論文 / 著書情報
Article / Book Information

Title	ContextualCoder: Adaptive In-context Prompting for Programmatic Visual Question Answering
Authors	Ruoyue Shen, Nakamasa Inoue, Dayan Guan, Rizhao Cai, Alex.C Kot, Koichi Shinoda
Citation	IEEE Transactions on Multimedia, , , pp. 1-14
Pub. date	2025, 2
DOI	https://doi.org/10.1109/TMM.2025.3543043
Creative Commons	Information is in the article.

ContextualCoder: Adaptive In-context Prompting for Programmatic Visual Question Answering

Ruoyue Shen, *Student Member, IEEE,* Nakamasa Inoue, *Member, IEEE,* Dayan Guan, Rizhao Cai, Alex C. Kot, *Fellow, IEEE,* Koichi Shinoda, *Senior Member, IEEE*

***Abstract*—Visual Question Answering (VQA) presents a challenging task at the intersection of computer vision and natural language processing, aiming to bridge the semantic gap between visual perception and linguistic comprehension. Traditional VQA approaches do not distinguish between data processing and reasoning, limiting their interpretability and generalizability in complex and diverse scenarios. Conversely, Programmatic Visual Question Answering (PVQA) models leverage large language models (LLMs) to generate executable codes, providing answers with detailed and interpretable reasoning processes. However, existing PVQA models typically rely on simplistic input-output prompting, which struggles to elicit domain-specific knowledge from LLMs and often produces unclear or extraneous outputs. Furthermore, PVQA models typically rely on a basic in-context example (ICE) selection methodology that is heavily influenced by individual word similarity rather than the overall sentence context. This leads to suboptimal ICE selection and a reliance on dataset-specific ICE candidates. In this paper, we propose ContextualCoder, a novel prompting framework tailored for PVQA models. ContextualCoder leverages frozen LLMs for code generation and pre-trained visual models for code execution, eliminating the need for extensive training and enhancing model flexibility. By incorporating an innovative prompting methodology and a novel ICE selection strategy, ContextualCoder facilitates the use of diverse in-context information for code generation, thereby improving the performance of PVQA models. Our approach surpasses state-of-the-art models, as evidenced by comprehensive experiments across diverse VQA datasets, including multilingual scenarios.**

***Index Terms*—Visual question answering, prompting methods, large language models, code generation.**

I. Introduction

VISUAL Question Answering (VQA) has emerged as a challenging yet promising field at the intersection of computer vision and natural language processing (NLP). It is dedicated to crafting systems capable of understanding and answering questions pertaining to visual content, thereby bridging the semantic gap between visual perception and language comprehension. VQA methodologies have evolved along diverse paths. Feature-based methods [1]–[4] extract image and text features independently and then integrate them. Attention mechanisms [5]–[10] have been incorporated to selectively focus on relevant information. More recently, large-scale pre-trained language models [11]–[16] have been employed to improve contextual understanding and enhance overall performance. Despite the advancements, traditional VQA approaches [1]–[16] lack explicit decision-making processes (Figure 1a), making it impossible to audit reasoning steps and diagnose failures, which in turn limits their interpretability in complex or diverse scenarios. Additionally, these models heavily leaning on extensive annotated datasets and require substantial training or fine-tuning. As they become increasingly data and compute-hungry, these approach become less sustainable. Therefore, there is a pressing need to develop models capable of interpretable reasoning without extensive training requirements.

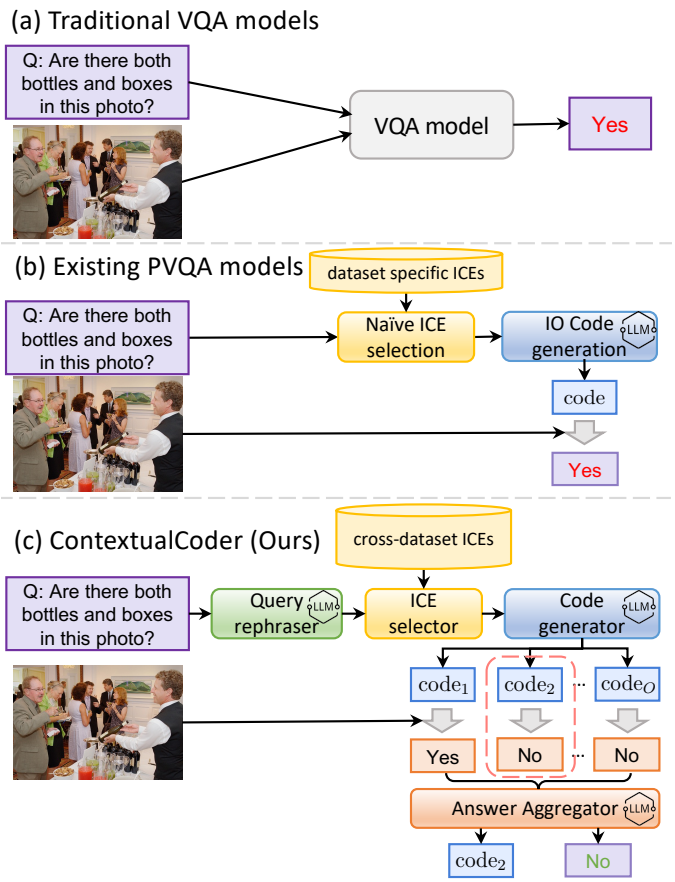

Fig. 1. Comparison of traditional VQA models, existing PVQA models, and our proposed ContextualCoder. Traditional VQA models predict answers directly, lacking detailed reasoning and interpretability. Existing PVQA models generate executable codes with LLMs but are limited by naive dataset-specific ICE selection and basic IO prompting. Our ContextualCoder enhances existing PVQA models with advanced prompting and the novel ICE selector, using multiple modules to provide richer in-context information and maximize code solution diversity.

Programmatic Visual Question Answering (PVQA) models [17]–[20] signify a notable departure from conventional approaches by integrating explicit code generation within the

model architecture, illustrated in Figure 1b. These models utilize large language models (LLMs) to generate input-correlated codes, which invoke off-the-shelf image processing modules as APIs. These approaches embed more interpretable reasoning within the code lines and eliminate the necessity for training. Employing in-context learning [21], PVQA models guide the output of LLMs by incorporating multiple in-context examples (ICEs) into the prompt as a demonstration. Each ICE consists of a question paired with the corresponding code to inform and direct LLM-based code generation. However, existing PVQA models suffer from several limitations: (i) they typically rely on simplistic input-output prompting, which hinders their ability to elicit domain-specific knowledge from LLMs, often leading to the generation of unclear or extraneous outputs; (ii) they employ an elementary ICE selection methodology based on sentence embedding similarity, which is heavily influenced by individual word similarity rather than the content of entire sentences. This approach leads to suboptimal ICE selection and results in a dependence on dataset-specific ICE candidates.

Recent studies have demonstrated the significant impact of advanced prompting strategies on enhancing the efficacy of LLMs across various NLP tasks, exemplified by techniques like Chain of Thought (CoT) [22] and Tree of Thoughts (ToT) [23]. Furthermore, investigations conducted by several researches [24]–[29] have elucidated the pivotal role of ICE selection strategies in influencing the performance of LLMs.

Inspired by these approaches, we introduce an innovative approach aimed at addressing the limitations identified in prior PVQA models by combining a novel code generation prompting strategy with an innovative ICE selection methodology. As shown in Figure 1c, the proposed ContextualCoder consists of four modules: the query rephraser, ICE selector, code generator, and answer aggregator. Each module's output serves as the contextual input to the subsequent module, enhancing its ability to produce more appropriate output. The query rephraser yields a series of expressions by rephrasing the input question. Subsequently, the ICE selector employs a selection algorithm to construct a set of demonstrations. Following this, the code generator utilizes rephrased queries and demonstrations to generate multiple codes representing diverse solutions. Finally, the answer aggregator selects the optimal answer and corresponding code based on the pre-execution of the generated codes.

This paper presents an extended version of our previous publication, PyramidCoder [30], published in ICIP 2024. The key enhancements and differences from the conference version are as follows: 1) the introduction of a novel ICE selection strategy aimed at identifying more appropriate ICEs to enrich contextual information; 2) in contrast to earlier PVQA models, including PyramidCoder, which rely on dataset-specific ICEs tailored to individual datasets, ContextualCoder leverages cross-dataset ICEs—a more generalized set of examples generated by integrating ICEs from multiple datasets. This approach enhances the model's generalization capabilities while reducing dataset dependency; and 3) a comparative analysis and discussion of model performance in multilingual scenarios. In summary, our contributions are threefold:

1) We introduce ContextualCoder, an innovative code generation prompting framework for visual question answering. It leverages frozen off-the-shelf models, eliminating the need for additional training.
2) We advance PVQA code generation through ContextualCoder by integrating both prompting methodologies and ICE selection strategies. This approach maximizes the diversity of code generation and ensures adaptability and independence from the pre-trained models employed.
3) We substantiate the superiority of our method over state-of-the-art models through rigorous experimentation across GQA, VQAv2, NLVR2 datasets with cross-dataset ICE annotations. Additionally, we pioneer the evaluation of PVQA model performance on the multilingual MaXM dataset, further solidifying the versatility and robustness of our approach.

II. Related Work

A. Visual Question Answering

Visual Question Answering (VQA) [31], [32] is a multimodal task where a system is presented with an image along with a natural language question regarding the image, with the objective of producing the corresponding answer. This task necessitates the ability to interpret the visual contents encapsulated within the image and comprehend the semantic information inherent in the question, thereby facilitating the generation of an accurate response.

Early VQA models rely on convolutional neural networks to extract image features and sequence models to process textual inputs [1]–[4]. Subsequently, architectural designs have incorporated attention mechanisms [5]–[10] to selectively focus on relevant image regions and words. Furthermore, transformer-based models with multimodal pretraining [11]–[16] have been adopted to enhance contextual understanding and leverage large-scale data resources effectively. Despite these advancements, these end-to-end models lack explicit decision-making processes, making it difficult to audit reasoning steps and diagnose failures, which limits interpretability in complex scenarios. These models also rely heavily on extensive annotated datasets and require substantial training, making them increasingly unsustainable as data and compute demands grow.

To enhance interpretability and eliminate the need for additional training, Programmatic Visual Question Answering (PVQA) methods [17]–[20] have emerged. These models leverage large language models to generate Python-like codes capable of solving the input question. These codes invoke modules or functions as APIs to capture logical steps and are subsequently executed to derive the answer. This integration of explicit program generation within the model architecture facilitates more structured and interpretable reasoning over both visual and textual inputs. Furthermore, PVQA models exhibit enhanced generalization capabilities, as they do not require additional training or extensive data, making them easily adaptable to new senarios.

B. Large Language Models

Large Language Models (LLMs) represent advanced artificial intelligence systems capable of comprehending, generating,

and interacting with human language at a sophisticated level. Utilizing transformer architectures [33], these models undergo extensive training on vast datasets, enabling them to tackle a wide range of natural language processing tasks.

While general-purpose LLMs such as GPT [21], PaLM [34], and Llama [35] have showcased proficiency in general text-related tasks, specialized LLMs tailored to specific domains have also emerged as powerful tools. LLMs dedicated to code generation, such as Codex [36], StarCoder [37], and CodeLlama [38], undergo training on vast collections of programming-related text, enabling them to automate coding tasks based on natural language instructions. Additionally, a class of LLMs that integrates vision and language inputs, exemplified by LLaVA [39], GPT-4V [40] and Gemini [41], presents a multimodal approach to comprehension and content generation, further advancing the capabilities of multimodal understanding and generation.

C. Prompting and In-context Learning

A prompt serves as guidance for the model, essentially providing the text input or instruction for the LLMs to generate the desired output. The formulation of a prompt is crucial in guiding the behavior of LLMs towards producing the intended outputs. With the emergence of in-context learning [21], prompts typically encompass the task definition, a series of in-context examples serving as a demonstration, and the naive input text.

Prompting methodologies, characterized by their resource efficiency and absence of gradient-based training, offer mechanisms to organize prompts for structured and focused generation [42]. Notably, the Chain of Thought (CoT) [22] approach epitomizes a prompting strategy wherein intermediate reasoning steps are generated prior to producing the final output. AutoCoT [43] automates the creation of demonstrations through zero-shot CoT and question clustering. Similarly, Complex-CoT [44] leverages reasoning complexity to curate efficient in-context examples, while ReAct [45] alternates between generating thoughts and actions to facilitate dynamic reasoning and adaptive action planning within the system. Noteworthy advancements also include the Tree of Thoughts (ToT) [23] framework, which enhances LLMs' decision-making processes by fostering exploration and self-evaluation of diverse thoughts and reasoning pathways within a structured tree framework.

The performance of LLMs has been shown to be influenced by the selection of in-context examples. Including different examples in the same prompt template results in variations in the quality of generated results. Consequently, prior approaches have been devised to enhance the quality of demonstrations, thereby refining the input prompt to effectively guide the generation process. Unsupervised methods, utilizing the closest neighbors based on L2 or cosine-similarity distance on sentence embeddings, are commonly employed for selecting in-context examples [24]–[26]. Additionally, factors such as diversity, perplexity, and the impact of examples are considered important indicators for selection [27]–[29]. Other approaches have been proposed to sort selected examples based on their distance to the input [24] or by utilizing entropy metrics [46].

Prompting strategies play a pivotal role in providing context and guidance for in-context examples, while the inclusion of in-context examples serves as demonstrations that further tailor the model's responses to meet task-specific requirements. In this paper, we propose a novel approach aimed at synergistically enhancing both prompting and in-context example selection, enabling LLMs to achieve higher levels of performance, adaptability, and controllability in visual question answering.

D. Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is a framework that integrates retrieval methods with language generation models to improve the generation of accurate and contextually relevant responses. It retrieves pertinent documents from an external corpus based on the input query, allowing the generative model to condition its responses on both the query and the retrieved information, thus improving the informativeness and precision of the generated content.

In RAG, the retrieval process leverages either sparse lexical retrieval methods, such as TF-IDF [47] or BM25 [48], or dense retrieval techniques that rely on the similarity of sentence embeddings between queries and candidate documents. SBERT [49] leverages a siamese and triplet network architecture to produce semantically meaningful sentence embeddings. SimCSE [50] combines unsupervised contrastive learning, where the same input sentence is contrasted with itself using dropout noise, with supervised contrastive learning that leverages "entailment" and "contradiction" pairs from natural language inference data. RankCSE [51] further refines the generation of semantically discriminative sentence embeddings by integrating ranking consistency and ranking distillation within a unified contrastive learning framework. BGE embedding [52] is a general-purpose embedding model trained on large-scale paired data using contrastive learning and multi-task learning.

While these methods generally perform well, they may not be able to finely distinguish the relevance of documents when processing semantically complex or fine-grained queries. To address this, a re-ranking stage, employing deep learning models, is introduced to more accurately assess the relevance between queries and retrieved documents. These re-ranking models take the query and candidate documents as input and output similarity scores [49], [53], [54], offering an efficient and lightweight solution. Furthermore, prompting-based methods for guiding LLMs in reranking [52], [55], [56] have shown strong performance but at a higher computational cost. The final ranked results are then combined with the input query and fed into the generative model to produce coherent, factually informed responses.

In the context of in-context learning, the quality and relevance of selected ICEs influence the model's output. Sentence embedding similarity and re-ranking techniques can enhance the ICE selection process by identifying semantically rich ICEs that align more closely with the query, thereby improving the model's capacity to generate accurate and coherent outputs. A comparative analysis of applying various sentence embedding methods, re-ranking strategies, and our proposed ICE selection approach is presented in Section V-H.

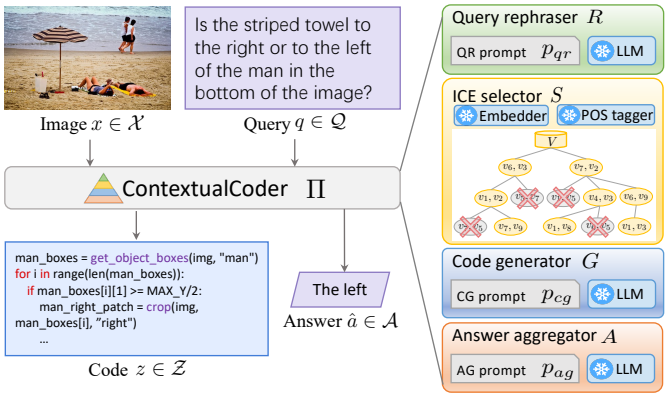

Fig. 2. **Structure of ContextualCoder.** The ContextualCoder Π receives an image x and a query q as inputs. It then produces a code z and predicts an answer $\hat{a}$ for the given image-query pair. The ContextualCoder consists of four modules: a query rephraser R, an ICE selector S, a code generator G, and an answer aggregator A. Each module's output serves as contextual information for subsequent modules.

Algorithm 1 ContextualCoder $\Pi(q, x)$

Input: Query $q \in \mathcal{Q}$, Image $x \in \mathcal{X}$, LLM π, ICE set $V \subset \mathcal{T}$
Output: Code $z \in \mathcal{Z}$, Answer $\hat{a} \in \mathcal{A}$

```
Z ← ∅
[r_1, r_2, ··· , r_M] ← R(q) # Query rephraser
for i = 1, ··· , M do
    [s_i1, s_i2, ··· , s_iN] ← S(r_i, V) # ICE selector
    for j = 1, ··· , N do
        [z_ij1, z_ij2, ··· , z_ijO] ← G(r_i, s_ij) # Code generator
        for k = 1, ··· , O do
            a_ijk ← Λ(x, z_ijk) # Python executor
            Z ← Z ∪ {(z_ijk, a_ijk)}
        end for
    end for
end for
σ ← A_ans(Z) # Answer aggregator
τ ← A_code(Z_σ) # Answer aggregator
return Code z = z_τ, Answer â = a_τ
```

III. Methodology

This section introduces ContextualCoder, a Programmatic VQA model characterized by a hierarchical prompting strategy and modular architecture. As illustrated in Figure 2, ContextualCoder operates by receiving an image and a query as input, generating the Python code that represents logical thought process. This code invokes image processing modules via predefined APIs to derive the answer. The model's architecture comprises four modules: the query rephraser R, ICE selector S, code generator G, and answer aggregator A. Modules R, G, and A utilize a single frozen LLM alongside module-specific prompt templates to produce required textual outputs. Additionally, the ICE selector S incorporates a search algorithm leveraging the capabilities of a sentence embedder and part-of-speech tagger to meticulously select appropriate demonstrations. Notably, all modules operate without the need for additional training, instead capitalizing on off-the-shelf pre-trained models and leveraging outputs from preceding modules as contextual inputs to enhance the quality of code generation. Subsequent sections delineate the overall framework in Section III-A, followed by comprehensive descriptions of individual modules from Section III-B to Section III-E.

A. Overall Framework

We follow the notation established in prior research [18], [19], [57]. Let $x \in \mathcal{X}$ represent an input image and $q \in \mathcal{Q}$ represent a textual query, where $\mathcal{X}$ denotes a set of images, and $\mathcal{Q}$ denotes a set of textual queries. The objective is to produce a code $z \in \mathcal{Z}$ that yields the answer $a \in \mathcal{A}$ corresponding to the given query and image, where $\mathcal{Z}$ denotes a set of executable codes and $\mathcal{A}$ denotes a set of answers. This study assumes that $\mathcal{Z}$ is a set of Python codes. Note that the three sets $\mathcal{Q}$, $\mathcal{Z}$, and $\mathcal{A}$ are subsets of the whole set of texts, denoted as $\mathcal{T}$. To establish mappings among these subsets, a frozen LLM $\pi : \mathcal{T} \to \mathcal{T}$ is employed. This LLM is pre-trained on a substantial text corpus comprising both natural language and programming code.

The ContextualCoder framework consists of two stages: the code generation stage and the code execution stage. During the code generation stage, a code z is synthesized corresponding to the input query and image as

$$z = \Pi(q, x), \tag{1}$$

where $\Pi : \mathcal{Q} \times \mathcal{X} \to \mathcal{Z}$ is the code generation process. In the code execution stage, the generated code is executed with the input image to obtain an answer a as

$$a = \Lambda(x, z), \tag{2}$$

where $\Lambda : \mathcal{X} \times \mathcal{Z} \to \mathcal{A}$ serves as the Python execution engine responsible for processing the image and the generated code to yield the predicted answer. This study centers on investigating the design of the code generation process Π, aiming to enhance the quality of code generation and VQA performance within the ContextualCoder framework.

The methodology delineated above for code generation, denoted as Π, is summarized in Algorithm 1 and illustrated in Figure 3, characterized by four steps. Initially, within the context of an input query $q \in \mathcal{Q}$, the query rephraser R is invoked, yielding a collection of rephrased queries $\{r_i\}_{i=1}^{M}$. Each r_i, except r_0, is a rephrased version of q, tailored to facilitate more effective coding by affording a more precise and nuanced comprehension of the original query. Subsequently, each rephrased query r_i serves as the contextual input for the ICE selector S, tasked with identifying appropriate in-context examples to construct diverse and suitable demonstrations $\{s_{ij}\}_{j=1}^{N}$. The third module, code generator G, is applied to get multiple candidate codes $\{z_{ijk}\}_{k=1}^{O}$ using the prompt that encapsulates prior contextual information s_{ij} and r_i. In this stage, the generated candidate codes undergo pre-execution to derive corresponding candidate answers $a_{ijk} = \Lambda(x, z_{ijk})$. Finally, the answer aggregator A is applied to the ensemble of candidate code-answer pairs to select the definitive answer a_τ and its corresponding optimal code z_τ, where τ denotes the index of the chosen pair. As shown in Figure 2, the ContextualCoder implements all modules utilizing frozen pre-trained models, thereby eliminating the need for additional training. The output of preceding modules serves as the

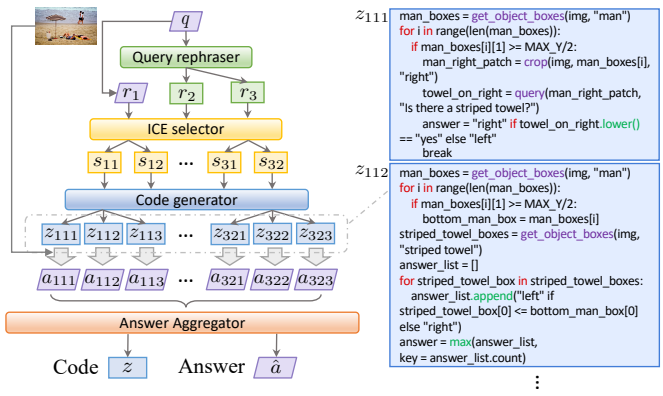

Fig. 3. **Framework overview.** For each input query, the ContextualCoder generates variable query expressions r_i and selects various demonstrations s_{ij} to provide contextual information for generating multiple candidate codes z_{ijk}. Finally, by aggregating these codes z_{ijk} with their corresponding pre-executed results a_{ijk}, the final answer $\hat{a}$ and an interpretable code representation z are obtained.

contextual input for subsequent modules, offering diverse and enriched contextual information for code generation.

B. Query Rephreaser Module

The query rephraser module within ContextualCoder is designed to augment both the interpretability and robustness of the model's comprehension regarding input queries. Given the diverse linguistic forms that may be employed for questions in the VQA task, a precise understanding of varied query expressions becomes imperative. It is widely acknowledged that the quality of responses generated by LLMs is influenced by the input prompt, including input queries. The query rephraser module addresses this challenge by dynamically reformulating input queries into diverse expressions while preserving the underlying semantics. Consequently, the subsequent stage of code generation benefits from a comprehensive and adaptable understanding of user queries, thereby mitigating potential misinterpretations and enhancing overall task performance.

The query rephraser is formally defined as

$$R(q) \triangleq \pi(p_{\text{qr}} + s_{\text{qr}} + q), \tag{3}$$

where q is an input query, p_{qr} is the query rephraser prompt as shown in Figure 4a, and s_{qr} denotes several in-context examples utilized for rephrasing. The operation $+$ indicates the concatenation of texts.

C. ICE Selector Module

Although in-context learning [21]–[23] has proven effective in enhancing the performance of LLMs, the in-context example (ICE) selection methods used in PVQA models are often simplistic, typically relying solely on sentence embedding similarity. Furthermore, the ICEs of PVQA models are often customized for specific datasets to achieve optimal performance, indicating that current PVQA models are sensitive to the source of ICEs and are underutilizing the available data. To address these issues, the proposed ICE selector module, denoted as $S(r, V)$, employs a search algorithm to select multiple diverse

(a) Query Rephraser Prompt p_{qr}

```
Rephrase the input question so that it has different
expressions. You can build different versions by changing
the word order of sentences, using clauses, etc. The
question is asking about the content of an image, make
sure the meaning of the question is not changed. Maximum
{n_queries} sentences.
```

(b) Code Generator Prompt p_{cg}

```
You will be provided with a question asking about an
image. Write up to {m_codes} versions of Python code to
answer this question.
- Only use the constants and APIs delimited by triple
  backticks and Python grammar.
- Make sure the code string keeps the correct 4 spaces
  indentation of Python.
- Make sure all the variables are defined before use.
- Make sure all the parameters are given when calling
  a function.
- Only output the codes in list format:["code1","code2"]
- Do not output other explainable words.
```

(c) Answer Aggregator Prompt for Answer Selection p_{aga}

```
You will be provided with a question asking about an
image. Multiple codes have been executed to generate
answers for the question, which are collected in an answer
list. Your task is to choose the correct answer.
- Evaluate each answer based on its relevance to the
  question and its frequency.
- You must choose one answer, and if you have no idea,
  choose the most frequently occurring one.
```

(d) Answer Aggregator Prompt for Code Selection p_{agc}

```
Given a question related to an image, you will receive
multiple versions of Python code as potential choices.
Your task is to select the most clear and user-friendly
code.
- Choose only one code from the provided list and return
  its index.
- Consider simplicity for straightforward questions and
  complexity for more challenging ones.
- The selected code should be logically correct to solve
  the question and run without errors.
```

Fig. 4. Prompt definitions of query rephraser, code generator and answer aggregator. These models utilize the same LLM with different prompt templates. By using varied prompts as input, distinct expected outputs can be achieved despite the models being identical.

demonstrations s_{cg} from a set of cross-dataset ICEs V. This selection is based on both semantic and syntactic similarity between the rephrased query r and each ICE $v \in V$. These demonstrations are then utilized to construct in-context prompts for subsequent code generation. A detailed explanation of the ICE selector module is provided in Section IV.

D. Code Generator Module

The development of the code generator module is driven by the recognition of the intricate nature of problem-solving, which is often characterized by a diversity and multiplicity of viable solutions. Traditional PVQA approaches, as evidenced by prior works [17]–[19], predominantly focus on the generation of a singular Python code in response to a given query. However, this conventional paradigm tends to overlook the inherent variability in problem-solving methodologies, thereby failing to capture the diversity of feasible solutions that can lead to the correct answer. By augmenting the code generation process to encompass the capacity of producing multiple codes for a single query input, the objective of the code generator is to exploit the richness of possible solutions associated with a given query. This approach promotes a more nuanced and adaptable framework for code generation, particularly in the context of complex problem-solving scenarios.

```
get_object_boxes(img: Image, object: str) -> List(center_x
: float, center_y: float, width: float, height: float)
- Detects the object in the image and returns its
  bounding box.

crop(img: Image, box: [float, float, float, float], type:
str) -> List(Image)
- Crops the image according to the box and crop type.
- Type is one of ("object", "left", "right", "top",
  "bottom", "front").
- When type = "object", it will only return the object
  region.
- When type = "front", it will return the region in
  front of the object.
- When type is equal to other values, it will return the
  left/right/top/bottom image part of the object, note
  that the object itself is not included in this case.

query(img: Image, question: str) -> str
- Queries the image and returns an answer to the question
about the image content.

translate(text: str, source_lang: str, target_lang: str)
-> str
- Translates the input text from the source language to
the target language and returns the translated text.
```

Fig. 5. API reference p_{api}. The generated code by PVQA models calls different APIs to collaborate and perform various functions in order to get the final answer.

We define the code generator by

$$G(r, s_{\text{cg}}) \triangleq \pi(p_{\text{cg}} + p_{\text{api}} + s_{\text{cg}} + r), \tag{4}$$

where r is a rephrased query, p_{cg} is the code generator prompt in Figure 4b, p_{api} is API reference texts detailed in Figure 5, and s_{cg} is the demonstration generated by the ICE selector.

E. Answer Aggregator Module

Conventional methods typically rely on majority voting to consolidate solutions, where the answer with the highest occurrence frequency is deemed the ultimate output. Nevertheless, this approach confronts challenges in scenarios where unsuccessful code execution results in the default answer attaining majority status, or when multiple answers achieve equal predominance. In response to these constraints, the answer aggregator module leverages the capabilities of LLMs to determine the final output, taking into account not only frequency but also the semantic coherence between the query and potential answers. During pre-execution, each code z_k generates a candidate answer a_k. Subsequently, the answer aggregator proceeds in two stages: firstly, selecting the definitive answer from the candidate answer pool, and secondly, identifying the most suitable code that produced this final answer during pre-execution. This sequential methodology minimizes input token consumption and mitigates potential mismatches between answers and their corresponding code that might be caused by hallucinations.

Given a set of candidate answer-code pairs $Z = \{(z_k, a_k)\}_{k=1}^{O}$, we define the answer aggregator by

$$A_{\text{ans}}(Z) \triangleq \pi(p_{\text{aga}} + s_{\text{aga}} + [a_1, a_2, \cdots, a_O]), \tag{5}$$

$$A_{\text{code}}(Z_{\boldsymbol{\sigma}}) \triangleq \pi(p_{\text{agc}} + s_{\text{agc}} + [z_{\sigma_1}, z_{\sigma_2}, \cdots, z_{\sigma_l}]), \tag{6}$$

where p_{aga} is the prompt used by the answer aggregator to select the final answer shown in Figure 4c, and p_{agc} is the prompt for choosing the optimal code shown in Figure 4d. The operation [] represents the aggregation of elements into a list and subsequent conversion into string format. s_{aga} and s_{agc} serve as in-context demonstrations. $Z_{\boldsymbol{\sigma}} = \{z_{\sigma_l}\}_{l=1}^{L}$ is a subset of Z, where $\boldsymbol{\sigma} = \{\sigma_l\}_{l=1}^{L}$ is the output of A_{ans} and represents the set of code indices where the pre-execution output for the code matches the selected answer.

Algorithm 2 ICE selector $S(r, V)$

Input: Query $r \in \mathcal{Q}$, ICE set $V \subset \mathcal{T}$, tree depth T, current branch s, difference threshold θ, demonstration limit N
Output: Demonstration set $D_T \subset \mathcal{T}$

```
D_0 ← ∅
for t = 1, ··· , T do
    V' = Filter(V, s)
    D'_t ← {[s, v] | s ∈ D_{t−1}, v_t ∈ S_gen(r, V')}
    D_t ← {[s, v] | s, v ∈ D'_t, S_eval(s, v) > θ}
    if |D'_t| = 0 or (|D_t| = 0 and θ = 1) then
        ▷ switch S_gen and return to the previous level
    else if |D_t| = 0 then
        ▷ increase threshold θ
    end if
end for
D_T ← arg max_{D⊂D_t,|D|=N} Σ_{s,v∈D} S_eval(s, v)
return D_T
```

IV. ICE Selector

A. ICE Search Algorithm

As introduced in Section III-C, the ICE selector module aims to select appropriate ICEs to generate diverse demonstrations for code generation. The algorithm for the ICE selector is outlined in Algorithm 2. The selection process constructs a tree structure, where each node v represents a subset of ICEs, and each branch reaching the maximum layer s_T represents a complete set of ICEs forming one demonstration, with T representing the tree depth. Given the query r and the candidate ICE set V, each layer in the ICE selector iteratively calls a node generator S_{gen} and a node evaluator S_{eval}. The node generator S_{gen} selects appropriate ICE subsets to build nodes based on various similarity criteria. Subsequently, the node evaluator S_{eval} compares the similarity between each generated node and its parent nodes to ensure diversity within each demonstration. To maintain a requisite number of nodes at each layer, Algorithm 2 incorporates an automatic adjustment mechanism for the similarity function in S_{gen} and the difference threshold θ in S_{eval} module.The following sections provide details on the node generator in Section IV-B and the node evaluator in Section IV-C.

B. Node Generator

The node generator in the ICE selector leverages both semantic and syntactic characteristics to evaluate the similarity between an input query r and the candidate ICE set V'. The strategy of including syntactic characteristics arises from the observation that questions with similar structures often require analogous logical frameworks for code generation.

The node generator is defined as

$$S_{\text{gen}}(r, V') \triangleq \text{Div}(\text{Shuffle}(E)), b), \tag{7}$$

$$E = \text{Select}(\text{Sim}(r, V'), n \times b), \tag{8}$$

$$\text{Sim}(r, V') = \begin{cases} \text{Sem}(r, V') & t\%2 = 0 \\ \text{Syn}(r, V') & \text{otherwise} \end{cases}, \tag{9}$$

1) Similarity Computation: The function $\text{Sim}(r, V')$ calculates the similarity S between the input question and the example question in each ICE, alternating between semantic similarity Sem(), which is based on the cosine distance of sentence embeddings obtained from SBERT [49], and syntactic similarity Syn(), measured by the edit distance of part-of-speech label sequences. Alternating between these two similarity measures at each layer ensures diversity in the selection of examples.

2) Example Selection: Subsequently, $n \times b$ examples with the highest similarity scores are selected from the ICE set V' to form b nodes, where n is the number of examples per node. These examples are chosen based on their similarity scores $S = \text{Sim}(r, V')$ to the input query r. The selection function takes S and the length of the selection list $n \times b$ as inputs and selects the top $n \times b$ examples for the subsequent node-building process. It is defined as

$$\text{Select}(S, n \times b) = \underset{S' \subset S, |S'| = n \times b}{\arg\max} \sum_{s_i \in S'} s_i. \tag{10}$$

3) Random Shuffling: The selected examples E are then randomly shuffled to introduce randomness. The Fisher-Yates shuffle algorithm [58] is employed as the function $E' = \text{Shuffle}(E)$, as detailed in Algorithm 3. This algorithm iterates through the sequence from the last element to the first. For each element at position i, a random integer j between 1 and i is selected, and the elements at positions i and j are swapped. This process ensures that each permutation of the sequence is equally probable, resulting in a uniformly random shuffle of the elements.

Algorithm 3 Shuffle(E)

Input: A sequence $E = (e_1, e_2, \ldots, e_{n \times b})$
Output: A permuted sequence E'
  for $i = n \times b$ to 2 **do**
    $j \leftarrow$ random integer such that $1 \leq j \leq i$
    swap e_i and e_j
  end for

4) Node Division: The function $\text{Div}(E', b)$ splits an in-context example list E' of length $n \times b$ into b sub-lists as nodes, each containing n items. It is defined as

$$\text{Div}(E', b) = \{E_i \mid 1 \leq i \leq b\}, \tag{11}$$

$$E_i = \{e_{(i-1) \times n + k} \mid 1 \leq k \leq n\}, \tag{12}$$

where b represents the number of nodes, $\{E_i\}_{i=1}^{b}$ denotes the i-th generated node, with i as the node index, and k as the index of an example within each node.

C. Node Evaluator

The node evaluator prunes nodes that exhibit significant similarity to their parent nodes, fostering diversity within the demonstration and mitigating computational overhead. The node generator prioritizes the similarity between the input query and example questions, whereas the node evaluator emphasizes the similarity of example code between the current node and its parent nodes.

The node evaluator is defined as

$$S_{\text{eval}}(s, v) \triangleq \text{Diff}(\text{AST}(s), \text{AST}(v)), \tag{13}$$

where s is the branch containing parent nodes, and v is the newly generated node requiring evaluation.

1) AST Extraction: The AST() function begins by extracting the Abstract Syntax Tree (AST) from the input code c, which represents the abstract syntactic structure of the code in a tree format. This AST is then converted into a flattened string, which is subsequently split into a list of strings for the purpose of difference comparison, as outlined in Algorithm 4. To obtain the AST from the input code, the Python function ast.parse() from the ast module is employed. This function returns the root node n of the AST, representing the parsed code. The flattening of the syntax tree n into a string f is achieved by the Flatten() function, which recursively concatenates the string representations of child nodes $\text{Flatten}(i, l+1)$, along with the current node's name, $\text{name}(n)$. In this process, l denotes the level of the node, and it governs the number of times the indentation string d is repeated to nodes at different levels in the flattened AST string. Once flattened, the AST string is split into a list s, where each element corresponds to the AST representation of a line of code.

Algorithm 4 AST(c)

Input: code string c, indentation string $d =$ ‘ ’
Output: flattened AST string list s
  $n \leftarrow$ ast.parse(c) # Obtain the root node of the AST
  function Flatten(n, l)
    $m \leftarrow$ get_children(n) # Retrieve the child nodes
    if $m \neq$ null **then**
      $f \leftarrow \text{concat}(\text{repeat}(d, l), \text{name}(n), \text{‘(’})$
      for each i in m **do**
        $f \leftarrow \text{concat}(f, \text{Flatten}(i, l+1), \text{‘,’})$
      end for
      $f \leftarrow \text{concat}(f, \text{‘)’})$
    else
      $f \leftarrow \text{concat}(\text{name}(n), \text{‘()’})$ # Leaf node case
    end if
    return f
  end function
  $f \leftarrow$ Flatten(n, 0) # Start from the root node with level 0
  $s \leftarrow$ split(f, ‘)’) # Split into a list of lines

2) Syntactic Difference: The $\text{Diff}(a, b)$ function computes the difference between two sequence of flattened AST string lists to capture syntactic differences. Given two sequences of flattened string lists, a and b, the function $\text{Diff}(a, b)$ calculates

the percentage of AST string lines in a that are not present in b.

$$\text{Diff}(a, b) = 1 - \frac{|a \cap b|}{|a|}. \tag{14}$$

V. Experiments

A. Experimental Settings

Four VQA datasets, namely GQA [59], VQAv2 [32], NLVR2 [60], and MaXM [61], are employed for assessment in the conducted experiment. The GQA dataset is recognized for its challenging questions that require spatial reasoning, commonsense reasoning, and a detailed understanding of images. The VQAv2 dataset consists of images from the COCO dataset paired with open-ended questions, encompassing a range of complexities and topics, with answers expected in free-form text. The NLVR2 dataset consists of statements about pairs of images, each annotated as true or false based on the visual content. For our experiments, these statements are reformulated into a query format. The MaXM dataset is a test-only multilingual VQA benchmark available in seven diverse languages: English (en), French (fr), Hindi (hi), Hebrew (iw), Romanian (ro), Thai (th), Chinese (zh). It comprises over 200 samples in each language and addresses the limitations of predominantly English-focused VQA research by incorporating typologically, genealogically, and geographically diverse languages. The evaluation metric for all experiments is the exact match accuracy for case-insensitive answers in visual question answering.

To conserve computational resources, we utilize the same randomly sampled subsets as the baseline [19] for evaluation purposes. In the results table, the GQA val2000 (GQA-2k) subset comprises 2000 samples randomly selected from the GQA validation set, and the VQAv2 val4000 (VQAv2-4k) subset includes 4000 examples randomly sampled from the VQAv2 validation set.

B. Implementation Details

To evaluate the insensitivity of ContextualCoder to the underlying base LLM model π, we employ a code-generation specific open-sourced LLM, CodeLlama-7b-Python [38], and two closed source general LLMs, GPT-3.5 Turbo [21] and Gemini-Pro [41], for performance assessment.

We validate the efficacy of our approach by employing state-of-the-art PVQA models, CodeVQA [30] and PyramidCoder [19], as our baseline framework. Utilizing identical image processing modules as those used in baselines, we employ GroundingDINO [62] for object detection via `get_object_boxes()`. For image captioning, we leverage BLIP [63] within `query()`. Furthermore, the question-answering module incorporated within `query()` employs the same LLM as the one used in the code generation stage.

Following the baseline methods, we wrote codes for 50 randomly sampled questions from the training sets of the GQA, VQAv2, and NLVR2 datasets to create respective dataset-specific ICEs. These ICEs were subsequently combined into a larger set, forming cross-dataset ICEs comprising 150 ICEs that are not tied to any specific dataset. Given that all available ICEs are in English, we utilize Gemini-Pro [41] through `translate()` for translation exclusively for the MaXM dataset to ensure seamless adaptation to multilingual VQA. For demonstration construction, in line with the baseline approach, 12 ICEs are sampled to construct a demonstration for datasets with single image input (GQA, VQAv2, and MaXM), while 6 examples are employed for datasets with image set input (NLVR2).

The limitations for the number of rephrased queries M, demonstrations N, and code generation O are set to 3. In the ICE selector, the maximum number of nodes in each layer b is fixed at 3. For datasets with image set input, the tree depth T is set to 3, and the number of examples in one node n is 2. Conversely, for single-image input datasets, T is set to 4, with n being 3.

TABLE I
Comparison with SOTA methods for visual question answering. Accuracy (%) on three datasets is reported. Code: ✓ indicates PVQA methods with code generation. LLM: large language model types. The highest scores are in bold, and the second-highest are underlined.

Method	Code	LLM	GQA val2000	GQA testdev	VQAv2 val4000	NLVR2 test
Gemini Vision [41]	–	Gemini 1.0 Pro	43.6	37.9	49.3	52.3
BLIP-2 [64]	–	Flan-T5-XXL	–	44.7	65.0	–
PnP-VQA [14]	–	text-davinci-003	52.1	**46.6**	<u>66.8</u>	63.4
ViperGPT [18]	✓	CodeLlama-7b	32.7	26.6	35.6	56.2
VisProg [17]	✓		50.8	41.2	57.0	55.5
CodeVQA [19]	✓		48.0	42.8	61.3	57.8
PyramidCoder [30]	✓		51.3	43.4	62.8	60.7
ContextualCoder	✓		51.6	43.9	62.8	63.1
CodeVQA [19]	✓	GPT3.5 Turbo	46.6	38.9	53.7	62.2
ContextualCoder	✓		<u>53.8</u>	42.7	61.3	<u>67.9</u>
CodeVQA [19]	✓	Gemini 1.0 Pro	51.5	43.0	61.1	65.5
ContextualCoder	✓		**56.0**	<u>46.5</u>	**67.2**	**72.9**

C. VQA Performance

Table I presents the VQA performance across three datasets. It's worth noting that this comparison may not be entirely fair to our method, as previous PVQA models utilize dataset-specific ICEs following their original settings, whereas ContextualCoder leverages cross-dataset ICEs. As discussed in Section V-I, dataset-specific ICEs generally achieve better performance because they align more closely with the target dataset's characteristics and biases. Other experimental settings are the same for all experiments. Compared to previous PVQA baselines, ContextualCoder demonstrates performance enhancements across diverse underlying LLMs and datasets, indicating its effectiveness. When leveraging Gemini as the backbone, ContextualCoder showcases significant relative improvements compared to the CodeVQA baseline: 4.5 points on the GQA val2000 set, 3.5 points on the GQA test set, 6.1 points on the VQAv2 val4000 set, and 7.4 points on the NLVR2 test set.

Additionally, we conduct comparisons with prior works that utilize pre-trained language models for answer prediction without code generation, including Gemini Pro Vision [41], zero-shot BLIP-2 [64], and few-shot PnP-VQA [14]. Notably, PVQA models using Gemini for intermediate code

TABLE II
ACCURACY (%) ON THE MULTILINGUAL MAXM DATASET. LLM: GEMINI STANDS FOR GEMINI 1.0 PRO, CODELLAMA FOR CODELLAMA-7B-PYTHON, AND GPT3.5 FOR GPT3.5 TURBO. THE HIGHEST SCORES ARE IN BOLD, AND THE SECOND-HIGHEST ARE UNDERLINED.

Method	Code	LLM	Language						
			en	fr	hi	iw	ro	th	zh
Gemini Vision [41]	–	Gemini	33.9	27.3	15.0	31.1	25.4	19.0	12.3
OFA-large [65]	–	–	35.6	13.3	**41.5**	31.7	26.4	29.5	17.6
ViperGPT [18]	✓	CodeLlama	23.0	17.4	10.8	16.1	19.0	11.9	2.7
VisProg [17]	✓		23.7	9.1	14.2	21.1	19.0	13.1	9.7
CodeVQA [19]	✓		34.2	29.9	16.9	31.1	30.6	21.3	21.3
PyramidCoder [30]	✓		35.6	31.8	17.3	31.8	30.6	23.9	21.8
ContextualCoder	✓		36.2	**34.1**	19.6	34.3	**33.1**	27.6	24.9
CodeVQA [19]	✓	GPT3.5	31.9	27.3	18.1	24.6	23.9	19.8	14.8
ContextualCoder	✓		34.6	32.2	22.3	27.5	29.9	24.3	17.7
CodeVQA [19]	✓	Gemini	29.6	26.5	18.1	29.6	27.1	21.6	20.9
ContextualCoder	✓		**37.4**	32.2	23.5	**43.6**	32.0	**34.7**	**27.4**

TABLE III
ACCURACY (%) WITH OR WITHOUT THE TRANSLATION MODULE ON THE MULTILINGUAL MAXM DATASET USING CODELLAMA BASE LLM.

Method	Translator	Language					
		fr	hi	iw	ro	th	zh
CodeVQA [19]	✗	29.6	5.0	9.6	11.6	15.3	10.5
CodeVQA [19]	✓	29.9	16.9	31.1	30.6	21.3	21.3
ContextualCoder	✗	28.4	7.3	12.5	11.6	14.9	9.0
ContextualCoder	✓	**34.1**	**19.6**	**34.3**	**33.1**	**27.6**	**24.9**

generation show improved VQA performance compared to directly using the multi-modality LLM (Gemini Pro Vision) for visual question answering. Comparative analysis against all previous methods indicates that ContextualCoder consistently matches or outperforms previous methods, highlighting the efficacy of ContextualCoder. Furthermore, it's noteworthy that ContextualCoder exhibits more pronounced enhancements when deployed on a stronger base LLM (CodeLlama versus Gemini). We attribute this phenomenon to the greater potential inherent in stronger models, which can be more effectively activated by our prompting strategy.

D. Multilingual VQA performance

To evaluate the performance of ContextualCoder in a multilingual context, experiments were conducted using the multilingual MaXM dataset, as outlined in Table II. Notably, using available English ICEs directly on other languages yields poor performance due to disparities between ICE and input-output languages (Table III). To address this limitation, a translation module that can translate the text from the source language to the target language is included as an API to facilitate language unification.

The multimodal large language model Gemini Pro Vision [41] and one of the state-of-the-art VQA models, OFA-Large [65], are utilized as baselines. Note that OFA-Large first translates the input question to English, applies the model, and then translates the answer back to the target language. The results indicate consistent enhancements in VQA performance across various languages and with different base LLMs when employing ContextualCoder in comparison to the PVQA baselines. Notably, ContextualCoder outperformed all the baselines in every language except Hindi (hi). Possible reasons for the performance variation in Hindi include data scarcity and linguistic differences, which may limit the effectiveness of text processing and consequently affect the performance of PVQA models. This result highlights the effectiveness of ContextualCoder in a multilingual context and further underscores its robustness and adaptability.

TABLE IV
ACCURACY (%) ON REPRESENTATIVE PROMPTING METHODS.

Method	GQA val2000	VQAv2 val4000
Input-Output Prompting (Default)	47.5	60.0
Chain-of-thought [22]	48.1	59.1
Tree-of-thought [23]	48.3	60.5
ContextualCoder (Ours)	**51.6**	**62.8**

TABLE V
ABLATION STUDY WITH RESPECT TO THE FOUR MODULES OF CONTEXTUALCODER. ACCURACY (%) IS REPORTED.

Method	GQA val2000	VQAv2 val4000	MaXM en
ContextualCoder	**51.6**	**62.8**	**36.2**
w/o query rephraser	50.8	60.4	35.4
w/o ICE selector	49.1	61.3	34.6
w/o code generator	50.4	59.6	34.2
w/o answer aggregator	51.4	61.8	35.4

E. Compared with other prompting methods

Table IV provides a comparative analysis of our ContextualCoder against representative prompting methodologies, namely input-output prompting, few-shot CoT, and few-shot ToT with a creative writing setup. The experiments are conducted on the GQA val2000 and VQAv2 val4000 datasets with CodeLlama as the base LLM. All methods utilize identical cross-dataset in-context examples and configurations. The results indicate that the preceding prompting techniques offer limited improvements over default IO prompting, primarily due to their inability to identify suitable ICEs and their tendency to generate homogeneous codes. In contrast, ContextualCoder surpasses these methods by producing diverse code candidates with richer contextual information, thereby providing more substantial reasoning evidence.

F. Ablation Study

The ablation study is conducted across three diverse datasets: GQA val2000, VQAv2 val4000, and MaXM en, utilizing CodeLlama as the base LLM. The comprehensive results are summarized in Table V. The first row showcases the performance of the complete ContextualCoder framework. To demonstrate the significance of each component, the query rephraser, ICE selector, code generator, and answer aggregator are individually omitted from the entire framework. The results clearly indicate that the exclusion of any of these modules results in a decrease in VQA accuracy across all datasets.

TABLE VI
ABLATION STUDY OF KEY COMPONENTS OF THE ICE SELECTOR MODULE. ACCURACY (%) IS REPORTED.

PVQA model	Method	GQA-2k	VQAv2-4k
ContextualCoder	ICE selector	51.6	62.8
	w/o Tree-of-Thoughts	50.3	60.4
	w/o syntactic similarity	49.4	61.8
	w/o node generator	48.0	59.9
	w/o node evaluator	50.8	61.9

TABLE VII
ACCURACY (%) ON DIFFERENT ICE SELECTION METHODS.

PVQA model	Selection Method	GQA-2k	VQAv2-4k
CodeVQA [19]	Random selection	47.3	59.7
	Naive selection	47.5	60.5
	Manual selection	48.0	61.3
	ICE selector	49.7	61.7
ContextualCoder	Random selection	47.4	60.6
	Naive selection	48.8	61.1
	Manual selection	51.3	**62.8**
	ICE selector	**51.6**	**62.8**

G. Ablation Study of ICE selector

As a critical component of ContextualCoder, we conducted ablation experiments to evaluate the effectiveness of each design element of the ICE selector. The ICE selector is composed of four key elements: the Tree-of-Thoughts selection process, syntactic similarity, node generator, and node evaluator, each of which was individually ablated, as presented in Table VI. In the configuration without the Tree-of-Thoughts selection process, the ICE selector generates the entire set of ICEs at once, bypassing the iterative layered selection process. In the absence of syntactic similarity, the module relies solely on semantic similarity for selecting ICEs. When the node generator is excluded, each node, representing an ICE subset, is generated through random sampling. Finally, when the node evaluator is removed, a random selection of up to O nodes is retained at each layer. The results demonstrate that all four components are essential to the overall performance of ContextualCoder, as their removal leads to significant performance degradation, highlighting their contributions to the selection process.

H. Comparison of ICE selection methods

1) ***Baseline Selection Methods****:* A comparison of various ICE selection methods is conducted using CodeVQA and ContextualCoder with general ICEs, as shown in Table VII. The random selection method randomly selects ICEs, the naive selection method uses sentence embeddings from SBERT [49] to select ICEs, while manual selection is performed by humans. The results indicate that our proposed ICE selector significantly outperforms the naive selection method and achieves human-level selection performance.

2) ***Sentence Embedding Methods****:* Cosine similarity is widely regarded as a standard in natural language processing for measuring sentence similarity, with the cosine similarity of sentence embeddings being a commonly used approach for ICE selection [17]–[20]. To investigate whether more advanced sentence embeddings can mitigate the issue of suboptimal in-context example selection, we apply various representative sentence embedding methods, as shown in Table VIII. The results demonstrate that modern sentence embedding methods enhance ICE selection for the GQA dataset; however, the improvement is less significant for the VQAv2 dataset. This divergence arises because the GQA dataset typically contains longer and more complex questions involving spatial and commonsense reasoning, whereas VQAv2 questions tend to be shorter and more repetitive, making them harder to differentiate in the embedding space. Notably, none of the tested methods outperformed our proposed ICE selector, particularly on the VQAv2 dataset.

TABLE VIII
COMPARATION OF THE ICE SELECTOR MODULE AND REPRESENTATIVE SENTENCE EMBEDDING METHODS. ACCURACY (%) IS REPORTED.

PVQA model	Method	GQA-2k	VQAv2-4k
CodeVQA [19]	SBERT [49]	47.5	60.5
	SimCSE [50]	48.7	59.9
	RankCSE [51]	49.4	59.8
	BGE Embedding [52]	49.4	60.1
	ICE selector (Ours)	**49.7**	**61.7**
ContextualCoder	SBERT [49]	48.8	61.1
	SimCSE [50]	50.1	60.7
	RankCSE [51]	48.4	60.9
	BGE Embedding [52]	49.7	60.2
	ICE selector (Ours)	**51.6**	**62.8**

TABLE IX
COMPARATION OF THE ICE SELECTOR MODULE AND REPRESENTATIVE RERANKING METHODS USED IN RAG. ACCURACY (%) IS REPORTED.

PVQA model	Method	GQA-2k	VQAv2-4k
CodeVQA [19]	BGE reranker [52]	48.6	60.3
	ColBERT [53]	49.4	60.2
	BGE M3 [66]	49.5	60.4
	ICE selector (Ours)	**49.7**	**61.7**
ContextualCoder	BGE reranker [52]	50.4	61.4
	ColBERT [53]	47.9	60.4
	BGE M3 [66]	50.6	60.7
	ICE selector (Ours)	**51.6**	**62.8**

3) ***RAG Re-ranking Methods****:* RAG re-ranking methods can also be employed for ICE selection by retrieving candidate examples based on relevance to the query and re-ranking them using a generative model to prioritize examples that are closely related to the input query. A comparison with representative RAG re-ranking methods is provided in Table IX, including the cross-encoder-based reranker BGE [52], the late-interaction reranker ColBERT [53], and BGE M3 [66], which integrates three common retrieval functionalities. Similar to sentence embedding methods, the results indicate that while modern re-ranking methods achieve performance close to that of the ICE selector on the GQA dataset, they still fall slightly short. Additionally, all tested methods show limited performance improvement on the VQAv2 dataset due to its shorter and more repetitive questions, which pose challenges for effective re-ranking.

TABLE X
COMPARISON OF PVQA METHODS UTILIZING DIFFERENT SOURCES OF ICES, WITH ACCURACY (%) REPORTED.

PVQA model	ICE source	GQA-2k	VQAv2-4k	NLVR2-test
CodeVQA [19]	GQA	48.0	58.8	52.8
	VQAv2	47.8	61.3	53.8
	NLVR2	34.2	43.4	57.8
	cross-dataset	47.5	60.5	53.8
ContextualCoder	GQA	**51.8**	61.1	53.2
	VQAv2	47.7	**63.3**	53.9
	NLVR2	35.4	45.7	**63.2**
	cross-dataset	51.6	62.8	63.1

TABLE XI
COMPARISON OF PVQA METHODS UTILIZING DATASET-SPECIFIC AND CROSS-DATASET ICES, WITH ACCURACY (%) REPORTED.

ICE source	PVQA model	GQA-2k	VQAv2-4k
Dataset-specific	CodeVQA [19]	48.0	61.3
	PyramidCoder [30]	51.3	62.8
	ContextualCoder (Ours)	**51.8**	**63.3**
Cross-dataset	CodeVQA [19]	47.5	60.5
	PyramidCoder [30]	48.5	60.8
	ContextualCoder (Ours)	**51.6**	**62.8**

I. Dataset-specific and Cross-dataset ICEs Analysis

Previous PVQA models rely on dataset-specific ICEs, which are examples selected from the same dataset as the target question. We conducted experiments to evaluate whether ICEs from different datasets, or a combination of them, can provide sufficient contextual guidance for code generation. As shown in Table X, using ICEs from different datasets results in significant performance degradation. This decline is attributed to the distinct biases embedded in the ICEs, reflecting the unique characteristics of each dataset. For example, GQA emphasizes complex reasoning with longer questions, VQAv2 covers a wide range of topics, and NLVR2 focuses on image comparison, each leading to specialized code structures in their respective ICEs. Notably, when using cross-dataset ICEs that include ICEs from both the target dataset and other datasets, the proposed ContextualCoder maintains performance with an average performance percentage decrease of only 0.45%, compared to CodeVQA's 6.9%. These results highlight the superior ICE selection abilities of our model, even when dealing with more general and complex ICE candidates.

Cross-dataset ICEs, which combine ICEs from multiple datasets, mitigate the issue of dataset-specific bias by exposing the model to a broader array of question types and reasoning patterns. Table XI presents a comparison of PVQA models using dataset-specific and cross-dataset ICEs as sources. Previous PVQA models struggled to select appropriate ICEs from cross-dataset ICEs, leading to notable performance declines. In contrast, our proposed ContextualCoder, quipped with a novel ICE selection module, maintains strong performance even when transitioning from dataset-specific to cross-dataset ICEs. It exhibits a performance percentage decrease of only 0.59%, significantly outperforming other models like CodeVQA (1.17%) and PyramidCoder (4.32%). This consistency underscores

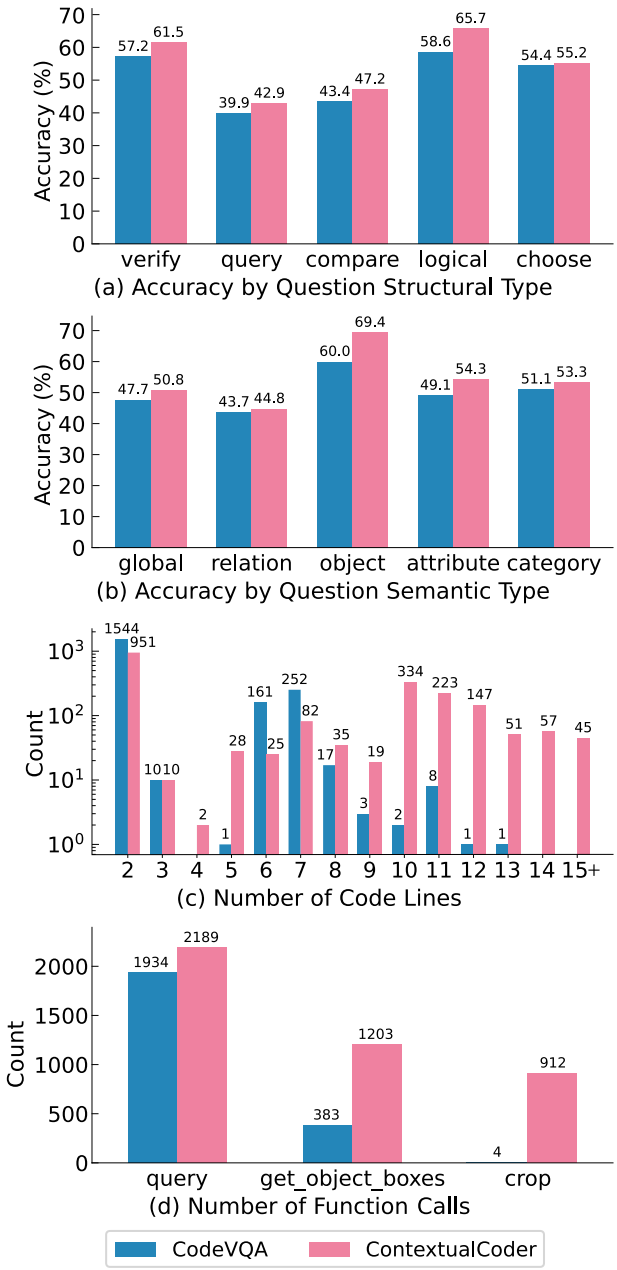

Fig. 6. Statistical comparison between CodeVQA and ContextualCoder on GQA val2000 set. (a) and (b) present the accuracy categorized by question structural and semantic types. (c) provides the distribution of the number of code lines. (d) displays the frequency of function calls in the generated code.

ContextualCoder's enhanced adaptability to varied ICE contexts, demonstrating its potential for optimal ICE selection. Notably, ContextualCoder, even when using diverse cross-dataset ICEs, outperforms baseline models using biased dataset-specific ICEs. This highlights ContextualCoder's stronger generalization abilities and its consistent performance.

J. Statistical Analysis

Figure 6 compares the statistical performance of the CodeVQA baseline with ContextualCoder. Figure 6 (a) presents QA accuracy across various question structural types. Significant improvements in accuracy are evident across all question categories with the implementation of ContextualCoder, particularly

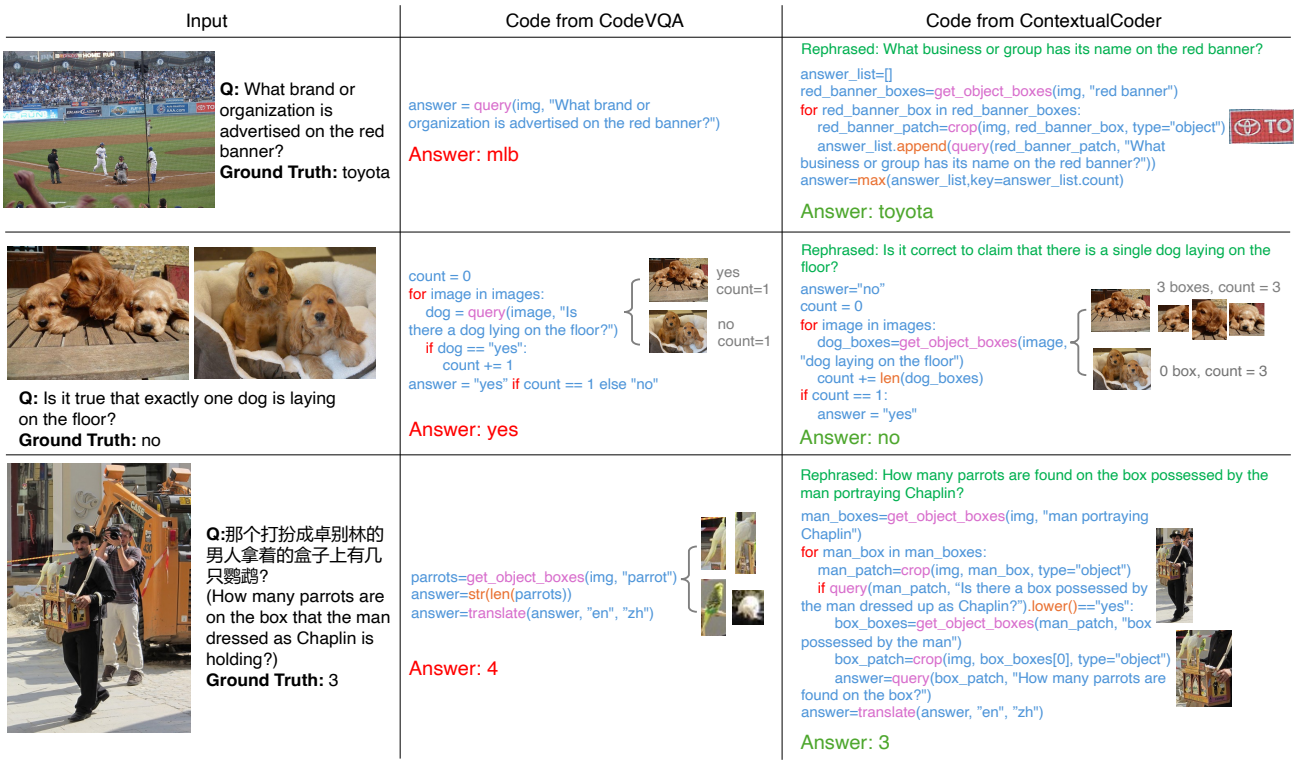

Fig. 7. Qualitative examples from the VQAv2, NLVR2, and MaXM datasets. Notably, the second example from NLVR2 involves an image pair as input, highlighting the visual comparison aspect, while the third example from MaXM showcases the model's multilingual capability by providing a Chinese question.

in logical inference questions, such as "logical," "verify," and "compare" types. Figure 6 (b) addresses semantic question types, demonstrating that ContextualCoder significantly enhances performance in visual component-related questions, notably the "object" and "attribute" types. Additionally, there is heightened performance in questions requiring global information, such as "category," "global," and "relation" types. These results indicate ContextualCoder's robust capability in handling complex reasoning and comprehending visual content, showcasing its versatility and efficacy across diverse QA scenarios.

Figure 6 (c) presents the distribution of line numbers in generated codes. Unlike the CodeVQA baseline, which predominantly produces two-line codes, ContextualCoder generates a more diverse distribution of line counts, often producing longer and more complex code to capture the finer details of the input question and facilitate a more thorough breakdown. Additionally, Figure 6 (d) shows that the complexity of ContextualCoder's generated code is attributed to a higher frequency of function calls, which are employed to more effectively decompose the question. This is particularly evident in functions designed for question breakdown, such as `get_object_boxes()` and `crop()`, which are frequently invoked in ContextualCoder but are rarely called in the CodeVQA baseline.

K. Qualitative examples

Diverse qualitative examples are depicted in Figure 7, showcasing the input, generated code, execution process, and predicted answers for both the CodeVQA baseline and the proposed ContextualCoder. Notably, the ContextualCoder demonstrates an enhanced ability to comprehend the question context and identify key elements, resulting in more coherent and diverse code outputs. When confronted with queries of distinct content, the generated code exhibits varying degrees of complexity and employs different APIs to construct disparate code solutions.

VI. Conclusion

In this paper, we proposed ContextualCoder, a novel hierarchical framework designed to enhance the capabilities of Programmatic Visual Question Answering. By employing frozen large language models and off-the-shelf models, ContextualCoder eliminates the need for additional training and ensures flexibility. The innovative prompting methodology and refined in-context example selection strategy address the limitations of previous PVQA models, resulting in more effective and diverse code generation and state-of-the-art PVQA performance. Extensive experimentation across various VQA datasets, including multilingual contexts, demonstrates the effectiveness and adaptability of ContextualCoder.

Acknowledgments

This work is supported by project JPNP18002, commissioned by the New Energy and Industrial Technology Development Organization (NEDO), JSPS KAKENHI Grant Number JP23H00490, and JST SPRING Grant Number JPMJSP2106 and JPMJSP2180.

References

[1] M. Malinowski, M. Rohrbach, and M. Fritz, "Ask your neurons: A neural-based approach to answering questions about images," in *ICCV*, 2015, pp. 1–9.

[2] H. Zhong, J. Chen, C. Shen, H. Zhang, J. Huang, and X.-S. Hua, "Self-adaptive neural module transformer for visual question answering," *IEEE Transactions on Multimedia*, vol. 23, pp. 1264–1273, 2020.

[3] H. Zhang and W. Wu, "Context relation fusion model for visual question answering," in *ICIP*. IEEE, 2022, pp. 2112–2116.

[4] S. Zhang, Y. Chen, Y. Sun, F. Wang, H. Shi, and H. Wang, "LOIS: Looking out of instance semantics for visual question answering," *IEEE Transactions on Multimedia*, vol. 26, pp. 6202–6214, 2024.

[5] Z. Yang, X. He, J. Gao *et al.*, "Stacked attention networks for image question answering," in *CVPR*, 2016, pp. 21–29.

[6] P. Anderson, X. He, C. Buehler *et al.*, "Bottom-up and top-down attention for image captioning and visual question answering," in *CVPR*, 2018, pp. 6077–6086.

[7] H. Xu and K. Saenko, "Ask, attend and answer: Exploring question-guided spatial attention for visual question answering," in *ECCV*, 2016, pp. 451–466.

[8] L. Gao, Y. Lei, P. Zeng, J. Song, M. Wang, and H. T. Shen, "Hierarchical representation network with auxiliary tasks for video captioning and video question answering," *IEEE Transactions on Image Processing*, vol. 31, pp. 202–215, 2021.

[9] X. Wu, J. Lu, Z. Li, and F. Xiong, "Ques-to-visual guided visual question answering," in *ICIP*. IEEE, 2022, pp. 4193–4197.

[10] B. Qin, H. Hu, and Y. Zhuang, "Deep residual weight-sharing attention network with low-rank attention for visual question answering," *IEEE Transactions on Multimedia*, vol. 25, pp. 4282–4295, 2023.

[11] L. H. Li, H. You, Z. Wang *et al.*, "Unsupervised vision-and-language pre-training without parallel images and captions," in *NAACL*, 2021, pp. 5339–5350.

[12] Y.-C. Chen, L. Li, L. Yu, A. El Kholy, F. Ahmed, Z. Gan, Y. Cheng, and J. Liu, "Uniter: Universal image-text representation learning," in *ECCV*. Springer, 2020, pp. 104–120.

[13] J. Cho, J. Lei, H. Tan, and M. Bansal, "Unifying vision-and-language tasks via text generation," in *ICML*. PMLR, 2021, pp. 1931–1942.

[14] A. M. H. Tiong, J. Li, B. Li, S. Savarese *et al.*, "Plug-and-play vqa: Zero-shot vqa by conjoining large pre-trained models with zero training," in *EMNLP Findings*, 2022, pp. 951–967.

[15] A. Mao, Z. Yang, K. Lin, J. Xuan, and Y.-J. Liu, "Positional attention guided transformer-like architecture for visual question answering," *IEEE Transactions on Multimedia*, vol. 25, pp. 6997–7009, 2022.

[16] X. Yang, F. Liu, and G. Lin, "Effective end-to-end vision language pretraining with semantic visual loss," *IEEE Transactions on Multimedia*, vol. 25, pp. 8408–8417, 2023.

[17] T. Gupta and A. Kembhavi, "Visual programming: Compositional visual reasoning without training," in *CVPR*, 2023, pp. 14 953–14 962.

[18] D. Suris, S. Menon, and C. Vondrick, "Vipergpt: Visual inference via python execution for reasoning," in *ICCV*, 2023, pp. 11 888–11 898.

[19] S. Subramanian, M. Narasimhan *et al.*, "Modular visual question answering via code generation," in *ACL*, 2023, pp. 747–761.

[20] J. Ge, S. Subramanian, B. Shi, R. Herzig, and T. Darrell, "Recursive visual programming," *arXiv preprint arXiv:2312.02249*, 2023.

[21] T. Brown, B. Mann, N. Ryder *et al.*, "Language models are few-shot learners," in *NeurIPS*, 2020, pp. 1877–1901.

[22] J. Wei, X. Wang, D. Schuurmans *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, vol. 35, 2022, pp. 24 824–24 837.

[23] S. Yao, D. Yu, J. Zhao *et al.*, "Tree of Thoughts: Deliberate problem solving with large language models," in *NeurIPS*, 2023.

[24] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for GPT-3?" in *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, May 2022, pp. 100–114.

[25] T. Sorensen, J. Robinson, C. Rytting, A. Shaw, K. Rogers, A. Delorey, M. Khalil, N. Fulda, and D. Wingate, "An information-theoretic approach to prompt engineering without ground truth labels," in *ACL*, 2022, pp. 819–862.

[26] E. Tanwar, S. Dutta, M. Borthakur, and T. Chakraborty, "Multilingual llms are better cross-lingual in-context learners with alignment," in *ACL*, 2023, pp. 6292–6307.

[27] I. Levy, B. Bogin, and J. Berant, "Diverse demonstrations improve in-context compositional generalization," in *ACL*, 2023, pp. 1401–1422.

[28] H. Gonen, S. Iyer, T. Blevins, N. A. Smith, and L. Zettlemoyer, "Demystifying prompts in language models via perplexity estimation," in *EMNLP*, 2023, pp. 10 136–10 148.

[29] T. Nguyen and E. Wong, "In-context example selection with influences," *arXiv preprint arXiv:2302.11042*, 2023.

[30] R. Shen, N. Inoue, and K. Shinoda, "Pyramid coder: Hierarchical code generator for compositional visual question answering," in *2024 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2024, pp. 430–436.

[31] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. L. Zitnick, and D. Parikh, "VQA: Visual question answering," in *ICCV*, 2015, pp. 2425–2433.

[32] Y. Goyal, T. Khot *et al.*, "Making the v in vqa matter: Elevating the role of image understanding in visual question answering," in *CVPR*, 2017, pp. 6904–6913.

[33] T. Le, H. T. Nguyen, and M. Le Nguyen, "Vision and text transformer for predicting answerability on visual question answering," in *ICIP*. IEEE, 2021, pp. 934–938.

[34] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[35] H. Touvron, L. Martin, K. Stone *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[36] M. Chen, J. Tworek *et al.*, "Evaluating large language models trained on code," *arXiv2107.03374*, 2021.

[37] R. Li, L. B. allal, Y. Zi, *et al.*, "Starcoder: may the source be with you!" *Transactions on Machine Learning Research*, 2023.

[38] B. Roziere, J. Gehring, F. Gloeckle *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[39] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," *Advances in neural information processing systems*, vol. 36, 2024.

[40] Z. Yang, L. Li, K. Lin, J. Wang, C.-C. Lin, Z. Liu, and L. Wang, "The dawn of lmms: Preliminary explorations with gpt-4v (ision)," *arXiv preprint arXiv:2309.17421*, vol. 9, no. 1, p. 1, 2023.

[41] G. Team, R. Anil *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[42] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, and Z. Sui, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[43] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *ICLR*, 2023.

[44] Y. Fu, H. Peng, A. Sabharwal *et al.*, "Complexity-based prompting for multi-step reasoning," in *ICLR*, 2023.

[45] S. Yao, J. Zhao, D. Yu *et al.*, "React: Synergizing reasoning and acting in language models," in *ICLR*, 2023.

[46] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," in *ACL*, 2022, pp. 8086–8098.

[47] G. G. Chowdhury, *Introduction to modern information retrieval*. Facet publishing, 2010.

[48] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[49] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084

[50] T. Gao, X. Yao, and D. Chen, "Simcse: Simple contrastive learning of sentence embeddings," in *EMNLP 2021-2021 Conference on Empirical Methods in Natural Language Processing, Proceedings*, 2021.

[51] J. Liu, J. Liu, Q. Wang, J. Wang, W. Wu, Y. Xian, D. Zhao, K. Chen, and R. Yan, "Rankcse: Unsupervised sentence representations learning via learning to rank," *arXiv preprint arXiv:2305.16726*, 2023.

[52] S. Xiao, Z. Liu, P. Zhang, N. Muennighoff, D. Lian, and J.-Y. Nie, "C-pack: Packaged resources to advance general chinese embedding," *arXiv preprint arXiv:2309.07597*, 2023.

[53] O. Khattab and M. Zaharia, "Colbert: Efficient and effective passage search via contextualized late interaction over bert," in *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, 2020, pp. 39–48.

[54] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," *arXiv preprint arXiv:2004.04906*, 2020.

[55] R. Pradeep, S. Sharifymoghaddam, and J. Lin, "RankVicuna: Zero-shot listwise document reranking with open-source large language models," *arXiv:2309.15088*, 2023.

[56] W. Sun, L. Yan, X. Ma, S. Wang, P. Ren, Z. Chen, D. Yin, and Z. Ren, "Is chatgpt good at search? investigating large language models as re-ranking agents," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 14 918–14 937.

[57] J. Johnson, B. Hariharan, L. van der Maaten *et al.*, "Inferring and executing programs for visual reasoning," in *ICCV*, 2017, pp. 2989–2998.
[58] M. Eberl, "Fisher–yates shuffle," *Archive of Formal Proofs*, September 2016, https://isa-afp.org/entries/Fisher_Yates.html, Formal proof development.
[59] D. A. Hudson and C. D. Manning, "GQA: A new dataset for real-world visual reasoning and compositional question answering," in *ICCV*, 2019, pp. 6700–6709.
[60] A. Suhr, S. Zhou, A. Zhang, I. Zhang, H. Bai, and Y. Artzi, "A corpus for reasoning about natural language grounded in photographs," in *ACL*, 2019, pp. 6418–6428.
[61] S. Changpinyo, L. Xue, I. Szpektor, A. V. Thapliyal, J. Amelot, M. Yarom, X. Chen, and R. Soricut, "MaXM: Towards multilingual visual question answering," in *EMNLP*. Association for Computational Linguistics, 2023, pp. 2667–2682.
[62] S. Liu, Z. Zeng, T. Ren *et al.*, "Grounding DINO: Marrying DINO with grounded pre-training for open-set object detection," *arXiv preprint arXiv:2303.05499*, 2023.
[63] J. Li, D. Li, C. Xiong, and S. Hoi, "BLIP: Bootstrapping language-image pre-training for unified vision-language understanding and generation," in *ICML*. PMLR, 2022, pp. 12 888–12 900.
[64] J. Li, D. Li, S. Savarese, and S. Hoi, "BLIP-2: bootstrapping language-image pre-training with frozen image encoders and large language models," in *ICML*, 2023, pp. 19 730–19 742.
[65] P. Wang, A. Yang, R. Men, J. Lin, S. Bai, Z. Li, J. Ma, C. Zhou, J. Zhou, and H. Yang, "Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework," in *ICML*. PMLR, 2022, pp. 23 318–23 340.
[66] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," *arXiv preprint arXiv:2402.03216*, 2024.

VII. Biography

Ruoyue Shen received her B.E. degree in software engineering from the University of Electronic Science and Technology of China, China, in 2020, and her M.S. degree in computer science from the Tokyo Institute of Technology, Japan, in 2022. She is currently pursuing a Ph.D. in Artificial Intelligence at the Institute of Science Tokyo (formerly Tokyo Institute of Technology), Japan. Her research interests include multimodality, computer vision, natural language processing, and vision-language understanding.

Nakamasa Inoue (Member, IEEE) received his B.E., M.E., and Ph.D. degrees in computer science from Tokyo Institute of Technology in 2009, 2011, and 2014, respectively. He is an Assistant Professor in the Department of Computer Science at the Institute of Science Tokyo (formerly Tokyo Institute of Technology), Japan. His main research interests lie in multimedia processing, including video retrieval, image recognition, and speech recognition.

Dayan Guan received his Ph.D. from Zhejiang University, China, in 2019. he was a Research Fellow at Nanyang Technological University from 2019 to 2022 and from 2023 to 2024. Between these periods, he served as a Research Scientist at MBZUAI from 2022 to 2023. He is an Associate Professor at the Harbin Institute of Technology, China. His research interests include scene understanding, unsupervised learning, and multimodality.

Rizhao Cai received his B.E. degree in electronic information engineering from Shenzhen University, China, in 2018 and his Ph.D. degree from Nanyang Technological University, Singapore, in 2024. He has been working as a Project Officer at the ROSE Laboratory and the NTU-PKU Joint Research Institute, leading projects that are collaborated with industrial partners. His research interests include computer vision and biometric security.

Alex C. Kot (Fellow, IEEE) has been with the Nanyang Technological University, Singapore since 1991. He was Head of the Division of Information Engineering and Vice Dean of Research at the School of Electrical and Electronic Engineering. Subsequently, he served as Associate Dean for the College of Engineering for eight years. He is currently a Professor and Director of the Rapid-Rich Object SEarch (ROSE) Lab and NTUPKU Joint Research Institute. He has published extensively in the areas of signal processing, biometrics, image forensics and security, and computer vision and machine learning. Dr. Kot served as Associate Editor for more than ten journals, mostly for IEEE Transactions. He served the IEEE SP Society in various capacities, such as the General Co-Chair for the 2004 IEEE International Conference on Image Processing and the Vice-President for the IEEE Signal Processing Society. He received the Best Teacher of the Year Award and is a co-author for several Best Paper Awards, including ICPR, IEEE WIFS and IWDW, CVPR Precognition Workshop, and VCIP. He was elected as the IEEE Distinguished Lecturer for the Signal Processing Society and the Circuits and Systems Society. He is a Fellow of IEEE and a Fellow of the Academy of Engineering, Singapore.

Koichi Shinoda (Senior Member, IEEE) received the B.S. and M.S. degrees in physics from the University of Tokyo, Tokyo, Japan, in 1987 and 1989, respectively, and the D.Eng. degree in computer science from the Tokyo Institute of Technology, Tokyo, in 2001. In 1989, he joined NEC Corporation, Kawasaki, Japan, where he was involved in research on automatic speech recognition. From 1997 to 1998, he was a Visiting Scholar with Bell Labs, Lucent Technologies, Murray Hill, NJ, USA. From October 2001 to March 2003, he was an Associate Professor at the University of Tokyo. He is currently a Professor at the Institute of Science Tokyo (formerly Tokyo Institute of Technology). His research interests include speech recognition, video information retrieval, and machine learning. Mr. Shinoda is a Fellow of IEICE and a Senior Member of the Information Processing Society of Japan (IPSJ). He was the Chair of SIG-Spoken Language Processing (SLP) in the Information Processing Society of Japan from 2014 to 2016 and one of the general co-chairs of the Asia Pacific Signal and Information Processing Association (APSIPA) 2021.