A. Appendix

We used an LLM as a general-purpose writing assistant for minor edits (clarity, grammar, or phrasing) and for generating alternative phrasings of paragraphs we had already drafted.

A.1. Details of VIRO

A.1.1. Input Arguments

We provide the details of VIRO by extracting the program generator prompts. The following functions are available in our framework for reasoning:

- **FIND(object_name=**'object_name'): Returns all objects matching the object name which are clearly detectable, excluding non-object entities (e.g., living room, field, wall).
- **LOCATE(object=**objects, **position=**'location'): Returns objects positioned at a specified absolute location, independent of other objects in the 2D space (e.g., 'right', 'at the bottom', 'on left', '9 o clock', 'outmost right', 'top', 'uppermost', 'middle', 'center').
- **ORDER(object=**objects, **criteria=**['left'|'right'|'top'|'bottom'], **rank=**number): Returns the object positioned at the specified `rank` when sorted only by the given criteria ('left', 'right', 'top', 'bottom'), counting from the end.
- **ABSOLUTE_DEPTH(object=**objects, **criteria=**['front'|'behind']): Returns objects from `objects` positioned absolutely closest (front) or farthest (behind) in the 3D space (depth information).
- **SIZE(object=**objects, **criteria=**['big'|'small']): Returns objects filtered by relative size only by the given criteria ('big', 'small').
- **PROPERTY(object=**objects, **value=**'attribute'): Filters objects based on their intrinsic attributes (e.g., color and patterns: 'red', 'striped', clothing: 'wearing a blue shirt', states or actions: 'standing', 'sitting', 'turned on', 'open').
- **FIND_DIRECTION(object=**objects1, **reference_object=**objects2, **direction=**['left'|'right'|'top'|'bottom']): Returns objects from `objects1` positioned next to objects in `objects2` only by the given criteria ('left', 'right', 'top', 'bottom').
- **FIND_NEAR(object=**objects1, **reference_object=**objects2): Returns objects from `objects1` that are spatially close to any object in `objects2`.
- **FIND_INSIDE(object=**objects1, **reference_object=**objects2): Returns objects from `objects1` that are strictly inside the reference object `objects2`.
- **RELATIVE_DEPTH(object=**objects1, **reference_object=**objects2, **criteria=**['front'|'behind']): Returns objects from `objects1` positioned in depth relative to objects in `objects2` only by the given criteria ('front', 'behind').
- **RESULT(object=**answer_object): Pre-processes the final selected object to the final answer form.

A.1.2. Verification Module

Attribute Operator. We use the `PROPERTY` operator to filter visual attributes using the CLIP and GroundingDINO-T model. CLIP has varying thresholds depending on the given image, which makes filtering based solely on similarity scores challenging. To improve the filtering process, we first apply a softmax transformation to the CLIP similarity scores of the candidate regions from the OVD. We then integrate GroundingDINO-T scores into the filtering process, combining them with the softmax CLIP scores via a weighted sum. Finally, we set an adaptive threshold based on the number of candidates from `FIND` operator.

Relative Spatial Operators. Relative spatial operators take at least two arguments: `object` and `reference_object`. The `reference_object` serves as the spatial standard, and the `object` denotes the entity whose position is specified relative to this reference. As discussed in Section 3.2.1, we use logical verification in `RELATIVE_DEPTH`, similar to `FIND_DIRECTION`, but with the relative depth values 'front' or 'behind'. Additionally, in `FIND_INSIDE`, if there is no intersection area between the `object` and `reference_object`, the proposal is rejected.

A.1.3. Adaptive Threshold

To compute the filtering score $S(l|I_j)$ in equation (2), for each candidate region I_j with label l, we leverage a pre-trained VLM trained with contrastive loss, such as CLIP. CLIP is designed to align visual and textual representations in a shared embedding space, enabling effective discrimination between semantically relevant (positive) and irrelevant (negative) pairs. However, one of the main challenges when using CLIP for verification is its inherent bias toward labels that appear frequently in its training data. Labels like 'person' or 'car' typically receive higher confidence scores compared to more specialized or rare objects, making a fixed threshold inappropriate for fair evaluation across different object categories. To address this label-specific bias, we implement an adaptive thresholding mechanism that calibrates the decision boundary for each target

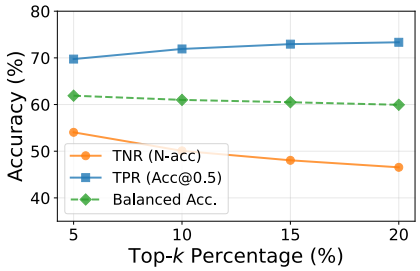

Figure A1. Analysis of k adaptive threshold which illustrates the trade-off between TPR and TNR.

label individually. We utilize ImageNet as an auxiliary calibration dataset, computing verification scores for a representative sample of images containing various object categories.

We collect 5 images per class in ImageNet (a total of 5,000 images) and use GroundingDINO to crop out only the relevant class objects (denoted as D_A). This process helps minimize bias from background elements, where many images contain a person even when the target class is not 'person'. For each target label l, we analyze the distribution of verification scores $S(l|I)$ across this preprocessed dataset to determine CLIP's typical confidence range for that specific category.

The calibration process employs a top-k selection strategy to determine the appropriate threshold δ_l Specifically, we rank all verification scores computed on the auxiliary dataset for target label l where $d \in D_A$, and the score is defined as:

$$S(l \mid d_i) = \frac{1}{K} \sum_{k=1}^{K} \frac{\exp(\text{sim}(d_i, l)/\tau)}{\exp(\text{sim}(d_i, l)/\tau) + \exp(\text{sim}(d_i, c_k)/\tau)} \,. \tag{5}$$

The set of verification scores for the entire dataset is represented as:

$$\mathcal{S}(l|D_A) = \{S(l|d_1), S(l|d_2), \ldots, S(l|d_n)\} \,, \tag{6}$$

where n is the number of auxiliary dataset. The threshold δ_l selects the top-k% highest scoring samples, where k is a hyper-parameter (typically set as 10). This strategy effectively captures the performance level that CLIP consistently achieves for high-confidence predictions of the target category, providing a data-driven threshold that reflects CLIP's inherent capability for that specific label.

We also investigate the sensitivity of our pipeline to k. As shown in Figure A1, VIRO's performance remains highly robust to changes in the Top-k percentage. Although a slight increase in TPR is observed with more candidates (as the ground-truth box is more likely to be included), the overall Balanced Accuracy stays stable across different values of k.

A.1.4. Justification for Neuro-symbolic Verification

VIRO addresses two fundamental limitations in existing approaches: *unreliable verification* and *cascading errors*. First, free-form Python approaches rely on LLM-generated code for verification (*e.g.*, `if` blocks), which is not explicitly enforced and thus prone to being incorrectly generated or omitted. In contrast, VIRO restricts program generation to a pre-defined set of verifiable primitives. Furthermore, methods like NAVER perform verification only after all reasoning steps, making them prone to cascading errors: inevitable false positives from open-vocabulary detectors propagate through intermediate reasoning steps and corrupt final predictions. In contrast, VIRO's operator-level verification ensures high-precision grounding by filtering out incorrect candidates.

A.2. Program Validator

The Program Validator serves as a safeguard against errors in LLM-generated programs. Its primary purpose is to detect syntactic, structural, and logical mistakes before program execution, and to provide structured feedback tailored to the specific type of error. Since LLM outputs are not guaranteed to be flawless, the validator detects invalid programs at pre-execution time and provides feedback that guides the LLM to regenerate a correct program.

A valid program must follow a simple and strict format: each line is written as `VAR = OP(ARG=..., ...)` and executed sequentially in order, while the program always terminates with a line that must take the form `FINAL_RESULT = RESULT(object=VAR)`. Given this constrained design, the validator enforces the following properties:

- **Syntax enforcement:** all lines except the last must take the form `VAR = OP(ARG=..., ...)`, and the last line must take the form `FINAL_RESULT = RESULT(object=VAR)`.

- **Variable tracking:** at line t, only variables defined in lines $1, \ldots, t-1$ may be referenced, and redefinition of an existing variable is disallowed to prevent overwriting outputs from earlier steps.
- **Argument typing:** arguments are checked for valid type (variable, string, or number). Certain arguments have additional constraints, e.g., `rank` must be a positive integer and `criteria` must be chosen from a predefined set (e.g., `left`, `right`, `top`, `bottom`).
- **Operator constraints:** each operator must belong to the predefined functions (e.g., `FIND`, `PROPERTY`), and each function must include all of its required arguments.
- **Output format:** the program must end with a single line in the form `FINAL_RESULT = RESULT(object=VAR)`, and no other `RESULT` may appear earlier in the program.

Through these checks, the Program Validator acts as a reliable feedback mechanism, reducing execution errors caused by incorrect code generated by the LLM. Both our method and ViperGPT [34] use programs generated by the LLM prior to execution, but the scope of validation differs fundamentally. ViperGPT only ensures that the generated code is syntactically valid Python code, which allows issues such as undefined variables, incorrect argument types, or missing required arguments to pass through unnoticed until runtime. In contrast, our validator applies strict structural and semantic checks before execution, ensuring correct format, valid variables, and a proper output statement. These checks reduce execution errors and provide targeted feedback for correction.

A.3. Experimental Setup

A.3.1. Datasets

RefCOCO/+/g. RefCOCO [43] primarily targets location-based expressions, while RefCOCO+ [43] focuses on attribute-based descriptions by prohibiting the use of absolute location words. RefCOCOg [26], in contrast, contains longer and more complex expressions, often combining both spatial relations and attributes. For RefCOCO and RefCOCO+, results are reported separately on two test splits: TestA, where the referent is a person, and TestB, where the referent is a non-person object, highlighting object-centric grounding. For RefCOCOg, results are reported on the standard single test split. All of these benchmarks are built upon the MSCOCO [23] image dataset.

gRefCOCO no-target split. gRefCOCO [11, 24] is a referring expression dataset that explicitly includes no-target queries, allowing evaluation of systems that must either localize the referred region or abstain when no valid target exists. To avoid trivial negatives, no-target expressions are constrained to be contextually related to the image, and annotators may reuse deceptive expressions from the same split when needed.

RefAdv. RefAdv [1] is an out-of-distribution (OOD) adversarial test set derived from Ref-Hard, the structure-dependent subset of RefCOCOg, designed to evaluate whether referring expression models truly leverage syntactic structure rather than rely on lexical cues. It reveals models' lack of syntactic understanding and limited generalization, even for those that perform well on the original RefCOCOg benchmark.

RefEgo. RefEgo [18] is a video-based referring expression comprehension (REC) dataset built upon first-person videos from the Ego4D [8], designed to evaluate models' ability to localize objects described in natural language under realistic conditions. The dataset contains clips, which are selected to feature rapid egocentric camera motion and multiple similar objects, and some include frames where the target object is absent, requiring models to detect no-target situations. Unlike prior REC datasets built from web-curated images, RefEgo focuses on testing language–vision grounding and generalization in real-world environments.

A.3.2. Evaluation Metrics

We assess both *no-target robustness* and standard REC accuracy, which jointly require addressing localization and classification. Following a binary classification view, we define outcomes based on the confusion matrix: **True Positive (TP)**: a target is present and the model correctly localizes it (Intersection-over-Union (IoU) > 0.5); **True Negative (TN)**: target is absent and the model correctly predicts its absence; **False Positive (FP)**: target is absent but the model incorrectly outputs a bounding box; and **False Negative (FN)**: a target is present but the model either predicts 'no-target' or localizes it incorrectly (IoU < 0.5).

A.3.3. Baseline Details

Fully supervised REC baselines are trained with full annotations on RefCOCO/+/g. GREC [11] further incorporates the gRefCOCO no-target dataset to address target-absent cases. By contrast, Qwen2.5-VL-72B-AWQ [2] is evaluated without fine-tuning on gRefCOCO. To enable no-target prediction, we append a negative instruction to its prompt, *e.g.*, "if there is no object, return []", thereby leveraging its inherent reasoning capability. The full prompt for Qwen2.5-VL is as follows: Detect

{description} in the image. Return a JSON list of bounding boxes as [x1, y1, x2, y2] in pixel coordinates relative to the input image. If there is no object, return [].

Proposal-based REC first parses the referring expression to isolate key linguistic components before matching them against pre-generated object proposals. **ReCLIP** [33] employs a syntactic parser to extract noun chunks, while **GroundVLP** [31] utilizes a traditional NLP toolbox to identify the main object. More recent approaches leverage the advanced capabilities of Large Language Models (LLMs) for this task; **SS-CLIP/SS-FLAVA** [10], two variants of the method in [10], use an LLM to parse the main object from the query. After this initial parsing stage, each method employs its unique mechanism to map the extracted components to the most relevant regions in given detected proposals from MAttNet [44]. These are architecturally constrained to a rich pool of candidate regions by Faster RCNN, making a direct comparison on target-absent task unfair.

Compositional reasoning REC parses complex queries into explicit programs. By generating and then executing these programs, they transparently handle multi-step compositional logic to derive the final result. We compare our approach with state-of-the-art methods in this domain, including **ViperGPT** [34], **HYDRA** [16], and **NAVER** [3], to benchmark its compositional reasoning capabilities.

A.4. Standard REC Accuracy Under Forced Prediction

Our framework is evaluated under a target-present assumption, wherein each query is guaranteed to have a corresponding target in the image. In this setting, our approach achieves high accuracy on standard benchmarks, as summarized in Table A1.

Table A1. Comparison of REC methods on standard benchmarks in terms of TPR (Acc@0.5), under a forced prediction setting.

	RefCOCO			RefCOCO+			RefCOCOg	
Method	Val	TestA	TestB	Val	TestA	TestB	Val	Test
Fully Supervised REC								
Qwen2.5-VL-72B [2]	92.7	94.6	89.7	88.9	92.2	83.7	89.9	90.3
GREC-MDETR-R101 [15]	86.8	89.6	81.4	79.5	84.1	70.6	81.6	80.9
GREC-UNINEXT-R50 [38]	89.7	91.5	86.9	79.8	85.2	72.8	84.0	84.3
Proposal-based REC								
ReCLIP [33]	45.8	47.0	45.2	45.3	48.5	42.7	57.0	56.2
SS-CLIP [10]	60.6	66.5	54.9	55.5	62.6	45.7	59.9	59.9
SS-FLAVA [10]	52.5	52.7	52.9	50.8	53.4	47.6	61.3	60.9
GroundVLP [31]	52.6	61.3	43.5	56.4	64.8	47.4	64.3	63.5
Detector-based REC								
GLIP-L [21]	47.5	52.6	41.8	44.1	48.6	39.8	51.9	52.6
GroundingDINO-T [25]	50.4	57.2	43.2	51.4	57.6	45.8	60.4	59.5
Compositional Reasoning REC with GLIP								
ViperGPT [34]	66.9	72.0	59.9	59.6	65.7	63.0	69.3	69.6
HYDRA [16]	68.0	73.1	62.5	55.8	60.6	50.6	67.2	67.6
NAVER [3]	69.6	73.4	64.4	59.0	62.7	56.4	70.7	70.0
VIRO (Ours)	71.4	75.7	63.8	59.3	66.2	51.3	70.6	71.5
Compositional Reasoning REC with GroundingDINO								
ViperGPT [34]	62.2	66.7	54.6	55.4	61.7	50.4	66.0	65.7
HYDRA [16]	62.0	62.8	62.4	55.0	58.4	51.6	66.4	67.1
NAVER [3]	61.1	64.2	58.2	56.4	60.1	51.8	68.4	68.4
GDINO-FLORA‡ [4]	73.7	78.5	67.8	63.2	71.6	53.5	72.5	72.1
VIRO (Ours)	71.4	75.0	64.4	59.5	65.8	50.1	69.6	70.3

‡ Official code has not been released as of September 25, 2025.

A.5. Handling No-Target Cases and Errors in Baselines

This section defines how we count no-target predictions and attribute errors for the baselines evaluated in our study. For **detector-based baselines**, if the detector returns no bounding boxes for the query, we label the prediction as no-target (correct). For **compositional-reasoning REC** baselines, prior implementations do not explicitly consider no-target cases; we therefore specify our handling of no-target outcomes and errors. Specifically, if the framework's final output is the empty set, we count the prediction as *no-target (correct)*. Further method-specific details are provided below.

ViperGPT [34]. In our evaluation of ViperGPT, an instance is classified as a no-target if the final code execution resulted in an empty list. Errors are identified when the `exec(compile(code, 'Codex', 'exec'), globals())` call returned None, which typically signifies a failure during code execution. ViperGPT's instructions enforce a fallback that retrieves the entire image when an object is not detected (e.g., when `len(object)==0`). This design choice creates ambiguity in identifying no-targets case. While likely intended to prevent downstream Python errors, this execution-safety behavior makes it difficult to distinguish a true 'no object found' scenario. We therefore adopt a conservative rule: treat only an empty final result as no-target, and treat None as error.

HYDRA [16]. For HYDRA, we adapt a strict criterion for no-target cases: we label no-target (correct) only when the number of detected main objects or relation-forming objects is exactly zero (*i.e.*, `len(...)==0`). Conversely, any case with `len(...)` $\geq$ `1` is classified as an error. While it could potentially represent a no-target scenario (*e.g.*, the model detected objects but failed to establish the queried relationship), it is not possible to definitively confirm this. Therefore, to maintain a conservative count of true no-target instances, we opt to classify these ambiguous cases as errors.

NAVER [3]. The NAVER pipeline operates through four distinct stages, each equipped with a self-correction mechanism: Perception, Logic Generation, Logic Reasoning, and Logic Answering. If the Perception stage fails to produce region proposals, it retries with a lower object detection threshold. In the Logic Generation stage, a failure prompts the LLM to retry the generation process. If the Logic Reasoning stage encounters a code error or yields no-target candidates, the pipeline reverts to the Logic Generation stage to create a new logical program. Finally, in the Answering stage, a Multimodal Large Language Model (MLLM)—specifically InternVL2-8B [5]—is prompted with the question, "Does the object meet the query?" If the MLLM answers 'no,' this stage is also retried.

After a maximum of five retry attempts across the pipeline, a query is classified as a no-target under one of three conditions: (i) perception failure (no detection), (ii) logic reasoning fails to establish the required relations (e.g., `contains`, `inside`, `is`), and (iii) the MLLM consistently answers 'no,' in answering state, indicating that no identified object satisfies the query.

Based on our implementation of the official code (at the time of our experiments), we observe that the vast majority of processing errors occur during the Logic Reasoning stage, specifically during the translation to or execution of the Probabilistic Problog program. Further analysis of NAVER's no-target cases is provided in Appendix A.6.3.

A.6. Detailed Experimental Results

We provide full benchmark results and baseline details across all evaluation splits for completeness.

A.6.1. Baseline Implementation for No-Target Cases

We evaluate our model's ability to handle negative cases where the target is not present in the image. The results on the gRefCOCO no-target split are shown in Table A2.

Table A2. Quantitative evaluation of no-target robustness on the gRefCOCO no-target split, reported as TNR (N-acc) (%).

Method	Val	TestA	TestB
Fully Supervised REC			
GREC-MDETR-R101 [15]	36.3	34.5	31.0
GREC-UNINEXT-R50 [38]	50.6	49.3	48.2
Detector-based REC			
GLIP-L [21]	14.8	21.7	18.2
GroundingDINO-T [25]	2.0	22.8	16.0
Compositional Reasoning REC with GLIP			
ViperGPT [34]	0.3	0.1	0.1
HYDRA [16]	7.2	6.3	6.1
NAVER [3]	14.9	13.9	13.0
VIRO (Ours)	56.7	50.1	53.2
Compositional Reasoning REC with GroundingDINO			
ViperGPT [34]	0.3	0.2	0.1
HYDRA [16]	8.6	7.5	7.0
NAVER [3]	3.0	3.4	1.8
VIRO (Ours)	56.5	50.2	52.9

A.6.2. Baseline Implementation for Standard REC Benchmarks

The detailed results for the RefCOCO, RefCOCO+, and RefCOCOg datasets are provided in Table A3, Table A4, and Table A5, respectively.

Table A3. Quantitative results on the RefCOCO dataset. Accuracy (%) is evaluated under two conditions: including all predictions (Inc.↑) and excluding model failure cases (Exc.↑). FR denotes the overall Failure Rate (%).

	Val			TestA			TestB		
Method	FR ↓	Exc.↑	Inc.↑	FR ↓	Exc.↑	Inc.↑	FR ↓	Exc.↑	Inc.↑
Detector-based REC									
GLIP-L [21]	0.00	47.5	47.5	0.00	52.6	52.6	0.00	41.8	41.8
GroundingDINO-T [25]	0.00	50.4	50.4	0.00	57.2	57.2	0.00	43.2	43.2
Compositional Reasoning REC with GLIP									
ViperGPT [34]	4.12	66.9	64.1	3.32	72.0	69.6	5.16	59.9	56.8
HYDRA [16]	19.60	68.0	54.7	17.27	73.1	60.5	24.14	62.5	47.4
NAVER [3]	17.97	69.6	57.1	14.85	73.4	62.5	20.71	64.4	51.1
VIRO (Ours)	0.09	68.1	68.1	0.07	72.8	72.8	0.14	60.6	60.5
Compositional Reasoning REC with GroundingDINO									
ViperGPT [34]	3.90	62.2	59.8	3.45	66.7	64.4	4.71	54.6	52.0
HYDRA [16]	26.13	62.0	45.8	28.46	62.8	44.9	34.6	62.4	40.8
NAVER [3]	8.77	61.1	55.7	6.01	64.2	60.3	10.70	58.2	52.0
VIRO (Ours)	0.09	68.2	68.1	0.07	71.9	71.9	0.14	60.8	60.8

Table A4. Quantitative results on the RefCOCO+ dataset. Accuracy (%) is evaluated under two conditions: including all predictions (Inc.↑) and excluding model failure cases (Exc.↑). FR denotes the overall Failure Rate (%).

	Val			TestA			TestB		
Method	FR ↓	Exc.↑	Inc.↑	FR ↓	Exc.↑	Inc.↑	FR ↓	Exc.↑	Inc.↑
Detector-based REC									
GLIP-L [21]	0.00	44.1	44.1	0.00	48.6	48.6	0.00	39.8	39.8
GroundingDINO-T [25]	0.00	51.4	51.4	0.00	57.6	57.6	0.00	45.8	45.8
Compositional Reasoning REC with GLIP									
ViperGPT [34]	6.97	59.6	55.4	3.28	65.7	63.6	6.73	53.0	49.4
HYDRA [16]	24.40	55.8	42.2	17.06	60.6	50.3	30.91	50.6	35.0
NAVER [3]	26.44	59.0	43.4	14.67	62.7	53.5	33.59	56.4	37.5
VIRO (Ours)	0.08	56.1	56.0	0.07	63.7	63.7	0.12	47.0	47.0
Compositional Reasoning REC with GroundingDINO									
ViperGPT [34]	6.71	55.4	51.7	6.83	61.7	57.5	6.46	50.4	47.1
HYDRA [16]	33.16	55.0	36.8	35.96	58.4	37.4	29.72	51.6	36.3
NAVER [3]	14.50	56.4	48.2	7.39	60.1	55.7	20.82	51.8	41.0
VIRO (Ours)	0.08	56.6	56.5	0.00	63.3	63.3	0.12	46.2	46.2

A.6.3. Analysis of NAVER

The Perception, Logic Reasoning, and Logic Answering modules in NAVER invoke self-correction when no valid target is found, effectively forcing an object prediction. To evaluate performance on the no-target split, we disable this behavior in all three modules and adopt an *early-exit* policy that stops the pipeline when "no-target" is detected. We report TNR on the gRefCOCO testA no-target set, and TPR on the RefCOCO testA set. As shown in Table A6, disabling self-correction improves TNR but at the cost of TPR. The module-wise distribution of early exits—indicating which component identifies the target's absence)—is summarized in Table A7. This analysis is intended to isolate no-target behavior and does not reflect the default official setting.

Table A5. Quantitative results on the RefCOCOg dataset. Accuracy (%) is evaluated under two conditions: including all predictions (Inc.↑) and excluding model failure cases (Exc.↑). FR denotes the overall Failure Rate (%).

	Val			Test		
Method	FR ↓	Exc.↑	Inc.↑	FR ↓	Exc.↑	Inc.↑
Detector-based REC						
GLIP-L [21]	0.00	51.9	51.9	0.00	52.6	52.6
GroundingDINO-T [25]	0.00	60.4	60.4	0.00	59.5	59.5
Compositional Reasoning REC with GLIP						
ViperGPT [34]	7.40	69.3	64.2	6.31	69.6	65.2
HYDRA [16]	23.98	67.2	51.1	21.44	67.6	53.1
NAVER [3]	28.44	70.7	50.6	25.90	70.0	51.9
VIRO (Ours)	0.28	66.2	66.0	0.38	67.0	66.8
Compositional Reasoning REC with GroundingDINO						
ViperGPT [34]	6.55	66.0	61.7	6.03	65.7	61.7
HYDRA [16]	34.07	66.4	43.8	32.37	67.1	45.4
NAVER [3]	38.50	68.4	42.1	9.74	68.4	55.8
VIRO (Ours)	0.24	66.2	66.0	0.38	66.6	66.3

Table A6. Impact of the self-correction on the NAVER framework. Performance is evaluated on gRefCOCO no-target TestA (TNR) and RefCOCO TestA (TPR) datasets.

Method	Balanced Acc ↑	TNR (gRef) ↑	TPR (Ref) ↑
NAVER w/ self-correction	33.8	3.4	64.2
NAVER w/o self-correction	63.2	71.6	54.8

Table A7. Frequency (%) of early exits per NAVER module contributing to TNR on the gRefCOCO no-target TestA split.

Module	**Frequency (%)**
Perception	67.8
Logic Reasoning	11.7
Logic Answering	20.5

A.6.4. Results on RefAdv dataset

To further evaluate the robustness of VIRO to linguistic perturbations, we analyze performance on the RefAdv benchmark, which introduces adversarially reordered referring expressions designed to test whether models rely on true syntactic understanding rather than superficial lexical cues. As shown in Table A8, VIRO achieves 64.2% Exc. and 63.8% Inc. accuracy on RefAdv, substantially outperforming prior compositional baselines, whose accuracy drops sharply due to frequent program-generation or execution failures.

A.6.5. Results on RefEgo validation set

In this section, we provide a detailed performance analysis on the RefEgo validation set in Table A9, extending the test set results presented in the main paper. While VIRO is not explicitly trained on egocentric video data, it significantly outperforms other off-the-shelf REC models such as OFA and MDETR. Notably, VIRO achieves 49.6% in Acc@0.5+n, demonstrating its strong zero-shot transferability in complex grounding scenarios without any task-specific fine-tuning.

A.7. Runtime Analysis with LLM backbones

We analyze the end-to-end latency of VIRO by decomposing the per-query runtime into two distinct stages: *pre-execution* and *execution*. As shown in Table A10, API-based models significantly reduce pre-execution overhead while maintaining competitive TPR. While a large-scale local LLM (Qwen2.5-72B-Instruct-AWQ) requires 12.21s for pre-execution, API-based acceleration via GPT-4o reduces this latency to 1.06s—a nearly 11.5× speedup—while even achieving higher TPR (72.3%). Consequently, the total runtime becomes dominated solely by the execution stage (fixed at 0.71s), rather than the reasoning

Table A8. Quantitative results with GroundingDINO on the RefAdv dataset. Accuracy (%) is evaluated under two conditions: including all dataset (Inc.↑) and excluding model failure cases (Exc.↑). FR denotes the overall Failure Rate (%).

	Test		
Method	FR ↓	Exc. ↑	Inc. ↑
Detector-based REC			
GLIP-L [21]	0.00	55.7	55.7
GroundingDINO-T [25]	0.00	60.5	60.5
Compositional Reasoning REC			
ViperGPT [34]	11.47	64.8	57.3
HYDRA [16]	36.61	65.8	41.7
NAVER [3]	13.23	64.0	55.5
VIRO (Ours)	0.59	64.2	63.8

Table A9. Comparison of video-based REC performance on the RefEgo validation set.

	All Frames		Target-present
Method	mSTIoU	Acc@0.5+n	TPR (Acc@0.5)
Fully Supervised with RefEgo			
MDETR+BH [18]	37.9	51.9	52.9
Off-the-shelf RefCOCOg model			
OFA [36]	16.9	30.8	29.6
MDETR [15]	17.4	28.3	25.2
Compositional Reasoning REC with GroundingDINO			
ViperGPT [34]	9.5	16.4	14.3
VIRO (Ours)	23.0	49.6	33.9

overhead. This efficiency is a direct consequence of VIRO's decoupled design, which enables flexible backbone selection to optimize latency-accuracy trade-offs.

Table A10. Pre-execution (Pre-exec.) runtime and accuracy comparison across different LLM backbones on RefCOCO testA. **The execution stage maintains a constant latency of 0.71s across all models**, reflecting VIRO's decoupled architecture.

LLM Backbone	**Pre-exec. (s)**	**TPR (%)**
Llama3.1-8B-Instruct (Local)	2.07	66.3
Qwen2.5-72B-Instruct-AWQ (Local)	12.21	71.9
GPT-4o mini (API)	1.37	69.1
GPT-4o (API)	1.06	72.3

A.8. Models Used in Compositional Baselines and VRAM Usage

Table A11 details the pre-trained vision-language components employed by each compositional baseline and their corresponding peak VRAM consumption, measured on an NVIDIA RTX A6000 (48GB). To ensure a focused comparison on the vision-language backbone efficiency, the reported memory excludes the VRAM occupied by the LLM itself.

Existing compositional baselines typically rely on heavy Multimodal LLMs (MLLMs), which incur significant memory and computational overhead. To avoid such overhead, VIRO is designed to operate without a MLLM, instead utilizing a more lightweight verification process. Furthermore, unlike NAVER, which requires an additional SegmentAnything [17] model, VIRO maintains a minimal set of components, requiring only 9.7GB of VRAM. Note that VRAM usage among baselines may vary depending on specific model-loading strategies, such as full pre-loading versus modular execution.

Table A11. Comparison of visual-language components and peak VRAM usage. Memory is measured during inference, excluding the LLM backbone.

	ViperGPT [34]	**HYDRA** [16]	**NAVER** [3]	**VIRO (Ours)**
Multimodal Encoder	X-VLM [45]	X-VLM [45]	X-VLM [45]	CLIP [28]
Open-set Detector	GDINO-T [25]	GDINO-T [25]	GDINO-T [25]	GDINO-T [25]
Depth Estimator	DPT-Large [29]	DPT-Large [29]	DepthAnythingV2 [40]	DepthAnythingV2 [40]
Multimodal LLM	BLIP-2 [20]	BLIP-2 [20]	InternVL2 [5]	N/A
Mask Generator	N/A	N/A	SegmentAnything [17]	N/A
VLM Memory Load	35.0 GB	21.8GB	29.9 GB	9.7 GB

A.9. Further Ablation Analysis

A.9.1. LLM Ablation Studies

To demonstrate that VIRO's performance stems from its architectural design rather than a specific LLM, we evaluate its robustness and scalability across various backbones, ranging from the Llama-3.1-8B [7] to the GPT-4o/mini [13].

First, we conduct an ablation study in which we replace the primary LLM backbone used in our main experiments with Llama-3.1-8B-Instruct [7], while keeping all other components and hyperparameters unchanged. The results, presented in Table A12, show that VIRO achieves a lower program-generation failure rate and maintains strong performance with Llama-3.1-8B. In contrast, ViperGPT and NAVER exhibit very high failure rates, which leads to substantially reduced Inc. TPR. We omit the results for HYDRA with Llama-3.1-8B-Instruct, as the model failed on nearly all queries; specifically, it struggled to adhere to HYDRA's iterative generation instructions, where even a minor formatting error caused the entire execution pipeline to collapse.

Table A12. Performance comparison on standard REC benchmarks using Llama3.1-8B-Instruct [7] as an alternative LLM backbone. We report results on the testA splits of RefCOCO/+ and the test split of RefCOCOg. Accuracy is measured both including (Inc.↑) and excluding (Exc.↑) failure cases. The failure rate (%) is reported for RefCOCOg.

		RefCOCO		RefCOCO+		RefCOCOg	
Method	Failure Rate ↓	Exc. ↑	Inc. ↑	Exc. ↑	Inc. ↑	Exc. ↑	Inc. ↑
Detector-based REC							
GLIP-L [21]	0.00	52.6	52.6	48.6	48.6	52.6	52.6
GroundingDINO-T [25]	0.00	57.2	57.2	57.6	57.6	59.5	59.5
Compositional Reasoning REC with GLIP							
ViperGPT [34]	40.09	65.1	48.4	53.2	41.3	57.6	35.1
NAVER [3]	70.15	72.3	32.4	65.1	28.6	69.6	20.8
VIRO (Ours)	0.34	68.2	68.2	56.6	56.6	61.4	61.4
Compositional Reasoning REC with GroundingDINO							
ViperGPT [34]	41.82	59.1	43.7	48.0	37.5	52.1	31.7
NAVER [3]	54.84	61.3	43.1	59.7	40.9	64.7	29.2
VIRO (Ours)	0.34	66.3	66.3	55.3	55.3	60.9	60.9

We further explore the scalability of our framework using GPT-4o and GPT-4o mini, while simultaneously clarifying the performance gap observed with the official NAVER results. As shown in Table A13, the high performance of NAVER (91.7%) stems from their use of a GroundingDINO-B detector fine-tuned on RefCOCO. To ensure a strictly zero-shot evaluation, we instead utilize GroundingDINO-T, which has not been exposed to the RefCOCO training set. Notably, when evaluated using the same zero-shot detector, VIRO consistently outperforms NAVER, confirming that our framework provides stronger zero-shot performance.

A.9.2. Early-exit Mechanism.

As shown in Table A14, incorporating the early-exit reduces the average latency focusing on the program execution phase to 0.52 seconds per query on the gRefCOCO no-target testA split. Since operators run sequentially, unmet intermediate conditions trigger immediate termination of the execution pipeline. For example, in the query "an elephant to the left of a

Table A13. Comparison of performance on the RefCOCO testA split using GPT-4o and GPT-4o mini. For a fair comparison, all methods are evaluated in a zero-shot setting except for the official NAVER entry using a fine-tuned detector.

Method	LLM	Detector	Acc@0.5
NAVER (Official)	GPT-4o-mini	GroundingDINO-B*	91.7
Fair Comparison: Zero-shot Setting			
NAVER (Official)	GPT-4o-mini	GroundingDINO-T	64.9
VIRO (Ours)	GPT-4o-mini	GroundingDINO-T	69.1
VIRO (Ours)	GPT-4o	GroundingDINO-T	**72.3**

*Fine-tuned on the RefCOCO training set.

man," the program exits as soon as the elephant is not found, thereby pruning redundant downstream visual operations. This mechanism effectively boosts throughput in scenarios where no-target cases are frequent.

Table A14. Ablation study on the early-exit mechanism. Latency and FPS are measured solely for the program execution stage, evaluated on the gRefCOCO no-target testA split.

Early-exit	Latency ↓	FPS ↑
Enabled	0.52	1.92
Disabled	0.58	1.79

A.10. Extension to Visual Question Answering via GQA

As a preliminary extension beyond REC, we adapt VIRO to the GQA dataset [12], a compositional VQA benchmark, as shown in Table A15. For a fair comparison, we utilize the same backbone models (BLIP-2, GLIP, and GPT-3.5-turbo) following the setup in HYDRA. For this task, we introduce two new operators: `ASK` for QA generation and `CROP_DIRECTION` for spatial focusing. This adaptation is implemented modularly by (i) defining the corresponding Python logic and (ii) registering the operators in the in-context prompt, allowing VIRO to support this task without retraining.

Table A15. Comparison of accuracy (%) on the GQA dataset.

Method	Acc
Trained RL Controller	
HYDRA (Official)	47.9
Zero-shot Setting	
ViperGPT (Official)	37.9
VIRO (Ours)	**45.4**

A.11. Qualitative Analysis

We present qualitative examples from RefCOCO validation set. Figure A2 demonstrates VIRO's ability to suppress false positives (FPs) from open-vocabulary detectors via CLIP-based verification, compared against previous REC methods.

A.12. VIRO Execution Examples

Figures A3, A4, and A5 show our program's execution process.

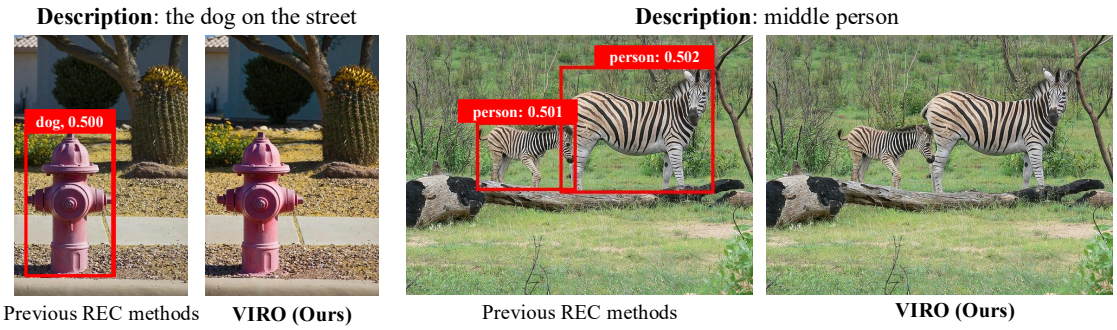

Figure A2. RefCOCO validation examples for false-positive suppression. Prior REC methods (left) yield spurious detections (red), whereas VIRO (right) rejects them via CLIP-based verification.

Query: the blade thing on the top jet

```
OBJ0 = FIND(object_name="blade")
OBJ1 = FIND(object_name="jet")
ANSWER0 = FIND_DIRECTION(object=OBJ0, reference_object=OBJ1, direction="top")
FINAL_RESULT = RESULT(object=ANSWER0)
```

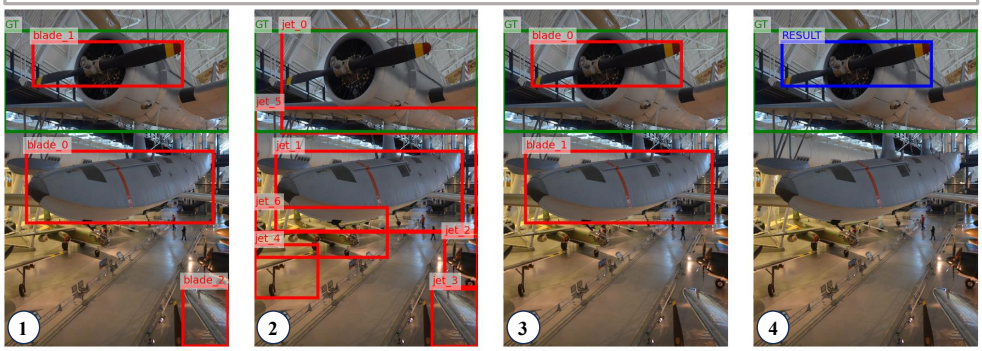

Figure A3. Program generated for the query "the blade thing on the top jet" (top), along with its sequential execution from step 1 to step 4 (bottom).

Query: cup to the top right of muffin

```
OBJ0 = FIND(object_name="cup")
OBJ1 = FIND(object_name="muffin")
OBJ2 = FIND_DIRECTION(object=OBJ0, reference_object=OBJ1, direction="top")
ANSWER0 = FIND_DIRECTION(object=OBJ2, reference_object=OBJ1, direction="right")
FINAL_RESULT = RESULT(object=ANSWER0)
```

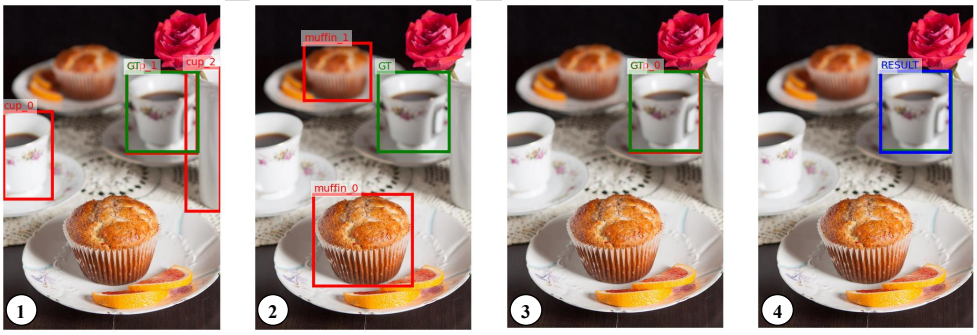

Figure A4. Program generated for the query "cup to the top right of muffin" (top), along with its sequential execution from step 1 to step 4 (bottom).

Query: man at head of table white shirt closest

```
OBJ0 = FIND(object_name="man")
OBJ1 = PROPERTY(object=OBJ0, value="wearing white shirt")
OBJ2 = FIND(object_name="table")
OBJ3 = FIND_DIRECTION(object=OBJ1, reference_object=OBJ2, direction="top")
ANSWER0 = ABSOLUTE_DEPTH(object=OBJ3, criteria="front")
FINAL_RESULT = RESULT(object=ANSWER0)
```

Figure A5. Program generated for the query “man at head of table white shirt closest” (top), along with its sequential execution from step 1 to step 6 (bottom).