Extended Abstract

Motivation Vision-based code generation addresses the practical need to convert visual programming interfaces, code screenshots, and handwritten sketches into functional code. While previous approaches using supervised learning or sparse reward signals from unit tests haven't effectively handled real-world visual variations like different fonts and formatting styles, and traditional reinforcement learning methods like PPO require computationally expensive separate value function models, our work applies Group Relative Policy Optimization (GRPO) to achieve efficient vision-to-code generation that maintains computational efficiency while improving functional correctness for practical deployment.

Method We employ GRPO, a novel reinforcement learning approach proposed by DeepSeekMath that samples multiple outputs for each training example and calculates advantages based on relative performance within each group, eliminating the need for a separate value function model used in PPO. To evaluate GRPO's effectiveness, we compare against three baselines: zero-shot performance, supervised fine-tuning (SFT) on successful completions, and SFT followed by Direct Preference Optimization (DPO) using preference pairs constructed from completion quality rankings. Our approach uses the GRPO objective function, where the group loss function incorporates clipped policy ratios and group-wise advantages. We define two primary reward functions: format reward for syntactic validity verification through AST checking, assigning +1 for valid syntax and 0 otherwise and accuracy reward for functional correctness evaluation by executing generated code against HumanEval test cases, assigning +1 for complete test passage and 0 otherwise. The model is prompted to generate responses in a structured format with `<think>` and `<answer>` sections, encouraging step-by-step reasoning before code generation.

Implementation Training utilized DeepSpeed ZeRO Stage 3 and Flash Attention 2 across 8 A100 GPUs for 24 hours with hyperparameters $\beta = 0.1$ (KL penalty) and $\epsilon_1 = 0.2$ (policy clipping). We compare against three baselines: zero-shot Qwen2.5-VL-3B-Instruct, supervised fine-tuning (SFT) on successful completions, and SFT+DPO using preference pairs constructed from completion quality rankings. All methods train on equivalent computational budgets to ensure fair comparison.

Results Our experimental evaluation on the augmented HumanEval dataset demonstrates GRPO's superior effectiveness for vision-based code generation. The baseline Qwen2.5-VL-3B-Instruct model achieves 45% execution success rate in zero-shot vision-to-code generation. SFT improves performance to 50% primarily by addressing response formatting issues, while SFT+DPO reaches 52% with only marginal gains due to sample inefficiency under equivalent computational budgets. GRPO achieves the highest performance at 55% success rate, representing a 10 percentage point improvement over the baseline and demonstrating superior learning efficiency compared to preference-based methods. Training analysis reveals that GRPO's execution reward improved by 83% throughout training (from 0.3 to 0.55), validating the effectiveness of group-based advantage estimation for navigating the sparse reward landscape of functional code generation from visual inputs.

Discussion Our results demonstrate that GRPO effectively addresses key challenges in vision-based code generation, achieving a 22% relative improvement over an untrained baseline through group-based advantage estimation, from 45% to 55%. The superior performance compared to SFT+DPO validates that relative ranking provides more stable learning signals than preference-based methods for sparse reward tasks. While the computational efficiency gains over traditional PPO make GRPO attractive for scaling, the modest absolute performance (55%) highlights the inherent difficulty of vision-to-code generation and suggests future work should explore more sophisticated reward functions and larger model architectures.

Conclusion We successfully demonstrated GRPO's effectiveness for vision-based code generation, establishing group-based reinforcement learning as a computationally efficient alternative to traditional methods. Our work reveals that relative ranking approaches provide superior sample efficiency over preference-based methods in sparse reward environments, offering a practical foundation for scaling multimodal programming tasks to larger models and real-world deployment scenarios.

Teaching Models to Reason about Vision-Based Code Generation using GRPO

Soham V. Govande
Department of Computer Science
Stanford University
govande@stanford.edu

Taeuk Kang
Department of Computer Science
Stanford University
taeuk@stanford.edu

Andrew Shi
Department of Computer Science
Stanford University
acshi@stanford.edu

Abstract

Vision-based code generation, which converts visual programming interfaces and code screenshots into functional code, faces significant challenges in handling real-world visual variations while maintaining computational efficiency. Traditional supervised learning approaches struggle with diverse formatting styles, while reinforcement learning methods like Proximal Policy Optimization (PPO) require computationally expensive separate value function models. We address these limitations by applying Group Relative Policy Optimization (GRPO) to vision-to-code generation, leveraging its ability to calculate advantages based on relative performance within sample groups without requiring separate value functions. Our approach employs Qwen2.5-VL-3B-Instruct as the backbone model, training on augmented HumanEval datasets with diverse visual presentations including varied fonts, syntax highlighting themes, and formatting styles. We define dual reward functions for syntactic validity and functional correctness, using structured prompting with reasoning sections to encourage step-by-step code generation. Experimental results demonstrate GRPO's superior effectiveness, achieving 55% execution success rate compared to 45% for the baseline model, 50% for supervised fine-tuning, and 52% for SFT+DPO. Our work establishes that group-based reinforcement learning can effectively bridge visual understanding and code generation, providing a computationally efficient foundation for multimodal programming tasks while revealing important insights about the superior sample efficiency of relative ranking approaches over preference-based methods in sparse reward environments.

1 Introduction

Vision-based code generation represents a critical challenge with significant practical applications across software development workflows. In real-world scenarios, developers frequently need to convert visual programming interfaces, screenshots of code snippets, handwritten algorithmic sketches, or design mockups into functional implementations. This capability is essential for automating UI-to-code translation in web development, enabling rapid prototyping from visual designs, assisting developers with accessibility needs who rely on screen readers, and facilitating code documentation where images contain algorithmic logic. Additionally, educational platforms could leverage this technology to automatically generate code solutions from visual programming problems or student-drawn flowcharts.

While previous approaches have primarily relied on supervised learning for UI-to-code translation Beltramelli (2017) or utilized sparse reward signals from unit-test pass rates Shojaee et al. (2023), they have not effectively addressed the need for robust vision-to-code generation that can handle visual variations such as different fonts, themes, and formatting styles commonly encountered in real development environments. Traditional reinforcement learning methods like Proximal Policy Optimization (PPO) (Schulman et al., 2017) require separate value function models, increasing computational overhead and memory requirements.

Our work addresses these limitations by applying Group Relative Policy Optimization (GRPO), a recent RL technique with demonstrated success for reasoning-intensive tasks, for fine-tuning VLMs on vision-to-code problems. GRPO is characterized by the removal of the critic model in an actor-critic network. Instead, the model itself acts as the policy and generates a set of multiple potential solutions, and a reward function is employed to evaluate the quality of each response (Shao et al., 2024). The benefit is two-fold: (1) GRPO is more stable during training because there are fewer components to manage in the RL training pipeline and (2) GRPO is more compute-efficient to implement compared to our baselines, which are vanilla foundation models and foundation models fine-tuned using DPO and PPO. Our central hypothesis is that GRPO's mechanism of comparing groups of generated solutions and using relative rankings provides a more stable and effective learning signal compared to standard RL algorithms, particularly given the sparse nature of execution-based rewards and the large, complex action space of code generation from noisy visual inputs. We consider this study successful if GRPO outperforms baselines in this novel setting with statistical significance, yielding more robust models for visual code understanding and generation, and contributing to the field of multimodal reasoning with reinforcement learning. If successful, GRPO's streamlined architecture without a critic network and its computational efficiency make it especially useful for researchers and organizations who have limited computational resources but still need to deploy advanced vision-to-code capabilities for further research or product deployment.

2 Related Work

The problem of translating visual representations into executable code lies at the intersection of vision–language modeling and reinforcement learning (RL). Early systems such as *Pix2Code* (Beltramelli, 2017) and *Sketch2Code* (Microsoft, 2018) demonstrated that user–interface screenshots or hand–drawn wireframes can be mapped to HTML via convolutional encoders and autoregressive decoders, but relied solely on supervised learning. More recent encoder–decoder architectures like *Pix2Struct* (Lee et al., 2023) pretrain vision transformers (ViTs) with text decoders to emit simplified markup, showing the scalability of vision–text pretraining yet still omitting functional execution signals.

Functional-correctness rewards for code generation have been extensively studied in text–only settings. *CodeRL* (Le et al., 2022), PPOCoder (Shojaee et al., 2023), and StepCoder (Dou et al., 2024) treat unit-test pass rates as sparse rewards and fine-tune large language models (LLMs) with actor-critic or PPO, yielding substantial gains on *LeetCode* benchmarks. However, these methods assume the prompt is already in textual form.

Combining vision input with RL fine-tuning was explored by Soselia et al.'s *ViCT*(Soselia et al., 2023), which applies an actor-critic objective to match rendered HTML against a screenshot via a learned visual critic, optimizing layout fidelity rather than semantic correctness.

Traditional policy gradient methods like PPO (Schulman et al. (2017)) have been applied in language model fine-tuning (e.g. InstructGPT and RLHF setups), but they typically require training a separate value function (or critic) to estimate expected reward. This can be computationally expensive, effectively doubling the number of model forward passes and parameters to train (one for the policy, one for the value). GRPO, introduced by *DeepSeekMath* (Shao et al., 2024), is a is a variant of PPO that foregoes the need for a learned critic model by using a novel group-based advantage estimation. The idea is to sample multiple candidate outputs for each input (problem) in a training batch. For example, in our work, the model generates 8 code solutions for the same visual prompt. These outputs are then scored by the reward function (in our case, checks for syntax and correctness). Instead of comparing each output's reward to a value baseline, GRPO computes relative advantages within the group of samples: an output with a higher reward than its peers in the group gets a positive advantage, and vice versa. This relative ranking provides a baseline implicitly (the group's average performance) without an explicit value estimator.

GRPO has been extended to visual math reasoning in *VL-Rethinker* (Wang et al., 2025), where grouped rollouts and relative ranking of solutions stabilize RL on sparse pass/fail feedback.

Our work is the first to fuse these two lines of research: we feed rendered programming problems to a ViT-LLM backbone and apply GRPO with an executable reward. Unlike ViCT, our reward is based on *functional* correctness, and unlike prior GRPO studies, the task involves generating syntactically valid, logically coherent code from noisy visual text. This setting demands robustness to font, theme, and image artifacts while leveraging RL to navigate an expansive action space, which is a gap not addressed by existing literature. Additionally, we apply other recent research in language modeling to vision, such as test-time scaling Muennighoff et al. (2025), where extra test-time compute within a budget has been shown to force models to double-check their answer and improve performance.

Our GRPO trainer and implementation builds off of the VLM-R1 repository Shen et al. (2025).

3 Methods

3.1 Data

3.1.1 Dataset Foundation

We base our training data on the HumanEval dataset, which contains 164 Python programming problems originally designed for evaluating functional correctness of code generation models. Each problem in HumanEval includes a function signature, docstring with problem description, and a comprehensive test suite for automatic evaluation. To adapt this text-based dataset for vision-to-code generation, we convert each programming problem into visual representations by rendering the code as PNG images.

3.1.2 Visual Rendering and Augmentation

The core innovation in our data preparation involves extensive visual augmentation to ensure model robustness across diverse visual presentations of code. We systematically vary multiple visual dimensions to simulate real-world scenarios where developers encounter code in different formats, editors, and display settings.

Our augmentation strategy encompasses four key dimensions.

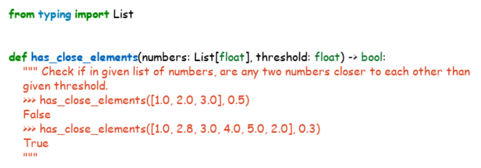
Font Size Variations: We render images using font sizes ranging from 10pt to 16pt (specifically 10, 12, 14, and 16pt) to account for different display preferences and accessibility requirements that developers commonly encounter.

Line Number Display: We generate versions both with and without line numbers enabled, as different code editors and documentation systems may present code in either format.

Syntax Highlighting Themes: We employ five distinct `pygments` themes to cover the spectrum from dark to light color schemes. Two dark themes are included: "monokai" and "github-dark", both of which are commonly used themes across code editors. Three bright themes, "default", "vs", and "colorful", are also used.

Font Family Variations: We utilize four different fonts to ensure robustness across typography choices. Traditional monospace fonts: "Liberation Mono" and "DejaVu Sans Mono" are intended to mimic the standard integrated development environments, code editors, and terminals. We also included fonts that are not commonly used for code to allow the model to learn diverse sets of code formatting. "Lobster" has a bold, condensed script typeface that mimics an informal handwriting and calligraphy style. "Comic Neue" is a reinterpretation of the famous "Comic Sans" font, a sans serif font that has a strong handwritten and rounded feel.

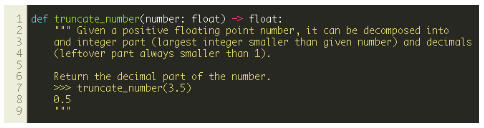
This augmentation strategy generates all possible combinations of these visual parameters, resulting in $4 \times 2 \times 5 \times 4 = 160$ unique visual variations for each of the 164 HumanEval problems, yielding a total dataset of 26,240 training samples.



```python
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

(a) Font size 16, colorful theme, handwritten font, and no line numbers



```python
def truncate_number(number: float) -> float:
    """ Given a positive floating point number, it can be decomposed into
    and integer part (largest integer smaller than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """
```

(b) Font size 14, monokai theme, monospace font, and line numbers

Figure 1: Sample data from the augmentation set

3.1.3 Data Format and Structure

Each training sample follows a structured format designed for multimodal instruction following:

- Image Input: A PNG rendering of the programming problem using pygments' ImageFormatter
- Text Prompt: A standardized instruction "First output the thinking process in <think> </think> tags and then output the final answer in <answer> </answer> tags. Output the final answer as ONLY Python code. Include the existing function heading in the Python code. Example of well-formatted answer: <answer>import math=2add_two(a):return a+b</answer>"
- Ground Truth: The canonical solution from the original HumanEval dataset

We deliberately replace the original HumanEval text prompts with our standardized instruction to force the model to rely entirely on visual input for problem comprehension, ensuring authentic vision-to-code learning rather than memorization of textual patterns. We also provided a one-shot example for the model to clearly observe the format that is required for the evaluation.

3.1.4 Test Case Integration

For reward computation during GRPO training, we append the complete test suites from HumanEval to each training sample. These test cases serve as ground truth for our accuracy reward function, enabling automatic evaluation of functional correctness by executing generated code against the provided unit tests.

3.1.5 Dataset Split

We partition the augmented dataset using an 80/20 split, allocating 80% of samples for training and 20% for validation. This split maintains the augmentation diversity across both sets, ensuring that validation accurately reflects model performance across the full range of visual variations encountered during training.

The resulting dataset provides comprehensive coverage of visual code presentation styles while maintaining the rigorous functional correctness standards of the original HumanEval benchmark, creating an ideal foundation for training robust vision-to-code generation models using reinforcement learning.

3.2 Experimental Setup

3.2.1 Model

We use Qwen2.5-VL-3B-Instruct as our backbone vision-language model, which is parameter-efficient and supports multimodal inputs, namely for videos, images, and natural language. The model combines a vision transformer encoder with a large language model decoder, enabling direct processing of image inputs alongside text prompts. We feed as input our rendered code images through the model's vision encoder and a text prompt. The model then generates code solutions through its autoregressive language decoder.

3.2.2 GRPO Algorithm

Our GRPO implementation follows the algorithm outlined by *DeepSeekMath*, with specific adaptations for vision-to-code generation. For each training example, we generate 8 candidate solutions (`num_generations` $= 8$), allowing sufficient diversity for group-based advantage estimation. Within each group of 8 outputs, we compute relative advantages by comparing each solution's reward against the group mean, eliminating the need for a separate value function. Finally, to train the model, we employ the GRPO objective with clipped probability ratios to ensure stable policy updates.

The GRPO algorithm is defined as:

$$J_{GRPO}(\theta) = \mathbb{E}_{\substack{q \sim P(Q) \\ \{a_i\}_{i=1}^{G} \sim \pi_{\theta_{old}}(A|q)}} \left[L_{group}(\theta, q, \{a_i\}_{i=1}^{G}) \right] - \beta D_{KL}(\pi_\theta || \pi_{ref})$$

where

$$L_{group}(\theta, q, \{a_i\}_{i=1}^{G}) = \sum_{i=1}^{G} \min\left(r_i(\theta) A_i, \text{clip}\left(r_i(\theta), 1 - \epsilon_1, 1 + \epsilon_1\right) A_i\right) \tag{1}$$

and

$$r_i(\theta) = \frac{\pi_\theta(a_i|q)}{\pi_{\theta_{old}}(a_i|q)} \tag{2}$$

3.2.3 SFT Algorithm

Our supervised fine-tuning (SFT) baseline provides a straightforward comparison point by training the model on successful code generations using standard cross-entropy loss. To maintain consistency with our other training approaches, we construct the SFT dataset using the same completion generation methodology employed for GRPO. For each training example, we generate up to 10 candidate completions from the base Qwen2.5-VL-3B-Instruct model and evaluate them using our dual reward function. We select only the completions that achieve successful task execution (passing all syntax validation and test cases) to form our positive training dataset. This filtering ensures that the model learns exclusively from high-quality, functionally correct examples rather than training on a mixture of successful and failed attempts. The SFT training process optimizes the standard language modeling objective, maximizing the likelihood of generating the selected successful completions given the visual programming prompts. This approach establishes a strong supervised learning baseline that demonstrates the effectiveness of learning from curated positive examples, providing a foundation for understanding the additional benefits that reinforcement learning methods like GRPO and DPO can provide through their more sophisticated optimization strategies.

3.2.4 SFT+DPO Algorithm

To ensure fair comparison with GRPO, we implement a DPO training pipeline that maintains equivalent data quality and training dynamics. Our DPO approach follows a two-stage process: supervised fine-tuning followed by preference optimization (SFT + DPO).

For preference dataset construction, we generate candidate completions using the same methodology as GRPO to ensure training equivalence. For each training example, we sample up to 10 completions from the model and evaluate them using our dual reward function. We then apply a decision tree approach to construct preference pairs: if at least one completion achieves successful task execution (passing all test cases), we select the highest-scoring successful completion as the "chosen" response and the lowest-scoring unsuccessful completion as the "rejected" response. Training examples where no successful completion is generated within 10 attempts are excluded from the preference dataset to maintain data quality.

The training process begins with supervised fine-tuning on all successful completions to establish a strong baseline policy. Subsequently, we apply DPO using the constructed preference pairs, optimizing the model to prefer functionally correct code generations over incorrect ones. This SFT+DPO pipeline ensures that our DPO baseline benefits from both the positive examples used in supervised learning and the contrastive signal provided by preference optimization, creating a robust comparison point for evaluating GRPO's effectiveness.

The DPO algorithm is defined as:

$$J_{\text{DPO}}(\theta) = \mathbb{E}_{\substack{q\sim P(Q)\\(a_w,a_l)\sim D}}\Big[L_{\text{pair}}(\theta,\, q,\, a_w,\, a_l)\Big] - \beta\, D_{KL}\big(\pi_\theta \,\|\, \pi_{\text{ref}}\big)$$

$$L_{\text{pair}}(\theta,\, q,\, a_w,\, a_l) = \log \sigma\Big(\beta\big(\log \pi_\theta(a_w \,|\, q) - \log \pi_\theta(a_l \,|\, q)\big)\Big) \tag{3}$$

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad , \quad s(a \mid q) = \log \frac{\pi_\theta(a \mid q)}{\pi_{\text{ref}}(a \mid q)} \tag{4}$$

3.2.5 Hyperparameters

We configure our training with method-specific hyperparameters to ensure fair comparison across all approaches.

We configure our GRPO training with the following key hyperparameters. First, the KL divergence weight (β), which controls the penalty for deviating from the initial policy is set to 0.1. The policy ratio clipping range (ϵ_1), which limits the policy ratio to prevent destructive updates is set to 0.2. The effective minibatch size is set to 16, using 8 generations per example and gradient accumulation of 2. We use a learning rate of 2e-5 with cosine annealing LR schedule. We train for 2 epochs over the full augmented dataset, the maximum input prompt length to 1024 tokens and using a temperature of 0.7 for each generation.

For DPO, the preference loss coefficient (β) is set to 0.1, matching our GRPO setting. We employ sigmoid loss for preference optimization. Training uses a learning rate of 5e-6 with cosine scheduling, batch size of 1 per device with gradient accumulation of 8 steps, and 3 training epochs. The maximum sequence length is set to 2048 tokens.

For SFT, the learning rate is set to 1e-4 with cosine scheduling, batch size of 1 per device with gradient accumulation of 8 steps, and 3 training epochs. The maximum sequence length matches DPO at 2048 tokens.

3.2.6 Reward Functions

We implement two reward functions for GRPO. First, we define the syntax reward (R_{syntax}) as a binary reward based on Python AST validation that returns +1 if the generated code parses successfully and 0 for syntax errors or ill-defined code. This reward ensures the model learns to generate syntactically valid Python code. Second, we define the execution reward ($R_{\text{execution}}$) as a binary reward based on functional correctness that returns +1 if the code passes all HumanEval test cases and 0 if any test case fails or execution errors occur. We execute the generated code in a sandboxed subprocess with a 5-second timeout.

Then, the we assign equal weight to the two reward functions, so the total reward for each generation is computed as $R_{\text{total}} = R_{\text{syntax}} + R_{\text{execution}}$.

3.2.7 Evaluation

We evaluate model performance on the held-out validation set of the augmented dataset, ensuring representation across all augmentation dimensions. During evaluation, we generate a single solution per problem using greedy decoding (`temperature` $= 0$) and measure both syntax validity and execution success rates separately. We track performance across different augmentation categories to assess robustness.

3.2.8 Baselines

To evaluate the effectiveness of GRPO on the vision-to-code task, we compare our model's performance to zero-shot Qwen2.5-VL-3B-Instruct, a model trained using the SFT algorithm from Section 3.2.3, and a model trained using the SFT+DPO algorithm from Section 3.2.4 on the augmented HumanEval dataset. All methods are trained on an equivalent number of tokens to ensure computational fairness. We adopt hyperparameters from LlamaFactory (Zheng et al., 2024) and maintain consistent training infrastructure across all approaches. During evaluation, each method uses identical data augmentation and protocols on the same held-out test set.

We specifically note that we do not use PPO as a baseline. It is considerably more difficult to fairly compare PPO to our model because it requires a well-defined value model, which we do not have for the HumanEval dataset.

3.2.9 Training Hardware

We used 8 NVIDIA A100 GPUs in a distributed training configuration and DeepSpeed ZeRO Stage 3 for model sharding across GPUs. We used the Flash Attention 2 kernel for efficient attention computation and bf16 mixed precision training for reduced memory footprint and faster computation. Training took approximately 24 hours for each model.

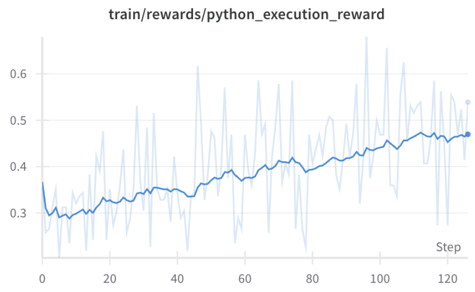
4 Results and Discussion

4.1 Training

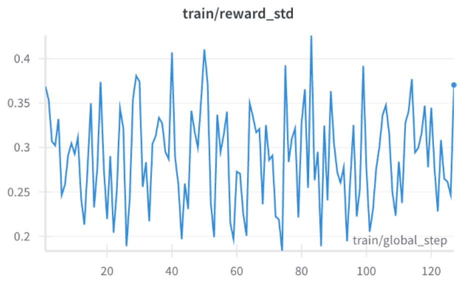
4.1.1 GRPO

Figure 2 shows the training reward progression and standard deviation of our GRPO-based vision-to-code generation model over 120 training steps. Out of the two reward function metrics, we noticed that the execution reward exhibited a consistent upward trajectory throughout training. Starting from approximately 0.3 at the beginning of training, the execution reward progressively increased, reaching around 0.55 towards the end of the training period on the smoothed reward curve. This shows an 83% improvement in execution success rate, demonstrating the effectiveness of GRPO in learning to generate functionally correct code from visual inputs. The standard deviation of the rewards showed fluctuations in the range of 0.2 to 0.4 throughout training, indicating variability in the model's performance across different visual augmentations and problem difficulties. This variability is reasonable given the diverse nature of our augmented dataset, which includes variations in fonts, themes, and formatting styles.

Compared to our initial experiments on other vision-language tasks like COCO image captioning, HumanEval proved to be a considerably more challenging benchmark. The difficulty stems from the precise nature of code generation, where even minor errors in syntax, logic, or response formatting can cause complete failure, unlike more forgiving tasks where partial correctness still may result in positive reward.



(a) Execution reward over steps



(b) Standard deviation of reward over steps

Figure 2: Training progression metrics for GRPO-based vision-to-code.

4.1.2 SFT Baseline

For our first baseline, we train a model using the SFT algorithm described in Section 3.2.3. The supervised fine-tuning baseline demonstrates reasonable convergence characteristics as shown in Figure 3. The model begins training with a loss of approximately 0.45, exhibiting rapid initial improvement within the first 10 training steps. Then, we see a consistent decay pattern, stabilizing around 0.275 after approximately 50 training steps. This represents around a 39% reduction in loss, indicating effective learning of the mapping between visual code representations and their corresponding functional implementations.

The training exhibits relatively low variance after the initial convergence phase, with the smoothed loss curve showing minimal fluctuations between steps 30-55. This suggests that the curated dataset of successful completions provides a consistent learning signal, which in turn allows the model to learn consistently from positive examples without the exploration-exploitation challenges inherent in RL-based approaches.

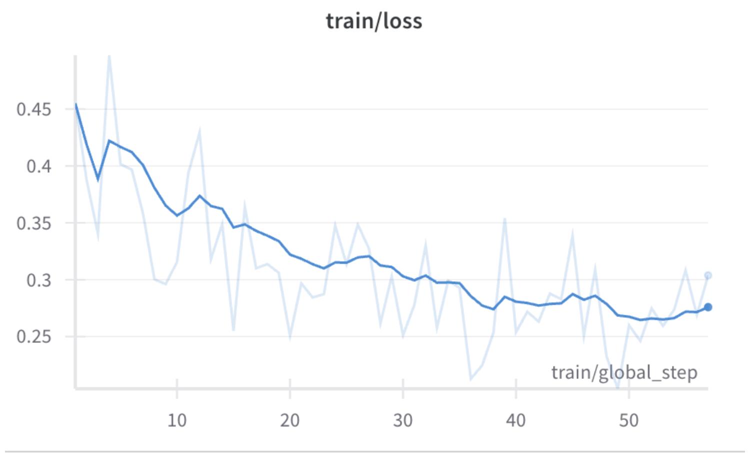


Figure 3: Training loss progression for SFT baseline on vision-to-code generation.

4.1.3 SFT+DPO Baseline

For our second baseline, we train a model using the SFT+DPO algorithm described in Section 3.2.4. We first use a copy of the SFT baseline model and then fine-tune it using DPO. The chosen rewards metric, which is shown in Figure 4, represents the model's implicit reward for preferred (functionally correct) completions over rejected ones and demonstrates a gradual improvement throughout training. Starting from approximately -0.03, the reward progressively increases to around +0.01 by the end of training, representing a shift from initially preferring incorrect completions to correctly distinguishing between functional and non-functional code.

The training progression exhibits higher variance compared to SFT, with notable spikes reaching up to 0.05 at certain points. From this, we see that the contrastive nature of preference learning, where the model must simultaneously learn to prefer correct solutions while actively rejecting incorrect ones, may lead to slightly more unstable learning patterns. However, the overall positive trend indicates that the model successfully learns to discriminate between high-quality and low-quality code generations. The positive final reward value suggests that the preference optimization successfully enhances the model's ability to generate functionally correct code beyond what pure supervised fine-tuning achieves.

4.2 Evaluation

This baseline performance shows that while the model possesses foundational visual code understanding, task-specific fine-tuning is necessary for practical deployment.

We evaluate all trained models on the held-out test set of our augmented HumanEval dataset, generating completions at temperature 0.7. Figure 5 shows the overall execution success rates across all visual augmentation variations.

The baseline Qwen2.5-VL-3B-Instruct model achieves a 45% success rate using zero-shot vision-to-code generation, demonstrating reasonable out-of-the-box capability but leaving room for improvement. The supervised fine-tuning model yields noticeable improvement, reaching 50% success rate. By analyzing the generated outputs for SFT, we find that the primary contribution of SFT lies in

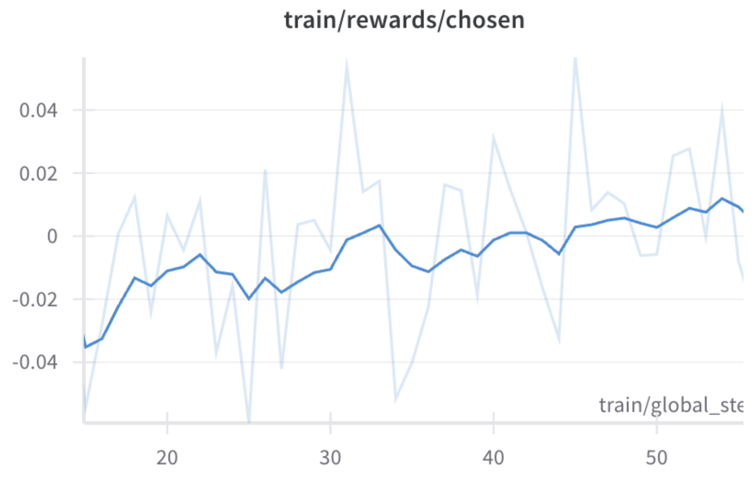


Figure 4: Chosen rewards progression during DPO training phase.

teaching proper response formatting. The baseline model frequently included unnecessary function headers in its responses or failed to follow the required output structure, leading to parsing failures even when the core functional logic was correct. By training exclusively on well-formatted successful completions, SFT effectively addresses these formatting issues, resulting in a 5-percent improvement purely from structural corrections. The SFT+DPO approach achieves 52% success rate, representing a marginal 2 percentage point improvement over SFT alone. GRPO emerges as the most effective approach, achieving 55% success rate and representing a 10 percentage point improvement over the baseline.

The difference between DPO and GRPO reveals a limitation of DPO on this downstream task in that it is less sample-efficient than GRPO when trained under equivalent computational budgets. As evidenced by the minimal increase in the chosen reward metric during training in Figure 4, DPO struggles to effectively leverage the preference pairs for substantial performance improvements within our training setup. The contrastive learning signal, while theoretically sound, may require considerably more training iterations to achieve meaningful gains in this vision-to-code task. This superior performance of GRPO validates our hypothesis that group-based advantage estimation provides more stable and informative learning signals for vision-based code generation. By comparing multiple generated solutions within each batch, GRPO efficiently explores the solution space and learns from relative performance differences, leading to more robust policy improvements despite the sparse reward signal inherent in code execution tasks.

The performance ordering (Baseline < SFT < SFT+DPO < GRPO) across our evaluation demonstrates that while supervised learning provides valuable formatting improvements, reinforcement learning approaches are necessary to achieve substantial gains in functional correctness. However, the choice of RL algorithm matters significantly: GRPO's group-based approach is more suitable for the high-variance, sparse-reward nature of vision-to-code generation compared to preference-based methods like DPO under computational constraints.

Sample generations for the vanilla baseline model, SFT model, SFT+DPO model, and GRPO model can be found in the Appendix.

4.3 Discussion

4.3.1 Reward Function Analysis

For GRPO, our empirical results revealed differing effectiveness of on our two reward metrics. While the execution reward showed substantial and consistent improvement throughout training, the syntax reward remained relatively stagnant for most of the training period. This divergence provides insights into the nature of each reward component and the challenges in vision-to-code generation. The stagnation in syntax reward may be due to sensitivity to specific formatting preferences rather than

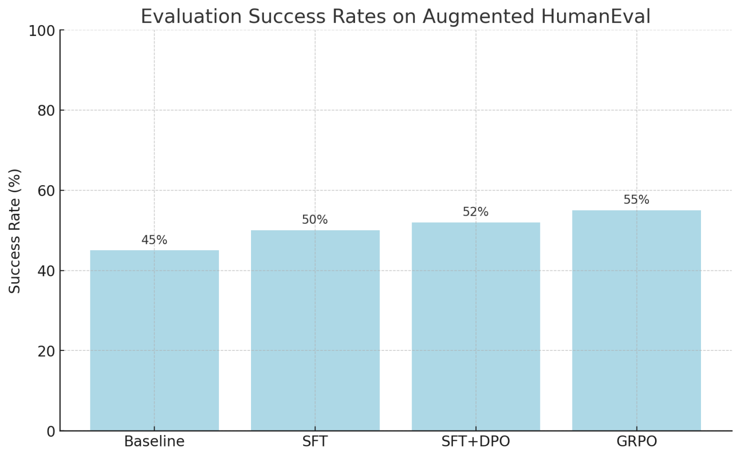


Figure 5: Performance of GRPO and baseline models on augmented test set.

general syntactic validity. Our training setup required models to learn nuanced formatting constraints, such as omitting function headers that were already visible in the image prompt. This requirement conflicts with typical pre-training patterns where models generate complete function definitions, suggesting that overwriting these deeply learned patterns requires extended training iterations. Once models successfully adapted to these formatting conventions, the syntax reward consistently reached 1.0, indicating that syntactic validity itself was not a limiting factor given Qwen2.5-VL's strong code generation foundation.

4.3.2 Visual Representation Limitations

Our analysis revealed that models occasionally fail to generate accurate code for edge cases within functions, particularly when handling complex conditional logic or boundary conditions. This limitation appears to stem from the lossy nature of visual text representation. When code is rendered as an image and then processed through vision encoders, subtle but critical details in the problem specification may be lost or distorted. The image-to-text reconstruction process inherently introduces noise that can obscure nuanced requirements, leading to functional incorrectness even when the general problem structure is understood.

This finding highlights a fundamental challenge in vision-based code generation: maintaining semantic precision through visual encoding. Unlike direct text processing where every character is preserved exactly, visual representations must balance between capturing the overall structure and preserving fine-grained details that are often crucial for correct code implementation.

4.3.3 Impact of Data Augmentation

We also ran ablation studies to demonstarte the importance of visual augmentation for generalization. When trained without our comprehensive augmentation strategy, models showed severe overfitting despite continued improvement in training rewards. Specifically, models trained on the base 164 HumanEval problems without visual variations plateaued at approximately 50% evaluation accuracy, compared to 55% with full augmentation.

This difference underscores that training robust vision-to-code generation requires exposure to diverse visual presentations during training. The model must learn to extract semantic content invariant to superficial visual changes such as font choices, color schemes, and formatting styles. Without this augmentation, models memorize specific visual patterns rather than learning the underlying mapping from visual code representations to functional implementations. Our results validate that comprehensive augmentation across fonts, themes, and formatting styles is essential for developing models that can handle the visual diversity encountered in real-world code screenshots and documentation.

5 Conclusion

5.1 Contributions

Our work successfully demonstrates that Group Relative Policy Optimization (GRPO) provides an effective and computationally efficient approach for vision-based code generation, establishing a new benchmark for multimodal programming tasks. Through comprehensive experimentation on an augmented HumanEval dataset with diverse visual presentations, we achieved a 55% execution success rate—representing a 22% relative improvement over the baseline Qwen2.5-VL-3B-Instruct model and outperforming both supervised fine-tuning (50%) and preference-based methods (52%).

This research makes several contribution to the intersection of computer vision and code generation. We applied GRPO to vision-to-code generation to demonstrate that group-based advantage estimation can effectively navigate the sparse reward landscape inherent in functional code generation from visual inputs. Second, we developed a comprehensive visual augmentation strategy that ensures model robustness across diverse real-world code presentations, including variations in fonts, syntax highlighting themes, and formatting styles. Third, our dual reward function design, combining syntactic validity through AST checking with functional correctness through test case execution, provides a robust evaluation framework for vision-based programming tasks.

5.2 Future Work

While our results are promising, the absolute performance of 55% execution success rate highlights the inherent difficulty of vision-to-code generation and suggests several avenues for future research. The computational efficiency gains of GRPO over traditional PPO make it particularly attractive for scaling to larger models and more complex programming tasks.

Following the DeepSeek-R1 approach (DeepSeek-AI et al. (2025)), implementing SFT before GRPO could provide a stronger initialization for the policy model. This staged approach might improve both convergence speed and final performance by first establishing basic vision-to-code capabilities through supervised learning before applying reinforcement learning optimization.

Developing more sophisticated reward functions that capture code quality metrics beyond functional correctness, such as efficiency, readability, style adherence, or security considerations, could produce more practically useful generated code.

Investigating GRPO's effectiveness with larger vision-language models (7B, 13B+ parameters) and more complex programming tasks would establish scaling laws and computational trade-offs for production deployment.

Extending beyond Python to languages like JavaScript, Java, C++, or domain-specific languages (SQL, HTML/CSS) would demonstrate the generalizability of GRPO for vision-based code generation. Each language presents unique syntactic and semantic challenges that could reveal the robustness of the approach.

Moving from rendered code to authentic handwritten sketches, whiteboard algorithms, or mobile app screenshots would bridge the gap to practical deployment scenarios. This includes handling real-world noise, perspective distortion, and informal notation styles that developers actually encounter.

6 Team Contributions

- **Soham Govande:** Running all experiments and devising reward functions
- **Taeuk Kang:** Creating datasets, experimenting with augmentation hyperpameters, and writing the paper
- **Andrew Shi:** Ideating reward functions, training runs for GRPO, and writing the final paper

Changes from Proposal Several significant modifications were made during implementation. We switched from the originally planned HuggingFaceH4/code_evaluation_prompts dataset to the HumanEval dataset, which provided more rigorous functional correctness evaluation through comprehensive test suites. Our model selection evolved from the proposed Qwen 2.5 500M to Qwen2.5-VL-3B-Instruct, a significantly larger model with native vision capabilities that better suited our multimodal requirements. The baseline comparison strategy was refined to include SFT and exclude PPO since we do not have a value model, while maintaining DPO as planned. We enhanced our reward function design from a simple execution-based metric to a dual reward system incorporating both syntactic validity through AST checking and functional correctness through test case execution. Additionally, we implemented structured prompting with reasoning sections to encourage step-by-step code generation, and our data augmentation strategy shifted from including Gaussian noise artifacts to focusing on comprehensive font, theme, and formatting variations that better reflect real-world visual code presentations.

References

Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. arXiv:1705.07962 [cs.LG] https://arxiv.org/abs/1705.07962

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback. arXiv:2402.01391 [cs.SE] https://arxiv.org/abs/2402.01391

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG] https://arxiv.org/abs/2207.01780

Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2023. Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding. arXiv:2210.03347 [cs.CL] https://arxiv.org/abs/2210.03347

Microsoft. 2018. Sketch2Code. https://github.com/Microsoft/ailab/tree/master/Sketch2Code. GitHub repository, last updated 30 May 2019.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. arXiv:2501.19393 [cs.CL] https://arxiv.org/abs/2501.19393

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the

Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300

Haozhan Shen, Peng Liu, Jingcheng Li, Chunxin Fang, Yibo Ma, Jiajia Liao, Qiaoli Shen, Zilun Zhang, Kangjia Zhao, Qianqian Zhang, Ruochen Xu, and Tiancheng Zhao. 2025. Vlm-r1: A stable and generalizable r1-style large vision-language model. *arXiv preprint arXiv:2504.07615* (2025).

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based Code Generation using Deep Reinforcement Learning. arXiv:2301.13816 [cs.LG] https://arxiv.org/abs/2301.13816

Davit Soselia, Khalid Saifullah, and Tianyi Zhou. 2023. Learning UI-to-Code Reverse Generator Using Visual Critic Without Rendering. arXiv:2305.14637 [cs.CV] https://arxiv.org/abs/2305.14637

Haozhe Wang, Chao Qu, Zuming Huang, Wei Chu, Fangzhen Lin, and Wenhu Chen. 2025. VL-Rethinker: Incentivizing Self-Reflection of Vision-Language Models with Reinforcement Learning. arXiv:2504.08837 [cs.LG] https://arxiv.org/abs/2504.08837

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. arXiv:2403.13372 [cs.CL] https://arxiv.org/abs/2403.13372

A Snippets of generated code

A.1 Baseline (Vanilla)

```
<answer>
def truncate_number(number: float) -> float:
"""""" Given a positive floating point number, it can be decomposed into
and integer part (largest integer smaller than given number) and decimals
(leftover part always smaller than 1).
Return the decimal part of the number.
>>> truncate_number(3.5)
0.5
"""""
    return number - int(number)
</answer>
```

A.2 SFT

```
<think>
To truncate a floating-point number, we need to separate the integer and decimal parts.
In Python, we can use the mod operator to get the decimal part. The integer part can be
obtained by subtracting the decimal part from the original number. Here’s the step-by-step
process:

Subtract the integer part from the original number to get the decimal part.
Use the mod operator (%) to get the decimal part.
Return the decimal part.
Given the function definition:
‘‘‘python
def truncate_number(number: float) -> float:
    """""" Given a positive floating point number, it can be decomposed into
    and integer part (largest integer smaller than given number) and decimals
    (leftover part always smaller than 1).

    Return the decimal part of the number.
    >>> truncate_number(3.5)
    0.5
    """""
‘‘‘
The decimal part of the number 3.5 is 0.5.
</think>
<answer>
def truncate_number(number: float) -> float:
    return number - int(number)
</answer>
```

A.3 SFT+DPO

```
<think>
1. The goal is to isolate the decimal part of a positive float.
2. The decimal part is what remains after removing the whole number (integer) portion.
3. I can obtain the integer part by casting the float to int, which truncates the decimal.
4. Subtracting the integer part from the original number leaves only the decimals.
5. Ensure this works for any positive float, e.g., 3.75 - 3 = 0.75.
6. Consider edge cases: input is already an integer (result should be 0).
</think>
<answer>
return number - int(number)
</answer>
```

A.4 GRPO

```
<think>
1. In Python, the ‘%‘ operator works with floats, returning the remainder of division.
2. For a positive ‘number‘, ‘number % 1‘ computes ‘number - floor(number)‘, which is
exactly the fractional part.
3. This is more concise than ‘number - int(number)‘ and directly conveys intent.
4. Be aware of minor floating-point precision artifacts inherent to IEEE 754.
5. Therefore, use ‘number % 1‘ to get the decimal part.
</think>
<answer>
return number % 1
</answer>
```