the code generation process toward more accurate solutions.

6 Property Test Analysis

In this section, we investigate generated property tests in depth by comparing two types of VQA property test cases (section 6.1) and evaluating the generated property test cases (section 6.2).

6.1 Basic vs Advanced Property Tests

Table 4 shows the accuracy and error analysis of two types of VQA property test cases using Llama3-8B. Advanced property test cases have higher accuracy compared to basic tests. Using advanced property test case generation produces almost twice as many errors as basic property test case generation. This is due to an extra semantic property test, which leads to more assertion errors. Advanced property test cases will be longer and more complicated than basic test cases, which causes more syntax errors (e.g., 31 $\rightarrow$ 36).

6.2 Generated Property Test Evaluation

We first evaluate our generated property tests on correctness by using the answers. If an answer passes the generated test, we count it as correct. We report this as accuracy in Table 5. We also examine the quality of our property test cases by using toxicity rate (Chen et al., 2022). If the produced results pass the test while the answer fails the test, we assume the test case is *toxic*. Advanced VQA property test cases have lower accuracy and higher toxic rates compared to basic VQA tests because they generate complicated property test cases that check semantic properties using tools.

Moreover, we present a 2×2 confusion matrix for the advanced property test cases generated on

Method	Dataset	Acc.	Toxic rate
Basic VQA	GQA	95.7%	0.03%
Advanced VQA	GQA	91.7%	0.04%

Table 5: Accuracy and toxic rate of generated property test cases on GQA with Llama3-8B. APIs are utilized in Advanced VQA property test cases, while only basic Python functions are used in Basic VQA.

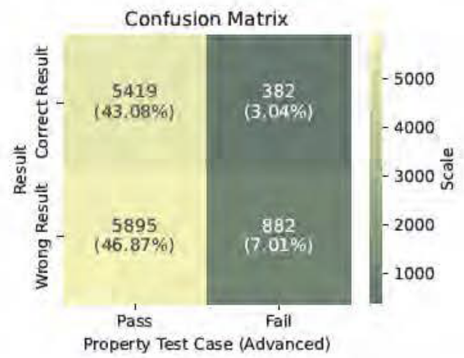


Figure 6: Confusion Matrix of the generated advanced property test cases on GQA using Llama3-8B. We show the counts of correct and incorrect results, further divided by whether they passed or did not pass the generated property test case.

GQA using Llama3-8B in Fig. 6. We define True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) as follows:

- TP: cases where the correct result passes the property case
- TN: cases where the wrong result fails the property case
- FP: cases where the wrong result passes the property case
- FN: cases where the correct result fails the property case

The matrix shows a high number of false positives, primarily due to the flexibility of VQA property test cases. For example, these tests often check for binary answers (yes or no), which can pass even if the result is incorrect. The confusion matrix for the basic property test case and for the visual grounding test case are provided in Appendix D.

Additionally, we conducted an experiment with the Oracle property tests on randomly sampled 100 subsets from GQA and RefCOCO. We created the oracle property tests by manually fixing errors in the generated property tests using Llama3-8B. As shown in Table 6, we can see an improvement when using oracle property tests. Our oracle property

Dataset	ViperGPT	PropTest	PropTest *w. Oracle Property Tests*
GQA (Acc.)	42.0	46.0	50.0
RefCOCO (IoU)	48.2	60.4	62.5

Table 6: Comparison of ViperGPT, PropTest, and PropTest with Oracle Property Tests for GQA (Accuracy) and RefCOCO (IoU). We sample 100 subsets from each benchmark and use Llama3-8B.

tests could still be further refined as we only limited ourselves to fixing mistakes in the automatically generated property tests. Importantly, this result shows that better property tests lead to further improvement under our method, generating better code and signaling that PropTest works for the right reasons.

7 Related Work

End-to-end vision language models (VLMs) are generally trained on large datasets containing images paired with text descriptions or instructions (Li et al., 2023; Alayrac et al., 2022; Yu et al., 2022; Driess et al., 2023; Li et al., 2022a; Liu et al., 2023; Guo et al., 2023; Wang et al., 2023). By learning correlations between visual features and linguistic patterns, VLMs can understand sophisticated relations between images and text using a single forward pass through a deep neural network. These models, however large, are still bounded by what functions can be learned and encoded in their model weights.

On the other hand, with the rise of LLMs for code generation in recent years (Chen et al., 2021; Roziere et al., 2023; Guo et al., 2024; Nijkamp et al., 2023; Luo et al., 2023), a recent set of methods in visual recognition have adopted the use of these models to solve visual tasks using a hybrid approach where VLMs and other computer vision models are used as tools by one of these code generation LLMs to generate a program that can solve a given task (Surís et al., 2023; Gupta and Kembhavi, 2023; Subramanian et al., 2023). This type of neuro-symbolic reasoning model was referred to as *Visual Programming* by Gupta and Kembhavi (2023). These methods lead to an executable program that decomposes complex visual reasoning queries into interpretable steps, which are then executed to produce results. These methods define APIs (tools) they use during the execution, with functions mapped to off-the-shelf vision modules such as object detectors (He et al., 2017; Li et al., 2022a), depth estimators (Ranftl et al., 2022), among many others. These methods benefit from not needing extra training while enhancing reasoning capabilities and interpretability. The performance of these methods depends on the tools or APIs the model leverages and the quality of the generated code. One line of work focuses on creating better and more diverse toolsets to improve accuracy (Yuan et al., 2024; Chen et al., 2023b; Wang et al., 2024). Efforts to enhance code quality have been made by code refinement techniques, incorporating various types of feedback, such as visual, textual, error-related, and human feedback (Gao et al., 2023). Self-tuning mechanisms have also been explored to optimize model hyperparameters automatically (Stanić et al., 2024). Training a code debugger to detect and fix the code has been investigated (Wu et al., 2024). Our method builds upon these findings, aiming to maximize the efficacy of VLMs (Li et al., 2023, 2022a) through property testing that is more specific to the visual domain.

Meanwhile, writing test cases is a common technique used by software developers to avoid writing code that contains programming errors. Similarly, it has enhanced code generation in code contest tasks. Test cases are used to detect errors and give feedback for self-refinement (Le et al., 2023; Chen et al., 2023a; Olausson et al., 2023). Another line of work generates test cases by mutating existing test inputs (Li et al., 2022b) or by using LLMs (Chen et al., 2022). Our research, however, differs from these methods by generating property tests that check different properties of the output, and utilizing these tests as an additional input when generating code.

8 Conclusion

This paper presents PropTest, a novel framework for leveraging property test code generation to improve the quality of generated program code in visual programming. PropTest shows consistent improvements on VQA and Visual Grounding with four open-source code generation LLMs. Interestingly, we find that common software development advice which dictates that we should first write testing code before implementing new functionality, also applies to LLM-based code generation.

Acknowledgements: Our work was partially funded by the Ken Kennedy Institute at Rice University and NSF Award #2221943, #2201710, #1845893.

9 Limitations

PropTest is an initial work that applies property test case generation for visual reasoning. Although the PropTest is a very promising framework for visual reasoning, there are several limitations that can be mentioned. First, PropTest requires an extra LLM inference to generate property test code, which will require extra time and resources, but we expect that as faster LLMs are supported in the future, this becomes less of an issue. Additionally, PropTest needs to design a specific property test case prompt depending on the type of the result (image or text). This can be resolved by adding an LLM that can design an automatic prompt depending on the task.

Although less common, the code generated for the property tests themselves could also contain logical errors which limits their usefulness, and additionally, the tools they rely upon could also introduce errors. These limitations can be resolved by integrating visual programming works focused on tool generation (Yuan et al., 2024; Wang et al., 2024) or self-refining (Gao et al., 2023; Stanić et al., 2024) to enhance the code generation skills. Finally, although the discussed datasets show strong performance, numerous visual reasoning tasks, such as video causal/temporal reasoning, remain to be explored in future research.

References

AI@Meta. 2024. Llama 3 model card.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. 2015. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2425–2433.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023a. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Zhenfang Chen, Rui Sun, Wenjun Liu, Yining Hong, and Chuang Gan. 2023b. Genome: Generative neuro-symbolic visual reasoning by growing and reusing modules. *arXiv preprint arXiv:2311.04901*.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. 2023. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jiaxian Guo, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Boyang Li, Dacheng Tao, and Steven Hoi. 2023. From images to textual prompts: Zero-shot visual question answering with frozen large language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10867–10877.

Tanmay Gupta and Aniruddha Kembhavi. 2023. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2961–2969.

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 (CVPR)*, pages 6693–6702.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*.

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. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Liunian Harold Li, Pengchuan Zhang, Haotian Zhang, Jianwei Yang, Chunyuan Li, Yiwu Zhong, Lijuan Wang, Lu Yuan, Lei Zhang, Jenq-Neng Hwang, et al. 2022a. Grounded language-image pre-training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10965–10975.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. 2023. Improved baselines with visual instruction tuning. *arXiv preprint arXiv:2310.03744*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations (ICLR)*.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations (ICLR)*.

René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. 2022. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(3):1623–1637.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Dustin Schwenk, Apoorv Khandelwal, Christopher Clark, Kenneth Marino, and Roozbeh Mottaghi. 2022. A-okvqa: A benchmark for visual question answering using world knowledge. In *European Conference on Computer Vision (ECCV)*, pages 146–162. Springer.

Aleksandar Stanić, Sergi Caelles, and Michael Tschannen. 2024. Towards truly zero-shot compositional visual reasoning with llms as programmers. *arXiv preprint arXiv:2401.01974*.

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. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 747–761, Toronto, Canada. Association for Computational Linguistics.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Weihan Wang, Qingsong Lv, Wenmeng Yu, Wenyi Hong, Ji Qi, Yan Wang, Junhui Ji, Zhuoyi Yang, Lei Zhao, Xixuan Song, et al. 2023. Cogvlm: Visual expert for pretrained language models. *arXiv preprint arXiv:2311.03079*.

Zhiruo Wang, Daniel Fried, and Graham Neubig. 2024. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*.

Xueqing Wu, Zongyu Lin, Songyan Zhao, Te-Lin Wu, Pan Lu, Nanyun Peng, and Kai-Wei Chang. 2024. Vdebugger: Harnessing execution feedback for debugging visual programs. *Preprint*, arXiv:2406.13444.

Jiahui Yu, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini, and Yonghui Wu. 2022. Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*.

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*, pages 69–85. Springer.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. 2024. Craft: Customizing llms by creating and retrieving from specialized toolsets. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Yan Zeng, Xinsong Zhang, and Hang Li. 2022. Multi-grained vision language pre-training: Aligning texts with visual concepts. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 25994–26009. PMLR.