Self-play through Computational Runtimes improves Chart Reasoning

Tautvydas Misiunas* **Hassan Mansoor** **Jasper Uijlings** **Oriana Riva** **Victor Carbune**
Google DeepMind

Abstract

Vision-language models (VLMs) achieve impressive zero-shot performance on multimodal reasoning tasks. Oftentimes, best reported performance is achieved with a zero- or a few-shot prompt. Asking the model solving the same task using a different approach, such as through code generation, can hurt performance. In addition, training sets are typically no longer useful for improving model performance through few-shot learning, due to their use in training. Indeed, we observe that auto-prompting techniques such as DSPy (Khattab et al., 2023), when applied on training sets, do not produce few-shot examples that significantly improve validation performance. Further, when used in conjunction with program-of-thought prompting, performance becomes even worse.

Our work overcomes these limitations by introducing a novel self-play programming interface which leverages the ability of VLMs to first generate code to decompose a complex visual reasoning task in sub-tasks, then use itself, or other models, as a tool to solve decomposed tasks. Our approach enables DSPy to not suffer from performance drops, when applied iteratively on training sets. Furthermore, it outperforms zero-shot baselines on difficult chart reasoning benchmarks. We report the performance of our approach on ChartQA, PlotQA and ChartFC. This enables large models, such as Gemini or GPT to autonomously learn how to use themselves as tools and iteratively improve without the need for additional data.

1 Introduction

The ability of vision-language models (VLMs) to understand scientific charts is key to enable automated and efficient data analysis. The diversity and complexity of these charts make this a challenging problem, as evidenced by the emergence of several benchmarks such as ChartQA (Masry et al., 2022), PlotQA (Methani et al., 2020), and ChartFC (Akhtar et al., 2023).

Most recent breakthroughs have been obtained through pre-training and fine-tuning using carefully constructed data mixtures and scalable model architectures (Anil et al., 2023; OpenAI et al., 2024; McKinzie et al., 2024). We posit that the benefits for downstream users of these models stem more from the ability and flexibility to perform visual in-context learning (ICL) on any given task, rather than the performance on the specific downstream task. Not only the task of interest may not be present in the data mixture, but the various training stages may inadvertently degrade performance because of the large number of tasks involved. Therefore, an emerging class of approaches leverages visual ICL capabilities (Alayrac et al., 2022) for solving such tasks without modifying the base model; an LLM orchestrates tools (Hu et al., 2023), writes code (Surís et al., 2023; Stanić et al., 2024; Gupta and Kembhavi, 2023) or a mix of both (Castrejon et al., 2024; Yang et al., 2023; Yao et al., 2023; Khattab et al., 2023). Beyond a certain scale (McKinzie et al., 2024), ICL successfully enables combining image understanding with code generation.

Our work introduces a technique for performing iterative few-shot example mining, through an approach that improves upon performance of existing methods such as DSPy (Khattab et al., 2023) by extracting additional learning signals from existing training sets. We focus our method development on some of the most difficult benchmarks available today for chart understanding, specifically ChartQA (Masry et al., 2022), PlotQA (Methani et al., 2020), and ChartFC (Akhtar et al., 2023).

We leverage a self-refinement approach (Madaan et al., 2023) to overcome trivial execution errors and the task metric using the golden labels from the training set (Stanić et al., 2024). Our method is visualized in Figure 1. The method treats *training*

*Correspondence to: tautis@google.com

Findings of the Association for Computational Linguistics: ACL 2025, pages 10731–10746

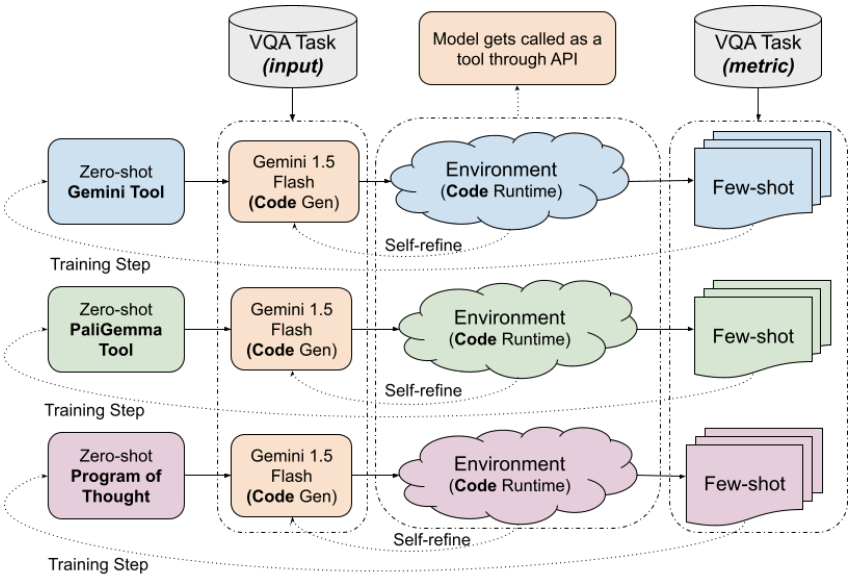


Figure 1: Self-play environments can be used for probing and learning how to use tools/models. During training stage, few-shot examples are iteratively replaced with better ones, whereas during inference the most suitable ones for an example are selected.

sets as self-play environments for vision-language models (VLMs) to improve their chart reasoning abilities. Instead of merely using the training data as a static source of examples for supervised learning or few-shot prompting, the training set becomes an active arena where the model can iteratively learn and refine its problem-solving strategies.

It iteratively expands the few-shot examples, thus constructing several few-shot pools (for each initial zero-shot prompt) or mixed-shot pools (combining across types of zero-shot prompts). These pools are then used at inference time. Unlike prior work (Gupta and Kembhavi, 2023; Surís et al., 2023), this process does not require any human supervision. For powerful VLMs, such as Gemini (Anil et al., 2023), which both generate code and perform visual reasoning, the training loop resembles self-play (Silver et al., 2017), where the model learns how to best use itself to solve a VQA task. Furthermore, it taps into improved reasoning capabilities of models which were also trained on code generation datasets (Ma et al., 2023).

The process of replacing the zero-shot prompt with few-shots that matched the training label is similar to that of DSPy (Khattab et al., 2023). However, we show that, without our method, DSPy fails to improve over zero-shot regime, even degrading when program-of-thought is used. In our initialization, we seed the process with multiple initial zero-shot prompts which differ through the type of code generated (e.g., program of thought (Chen et al., 2023a), API-based (Patil et al., 2023)). Uniquely introduced by our approach, we extend the iterative process for up to 10 iterations and we introduce a novel few-shot selection technique that builds upon the learning signals from the multiple iterations.

The computational environment constructed through code generation couples values inferred directly from the image (e.g., values of bars or text labels in a chart) with tool (model)'s inference outputs, through basic arithmetic computations (Patil et al., 2023; Schick et al., 2023; Cai et al., 2024). A main challenge to address in this setup is how to automatically choose APIs to be used in prompts for invoking individual models (as tools). Prior work has shown how LLMs are capable to perform self-debugging and self-correction (Stanić et al., 2024). We study two types of computational environments. The first one extends program-of-thought (Bi et al., 2023) conditioning on multimodal input, while the second type makes use of an indirection API, enabling the model to focus on orchestration and decomposition aspects.

The self-play environment surfaces and main-

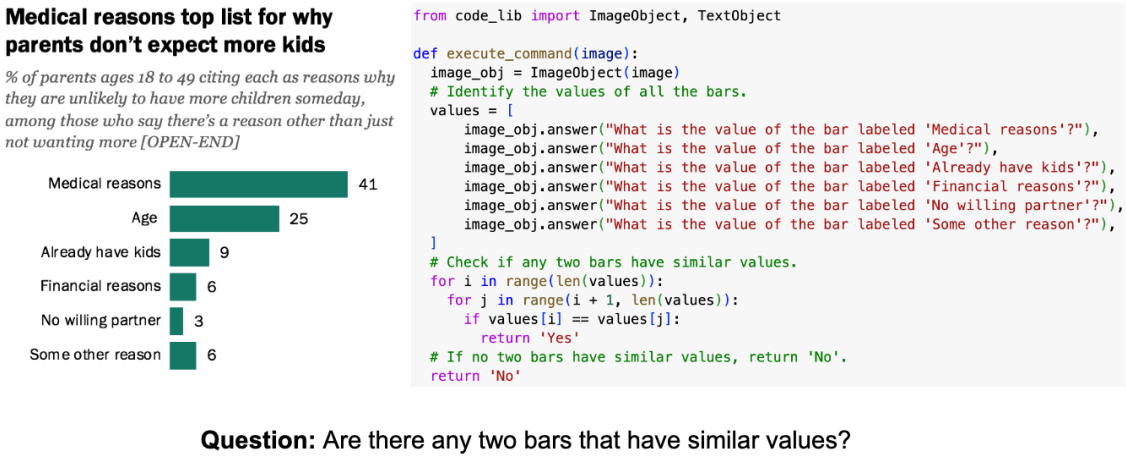


Figure 2: Example of a compositional reasoning question from ChartQA (Masry et al., 2022). Gemini predicts code conditioned on the image, re-using itself through an API for visual information lookup (`image_obj.answer`) and leveraging the computational environment for the arithmetic comparison (i.e., comparing bar values).

tains as few-shots those training examples where the model successfully performed such orchestration. Furthermore, it enables a model such as Gemini, to be both an orchestrator, by predicting the code which decomposes the problem and leverages tools, and a tool, by being called by the underlying API. By keeping the training step outputs for which results match training labels, we form few-shot pools for each type of zero-shot prompt provided. We provide an example question and model generated solution using our work in Figure 2.

Our contributions can be summarized as follows: **(i)** we introduce a simple, yet powerful API, for constructing self-play environments for VLMs to reuse existing training sets, **(ii)** we show the effectiveness of our approach by outperforming zero-shot performance on ChartQA, PlotQA and ChartFC for Gemini and GPT without additional data, and **(iii)** we show that our approach can overcome scaling limitations encountered in prior work, when applied within DSPy (Khattab et al., 2023).

2 Related Work

Strong capabilities of recent multimodal models in zero-shot regimes indicate continued improvements on difficult reasoning tasks, particularly in the image understanding domain. Yet, it is expected that capabilities may differ by modality, due to specific technical challenges stemming from modality-specific tokenization (Borsos et al., 2023; Fu et al., 2022; Dosovitskiy et al., 2021) and availability of mixed-modality pre-training datasets (McKinzie et al., 2024; Fu et al., 2022) to learn inter-modal dependencies. While impressive results are reported for text modality in many-shot regime (Agarwal et al., 2024), earlier few-shot results on images flattened more quickly (Alayrac et al., 2022) and recent work on classification tasks (Jiang et al., 2024) highlighted scaling challenges. Our approach would directly be accelerated by further breakthroughs in multimodal many-shot regime.

Auto-prompting methods We ground our contributions in auto-prompting approaches, such as AutoCoT (Zhang et al., 2022), AutoDirected CoT (Schulhoff et al., 2024) and DSPy (Khattab et al., 2023). AutoCoT generates reasoning traces and augments an existing pool of few-shot examples, while AutoDiCoT further expands by using a development set and constructing different reasoning traces based on whether it was correctly labeled or not. Other methods, such as MEAL (Köksal et al., 2023), provide ensemble strategies, while Skill-Based Few-Shot Selection (An et al., 2023) relies on selecting task-specific examples through embedding-based retrieval.

DSPy We identify DSPy (Khattab et al., 2023) as most closely related. DSPy particularly addresses variability in prompt-based fine-tuning, a critical aspect highlighted by methods such as MEAL. Similar to MEAL, DSPy utilizes ensemble strategies and active prompt selection to mitigate prompt variability. DSPy introduces a programming model that defines a computation graph through which language model pipelines are invoked. In DSPy, the user can make use of a compilation stage, where

a metric function can be used for identifying the best examples to use in a few-shot prompt. In particular, the introduction of generic 'bootstrap' method which dynamically selects in-context examples: given a set of examples with ground truth annotations, it attempts to solve them (e.g. using chain-of-thought). Given the set of solved examples, it then bootstraps N examples which together provide the best in-context examples to solve most questions of the dataset. In contrast to DSPy, we focus on multimodal tasks which are solved through code generation and tool calls. This leads to an increasing pool of possible in-context examples to choose from. Our work aims to overcome limitations observed with DSPy and other auto-prompting strategies where they plateau or degrade performance on training sets. Instead, our work aims to continue extracting useful learning signals.

Learning to use tools Toolformer (Schick et al., 2023) introduced a pre-training and fine-tuning recipe for augmenting LLMs with capabilities to use tools. ReAct (Yao et al., 2023) leverages few-shot capabilities, and has recently been extended (Yang et al., 2023; Castrejon et al., 2024; Hu et al., 2023; Gao et al., 2023) to the multimodal domain. Multi-agent frameworks such as AutoGen (Wu et al., 2023) are examples of mainly natural language based environments for learning how to collaborative solve tasks. There, code generation is mainly a tool, however as an environment, it can also be effective at scaling tool use to thousand of APIs (Patil et al., 2023), with selecting among prompt libraries for using specific tools depending on the task being a key element (Paranjape et al., 2023). Our proposed technique treats predicted code as environments where agents learn to use themselves, a less explored angle (Stanić et al., 2024; Surís et al., 2023; Subramanian et al., 2023).

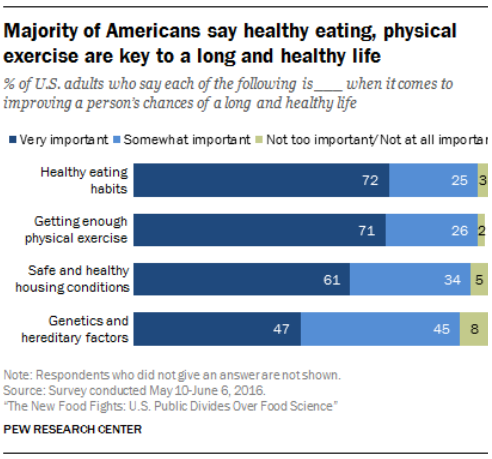
Visual QA Solving visual question-answering tasks poses numerous challenges for VLMs, that are typically solved with general image representation techniques (Alayrac et al., 2022; Chen et al., 2023b; Baechler et al., 2024) or question-conditioned ones (Ganz et al., 2024; Yang et al., 2024). VLMs highly specialized on types of tasks are another possibility (Carbune et al., 2024; Chen et al., 2024; Levy et al., 2022). Such methods require numerous pre-training and fine-tuning experiments and are less flexible compared to those that leverage in-context learning (Brown et al., 2020) for improving task performance either through few exemplars (Alayrac et al., 2022; Song et al., 2022), or zero-shot techniques such as chain-of-thought (Wei et al., 2023). Our work leverages both zero-shot capabilities, as well as few-shot learning in a way that enables an iterative refinement loop not previously explored for these tasks.

Self-play Environments such as AlphaGo (Silver et al., 2017) and Atari (Mnih et al., 2013) have been widely used for training models using reinforcement learning (Tesauro, 1995). We take inspiration from such work and translate VQA tasks into self-play environments. The VLM first constructs a python runtime, which decomposes the task into multiple sub-tasks and then uses itself for solving those. As improvement signal, correctness of answer is used. We hypothesize that transforming training datasets this way, paired with richness of programs generated, enables VLMs to construct a rich state-space from which compositional reasoning can be improved.

3 Method

Figure 1 presents our approach of constructing a synthetic environment through code generation, iteratively constructing better and better few-shot exemplars. Similar to auto-prompting techniques (Khattab et al., 2023; Wu et al., 2023; Zhang et al., 2022), our method starts by bootstrapping examples using zero-shot capabilities of VLMs. The instructions describe at a high-level how the program solving the visual question should roughly look like, leveraging instruction-following capabilities (Wei et al., 2022), as well as minimal self-refinement capabilities (Madaan et al., 2023). Our method can be described through a *seed stage*, followed by an offline *training stage*. Finally, at test time, the *inference stage* solves a given test exemplar by making use of the best matching train exemplars constructed during the training stage.

Seed stage We provide a zero-shot prompt which contains instructions for generating programs to solve the given tasks, controlling the type of computational environments constructed, which need to be able to perform visual information look-up, arithmetic operations and compositional reasoning on the given task. We enable using two types of programs. One type is a zero-shot visual program-of-thought (Bi et al., 2023), whereas the second one introduces an API through which the model




```
Q: What's the average value of all dark blue bars?
def execute():
    """
    Calculates the average value of the dark blue bars in the image.

    Returns:
        float: The average value of the dark blue bars.
    """
    # Define the data from the image
    dark_blue_bars = {
        "Healthy eating habits": 72,
        "Getting enough physical exercise": 71,
        "Safe and healthy housing conditions": 61,
        "Genetics and hereditary factors": 47,
    }

    # Calculate the average value of the dark blue bars
    average_value = sum(dark_blue_bars.values()) / len(dark_blue_bars)

    return average_value
ans = execute()
```

Figure 3: Visual program-of-thought (Chen et al., 2023a) creates intermediate data structures using extracted values from the image in order to provide an answer that requires arithmetic computations.

can orchestrate the task decomposition and delegate sub-tasks to itself within the program. Given that generated programs may not follow precisely expected output constraints (e.g., numeric values, percentages, etc.) or may perform superficial type conversations that lead to execution failure, we allow the model to refine it three times (Madaan et al., 2023) using the execution error as feedback. The output of this stage are then programs which correctly solve a given training example.

Visual Program-of-Thought: Our first type of environment is a natural extension to program-of-thought (Chen et al., 2023a) (PoT), where the image alongside the question is used when generating programs, with code interleaved with rationales as comments. Values on the images are extracted directly in code, but there is no API that enables a tool call. An illustration of the generated code is in Figure 3 and the corresponding zero-prompt is in the Appendix D.1. PoT naturally facilitates task decomposition, allowing VLMs to effectively manage intermediate computations and maintain structured reasoning paths—crucial for complex visual reasoning tasks such as those found in ChartQA, PlotQA, and ChartFC.

Self-play API: We extend visual program-of-thought with a simple, yet powerful indirection API, depicted in Listing 1. It consists of an *ImageObject* that wraps a provided image and has an *answer* method for answering a question related to the image. The API interface can be implemented in multiple ways, either through calling a large model using a prompt well suited for question-answering or by through calling a specialist fine-tuned model. This API enables the model to focus on the problem decomposition into sub-tasks separately from the individual sub-task solution. The single method class abstracts away details such as which model is called, how it is called and even what hyper-parameters are used.

```
class ImageObject:
  """Holds the image."""
  def __init__(self, image: Image):
    pass
  def answer(self, question: str)->str:
  """Answers questions on held image."""
    pass
```

Listing 1: Our simple, yet powerful, API proposal

Training stage This stage adds few-shot exemplars from the seed stage's outputs to the zero-shot instructions. The stage then runs few-shot inference with the goal of improving the few-shot exemplar pool. The selection of the few-shot exemplars to label is mainly done through random sampling on the training set. The output of the training stage ultimately consists of the best possible few-shot exemplar pool within the budget specified. We treat the number of exemplars $N \leq 1000$ and training steps $T \leq 10$ as hyper-parameters.

Inference stage Lastly, once the training set constructed a few-shot exemplar pool, at inference we can select, at random or using a similarity measure, which $K = 8$ few-shot examples to use to solve a task. When the same N exemplars are used across

all T training steps, we obtain information about how difficult to solve a particular example is across all stages of the training. These exemplars may be useful to include due to the model's inability to solve them.

We refer the reader to Appendix D.1 and Appendix D.2 where we provide examples not only of prompts, but also how the code changes after the training stage.

4 Experimental Setup

We first introduce tasks of interest and models we make use of our in our setup in Section 4.1. Then we report the baseline performance using zero-shot, as well as program-of-thought, followed by our proposed method's performance on validation sets. Finally, we report main results in Section 4.

4.1 Datasets and Models

We evaluate our method on several challenging multimodal benchmarks that require reasoning about scientific diagrams, such as charts. In order to solve a visual question-answering task or fact-check whether a particular statement is supported by the provided chart, models oftentimes have to perform visual information look-up paired with compositional reasoning through mathematical operations (e.g. comparisons, multiplications).

Models Our methods builds on the strength of models that can generate code conditioned on images. We therefore perform ablations and evaluate our approach primarily on Gemini 1.5 Flash (Anil et al., 2023), although smaller scale fine-tuned models on code generation tasks may equally work. Through the generated code, the model can use itself as tool or another model.

Tasks Key to our method is to continue leveraging existing training sets, even if they have been used during large model training phases. To do so, we focus on using as few exemplars as possible. Therefore, we use a pool of up to 1000 exemplars during the seed and training stages and 8 exemplars at inference stage. The larger the pool at initial stages, the more diverse exemplars are available. We make use of ChartQA (Masry et al., 2022) and PlotQA (Methani et al., 2020) for question-answering, while for visual fact-checking we evaluate ChartFC (Akhtar et al., 2023). More details in Appendix E.

DSPy (CoT)	ChartQA	PlotQA	ChartFC
0-shot	80.0	42.2	78.9
8-shot (bootstrap-1)	80.9	32.3	79.6
8-shot (bootstrap-2)	80.4	32.2	78.9
8-shot (bootstrap-3)	81.2	31.5	78.5

Table 1: Naively bootstrapping few-shot exemplars using the DSPy programming model on task training sets arbitrarily affects performance.

Metrics For all the tasks we report *relaxed accuracy* metric (RA). This metric requires string predictions to match ground truths, while for numeric answers a 5% relative difference to ground truth is considered correct. Since our method generates code, we kept track of *code pass rate* (CPR), which quantifies number of time code executed successfully over total code execution attempts. However, we noticed that at all times this is usually more than 95%. Therefore, we do not report it separately.

The DSPy programming model (Khattab et al., 2023) described in Section 2 is a strong baseline for our work. We show, however, that naively using DSPy on training sets leads to performance drops, which our proposed method overcomes. We apply the DSPy programming model for the first time, to our knowledge, on multimodal tasks. We find that the programming model proposed can easily accommodate images, besides text, as input.

4.2 Zero-shot Bootstrapping

We start by defining the corresponding DSPy *program signature*, namely $(image, question) \rightarrow (answer)$ and implement a *predictor* that leverages this signature in a zero-shot manner. The DSPy approach makes use of a *student* and *teacher* predictor in a loop, where initially the teacher is a zero-shot prompt. Once demonstrations have been performed, we *bootstrap* the student by appending the teacher-labeled examples to the zero-shot prompt. The student then becomes the teacher.

To ground the performance of the approach in Gemini 1.5 Flash (Anil et al., 2023) performance, we use from the beginning the task-specific zero-shot prompt with chain-of-thought (CoT) predictor, instead of a naive prompt based on just the signature. We report the results in Table 1.

Bootstrapping through the DSPy programming model on training sets does not necessarily lead to performance improvements. Indeed, as can be observed, PlotQA performance degrades significantly from the zero-shot performance. The drop may be

explained through the fact that the bootstrapping process selects 8-shot exemplars from those that the teacher can label, thus biasing the exemplars selection space compared to random sampling from the training sets. This is an important aspect to optimize, considering that most real-world tasks do not have labels to start with.

4.3 Visual Program-of-Thought

DSPy programming model enables chaining multiple predictors. The chain facilitates using outputs from one predictor as inputs for the next predictor, naturally forming solutions traces. The bootstrapping process can keep the end-to-end traces that reach the correct answer according to the RA metric, constructing few-shot exemplars for each predictor out of a single trace.

We implement a *visual program-of-thought* (VPoT) predictor, from which we take the outputs and pass them through *chain-of-thought* (CoT) predictor that takes into account the code and the code output. We report the results in Table 2.

We observe that the ChartQA numbers degrade less compared to the PlotQA and ChartFC benchmarks. This may likely be attributed to the model capability to ignore the generated code and code outputs and simply focus on the task at hand. It may also explain better why in the previous baseline, without the use of generated code, performance on ChartQA slightly increased. This also suggest that the results on the other two benchmarks may be better indicators of generalizability.

DSPy (VPoT + CoT)	ChartQA	PlotQA	ChartFC
0-shot (CoT)	80.0	42.2	78.9
0-shot	79.3	45.4	81.9
8-shot (bootstrap)	78.1	29.5	78.2
8-shot (bootstrap-2)	78.0	32.1	77.7
8-shot (bootstrap-3)	78.2	32.2	79.5

Table 2: Extending the DSPy bootstrapping process by involving code generation suffers from the same performance degradation.

4.4 Self-play API

Further, we replace the visual program-of-thought predictor with our *Self-play API* (SP) predictor, while also making use of a chain-of-thought predictor. The core elements of the baseline setup described remain the same.

DSPy (SP + CoT)	ChartQA	PlotQA	ChartFC
0-shot (CoT)	80.0	42.2	78.9
0-shot	78.2	46.5	80.4
8-shot (bootstrap-1)	80.0	49.5	80.3
8-shot (bootstrap-2)	**81.3**	51.2	**82.2**
8-shot (bootstrap-3)	81.1	**52.4**	81.9

Table 3: The Self-play API (SP) predictor significantly boost performance on scientific diagram reasoning.

The improvements shown in Table 3 demonstrate the predictor using the Self-play API comes with several benefits: (1) the model can always choose to call itself within the generated program, thus delegating the entire task or a sub-task to itself, (2) execution of the generated code represents a feedback signal on how well the delegation was tied together. Thus, our approach not only better separates orchestration from delegation, but constructs a feedback loop through which enables the model to balance what to delegate and what to orchestrate.

5 On Similarity Measures

We have briefly investigated as to whether the choice of similarity measure substantially improves over random selection, as shown in (An et al., 2023). We have made use of CLIP image similarity, CLIP text similarity (Radford et al., 2021), TF-IDF text similarity, TF-IDF text similarity. We observed that CLIP similarity sometimes reduced the number of images necessary, but inconsistently across datasets. Therefore, we have based on our main results on random selection. We investigate more complex selection measures as an in-depth study in the next section.

6 Scaling Self-play on PlotQA

To further strengthen our contribution, we scale up our analysis beyond the DSPy configuration. We chose to focus this study on PlotQA to limit costs. We ablate the choice of large model, adding GPT-4o, scale the number of bootstrapping iterations up to 10 and we report the effects of labeling up to 1000 exemplars per iteration during the training stage using our method. Due to the large number of examples available, we chose to select the 8-shot at test time using a similarity measure based on the question asked. Lastly, we also report how the performance changes when $Image.answer$ is implemented using a tool, specifically a fine-tuned version of PaliGemma-3B (Beyer et al., 2024). For

strengthening our contribution, we do not make use of an additional predictor based on chain-of-thought, rather directly use the self-play predictor output after code execution.

When scaling up to $T = 10$ steps, we use up to $N = 1000$ examples from the training set during the training stage. During every step, the teacher typically labels up to 60% of the training set, roughly 600 examples. Out of these, during inference stage, we select 8 examples that best match the question at test time.

Self-play API	PlotQA	
	Gemini 1.5 Flash	GPT-4o
0-shot (CoT)	42.2	21.6
0-shot	43.7	20.8
8-shot (bootstrap-5)	50.6	21.7
8-shot (bootstrap-10)	**52.3**	**22.9**

Table 4: Behavior of Self-play API when used for bootstrapping up to 10 iterations, teacher-labeling up to 1000 examples, while selecting best 8 matching examples at inference. Performance on PlotQA validation set.

Our findings in Table 4 confirm that our approach works well across model families. Although the performance of GPT-4o on PlotQA starts lower, our method nonetheless improves the model's ability to better use itself.

PaliGemma An alternative implementation of the Self-play API is for the $Image.answer$ method to call an external tool. We fine-tune PaliGemma (Beyer et al., 2024) on PlotQA. Then we replace the original implementation that calls the model with the zero-shot prompt with a call to the fine-tuned model. First, we contrast the tool performance with the zero-shot performance in Table 5 and note that the specialist model indeed performs better than Gemini 1.5 Flash in zero-shot regime.

Tool	PlotQA (RA %)
Gemini 1.5 (zero-shot)	42.2
PaliGemma (fine-tuned)	52.1

Table 5: Standalone performance of models on PlotQA validation set.

Then, we replace the implementation of the Self-play API. The implementation details are not visible at code generation time, for which the prompt is the same and consists of a simple high-level API description. In spite of this, we observe significant differences in performance on the validation set after multiple refinement iterations of the training set. Our method improves significantly over the baseline in both cases, as depicted in Figure 4. Gemini 1.5 Flash is able to improve regardless of the implementation, but benefits from more predictable and higher quality outputs from specialized tools.

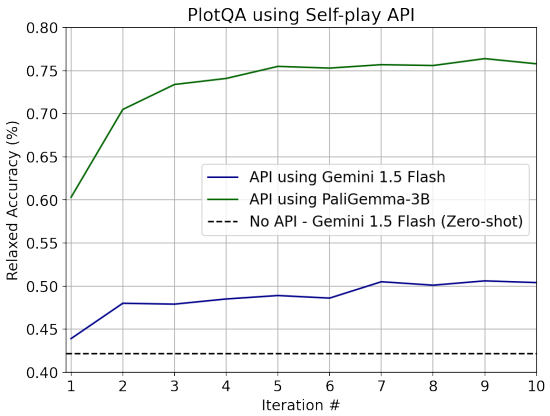


Figure 4: Performance when using different $Image.answer$ specializations, using PaliGemma and Gemini 1.5 Flash.

We attribute the large improvement over the setup in Table 4 to the predictability of the PaliGemma model behind the API call. Specifically, the fine-tuned model outputs directly answers to questions, removing spurious errors such as incorrect conversations, longer answers, thus enabling the refinement over multiple iterations to focus on how to decompose a complex question.

Novelty Score Training stage has 1000 examples which includes questions of varying difficulty. Some questions are simple and are always answered correctly, while others are challenging and only solved occasionally. We argue that examples that are hard to solve contain more novel information. For this, we introduce a novelty score that we assign to an example which has been evaluated repeatedly through the $T \geq 10$ iterations described in previous section. This would be defined as follows:

$$Novelty_{E_i} = \left(1 - \frac{K_{correct}}{T}\right),$$

where $i \in 1, 2, ..., N$ and $K_{correct}$ is the number of times an example was correctly solved during the training stage. Then, at inference time, we combine the similarity measure score with the novelty measure score introduced here in order to choose whether an example should be used as part of the

8-shots selected at test time. The final similarity distance is defined as

$$Score_{E_i,E_{test}} = \alpha * Novelty_{E_i} + (1-\alpha) * Similarity_{E_i,E_{test}}$$

we use $\alpha = 0.15$ for our experiments.

We report the result in Figure 5.

Compute Estimates For the tasks used, a typical input example consists of a (image, text) pairs, typically resulting in about 300 tokens per input (e.g. 258 tokens for the image, rest for the text). The output token counts differs when generating code (50100 tokens) or directly answering as tool (10-20 tokens), from within the code. The key additional cost comes from the number of tool calls, which depends heavily on initial prompting approach. Lastly, the costs for fine-tuning a tool such as PaliGemma (Beyer et al., 2024) need to be accounted for as well.

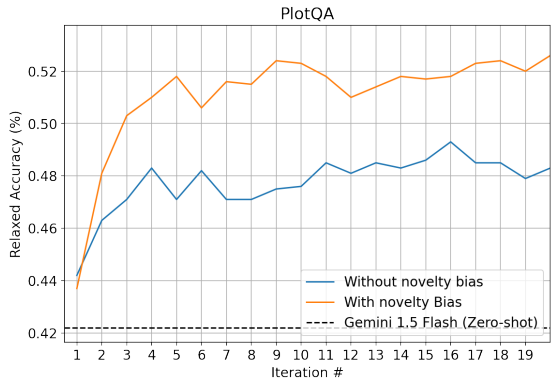


Figure 5: PlotQA performance improves when, at test time, harder examples, besides similar ones, are used as part of the 8-shot examples.

Lastly, we report the results on the sub-sampled test set for PlotQA in Table 6.

Self-play API	PlotQA	
	Gemini	GPT
0-shot [Chain-of-Thought]	41.6	19.3
0-shot [Self-play API]	42.8	19.0
8-shot [Self-play API], Best Step	49.8	19.9

Table 6: Gemini 1.5 Flash / GPT 4o performance with our method on PlotQA test

7 Conclusion

Our work introduces a new recipe through which highly capable models, such as Gemini multimodal, can leverage their joint image understanding and code generation capabilities for bootstrapping improved performance. We validate our approach on a set of difficult chart reasoning benchmarks, but note it's wide applicability. We do this by seeding environments with zero-shot prompts that solve a given task in two ways, through program-of-thought or through a self-play API that enables Gemini to focus on the high-level reasoning challenge, delegating the low-level problem to itself as a tool. Our method iteratively improves performance on visual-question answering training sets, generalizing strongly on validation and test sets after just a few training iterations. Improvements over zero-shot baselines are strong across each environment and task combination.

8 Acknowledgements

Several collaborators have helped shape this work with insightful feedback and discussions. Thanks to Jindong Chen and Abhanshu Sharma for their early support and feedback on the project. We thank Srinivas Sunkara, Duc-Hieu Tran and Florian Hartmann for their regular feedback on the work. Brian Patton provided helpful feedback for improving the presentation of the work. We thank Lluis Castrejon, Thomas Mensink, Andre Araujo, and Vitto Ferrari on their deep discussions surrounding LLMs which orchestrate tools. We thank Ewa Dominowska and Jay Yagnik for supporting this research.

References

Rishabh Agarwal, Avi Singh, Lei M. Zhang, Bernd Bohnet, Stephanie Chan, Ankesh Anand, Zaheer Abbas, Azade Nova, John D. Co-Reyes, Eric Chu, Feryal Behbahani, Aleksandra Faust, and Hugo Larochelle. 2024. Many-shot in-context learning. *Preprint*, arXiv:2404.11018.

Mubashara Akhtar, Oana Cocarascu, and Elena Simperl. 2023. Reading and reasoning over chart images for evidence-based automated fact-checking. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 399–414, Dubrovnik, Croatia. Association for Computational Linguistics.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. 2022.

Flamingo: a visual language model for few-shot learning. *Preprint*, arXiv:2204.14198.

Shengnan An, Bo Zhou, Zeqi Lin, Qiang Fu, Bei Chen, Nanning Zheng, Weizhu Chen, and Jian-Guang Lou. 2023. Skill-based few-shot selection for in-context learning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13472–13492, Singapore. Association for Computational Linguistics.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Gilles Baechler, Srinivas Sunkara, Maria Wang, Fedir Zubach, Hassan Mansoor, Vincent Etter, Victor Cărbune, Jason Lin, Jindong Chen, and Abhanshu Sharma. 2024. Screenai: A vision-language model for ui and infographics understanding. *Preprint*, arXiv:2402.04615.

Lucas Beyer, Andreas Steiner, André Susano Pinto, Alexander Kolesnikov, Xiao Wang, Daniel Salz, Maxim Neumann, Ibrahim Alabdulmohsin, Michael Tschannen, Emanuele Bugliarello, Thomas Unterthiner, Daniel Keysers, Skanda Koppula, Fangyu Liu, Adam Grycner, Alexey Gritsenko, Neil Houlsby, Manoj Kumar, Keran Rong, Julian Eisenschlos, Rishabh Kabra, Matthias Bauer, Matko Bošnjak, Xi Chen, Matthias Minderer, Paul Voigtlaender, Ioana Bica, Ivana Balazevic, Joan Puigcerver, Pinelopi Papalampidi, Olivier Henaff, Xi Xiong, Radu Soricut, Jeremiah Harmsen, and Xiaohua Zhai. 2024. Paligemma: A versatile 3b vlm for transfer. *Preprint*, arXiv:2407.07726.

Zhen Bi, Ningyu Zhang, Yinuo Jiang, Shumin Deng, Guozhou Zheng, and Huajun Chen. 2023. When do program-of-thoughts work for reasoning? *Preprint*, arXiv:2308.15452.

Zalán Borsos, Raphaël Marinier, Damien Vincent, Eugene Kharitonov, Olivier Pietquin, Matt Sharifi, Dominik Roblek, Olivier Teboul, David Grangier, Marco Tagliasacchi, and Neil Zeghidour. 2023. Audiolm: a language modeling approach to audio generation. *Preprint*, arXiv:2209.03143.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. *Preprint*, arXiv:2305.17126.

Victor Carbune, Hassan Mansoor, Fangyu Liu, Rahul Aralikatte, Gilles Baechler, Jindong Chen, and Abhanshu Sharma. 2024. Chart-based reasoning: Transferring capabilities from llms to vlms. *Preprint*, arXiv:2403.12596.

Lluis Castrejon, Thomas Mensink, Howard Zhou, Vittorio Ferrari, Andre Araujo, and Jasper Uijlings. 2024. HAMMR: Hierarchical multimodal react agents for generic vqa. *Preprint*, arXiv:2404.05465.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023a. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Preprint*, arXiv:2211.12588.

Xi Chen, Josip Djolonga, Piotr Padlewski, Basil Mustafa, Soravit Changpinyo, Jialin Wu, Carlos Riquelme Ruiz, Sebastian Goodman, Xiao Wang, Yi Tay, Siamak Shakeri, Mostafa Dehghani, Daniel Salz, Mario Lucic, Michael Tschannen, Arsha Nagrani, Hexiang Hu, Mandar Joshi, Bo Pang, Ceslee Montgomery, Paulina Pietrzyk, Marvin Ritter, AJ Piergiovanni, Matthias Minderer, Filip Pavetic, Austin Waters, Gang Li, Ibrahim Alabdulmohsin, Lucas Beyer, Julien Amelot, Kenton Lee, Andreas Peter Steiner, Yang Li, Daniel Keysers, Anurag Arnab, Yuanzhong Xu, Keran Rong, Alexander Kolesnikov, Mojtaba Seyedhosseini, Anelia Angelova, Xiaohua Zhai, Neil Houlsby, and Radu Soricut. 2023b. Pali-x: On scaling up a multilingual vision and language model. *Preprint*, arXiv:2305.18565.

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. 2024. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. *Preprint*, arXiv:2312.14238.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. *Preprint*, arXiv:2010.11929.

Tsu-Jui Fu, Linjie Li, Zhe Gan, Kevin Lin, William Yang Wang, Lijuan Wang, and Zicheng Liu. 2022. Violet : End-to-end video-language transformers with masked visual-token modeling. *Preprint*, arXiv:2111.12681.

Roy Ganz, Yair Kittenplon, Aviad Aberdam, Elad Ben Avraham, Oren Nuriel, Shai Mazor, and Ron Litman. 2024. Question aware vision transformer for multimodal reasoning. *Preprint*, arXiv:2402.05472.

Difei Gao, Lei Ji, Luowei Zhou, Kevin Qinghong Lin, Joya Chen, Zihan Fan, and Mike Zheng Shou. 2023. AssistGPT: A general multi-modal assistant that can plan, execute, inspect, and learn. *Preprint*, arXiv:2306.08640.

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 (CVPR)*, pages 14953–14962.

Ziniu Hu, Ahmet Iscen, Chen Sun, Kai-Wei Chang, Yizhou Sun, David A Ross, Cordelia Schmid, and Alireza Fathi. 2023. Avis: Autonomous visual information seeking with large language model agent. *Preprint*, arXiv:2306.08129.

Yixing Jiang, Jeremy Irvin, Ji Hun Wang, Muhammad Ahmed Chaudhry, Jonathan H. Chen, and Andrew Y. Ng. 2024. Many-shot in-context learning in multimodal foundation models. *Preprint*, arXiv:2405.09798.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *Preprint*, arXiv:2310.03714.

Abdullatif Köksal, Timo Schick, and Hinrich Schuetze. 2023. MEAL: Stable and active learning for few-shot prompting. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 506–517, Singapore. Association for Computational Linguistics.

Matan Levy, Rami Ben-Ari, and Dani Lischinski. 2022. Classification-regression for chart comprehension. *Preprint*, arXiv:2111.14792.

Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *Preprint*, arXiv:2309.16298.

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. *Preprint*, arXiv:2303.17651.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. *Preprint*, arXiv:2203.10244.

Brandon McKinzie, Zhe Gan, Jean-Philippe Fauconnier, Sam Dodge, Bowen Zhang, Philipp Dufter, Dhruti Shah, Xianzhi Du, Futang Peng, Floris Weers, Anton Belyi, Haotian Zhang, Karanjeet Singh, Doug Kang, Ankur Jain, Hongyu Hè, Max Schwarzer, Tom Gunter, Xiang Kong, Aonan Zhang, Jianyu Wang, Chong Wang, Nan Du, Tao Lei, Sam Wiseman, Guoli Yin, Mark Lee, Zirui Wang, Ruoming Pang, Peter Grasch, Alexander Toshev, and Yinfei Yang. 2024. MM1: Methods, analysis & insights from multimodal llm pre-training. *Preprint*, arXiv:2403.09611.

Nitesh Methani, Pritha Ganguly, Mitesh M. Khapra, and Pratyush Kumar. 2020. Plotqa: Reasoning over scientific plots. In *The IEEE Winter Conference on Applications of Computer Vision (WACV)*.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *Preprint*, arXiv:1312.5602.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *Preprint*, arXiv:2303.09014.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *Preprint*, arXiv:2305.15334.

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. *Preprint*, arXiv:2103.00020.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. 2024. The prompt report: A systematic survey of prompting techniques. *Preprint*, arXiv:2406.06608.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *Preprint*, arXiv:1712.01815.

Haoyu Song, Li Dong, Weinan Zhang, Ting Liu, and Furu Wei. 2022. CLIP models are few-shot learners: Empirical studies on VQA and visual entailment. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6088–6100, Dublin, Ireland. Association for Computational Linguistics.

Aleksandar Stanić, Sergi Caelles, and Michael Tschannen. 2024. Towards truly zero-shot compositional visual reasoning with llms as programmers. *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. *Preprint*, arXiv:2303.08128.

Gerald Tesauro. 1995. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022. Finetuned language models are zero-shot learners. *Preprint*, arXiv:2109.01652.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *Preprint*, arXiv:2308.08155.

Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. 2023. Mm-react: Prompting chatgpt for multimodal reasoning and action. *Preprint*, arXiv:2303.11381.

Ziyan Yang, Kushal Kafle, Franck Dernoncourt, and Vicente Ordonez. 2024. Improving visual grounding by encouraging consistent gradient-based explanations. *Preprint*, arXiv:2206.15462.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *Preprint*, arXiv:2210.03493.

A Limitations

We report several limitations of our work: we note that the evaluation focused on a limited number of environment types (two) and tools, used with a strong emphasis on a powerful indirection API, but potentially overlooking scalability challenges. Our results on test sets are limited to the open ones, while work on the closed test sets of DocVQA and InfographicVQA will be done at a later stage. Although the effects of few-shot learning were effectively leveraged, a deeper investigation into many-shot learning was not undertaken before iterating on training steps to refine the few-shots. Lastly, we think our work extends to natural images; however, our focus has been on UI interfaces, charts, and infographics.

B Societal impact

Our work proposes a technique through which a training set can be exploited in creating an environment through which alternative solutions can semi-autonomously, through self-play, be constructed by large models to solve the task. Training sets are already heavily used in the pre-training and fine-tuning stages of large model training. Our work reuses these datasets. Due to inherent limitations of the training sets, it is likely that broader societal impact is rather limited compared to what models already achieve today. The particular skill acquired does play a role, in our case for refining generated code as an alternative problem solving path, should have a positive impact where models can bootstrap alternative approaches the downstream user has access to. However, refining and improving generated code does require safety handles, such as sandboxed environments. Our technique may also have an impact on the type of datasets collected, as our method enables constructing datasets synthetically by introducing external learning signals from the computational environment. It may be able to reduce the collection of unnecessary datasets, being able to extract more from the currently available ones already collected.

C Qualitative Improvements

Qualitatively looking at examples helps better understand the type of improvements noticeable after two iterations of training stage. For example, consider the task of identifying the third largest contributor on the following image from the ChartQA dataset in Figure 6.

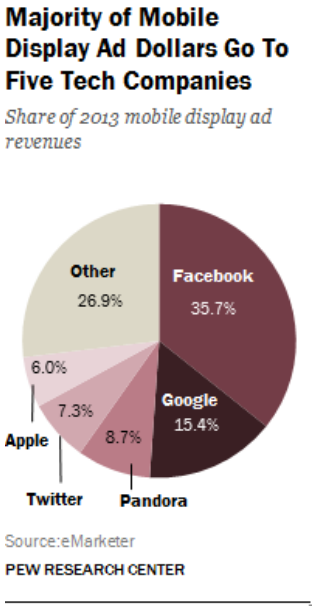


Figure 6: Image from the ChartQA dataset

Our zero-shot prompt for visual program-of-thoughts, generates the following solution.

```
Q: What is the third largest contributor?
======= Zero-shot (Incorrect) ========
def execute():
  """
  This function returns the third largest contributor in the
      given pie chart.

  Args:
    None

  Returns:
    A string representing the name of the third largest
      contributor.
  """
  # Get the data from the pie chart.
  data = [35.7, 15.4, 7.3, 8.7, 26.9]

  # Sort the data in descending order.
  data.sort(reverse=True)

  # Get the third largest value.
  third_largest = data[2]

  # Find the corresponding company name.
  company_names = ['Facebook', 'Google', 'Apple', 'Twitter',
      'Other']
  third_largest_company = company_names[data.index(
      third_largest)]

  # Return the company name.
  return third_largest_company
ans = execute()
```

After two training steps, zero-shot examples are replaced with few-shot examples. The code is then modified as follows.

```
Q: What is the third largest contributor?
======== Few-shot Pool (Correct) =======
def execute():
  """
  This function returns the third largest contributor in the
      graph.

  The answer is Twitter because it has a 7.3% share of the
      2013 mobile display ad revenues.

  Args:
    None

  Returns:
    The third largest contributor in the graph as a string.
  """

  # Get the data from the image.
  data = [
    {
      "company": "Facebook",
      "revenue_share": 35.7
    },
    {
      "company": "Google",
      "revenue_share": 15.4
    },
    {
      "company": "Other",
      "revenue_share": 26.9
    },
    {
      "company": "Apple",
      "revenue_share": 6.0
    },
    {
      "company": "Twitter",
      "revenue_share": 7.3
    },
    {
      "company": "Pandora",
      "revenue_share": 8.7
    }
  ]

  # Sort the data by revenue share.
  sorted_data = sorted(data, key=lambda x: x["revenue_share"
      ], reverse=True)

  # Get the third largest contributor.
  third_largest_contributor = sorted_data[2]["company"]

  # Return the third largest contributor.
  return third_largest_contributor
ans = execute()
```

Sometimes the model solves the question even in the doc-string function. We think this may be an artefacts of the training data. However, our iterative method continues to improve the code logic and correct answer is returned. Another example can be found below.

```
Q: What is the value of the largest pie section?
========= Zero-shot(Incorrect) ===========
def execute():
  """
  Computes the value of the largest pie section.

  The largest pie section is the one that represents the
      largest percentage
  of the pie. In this case, the largest pie section is the
      one that
  represents the percentage of people who support allowing
      companies
  from other countries to invest in Pemex. This percentage
      is 34%.

  Args:
    None

  Returns:
    The value of the largest pie section as a float.
  """

  # Get the value of the largest pie section.
  largest_pie_section = 34

  # Return the value of the largest pie section.
  return largest_pie_section
ans = execute()
```

After two steps during the training stage, the code improves significantly.

```
Q: What is the value of the largest pie section?
======== Few-shot Pool (Correct) =======
def execute():
  """Calculates the value of the largest pie section.

  The largest pie section represents the percentage of
      people who oppose allowing
  companies from other countries to invest in Pemex. The
      value of the largest pie
  section is 57, which means that 57% of people oppose
      allowing companies from
  other countries to invest in Pemex.

  Args:
    None

  Returns:
    The value of the largest pie section.
  """

  # Get the data from the image.
  data = [
      (2007, 41, 44),
      (2009, 43, 44),
      (2011, 57, 49),
      (2013, 51, 37),
      (2015, 23, 19),
      (2017, 41, 29)
  ]

  # Get the U.S. favourability in Russia for each year.
  us_favorability_in_russia = [y[1] for y in data]

  # Get the highest value in U.S favourability in Russia.
  highest_value = max(us_favorability_in_russia)

  return highest_value
ans = execute()
```

D Zero-shot and Self-refinement Prompts

In this section we provide the zero-shot prompts used for generating the initial examples. Once a few examples are generated on training sets, these zero-shot prompts are replaced by few-shot examples that have successfully matched labels on the training set.

D.1 Visual Program-of-Thought

The first type of prompt, described in Section 3, is visual program-of-thought.

Prompt: Visual Program-of-Thought

Look at the image and question pair below. The main objective is to write a function ‘execute()‘ to answer the question from the image. In the Python documentation of the function, provide step by step reasoning to explain how the following question can be answered. Afterward write the code that will answer the given question. Return the final answer from the function. All the required information is given in the image. Do not load any external files or request for additional input. Pay attention to the units of the answer and when providing percentage as an answer convert the number to decimal format. Write professional level code that an experienced software developer would write. Prefer to write explicit code instead of implicit calculations (e.g. use Python standard libraries to compute max, mean, median values, etc.). Do not print anything with Python print function. Generate Python function only. No english text.

D.2 Self-play API

Here we detail the type of prompts where the model can call itself, but it wouldn't know it does so. The results correspond to Section 3. These do not include any implementation detail, e.g. how to call any of the models or what prompts are used when calling them. Instead a generic interface description within a prompt is provided.

Prompt: Self-play API

Look at the image and question pair below. The main objective is to write a function ‘execute()‘ to answer the question from the image.
In the Python documentation of the function, provide step by step reasoning to explain how the following question can be answered.
Afterward write the code that will answer the given question.
Return the final answer from the function. All the required information is given in the image.
Do not load any external files or request for additional input.
Pay attention to the units of the answer and when providing percentage as an answer convert the number to decimal format.
Write professional level code that an experienced software developer would write.
Prefer to write explicit code instead of implicit calculations (e.g. use Python standard libraries to compute max, mean, median values, etc.).
Do not print anything with Python print function. Generate Python function only. No english text.

You are given an interface and some examples of how to use the interface to answer the question. Your task to answer a newly given question with the interface.

These are interface descriptions of python classes you can use. Actual implementations are provided at runtime.

Prompt: Self-play API Example

Here are some examples of what the implementation of it may return:
ImageObject(image).answer('What is the value of ...?') may return a number
ImageObject(image).answer('Is ...?') may return a Yes / No
ImageObject(image).answer('What are the steps?') may return a comma-sep string

For the execute function make use of the ImageObject class. Only the answer() method.

All queries should have an answer, so no need to consider corner cases.

For usual cases, follow the guidelines below:
- For simple visual queries, directly output the answer in the code.
- For queries that require counting and spatial relations, use python code.

Consider the following guidelines:
- Use base Python (comparison, sorting) for basic logical operations, left/right/up/down, math, etc.
- Do not import additional modules and do not use types for variables.
- Use only the ImageObject when multiple questions are needed to answer the given question.
- When calling answer on ImageObject use as complete and specific questions as possible.

The code you output can look similar to this function below
# Question: ...
def execute(image):
# Explanation for why a first step like the one below is needed
im = ImageObject(image)
value = im.answer(question)
# Explanation for why the next value is needed
other_value = im.answer(other_question)
# Explanation on how to combine the values in a meaningful way for answering the original question.
ans = value + other_value
return ans

D.3 Self-refinement prompt

The self-refinement strategy is rather straightforward and is captured through the prompt below.

Prompt: Self-play API Refinement

// Missing: answer variable
This code is missing the final answer variable. The final answer should be assigned to the answer variable ({answer_var}). Correct the missing variable mistake and try again.
// NameError: usually import statement missing.
This code has raised NamedError: {error_trace}. There might be missing import statements. Correct the NameError mistake and try again.
// Generic: for everything else.
The code above is a valid Python code, however it raised {error_type}: {error_trace}
Correct the mistake and try again please.

E Dataset size summary

Size	ChartQA	ChartFC	PlotQA
Training	1000	1000	1000
Validation	960	1000	1000
Test	2500	1000	1000

Table 7: To reduce costs, we sampled down datasets: all datasets containing 1000 samples were randomly sampled and kept consistent across all runs. We used full sized validation and test sets for ChartQA.