ARM-Thinker: Reinforcing Multimodal Generative Reward Models with Agentic Tool Use and Visual Reasoning

Supplementary Material

Outline

In the appendix, we provide additional supporting materials to facilitate a deeper understanding of our work. First, in Sec. A, we present an overview of the models, datasets, and benchmark statistics used throughout ARM-Thinker, including detailed descriptions of the training data employed for multimodal reward modeling. Second, Sec. D reports comprehensive statistics and analyses of the ARMBench-VL benchmark. Third, in Sec. C, we showcase a diverse set of qualitative response cases, with a particular focus on visual reasoning and WeMath examples. Fourth, in Sec. D, we describe the implementation details of the multimodal tools integrated into our agentic framework. Finally, in Sec. E, we provide the full list of prompts used in our experiments.

A. Model, Dataset and Benchmark Statistic

A.1. Models

In our study, we adopt the Qwen2.5 family of vision–language models as the backbone for both training and evaluation. Unless otherwise specified, the term *base model* refers to Qwen2.5-VL-7B [1], on top of which we build our agentic reward model. The resulting model is denoted as ARM-Thinker-7B (ARM-Thinker-7B), which augments the backbone with an explicit think–act–observe agent loop and multi-stage GRPO training, while keeping the underlying architecture size (7B parameters) unchanged.

For fair comparison, we evaluate ARM-Thinker-7B alongside a diverse set of strong baselines that cover general-purpose LVLMs, specialized reward models, and visual tool-use models. As general-purpose LVLMs, we include Qwen3-VL-8B [1], InternVL3-8B [51], InternVL3.5-8B [32], and the proprietary GPT-4o [10].

To specifically assess reward-modeling capability, we further compare with UnifiedReward-7B [33], a recent multimodal reward model designed to unify understanding and generation evaluation. This baseline is evaluated on the same reward benchmarks as ARM-Thinker-7B, including RewardBench-2, VL-RewardBench, and our proposed ARMBench-VL, allowing us to isolate the benefit of adding an explicit agent loop and tool-use to reward modeling.

For *think-with-images* and tool-assisted visual reasoning, we include several specialized models that are explicitly trained with visual tool-use supervision: DeepEyes [49], Pixel Reasoner [26], and Mini-o3 [11]. These models serve as strong baselines on V* Bench, HRBench-4K/8K, and MME-RealWorld, where performance heavily relies on iterative zoom-in or crop operations. By contrast, ARM-Thinker-7B acquires its tool-calling behavior purely through reward-based optimization, without curated tool-use demonstrations, yet achieves accuracy comparable to or better than these specialized systems.

Finally, on general multimodal math and logical reasoning benchmarks, we also report results for larger or more specialized reasoning models such as Gemma-3-27B [27] and InternVL3-8B [51]. These models offer an upper-bound reference for reasoning performance on MMMU, MathVista, MathVision, MathVerse, WeMath, and LogicVista, and help contextualize how much of ARM-Thinker-7B's gain comes from improved agentic reward modeling rather than sheer model scale.

A.2. Training Data

We use two stages of data for training ARM-Thinker-7B: a Supervised Fine-Tuning (SFT) stage and a GRPO stage.

SFT Data. The SFT stage combines (i) preference-style reward data from LLaVA-Critic for general multimodal QA, and (ii) tool-specific data covering image zoom-in (DeepEyes), instruction-following checking (MM-IFEngine), and document retrieval (MP-DocVQA). After filtering, the SFT dataset consists of approximately $\sim$40k samples from LLaVA-Critic(augmented by interchanging the *resp_1* and *resp_2* order), $\sim$4k from DeepEyes, $\sim$1k from MM-IFEngine, and $\sim$1k from MP-DocVQA.

GRPO Data. For GRPO training, we sample a subset of the SFT-prepared data as queries. Each query is rolled out with multiple trajectories per iteration. Across both GRPO stages, we sample $\sim$20k from DeepEyes, and $\sim$4k from MP-DocVQA. Notably, We do not include Multimodal Instruction Following task–related tool-use data here, because these tasks are primarily abundant rather than difficult. Their core challenge lies in selecting the appropriate tool rather than executing complex tool-use logic. In our experiments, we observe that once the model is trained with our framework, its tool-use capability generalizes naturally to such tasks without requiring explicit inclusion of this data. This further demonstrates the generalization strength of our approach. These two stages together provide comprehensive supervision for reward modeling, chain-of-thought reasoning, and tool-use behavior in ARM-Thinker-7B.

B. ARMBench-VL Statistics

To provide a clear overview of the scale and composition of ARMBench-VL, we summarize the dataset statistics across

Single judge for Instruction Following Task in ARMBench-VL

You will receive a response(named as 'text_0') which follows the user's instruction or requirement to the provided image. Your Task is to judge whether the response satisfies the constraint. If it does, you should mark it as 'True', otherwise 'False' for you think the response does not satisfy the constraint.

<start_of_instruction>
{*instruction*}
<end_of_instruction>

<start_of_text_0>
{*prediction*}
<end_of_text_0>

<start_of_constraint>
{*constraint*]}
<end_of_constraint>

Output Format (strict)
You should make the final judgment wrapped in <answer></answer> XML tags: <answer>Overall Judgment: True (or False)</answer>

Figure 5. **Single judge for Instruction Following Task in ARMBench-VL**

N-way Pairwise Judge for Tasks in ARMBench-VL

You will receive two responses (named as 'resp_1' and 'resp_2') which follow the user's instruction or requirement to the provided image (or document). Your Task is to judge which response is better. Note that correctness is most important. If both are not correct, you should choose the one that is more better from other aspects.

<start_of_instruction>
{*instruction*}
<end_of_instruction>

<start_of_resp_1>
{*prediction1*}
<end_of_resp_1>

<start_of_resp_2>
{*prediction2*}
<end_of_resp_2>

Output Format (strict)
You should make the final judgment wrapped in <answer></answer> XML tags: <answer>Overall Judgment: Answer X is better (X must be either 1 or 2). </answer>

Figure 6. **N-way Pairwise Judge for Tasks in ARMBench-VL.** The figure illustrates the 2-way judging setup as an example. Our benchmark also includes 4-way comparisons, which follow the same structure but contain additional candidate responses to be judged.

its three major task categories: (1) Fine-grained Perception, (2) Multimodal Long Document QA, and (3) Multimodal Instruction Following. Each task includes different combinations of single-response judging (single_rm) and pairwise comparison judging (pair_rm), including 2-way and 4-way evaluation settings where applicable.

Fixed Chain-of-Thought Prompt for Agent-based Evaluation in ARMBench-VL

Important Requirement:
[If for image-based tasks]
The given image is 'original_image'.

[If for document-based tasks]
The given document is named '{*doc_id*}'. The page indices in the combined image start from 1 at the top-left corner and increase horizontally from left to right, then continue to the next row from top to bottom.

You must output your reasoning inside `<think>...</think>`. After reasoning, either output the final answer within `<answer>...</answer>` or call a tool within `<tool_call>...</tool_call>`. You may call tools multiple times across turns to assist with judgment or verification, **but only one tool per turn**. If a tool call fails, you may retry or stop and give your final answer. Once no more tool calls are needed, provide your final answer or judgment within `<answer>...</answer>`.

Figure 7. **Fixed CoT Prompt for Agent Models in ARMBench-VL.** This is the fixed suffix prompt appended after each task when evaluating agent-style models such as ARM-Thinker-7B. It enforces explicit reasoning, structured answers, and controlled tool usage.

Long Response Generation Template in ARMBench-VL

Assume you are a helpful assistant. You are given a question and a solution, together with the image. You need to generate two responses to the question based on the solution. The language style of the response can be varied.
For the first response, provide a detailed analysis of the question. This should include a concise explanation of how to approach the problem and then present the correct solution. The answer should include the original correct solution, but avoid excessive analysis or length—focus on clarity and providing the correct final answer. The answer should be smoothly conveyed in the end.
For the other three responses, offer a detailed analysis of the question with a similar approach. However, concludes with an incorrect solution. The wrong solution should seem plausible but contain a mistake, misleading the reader. Both responses should be conveyed in a confident tone, and should not provide any information about the correctness of the solution.

<start_of_question>
{*question*}
<end_of_question>

<start_of_solution>
{*solution*}
<end_of_solution>

Directly give back four responses in the following format:
response_1: ... **response_2**: ... **response_3**: ... **response_4**: ...

Figure 8. **Long Response Generation Template for Tasks in ARMBench-VL.** This example shows the template for generating one correct and three incorrect yet plausible long-form responses (i.e. 4-way comparisons). The 2-way long response generation template follow the same structure.

For the **Fine-grained Perception** task, the benchmark contains a total of 550 samples. This task focuses heavily on pairwise comparison, with 295 general pair_rm items, 163 2-way pairwise items, and 92 4-way comparison items. These multi-candidate settings reflect the nuanced, fine-grained nature of visual perception evaluation.

The **Multimodal Long Document QA** task includes 460 samples. Since this task emphasizes long-context reasoning over document-rich multimodal inputs, it incorporates both single-response judging (173 items) and pairwise 2-

Caption-style Question Rewriting and Response Generation in ARMBench-VL

Assume you are a helpful assistant. You are given an image with a related question and a solution. Your task is to generate two responses based on the image and the solution, and turn the original question into a new form. The language style of the response can be varied.
For the new question, it should contain **"describe"** and **"in detail"**, and focus on the part of the image that is **related to the original question and solution**. The language style should be diverse, and the question should be concise, without being overly specific to a single attribute.
For example:
Original Question: What is the hat color of the man on the roof? **New Question**: Describe the man on the roof in detail.
Original Question: What animal is depicted in the tattoo on the woman's arm? **New Question**: Describe the woman in detail, focusing on her arm.
For the first response, provide a detailed description of the scene in the image, addressing the key elements. The description should smoothly integrate the correct solution (e.g., if the solution states the cat is white, the response should describe the cat as white).
For the second response, describe the scene with a similar structure, but introduce some details that are slightly misleading or incorrect. The incorrect detail should not be immediately obvious, but it must contradict the original correct solution (e.g., describing the cat in a different color).
Both responses should satisfy the new question, offering a detailed description of the relevant region of the image. You do not need to output the solution explicitly. Both responses should be conveyed confidently and should **not** reveal any information about correctness.

<start_of_question>
$\{question\}$
<end_of_question>

<start_of_solution>
$\{solution\}$
<end_of_solution>

Directly give back two responses in the following format:
new question: ...
first: ...
second: ...

Figure 9. **Caption-style Question Rewriting and Response Generation Template in ARMBench-VL.** This example demonstrates how original questions are rewritten into descriptive caption-style queries and paired with both correct and subtly incorrect image-grounded responses. The structure extends to more diverse rewriting tasks in our benchmark.

Table 6. **Summary of Data Statistics across the Three Tasks in ARMBench-VL.**

Task	Total	Single-RM	Pair-RM (2-way / 4-way)
Fine-grained Perception	550	–	295 (pair_rm), 163 (2-way), 92 (4-way)
Multimodal Long Document QA	460	173	287 (2-way only)
Multimodal Instruction Following	489	489	–

way comparisons (287 items). No 4-way setting is used in this task to ensure controlled difficulty for long-context evaluation.

The **Multimodal Instruction Following** task contains 489 samples, all evaluated via single-response judging. This is because instruction-following quality can be assessed reliably through constraint satisfaction without multi-way comparison.

Tab. 6 summarizes all statistics in a unified table for clarity.

Image:

Question:
Describe the setting and ambiance of this location, highlighting its architectural features and design.

Constraint:
Your response must consist of exactly 3 paragraphs.

GT: Two Paragraphs
Judgment: False

text_0:
The setting is a grand staircase that exudes architectural elegance, with its ornate wooden railing and intricate carvings leading up to the upper floor. The walls are adorned with gold wallpaper, adding a touch of opulence to the space. The ...\n\n....the hotel's commitment to providing an experience that is both comfortable and visually stunning.

Qwen2.5-VL-7B:

<think>\nThe response provided by text_0 consists of three distinct paragraphs, which aligns with the constraint specified in the instruction. The first paragraph describes the architectural features and design of the staircase, the second paragraph discusses the ambiance and historical context, and the third paragraph elaborates on the decorative elements and the inviting atmosphere. Each paragraph contributes to a comprehensive description of the setting without exceeding the required length.\n</think>
<answer>Overall Judgment: True</answer>

ARM-Thinker:

<think>The user wants me to determine if the response text (`text_0`) meets the constraints specified. One of the constraints is that the response must consist of exactly 3 paragraphs...to verify this, I'll use the check_whether_response_paragraph_number_in_range function with lower_bound=3 and upper_bound=3 since we're checking if there are exactly 3 paragraphs.</think>
<tool_call>\n{\"name\": \"check_whether_response_paragraph_number_in_range\", \"arguments\": {\"response\": \"text_0\", \"lower_bound\": 3, \"upper_bound\": 3}}\n</tool_call>.

(tool_result)Check result: False"

<think>...</think>
<answer> Overall Judgment: False</answer>

Figure 10. **Case of Multimodal Instruction Following Judgment Task in ARMBench-VL.**

Image:

Question:
How many birds are there in the sky?

GT: Three
Judgment: Answer 1 is better

Resp 1:
Scanning the sky in the image, three birds are clearly visible against the backdrop of scattered clouds. This count aligns with careful observation of the aerial elements in the urban landscape. The solution is three.

Resp 2:
Scanning the sky in the image, no birds are visible amidst the scattered clouds. The scene shows only patches of soft cloud formations against a clear backdrop, with no shapes or silhouettes resembling birds.

Qwen2.5-VL-7B:

<think>\nThe task is to determine how many birds are present in the sky based on the provided image. Both responses analyze the sky for birds but come to different conclusions. \n\n- Resp_1 states there are three birds visible in the sky.\n- Resp_2 states there are no birds visible in the sky.\n\nUpon examining the image, it is clear that there are indeed no birds visible in the sky. The sky contains scattered clouds, but no birds can be seen flying or perched anywhere. Therefore, Resp_2 is more accurate and aligned with the visual evidence in the image.\n</think>
<answer>Overall Judgment: Answer 2 is better.</answer>

ARM-Thinker:

<think>The user is asking how many birds are in the sky in the provided image...I'll start by zooming in on a section of the sky where birds might be visible. Let's focus on the upper left portion of the image since that's where some small dark spots (possibly birds) were mentioned.</think>

<tool_call>\n{\"name\": \"image_zoom_in_tool\", \"arguments\": {\"image\": \"original_image\", \"bbox_2d\": [0, 0, 500, 500]}}\n</tool_call>

(tool_result)This is the zoom-in image `observation_1`.

<think>...Wait, the tool call was for the original image zoomed into [0,0,500,500], which is the upper left quadrant. In that zoomed view, there are three birds visible. So resp_1 is correct. Therefore, the answer should be 1.</think>
<answer>Overall Judgment: Answer 1 is better.</answer>

Figure 11. **Case of Fine-Grained Image Perception Judgment Task in ARMBench-VL.**

C. Qualitative Case Study

In this part, we show more model response cases. Fig. 10 shows a case of Multimodal Instruction Following Judgment Task in ARMBench-VL, and Fig. 11 shows a case of Fine-Grained Image Perception Judgment Task in ARMBench-VL.

D. Implementation Details of Multimodal Tools

All tools in our system inherit from a common baseTool interface and expose a unified OpenAI-style function-calling schema. Each tool implements a standard *create-execute-release* lifecycle and returns a tool response object that can contain both textual feedback and images. Below we detail the three families of multimodal tools used in our experiments.

Document–level multimodal retrieval tools. To support long-document question answering, we implement two complementary tools that operate on pre-rendered page images stored under image root using the naming pattern `{filename}_{page}.{ext}`. Both tools are built on top of a shared retriever manager that lazily instantiates a CLIP-based encoder and a persistent vector database.

For dense retrieval, we use a SentenceTransformer implementation of CLIP-ViT-B/32, loaded from a local HuggingFace cache in offline mode. Given a batch of texts, the encoder produces 512-dimensional embeddings on GPU when available. These embeddings are stored and queried via a `chromadb.PersistentClient` configured with anonymized telemetry disabled. We use a single collection (`COLLECTION_NAME`) with metadata fields including the document identifier (`source`) and page index (`page`). A global `RetrieverManager` holds the collection and is initialized exactly once using an `asyncio.Lock` to avoid race conditions during concurrent tool calls.

DocPageSearchTool takes a document name and a natural-language query as input. At execution time, it first ensures that the retriever manager is initialized; if retrieval is not available, it returns a structured error message to the model. Otherwise, it queries the Chroma collection with the given query, restricting the filter to the specified document (`where={''source'': filename}`) and retrieving up to k results (default k is 5). From the returned metadata, the tool deduplicates and preserves the order of page indices, then resolves each page to an image path via a helper that tries multiple file extensions. Missing pages are reported with an explicit error listing all attempted paths.

For visualization, the tool loads the corresponding page images and horizontally concatenates them using a dedicated image utility. Each page is first resized to a fixed maximum long side (`RAG_IMAGE_MAX_SIDE`, default 1120 pixels) while preserving aspect ratio. The concatenation canvas width is the sum of individual widths plus a fixed padding, and the height is the maximum of the resized heights. To avoid excessively large tensors and potential OOM errors during training, we enforce a hard cap on the total pixel count (`MAX_CONCAT_PIXELS`); if the stitched image exceeds this budget, it is downsampled isotropically. The tool then returns a single stitched image that visually aggregates the top-k pages along with a textual description summarizing which pages were retrieved, and hints to the model that it may want to refine the query if the retrieved context is not relevant.

DocPageByIndexTool provides a complementary, deterministic interface that bypasses dense retrieval. It takes a filename and a image_idx and directly returns the corresponding page image. The tool validates that the index, resolves the page to a file path (again trying multiple extensions), and fails with a clear error if the image cannot be found or the index is out of range. The page image is loaded and resized using the same long-side constraint as above, and the final response includes a single page image plus a short textual confirmation of the selected page. In practice, the model often uses DocPageSearchTool to locate a coarse region of interest and then DocPageByIndexTool to inspect specific pages sequentially.

Image zoom-in tool. To support fine-grained visual inspection, we implement an ImageZoomInTool that crops a sub-region from an existing image. The tool is designed to be robust to noisy bounding boxes and to integrate seamlessly with our multimodal reasoning loop.

The tool operates over a per-instance `response_store` that contains an `imgs_map` mapping logical image keys (e.g., `''original_image''`) to concrete image paths. At execution time, the model specifies an `image` key and a 2D bounding box `bbox_2d`. The bounding box is expressed in normalized integer coordinates within $[0, 1000]$ along each axis, which makes it easier for the language model to reason about relative locations while still allowing precise cropping. The tool first resolves the image key; if the key is not found, it returns a detailed error listing all currently available image identifiers to guide the model towards a valid call.

We apply strict validation on the bounding box: we check that it consists of four numeric values, each in $[0, 1000]$, and that $x_1 < x_2$ and $y_1 < y_2$. The normalized coordinates are then converted into absolute pixel coordinates based on the underlying image size, which is obtained via a lightweight `fetch_image` helper compatible with Qwen-VL style inputs. A dedicated helper, `_maybe_resize_absolute_bbox`, clamps the box to the image boundaries, enforces reasonable aspect ratios, and ensures that the cropped region is not too small. In

particular, we require that both width and height of the final crop are at least `MIN_QWEN_DIMENSION` pixels (set to 28 in our experiments). If the original box is too small, we automatically expand it around its center while still respecting the image boundaries; any invalid or degenerate boxes are omitted.

Once a valid bounding box is obtained, the tool crops the image accordingly. For very small crops (e.g., thumbnails or tiny regions), we optionally upsample the crop by a factor of 2 using bicubic interpolation to improve readability. The tool returns the cropped image and an instructional text that (i) names the new observation (e.g., `observation_2`), (ii) reminds the model to continue its reasoning within `<think>...</think>`, and (iii) encourages additional tool calls or final answers as appropriate. This design makes the zoom-in tool composable and easy to chain with the document retrieval tools.

Textual instruction-following tools. The third family of tools targets fine-grained textual instruction-following and is used to automatically verify whether a generated response satisfies structural and lexical constraints. All such tools inherit from a shared **BaseInstructionFollowingTool**, which automatically constructs the function schema from a declarative `parameters` list and manages a per-instance `response_store`. The `response_store` exposes a `texts_map` that maps logical keys (e.g., ``text_0'') to full string outputs. A helper `_resolve_from_store` resolves these keys, and provides informative errors that enumerate available keys when resolution fails. Each concrete tool implements an asynchronous `_execute_logic` method that returns a boolean, which is then wrapped into a textual `ToolResponse` of the form "Check result: True/-False".

We implement several categories of instruction-following tools:

- **Length and segmentation constraints.** Tools such as ParagraphNumberInRangeTool, SentenceNumberInRangeTool, EachParagraphSentenceNumberInRangeTool, EachParagraphSentenceNumberInRangeListTool, WordCountInRangeTool, and EachParagraphWordCountInRangeTool check whether the total or per-paragraph number of paragraphs, sentences, or words falls within specified bounds. Sentences are segmented using NLTK's sentence tokenizer, and paragraphs are defined via blank-line separation. For poetry-like formatting, EachParagraphSentenceNumberInRangeTool automatically switches to a line-based heuristic.
- **Lexical and formatting constraints.** Tools including NotContainSubstringTool, NotContainSubstringsTool, EachSentenceBeginsWithTool, EachSentenceEndsWithTool, EachParagraphBeginsWithTool, EachParagraphEndsWithTool, ResponseBeginsWithTool, ResponseEndsWithTool, and NoArabicNumberTool enforce constraints on the presence or absence of certain substrings, required prefixes or suffixes at the sentence or paragraph level, or the absence of standalone Arabic numerals. All matching is done in a case-insensitive manner after light normalization that strips punctuation and ellipses from boundaries.
- **Keyword coverage.** EachKeywordMentionedInRangeTool and TotalKeywordsMentionedInRangeTool check how often specific keywords appear in the response. We support both individual per-keyword bounds as well as global bounds over the total mention count. A specialized matcher handles hashtags and other special characters robustly by constructing appropriate regular expressions.
- **Numeric precision.** PercentagePrecisionTool and NumberPrecisionTool verify that all percentage expressions or decimal numbers in the response have exactly a specified number of digits after the decimal point. This allows us to enforce formatting requirements such as "report all percentages with two decimal places" in an automatic and tool-based manner.

These tools are purely textual and do not manipulate images, but they are implemented within the same function-calling framework as the multimodal tools. This uniform design allows the policy to learn a single tool-use interface while exhibiting rich behaviors: retrieving and inspecting visual context, zooming into fine-grained regions, and verifying that its final textual outputs satisfy complex instruction-following constraints.

E. Prompts

Prompts Used in Evaluation. To ensure consistent and reproducible evaluation across all tasks in ARMBench-VL, we employ three types of standardized prompts. These prompts are used respectively for (1) constraint-based instruction following judgment, (2) pairwise response comparison, and (3) agent-style chain-of-thought evaluation with tool use.

Fig. 5 presents the prompt used for **single-response judging**, where the model must determine whether a given prediction satisfies the explicit constraint provided in the instruction. Fig. 6 illustrates the **pairwise (N-way) comparison prompt**, where the judge model evaluates two or more candidate responses and selects the better one, prioritizing correctness. Although the figure shows the 2-way setup, the same structure naturally extends to 4-way or more candidates in our benchmark. Finally, Fig. 7 shows the **fixed agent-style CoT prefix and suffix prompt** appended to every task when evaluating agentic models (e.g., ARM-Thinker-7B). This prompt unifies the handling of image-based and document-based inputs and enforces structured reasoning, controlled tool usage, and explicit answer formatting.

Together, these three prompts cover the full range of evaluation scenarios in ARMBench-VL, enabling fair comparison across standard judges, reward models, and agentic multimodal evaluators.

Prompts Used in Data Construction. To systematically construct training and evaluation data for ARMBench-VL, we further employ two standardized prompt templates targeting response generation and question rewriting.

Fig. 8 presents the **long response generation template**, where the model is given a question–solution pair and asked to produce one detailed response that faithfully follows the correct solution, together with three linguistically diverse but subtly incorrect responses. The incorrect responses are required to remain plausible while deviating from the ground-truth solution, providing hard negative examples for training and evaluating reward models and judges.

Fig. 9 illustrates the **caption-style question rewriting and response template**, which takes an image, an original question, and its solution as input. The prompt first rewrites the original question into a descriptive "describe ... in detail" style query focused on the region relevant to the solution, and then generates two image-grounded descriptions: one consistent with the solution and one subtly inconsistent in fine-grained details. This enables the construction of visually grounded contrastive pairs for judging nuanced description quality.

Together, these two data-construction prompts support scalable generation of controlled positive and negative examples across both text-centric reasoning tasks and image-centric caption-style tasks in ARMBench-VL.