

Imperfect Visual Verification for Code Edition : A Case Study on TikZ

Charly Reux, Mathieu Acher, Djamel Eddine Khelladi, Clément Quinton, Olivier Barais

▶ **To cite this version:**

Charly Reux, Mathieu Acher, Djamel Eddine Khelladi, Clément Quinton, Olivier Barais. Imperfect Visual Verification for Code Edition : A Case Study on TikZ. 2026. ⟨hal-05582911⟩

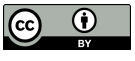
HAL Id: hal-05582911

https://hal.science/hal-05582911v1

Preprint submitted on 7 Apr 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Imperfect Visual Verification for Code Edition : A case study on TikZ

Charly Reux
Univ Rennes, Inria, IRISA, INSA
Rennes, France
charly.reux@inria.fr

Mathieu Acher
Univ Rennes, Inria, CNRS, IUF, IRISA
Rennes, France
mathieu.acher@irisa.fr

Djamel Eddine Khelladi
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
djamel-eddine.khelladi@irisa.fr

Clément Quinton
Univ. Lille, CNRS, Inria
Lille, France
clement.quinton@univ-lille.fr

Olivier Barais
Univ. Rennes, IRISA, Inria
Rennes, France
olivier.barais@irisa.fr

ABSTRACT

LLMs have significantly advanced code generation, enabling the synthesis of functional programs. While recent systems achieve strong performance on many coding benchmarks, tasks involving programs such as TikZ that generate visual artifacts remain challenging, in particular on visual code customization. Unlike generation from scratch, customization requires localized, semantics-preserving edits: the model must locate relevant code, modify it according to the instruction, and preserve the remaining structure and rendering. *Approaches based on post-hoc iterative refinement/correction* where a verifier provides feedback to guide corrections, have shown promise. However, in the case of programs with a visual outcome such as in TikZ, where correctness is harder or likely impossible to formalize and evaluate automatically, deterministic verifiers do not exist. Hence, developers can only rely on imperfect verifiers.

In this paper, we conduct an empirical study to answer: *to what extent can iterative refinement remain effective when the verifier itself is unreliable?* We use TikZ as a focused case study that isolates the core difficulties of the problem (weak code structure, fine-grained visual semantics, and difficult feature localization) in a controlled and challenging setting. We define visual code customization as an iterative editing problem with an imperfect oracle, and introduce a framework for analyzing such iterative refinements. We conduct a large-scale study and evaluate multiple LLM-based and tool-augmented visual verifiers within iterative refinement pipelines, and perform extensive manual annotation of refinement trajectories to assess verifier behavior and feedback quality.

Our findings show that even imperfect verifiers can determine with moderate accuracy whether visual instructions are applied to code, achieving F1-scores up to 0.815. Feedback improves iterative refinement, especially for weaker models, adding 11–20 perfect customizations for Qwen3-vl-30b-a3b-Instruct, while stronger models like Gemini-3 gain fewer improvements (+5) but benefit more from accurate verification that prevents premature acceptance. Feedback is effective only when it precisely identifies image issues, provides actionable guidance, addresses all relevant problems, and remains grounded in the original instruction.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Software prototyping**; **Visual languages**; • **Computing methodologies → Artificial intelligence**.

KEYWORDS

AI-based code generation, Large Language Models (LLMs), Code customization, Visual intent alignment, Evaluation

1 INTRODUCTION

Large language models (LLMs) have significantly advanced automated code generation, enabling the synthesis of functional programs from natural language descriptions [43]. While recent systems achieve strong performance on many coding benchmarks, a class of tasks remains challenging: programs that generate *visual artifacts* [23, 41, 45], such as TikZ, SVG, or interactive web visualizations (e.g., D3.js). In these settings, correctness is not solely determined by syntactic or functional properties of the code, but by high-level visual properties such as geometric structure, spatial relationships, and layout semantics. However, such properties are difficult to infer or verify from code alone.

A particularly challenging instance of this problem is *visual code customization*, where an existing program must be modified according to a high-level visual instruction (e.g., *"reduce the size of the green box on the left and put it parallel to the yellow one"*). Unlike generation from scratch, this customization requires localized, semantics-preserving edits: the model must identify where in the code a visual element is defined, modify it according to the instruction, and preserve the remaining structure and rendering. This problem arises in a wide range of domains, including web interfaces, SVG graphics, scientific diagrams, games, and 3D modeling. Although prior work on multimodal code generation (e.g., image-to-code or text-to-diagram synthesis) has shown promising results [12, 32, 40], recent studies indicate that LLMs still struggle to reliably perform such targeted visual modifications [31]. Indeed, Reux et al. showed that GPT-4o successfully completed only 26% of TikZ customization tasks.

One promising paradigm for improving generation quality is *post-hoc iterative refinement/correction* [13, 29], where an LLM proposes candidate solutions and a verifier provides feedback to guide subsequent improvements (see fig. 2). This paradigm underlies a growing class of systems, including coding agents and Reinforcement Learning with Verifiable Rewards (RLVR), where feedback signals are typically deterministic, boolean, and highly reliable. Examples include unit test execution for code generation, constraint checking for structured outputs, or formal proof checkers in theorem-proving environments. In such settings, refinement is effective because the verifier acts as a near-perfect oracle.

However, many real-world generation tasks lack such deterministic verification signals, making it challenging to design reliable refinement loops. When outputs are primarily visual, correctness is difficult (if not impossible) to formalize and evaluate automatically. For visual code such as TikZ, HTML/CSS, SVG, or SCAD, compilation or execution only ensures syntactic validity, not semantic correctness of the rendered image. As a result, refinement must rely on *imperfect verifiers*, often implemented as LLM- or VLM-based evaluators assessing visual similarity or instruction compliance. An imperfect verifier can be characterized along three dimensions: *(i)* false positives, where incorrect outputs are mistakenly accepted, *(ii)* false negatives, where correct or partially correct outputs are rejected, and *(iii)* feedback quality, defined as the extent to which feedback is specific, actionable, and aligned with the underlying error. Such imperfections make the refinement process challenging: false positives lead to premature termination, while poor feedback prevents meaningful improvement. This motivates the core research question of this work: *To what extent can iterative refinement remain effective when the verifier itself is unreliable?*

This paper investigates this question through a focused and controlled case study on TikZ. As detailed in Section 2, TikZ amplifies three core software engineering challenges while controlling away confounding factors such as multi-file architectures or framework-specific tooling: *(i)* weakly structured and highly compositional code, which complicates visual element localization; *(ii)* high sensitivity to small edits, where minor code changes produce subtle but critical visual differences; and *(iii)* abstract geometric constructs, which are difficult for current vision-language models to interpret reliably.

To address this problem, we frame visual code customization as an iterative editing process under an imperfect oracle, and we introduce a framework for analyzing such iterative refinements. Building on this framework, we conduct a large-scale empirical study using a refined version of the VTikZ dataset [31]. In particular, we evaluate multiple LLM-based and tool-augmented visual verifiers within iterative refinement pipelines, and we perform extensive manual annotation of refinement trajectories to assess verifier behavior and feedback quality. We selected Qwen-vl-30b-a3b and Gemini-3-flash as the backbone models for our evaluation.

The results yield three key insights. First, imperfect verifiers can correctly assess whether visual instructions are applied to code, achieving F1-scores up to 0.815. Second, feedback significantly improves refinement for weaker models, yielding gains of +11 to +20 perfect customizations for Qwen3-vl-30b-a3b-Instruct, whereas stronger models (Gemini-3) benefit less from feedback and more from accurate classification that prevents premature acceptance. Third, feedback is effective only when it is actionable: it must precisely identify visual issues, provide concrete guidance, cover all relevant problems, and remain grounded in the original instruction.

To summarize, this work makes the following contributions:

- We frame the general problem of iterative code customization with imperfect visual verifiers, TikZ being one challenging instance.
- We construct a large-scale dataset of manually annotated customization within iterative refinement trajectories.
- We provide the first empirical analysis of feedback on LLM refinement performance in the customization scenario.

```
\fill [BlueGrey700] (6,12)
  .. controls (5,11) and (5,8)  .. (6,7) .. controls (7,8)  and (7,9)  .. (8,10)
  .. controls (8,11) and (7,12) .. (6,12)-- cycle;
\fill [BlueGrey500] (0,20)
  .. controls (0,19)  and (0,18)  .. (1,17)
  .. controls (3,16)  and (4,16)  .. (4,15)
  .. controls (4,14)  and (2,15)  .. (1,15)
  .. controls (2,13)  and (3,12)  .. (5,10)
  .. controls (7,8)   and (11,6)  .. (14,5)
  .. controls (14,3)  and (14,1)  .. (15,0)
  .. controls (15,2)  and (15,3)  .. (16,4)
  .. controls (17,5)  and (18,6)  .. (20,6)
  .. controls (19,7)  and (16,7)  .. (15,6)
  .. controls (14,10) and (11,15) .. (9,17)
  .. controls (7,19)  and (3,20)  .. (0,20) -- cycle;
\fill [BlueGrey100] (0,20)
  .. controls (0,19) and (0,18) .. (1,17) .. controls (3,16) and (4,16) .. (4,15)
  .. controls (4,14) and (2,15) .. (1,15) .. controls (2,13) and (3,12) .. (5,10)
  .. controls (7,8)  and (11,6) .. (14,5) .. controls (13,8) and (7,8)  .. (6,12)
  .. controls (5,16) and (2,19) .. (0,20) -- cycle;
\fill [BlueGrey500] (3,13)
  .. controls (2,12) and (2,9)  .. (3,8) .. controls (4,9)  and (4,10) .. (5,11)
  .. controls (5,12) and (4,13) .. (3,13) -- cycle;
```

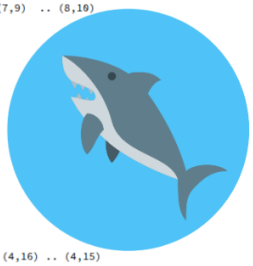


Figure 1: Excerpt of a TikZ code.

- We derive a taxonomy of feedback behaviors and failure modes of LLM-based visual verifiers through systematic manual analysis of refinement trajectories.
- We provide the dataset and a replication package of our experiments.

2 BACKGROUND AND MOTIVATION

This section introduces the necessary background and further motivates our work.

Problem formalization. We formalize iterative code customization as the problem of refining a code artifact c through a sequence of updates guided by an imperfect verifier. At each iteration, the generator proposes edits to the code, the verifier provides potentially noisy feedback and a score, and the process repeats until a stopping criterion or refinement budget is reached. The goal is to produce a final code artifact that best matches the intended visual outcome.

Let c_0 denote the initial code (a diagram). At iteration t, the generator produces the next code based on the current code, the verifier feedback, and the score:

$$c_{t+1} = G(c_t, f_t),$$

where $f_t = V(c_t)$ represents the feedback and score provided by the verifier.

For a complete refinement loop:

$$c_0 \xrightarrow{G} c_1 \xrightarrow{V} f_1 \xrightarrow{G} \dots \xrightarrow{G} c_n.$$

The quality of the final code c_n is measured with respect to the intended visual goal v^*:

$$Q(c_n, v^*),$$

The iterative refinement protocol we formalize in section 4.2 provides a concrete operational instance of this process, enabling empirical evaluation with real LLM generators and verifiers.

TikZ as a testbed. The following work focuses on the *TikZ* language, a LaTeXPackage that allows the creation of graphic elements, mainly for scientific diagrams, though it can also be used for drawings and cartoon characters. TikZ programs define visual elements such as shapes, edges, and layouts through code, making them a representative example of programs that generate visual artifacts.

We use TikZ as a *focused and controlled case study* for iterative refinement with imperfect verifiers. Unlike broader settings such

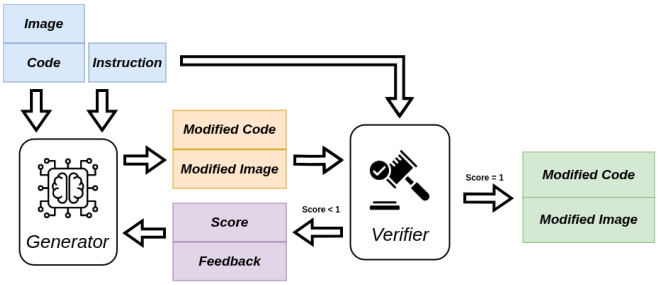


Figure 2: Post-hoc Iterative Refinement.

as web development, which involve multi-file structures, modular architectures, and mature visual tooling, TikZ controls away these confounding factors while *amplifying* the challenges central to our study: the code is weakly structured and lacks modular boundaries, making feature localization difficult; visual outputs require fine-grained geometric reasoning; and small code changes can produce subtle yet critical visual differences. The code is often sufficiently opaque that even advanced LLMs fail to interpret what it represents. For instance, given the code of Figure 1, GPT-4 misinterprets it as a "stylized abstract landscape" and Gemini-3 as a seagull, when it in fact represents a shark. TikZ thus provides an idealized yet practically relevant setting that isolates the mechanism of interest (imperfect visual verification and fine-grained code localization), providing analytical leverage for studying verifier–generator dynamics. Customizing TikZ code from high-level visual instructions requires translating visual-level requests into localized code changes while preserving the overall diagram structure.

Iterative refinement approach. To address this challenge, we explore *post-hoc iterative refinement*, illustrated in Figure 2. In this paradigm, a `Generator` attempts to modify the code according to the visual instruction, while a `Verifier` evaluates whether the instruction has been correctly applied, given the code and the rendered image. The verifier returns a score and textual feedback, which can be used to iteratively refine the generated solution.

For example, consider a simple customization such as removing a specific feature in the TikZ drawing in Figure 1. A user instruction might ask that the teeth of the shark be removed. The Generator may mistakenly identify the teeth and instead remove the fins. The verifier can detect this discrepancy and guide the generator toward the correct edit in subsequent iterations. Alternatively, the verifier may only deteriorate the code customization if incorrect feedback is further returned.

Challenges and research gap. While iterative refinement has shown promising results in domains such as code optimization, readability improvement, constrained generation, and mathematical reasoning [26]. To the best of our knowledge, its effectiveness in visual customization settings remains unclear. In our case, verifiers are inherently imperfect, as visual instructions may admit multiple valid solutions and cannot be evaluated through deterministic tests. As a result, it is uncertain whether LLM-based verifiers can reliably detect whether a visual instruction has been correctly applied, and whether their feedback can effectively guide refinement. Our work fills this gap.

3 STUDIED VERIFIERS

The verifiers we evaluate fall into two categories: (1) *basic verifiers*, which are variations of multimodal LLM-based verifiers where either the input modality is changed (*Text*, *Visual*, or *Text+Visual*) or the prompting technique is modified (*Property*); and (2) *complex verifiers*, which follow a more elaborate code-based approach. All system prompts and settings are available in the Git repository[1] Each verifier outputs a numeric score and textual feedback. For consistency, we map all scores to the interval $[0, 1]$. For most verifiers, during prompting, we ask models to produce an integer score from 0 to 100, then divide by 100, as manual testing showed that models are more likely to assign a perfect score when the range is $[0, 1]$ rather than $[0, 100]$. We also define specific scoring thresholds (0–49, 50–79, 80–99, 100), depending on the extent to which the customization is applied.

3.1 Simple LLM-based Verifier

These verifiers differ from each other in the modality provided to the multimodal model: 1) *Text*, where the initial and customized code are given; 2) *Visual*, where the initial and customized images are fed to the model; and 3) *Text+Visual*, where both modalities are provided (images and codes).

3.2 Property-based Verifier

This verifier follows a two-step procedure. First, given the instruction and the original image, it generates a list of properties that the customized image should satisfy. Then, each property is evaluated independently by prompting the same model in a new context to determine whether the property holds, using a prompt similar to that of the simple LLM-based verifiers. The final score is the average score of the per-property scores.

3.3 Agentic Vision Verifier

This verifier is based on recent work by Google, which leverages code execution to ground the LLM's reasoning[2]. The approach generates Python code to manipulate and/or analyze an image in order to extract more fine-grained information. It is analogous to visual programming and agentic approaches, and serves as a solid code-based baseline.

3.4 Segmentation and Tool-based Verifier

We implemented an approach that leverages the code-generation capabilities of LLMs to produce Python code. By prompting the model with a precise specification, we ensure that the generated code uses a set of oracle (functions). Each oracle is designed to perform a specific visual verification on a change between two images (e.g., color, shape, position).

3.4.1 Code Generation and Score Computation. The first step of this approach is to prompt the model to generate a verifier function that uses the available oracles. This function is then executed on the customized image and returns a score between 0 and 1.

[1] https://github.com/CharlyReux/verifier_problem
[2] https://blog.google/innovation-and-ai/technology/developers-tools/agentic-vision-gemini-3-flash/

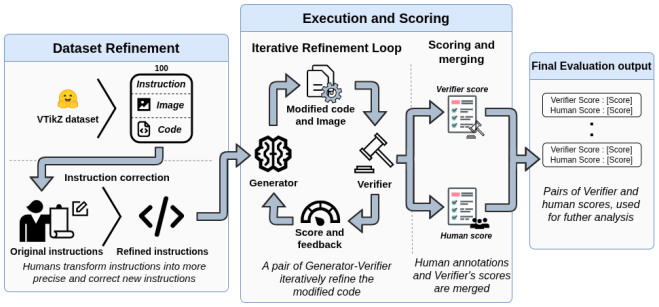


Figure 3: Complete evaluation protocol.

For example, to verify that a circle is changed to blue and its width is doubled, the *color* and *size* oracles can be combined using the **and** operator.

```
def test_valid_customization() -> bool:
 return color("circle", "blue")
    and size("circle", (2, 1))
```

Each oracle internally produces a probability, which are combined to compute a final score, computed using probabilistic Boolean combination.

3.4.2 Automatic Feedback Generation. Each oracle produces automatic feedback describing the detected discrepancy. When the generated verifier code combines multiple oracles using logical operators, feedback is reported only when the corresponding combined probability exceeds a predefined threshold of 0.9.

3.4.3 Oracle Catalog. Our oracle catalog includes checks for spatial relations (placement, position, alignment, mirrored), visual attributes (color, size, shape), containment and quantity (within, present, count), rotation (angle), and complex visual properties (visual_property). Each oracle leverages Gemini-2.5 segmentation to identify visual elements from textual descriptions.With this type of method, completeness for an oracle catalog is difficult to achieve, and there is a risk of not covering every possible use case. This limitation is inherent to the catalog-based approach.

4 EVALUATION METHODOLOGY

This section presents our evaluation methodology, summarized in fig. 3. We first refine the VTikZ dataset by correcting and improving the existing instructions. Next, we run the iterative post-hoc refinement loop using selected verifier/generator pairs. At each iteration, we collect the generated image (produced from the customized code) along with the score assigned by the verifier. The resulting images are then evaluated by human annotators, who are provided with the corresponding instruction and the original image, and asked to rate them on a 1–5 scale. This yields pairs of human and verifier scores, which form the basis of our subsequent analysis. We analyze verifier performance along two dimensions: scoring accuracy and feedback quality. Based on this distinction, we formulate the following research questions:

RQ1 *To what extent do verifiers correctly assess the application of a visual instruction to code?*
- RQ 1.1 Can they correctly detect whether a customization has been applied or not?
- RQ 1.2 Do their scores correlate with human judgments?

RQ2 *How does the feedback created and the score assigned by verifiers relate to performance increases in the customization task?*
- RQ 2.1 Does the quality of feedback matter?
- RQ 2.2 To what extent does the score matter to achieve better solutions?

RQ3 *How does feedback influence the interaction between the generator and the verifier?*
- RQ 3.1 Which types of feedback lead to errors, and which guide the generator toward better solutions?
- RQ 3.2 How do generator LLMs interpret and act upon the feedback they receive?
- RQ 3.3 What makes a Generator create a perfect solution?

4.1 Dataset

We base our evaluation on the existing *VTikZ* dataset [31]. We performed a manual pass to correct typographical errors, resolve ambiguities, and improve clarity. Every instruction specifies a visual modification to be applied to an existing TikZ program. The corresponding original code and rendered image serve as the starting point for the iterative refinement process.

4.2 Execution

We apply the iterative refinement to each customization. At every iteration, the generated code, rendered image, and verifier's score are recorded. This results in a sequence of refinement steps for each customization, which are later used for human annotation and quantitative analysis.

Formalization of the Iterative Refinement Protocol. For a given customization, we consider a Generator G and a Verifier V. The Generator G exposes two operations: *generate* and *adjust*. The *generate* operation takes a code and an instruction as input and produces a modified code. The *adjust* operation takes feedback and a score and returns an updated code. The Verifier V takes an instruction and a context as input, with the context containing the original and customized rendered images and/or their corresponding source codes. The verifier returns a score between 0 and 1 along with textual feedback.

The process for a pair (G, V) is outlined in Algorithm 1. Given an initial code and instruction, the generator G produces a customized code c' (l.3). An iterative refinement loop is then performed (l.5). At each iteration, a context is constructed (l.7) that includes the initial and generated images and codes (r, r', c, c'). This context is provided to the verifier V (l.8), which outputs feedback and a score (f, s). If the score satisfies a predefined threshold (set to 1 in our experiments), the process terminates (l.11). Otherwise, G uses (f, s) to produce an updated version of the code (l.13). The loop continues until the threshold is met or the maximum number of iterations T is reached. At each iteration, the current code and image (c', r'), together with the score s, are recorded (l.9) and returned upon termination. This process is executed on all the customizations in the dataset, and returns code-image-score triplets.

4.3 Manual Annotation

The annotation process is conducted through a web interface in which annotators evaluate individual refinement steps rather than only the final output. Upon connection, each annotator is shown

Input : Code c and instruction i
Output : List of (c', r', s)

$s \leftarrow 0$; $\tau \leftarrow 1$; $T \leftarrow 5$;
$r \leftarrow$ `Renderer(`c`)`;
$c' \leftarrow$ `G.generate(`c,i`)`;
$D \leftarrow []$;
for $j \leftarrow 0$ **to** $T - 1$ **do**
  $r' \leftarrow$ `Renderer(`c'`)`;
  $context \leftarrow (r, r', c, c')$;
  $(f, s) \leftarrow$ `V(`i,$context$`)`;
  D.append(c', r', s);
  if $s \geq \tau$ **then**
    return D
  end
  $c' \leftarrow$ `G.adjust(`f,s`)`;
end
return D

Algorithm 1: Iterative Refinement Execution

an image randomly sampled from all code-image-score triplets. The corresponding original image and customization instruction are also displayed. The annotator is then asked to rate how well the instruction was applied on an ordinal scale from 1 to 5.

- 1 - Not applied at all
- 2 - Slightly applied
- 3 - Moderately applied
- 4 - Mostly applied
- 5 - Perfectly applied

This gives us, for each step, triplets of *image*, *verifier score*, and *human score*.

Eight annotators participated in the manual annotation process. Although the dataset contains many generated images, several correspond to identical or very similar code modifications produced by the LLMs. As a result, the annotations cover 3907 distinct items, which correspond to a larger total of 15345 generated instances in the dataset. This deduplication significantly reduced the amount of manual annotation required while still providing coverage of a larger set of generated outputs. More details about the human-annotated data are provided in Table 1. We measure inter-annotator agreement using Krippendorff's α, obtaining a value of 0.898, indicating strong agreement and supporting the reliability of the annotations.

4.4 Evaluation Configurations

Because we evaluate *verification methods* rather than the *models* themselves, we conduct experiments using two models: *Qwen3-vl-30b-a3b-instruct* and *Gemini-3-flash-preview*, with the Gemini model configured with minimal thinking effort. These models were chosen for their strong coding and multimodal capabilities.Each model is used both as a generator and as a verifier. Unless specified otherwise, all models are run with a temperature of 0.5.

We first evaluate the four *basic verifiers* (*Simple* and *Property*) using Gemini-3 as the verifier model, paired with both Qwen and Gemini as generators. We then repeat the same experiment using Qwen as both the verifier and generator model. In addition to these configurations, we evaluate the two *complex verifiers* (Agentic Vision and Segmentation Tool-Based) paired with both Qwen and Gemini. Finally, we introduce two additional verifiers: (1) *SentenceVerifier*, a trivial verifier that always returns the message "The customization is not applied, please apply it.", and (2) *PreciseVerifier*, a CLIP-based verifier, executed last, which only returns a score measured using the maximum CLIP similarity between the customized image and the set of images annotated as perfect. Both of these new baselines allow us to assess the usefulness of the feedback and the score, which we interpret in Sections 5.1 and 5.2

Overall, this results in 20 generator-verifier pairs. Each pair is evaluated on the 100 TikZ customizations, with three repeated runs per customization and up to five refinement iterations per run.

Table 1: Human Annotation Statistics and Inter-Annotator Agreement

#Annotators	8
Total number of distinct items	3907
Total number of items	15345
Total number of distinct annotated items	4289
Total number of annotated items	17100
Avg. annotations per Item	1.11
Avg. distinct #items per annotator	536
Avg. #items per annotator	2138
Krippendorff's α	0.898

5 EVALUATION RESULTS

This section now presents our observed results answering our RQs.

5.1 *RQ1:* Assessment of Visual Instruction Application on Code

For this evaluation, we first focus on the ability of verifiers to identify perfect customizations. We therefore cast the task as a binary classification problem, where items rated 5 ("Perfectly applied") by human annotators are treated as positive instances, and all others as negative. We then evaluate how well verifier scores can predict these positives using F1-score. Figure 4 presents the precision-recall (PR) curves for all evaluated generator-verifier pairs. Overall, *simple LLM-based verifiers* (Text, Text+Visual, Text-only) as well as AgenticVision tend to assign high scores to most candidate customizations. As a result, their PR curves are concentrated in the high recall region, with no operating point (except near 0) achieving both high precision and low recall. This indicates a strong bias toward accepting outputs, yielding consistently high recall but limited precision.

Introducing more structured verification partially mitigates this behavior. The VisualPropertyVerifier reduces the frequency of uniformly maximal scores, leading to a more balanced score distribution. However, this comes at the cost of overall effectiveness, with both precision and recall decreasing relative to simple LLM-based approaches. A similar trade-off is observed for the Segmentation and Tool-based approach, which performs similarly to VisualPropertyVerifier but yields overall underwhelming results.

To further quantify these trends, the best achievable F1-score for each approach is computed over all decision thresholds. The corresponding precision and recall values are reported in Table 2 and

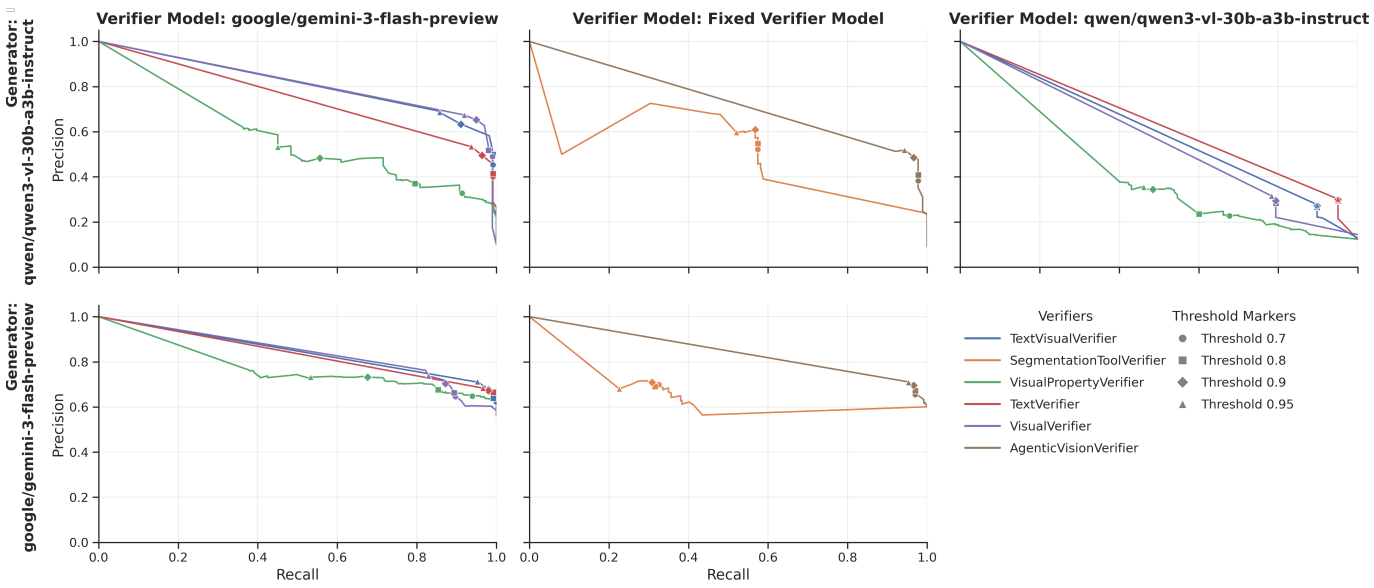


Figure 4: PR curves for each verifier.

Table 2: Precision/Recall/F1 Scores for the best thresholds for llm verifiers

Verifier Model	Generator	Text				Text+Visual				Visual				Visual Property			
		Threshold	Precision	Recall	F1	Threshold	Precision	Recall	F1	Threshold	Precision	Recall	F1	Threshold	Precision	Recall	F1
Gem.	Gem.	1.000	0.683	0.967	0.801	0.980	0.712	0.952	**0.815**	1.000	0.762	0.822	0.791	0.667	0.645	0.970	0.775
Gem.	Qwen	1.000	0.533	0.937	0.680	1.000	0.691	0.857	0.765	1.000	0.674	0.919	**0.778**	0.875	0.484	0.715	0.578
Qwen	Qwen	1.000	0.302	0.950	**0.458**	1.000	0.277	0.897	0.423	1.000	0.316	0.783	0.450	0.880	0.348	0.531	0.421

Table 3: Precision/Recall/F1 Scores for the best thresholds for fixed verifiers

Generator	Agentic Vision				Segmentation-Tool Verifier			
	Threshold	Precision	Recall	F1	Threshold	Precision	Recall	F1
Gem.	1.000	0.712	0.949	**0.814**	0.000	0.601	1.000	0.751
Qwen	0.980	0.519	0.943	**0.669**	0.920	0.609	0.568	0.587

Table 3. For readability, we abbreviate model names as follows: *Gem.* (Gemini-3-flash-preview) and *Qwen* (Qwen3-vl-30b-a3b-instruct). For *simple LLM-based Verifiers*, optimal F1-scores are consistently obtained at thresholds close to 1, confirming the tendency of these models to assign high scores, leading to high recall but relatively low precision.

Across all approaches, performance improves when paired with stronger generator models, a trend observed for both LLM-based verifiers (Table 2) and fixed approaches (Table 3). Notably, AgenticVision achieves the highest overall F1-score when paired with Gemini (0.814), but not with Qwen (0.669).

Table 4 reports Kendall correlation coefficients between verifier scores and all human judgments. In contrast to F1-scores, an opposite trend can be observed: correlation is higher when the generator model is weaker, suggesting that stronger generators produce outputs that are harder to discriminate. Input modality also plays a key role. When paired with weaker generators, combining textual and visual inputs (*Text+Visual*) yields the strongest alignment with human judgments (correlation up to 0.765) However, with stronger generators, relying solely on visual input (*Visual*) leads to better alignment, indicating that textual signals may introduce noise once baseline output quality is high.

Table 4: Kendall correlations for LLM and Fixed verifiers

Verifier Model	Generator	Text	Text+Visual	Visual	Visual Property
Gem.	Gem.	0.172	0.448	**0.456**	0.287
Gem.	Qwen	0.718	**0.765**	0.709	0.530
Qwen	Qwen	**0.620**	0.616	0.487	0.364

Generator	Agentic Vision	Precise Score	Segmentation-Tool Verifier
Gem.	0.441	**0.534**	0.003
Qwen	**0.718**	0.373	0.347

RQ1: Among the evaluated approaches, only Gemini-3-Flash is reliable at detecting whether a customization has been applied, achieving F1-scores up to 0.815. However, it tends to overestimate correctness, resulting in high recall but lower precision. Correlation with human judgments is moderate when paired with Gemini as the generator (0.456), but substantially higher with Qwen (0.765).

More complex verification strategies—such as code generation (AgenticVision, Segmentation-Tool-based) or decomposition into

multiple verification steps (VisualProperty) - do not improve performance and can even degrade both accuracy and alignment with human judgments.

Overall, for TikZ code, LLM-based verifiers (particularly Gemini-3-Flash) can be used to assess whether a visual instruction has been applied. However, this comes with a significant rate of false positives, reflecting a systematic bias toward high recall at the expense of precision.

5.2 *RQ2:* Overall Performances

Figure 5 reports the step-wise performance of all approaches, measured as the number of customizations that achieve a perfect score according to human annotators.

For Qwen as the **generator**, performance improves when paired with Gemini as the verifier, with gains of up to +16 and +17 of perfect customizations. The largest gains occur when Gemini has access to the generated code (TextVerifier, TextVisualVerifier), although gains remain noticeable with visual-only input (VisualVerifier). Compared to the baseline where Qwen acts as both generator and verifier, using Gemini leads to consistent gains, although they are more limited in the visual-only setting (+11). Pairing Qwen with the SentenceVerifier, which provides only coarse binary feedback ("customization not applied"), yields improvements comparable to being paired with itself, suggesting that minimal feedback can be useful for weaker models.

Improvements are more limited when Gemini is used as the **generator**. When paired with itself, the VisualVerifier yields the best results, but only marginally (+2). Unlike Qwen, performance decreases when using the SentenceVerifier. This difference can be explained by the higher initial performance of Gemini (around 75 correct customizations versus 20–24 for Qwen), which reduces the need for additional exploration and increases the risk of degrading already correct outputs during refinement.

More complex verification strategies, such as AgenticVision and the Segmentation-Tool-based approach, yield limited gains across both generators. Their performance remains comparable to, or slightly below, that of simpler LLM-based verifiers, indicating that added architectural complexity does not translate into improved refinement.

The PreciseVerifier yields the largest performance improvements, reaching up to 85 perfect customizations at step 4. By leveraging similarity scores with known perfect outputs, it effectively halts successful trajectories early while allowing further exploration on unresolved cases. However, Qwen benefits less from this mechanism, with only marginal improvements (+2).

RQ2: Feedback can improve refinement performance, but its impact strongly depends on the strength of the generator. For weaker models such as Qwen, informative feedback from stronger verifiers (e.g., Gemini-based) leads to substantial gains (33–36 perfect customizations), while coarse or indirect signals (e.g., PreciseVerifier, SentenceVerifier) result in lower performance (28–30).

For stronger models such as Gemini, the role of feedback differs: coarse feedback degrades performance, whereas more informative verifiers (TextVisualVerifier, VisualVerifier) yield modest improvements. The best results are obtained with PreciseVerifier, indicating that, at higher performance levels, accurate scoring and early stopping become more critical than exploratory feedback.

Overall, feedback is beneficial for improving performance in the TikZ customization task, especially for weaker models. However, for stronger models, the precision and reliability of the verifier play a more critical role than the feedback.

5.3 *RQ3:* Qualitative Feedback analysis

To better understand LLM behavior during iterative refinement, we conduct a manual analysis of verifier-generated feedback. We focus on feedback produced by *TextVisualVerifier*, *PropertyVerifier*, and *AgenticVisionVerifier*, as these approaches generate open-ended, visually grounded, natural language feedback. In contrast, other approaches either constrain feedback to predefined properties (*VisualPropertyVerifier*) or rely on fixed oracle responses (*Segmentation-Tool-based approach*).

We construct a characterization of feedback in customization trajectory steps. We consider three dimensions:

- (i) the quality of the initial customization (perfect vs. non-perfect)
- (ii) the correctness of the feedback (correct vs. erroneous with respect to the instruction or image)
- (iii) the quality of the subsequent customization (Worse, Unchanged, or Better)

For each output of a trajectory step, we analyze the feedback in terms of **aspects** (i.e., what the feedback contains) and **interpretations** (i.e., how it is acted upon by the generator). The analysis is conducted over **138 randomly sampled tuples** of the form ⟨initial customization, feedback, refined customization⟩. Sampling was continued until **no new feedback patterns or trajectory behaviors were observed**, suggesting that the main trajectory patterns are adequately represented. This enables us to identify recurring feedback patterns associated with improvement, stagnation, or degradation in customization quality.

We summarize this study in fig. 6, which depicts a decision tree over customization trajectories. Starting from an initial customization, trajectories branch on the observed quality (Perfect vs. Non-Perfect) and the correctness of the feedback, leading to subsequent customization steps. At each decision node, we report the frequency of transitions, and at the leaves, we detail the generators' interpretations and resulting actions (e.g., rightly guided, partially applied, neglected, or wrongly guided).

For incorrect feedback, the dominant failure modes are *hallucinations* on the visual content or instruction and *misunderstandings* of the instruction.

For correct feedback, we identify several aspects, specifically, it may be **imprecise**, **non-guiding**, **incomplete**, or **forgetful**, which can lead to partial, incorrect, or missing application in the next customization.

We define these aspects as follows:

- **Precision**: The feedback localizes and specifies issues clearly, rather than describing them in vague/abstract terms
- **Guidance**: The feedback goes beyond identifying errors and provides actionable directions for correction.

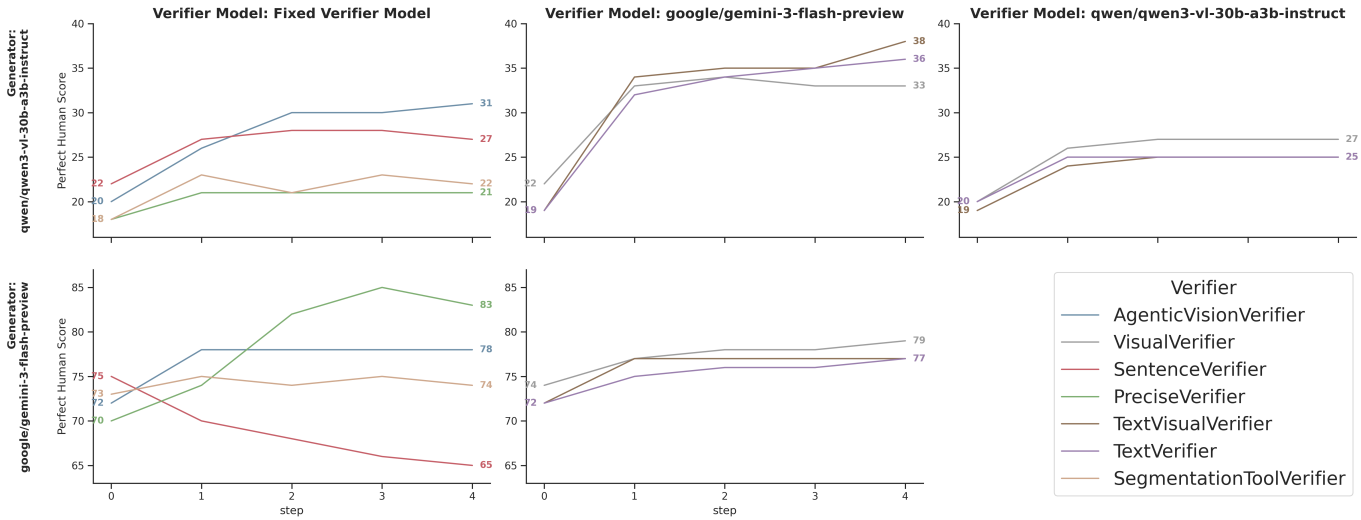


Figure 5: Total number of perfect customizations per step per verifier.

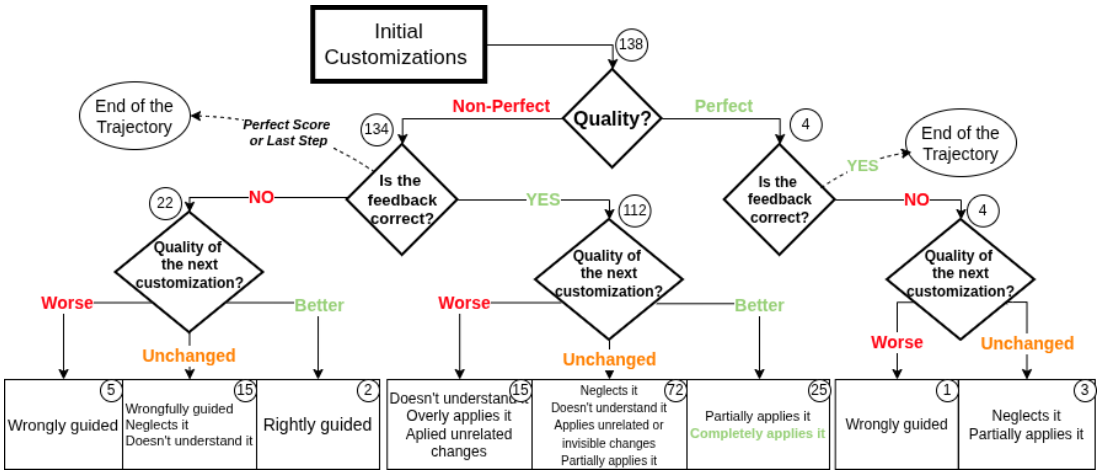


Figure 6: Observed Customization Trajectories, and subsequent interpretation/actions by the generator.

- **Completeness**: The feedback captures all necessary issues required to correctly refine the customization.
- **Forgetfulness**: The feedback "forgets" about the initial instruction, and only comments on the flaws of the diagram.

Independent of correctness, feedback may be interpreted and acted upon in different ways by the generator as it may be understood or not, and either applied, partly applied, or ignored. We also observe recurring behaviors such as **invisible changes**, **stubbornness**, or **neglect** where the generator fails to modify the code despite receiving feedback. This may stem either from an inability to operationalize the feedback or from overconfidence in the current solution.

Invisible changes are generally reformatting, adding comments, or shifting the entire picture. Together with **neglect**, these are the main reasons why the majority of the customizations happening exhibit the same quality as the previous customizations, and are mainly due to Qwen's inability to correctly apply the feedback.

Finally, we select the behaviors that led to perfect customization. For these we have observed most/all of the aspect described above: the feedback is leverage-able and states what must be modified (**guiding**), it explicitly describes the what is right and what is wrong w.r.t the instruction (**not forgetful**) and it provides detailed information (**Precise**), and finally, it states all the points which are to be fixed (**Complete**).

RQ3: Feedback that leads to errors is either (i) incorrect, due to **hallucinations** or **misunderstandings**, or (ii) correct but ineffective, due to being **imprecise**, **non-guiding** or **forgetful**. Generator LLMs exhibit diverse interpretation behaviors, ranging from correct understanding and application to failure modes such as misunderstanding, partial application, **neglect**, and **stubbornness**. Given that the generator LLM can operationalize the feedback, our study

shows that perfect customization is achieved only when the feedback is **guiding**, **not forgetful**, **Precise**, and **Complete**. Overall, the feedback influences the interactions in multiple ways, either negatively or positively, depending on its quality, and also on how LLM can interpret it.

6 RECOMMENDATIONS

We revisit the central question of this paper: *To what extent can iterative refinement remain effective when the verifier itself is unreliable?* The TikZ case study suggests a positive but conditional answer. **Iterative refinement remains effective under imperfect verification, but only when the verifier is reliable enough to avoid premature acceptance and provides actionable feedback**. Building on this insight, we derive two sets of recommendations : (i) practical guidelines for *TikZ customization*, grounded in our empirical results (Section 5), and (ii) broader implications for refinement pipelines relying on *imperfect visual verifiers*.

6.1 Recommendations for TikZ customization

Prefer simple multimodal LLM-based verifiers as a strong baseline. As shown in RQ1 (section 5.1), **Gemini-based verifiers consistently provide the strongest overall assessment results**, whereas more elaborate verification strategies based on decomposition, code generation, or segmentation do not systematically improve either detection quality or alignment with human judgments. Combined with the results of RQ2 and RQ3, this suggests that the advantage of multimodal verifiers in TikZ is not only that they can assess whether a customization has been applied, but also that they provide feedback that is easier for the generator to exploit during refinement.

Do not treat these verifiers as reliable or near-perfect oracles. Despite strong recall, RQ1 highlights a persistent bias toward false positives. In an iterative loop, such errors are particularly harmful because they terminate refinement too early. **Stopping policies should therefore be conservative, and verifier acceptance treated with caution**.

Adapt the verifier's role to the generator's strength. RQ2 shows that **weaker generators benefit primarily from informative feedback**, whereas **stronger generators benefit from accurate scoring and early stopping**. In practice, the verifier should act as a critic for weaker models, and a more selective stopping mechanism for stronger ones.

Explicitly optimize for feedback quality. RQ3 shows that failures stem not only from incorrect feedback, but also from correct yet ineffective feedback. Imprecise, non-actionable, incomplete, or forgetful feedback often leads to stagnation, partial application, or incorrect edits. For TikZ customization, **effective feedback must localize the problem, specify required changes, remain grounded in the original instruction, and cover all relevant corrections**.

Assess both requested change and preservation. Focusing solely on whether the requested modification is present is not sufficient. **Successful customization must preserve the semantics and structure of the original artifact.** Verifiers should therefore jointly assess the applied change and the absence of unintended side-effects.

6.2 Recommendations for imperfect visual verifiers

Although our empirical claims are carefully restricted to TikZ, the case study suggests broader recommendations for iterative refinement in domains where correctness cannot be checked through deterministic tests.

Separate decision quality from feedback quality. Two distinct capabilities must be considered: (i) the ability to correctly accept or reject outputs, and (ii) the ability to provide guidance that improves them. **A verifier can be a good scorer but a poor guide, or vice versa**. These dimensions should be evaluated independently.

Evaluate verifiers within the refinement loop. Standalone verification is insufficient. Our results show that **the same verifier can help or hinder depending on the generator and the interaction dynamics**. Evaluation should therefore consider final task success, stopping behavior, and the evolution of solutions across refinement steps.

Prioritize reducing false positives and improving actionability. False positives are harmful, as they prematurely terminate refinement. At the same time, **useful feedback is not merely correct: it must be precise, guiding, complete, and explicitly tied to the instruction being executed**. These properties constitute a more realistic target for imperfect visual verifiers than score agreement alone.

7 LIMITATIONS AND THREATS TO VALIDITY

Case study. Our study focuses on a single language, which limits generalization to other domains. While TikZ customization does not capture the scale and complexity of broader multimodal coding tasks (e.g., web development), it provides, as discussed in section 2, a focused and controlled setting that isolates the core difficulties of the problem while combining code generation, multimodal reasoning, and instruction understanding. Nonetheless, future replication on other languages and settings remains necessary.

Evaluated models. We evaluate only two models, namely Qwen3-VL-30B-A3B-Instruct and Gemini-3-Flash-Preview. This choice is primarily driven by the high annotation cost: even with two models, the study required 8 annotators and 4,289 annotations. The selected models represent different paradigms (*instruct* vs. *thinking*), which may limit direct comparability. Nevertheless, both are suitable for the customization task and provide complementary baselines. Importantly, our focus is on imperfect verifiers rather than on the generative models themselves.

Limitation to 5 steps. We restrict evaluation trajectories to five steps. This represents a trade-off between capturing iterative refinement dynamics (e.g., classification quality, trajectory patterns, and feedback quality) and maintaining a manageable annotation effort. Exploring longer trajectories could provide additional insights, but is left for future work.

Constrained feedback format. The system prompts we used enforce a specific feedback format. This limits the diversity of possible feedback. However, as shown in section 5.3, it is sufficient to assess the quality of LLMs' feedback in an iterative refinement setting.

8 RELATED WORK

Visual code customization sits at the intersection of iterative refinement, critic models, visual question answering, and visual code generation. While each area has seen substantial progress, none

directly addresses iterative refinement under an imperfect visual verifier - where feedback may yield false positives, false negatives, or lack the actionability needed to guide meaningful improvement.

Iterative refinement. It was initially introduced as a test-time compute strategy to improve LLM outputs, notably with Self-Refine [26], and has since been extended in approaches such as CRITIC [8]. Prior work has applied iterative refinement to code generation [5, 26, 33, 42, 47] and, more recently, to multimodal tasks [2, 18, 20, 42]. However, these studies do not explicitly consider iterative refinement for structured diagram customization at inference time. Only Xu et al. and Li et al. explore both multimodal reasoning and code generation in the context of chart creation, but target full code generation from existing data, rather than customization from existing code. In code generation, Post-Hoc iterative refinement typically relies on strong, programmatic feedback signals, similarly to RLVR-based systems. For instance, Self-Debug [5] and Reflexion [33] leverage multi-turn unit test execution to verify intermediate outputs, while Self-Edit [46] applies a single-turn execution-based feedback strategy. However, these approaches leverage deterministic verification, which is unavailable when correctness depends on properties of rendered diagrams. Agents also exhibit similar behavior to iterative refinement [1], by using external tools and verification loops. In contrast to our context, the verifications are usually more reliable. Other works investigate refinement strategies during training, using feedback loops to improve model parameters [4, 17, 22, 25], In contrast, our work focuses exclusively on refinement as a Post-Hoc strategy, applied at inference time. *Critic models* are LLMs fine-tuned to evaluate or verify outputs [14, 15, 18, 21, 22, 28, 30, 39]. They are often applied in single-turn generate-critique-refine pipelines, sometimes extending to multimodal domains (e.g., *VIS-Shepherd*, *ReLook*, *Volcano*, *IE-Critic-R1*). While these critics demonstrate task-specific evaluation, they operate in a single-turn fashion and do not explore multi-step refinement loops.

Visual Question Answering and UI critique. The task of Visual Question Answering (VQA) is related to verification in our setting, as verification can be framed as asking: *"Is the instruction <instruction> correctly applied to this image?"*. However, unlike standard VQA, our setting requires fine-grained spatial reasoning, instruction grounding, and contextual comparison between an initial and an edited image. Visual Programming (VP) approaches [7, 35, 37] decompose tasks into sequences of tool calls. While conceptually similar, these methods are primarily designed for natural images and generic visual primitives, limiting their applicability to diagram-level edits. For completeness, we also evaluate VP-like verifiers(see sections 3.3 and 3.4). The task of *UI critique*, which consists of generating design comments from a UI screenshot and design guidelines, is also related. Duan et al. propose a pipeline that generates feedback and corresponding bounding boxes, and iteratively refines both using a validation model. This approach shares similarities with the segmentation-tool-based verifier (section 3.4), but operates on different inputs.

Visual Code Generation. The intersection of code generation and visual inputs has been explored primarily through image-to-code approaches [3, 42, 48], and to a lesser extent in editing scenarios [9, 11]. However, these approaches focus on single-turn generation, often in a single pass, without any feedback mechanism. Only Xu et al. incorporate iterative refinement, but exclusively in the context of image-to-code generation rather than editing.

Refinement and Repair capability evaluation. Several works evaluate the refinement capabilities of LLMs, either by measuring their ability to improve their own outputs or by assessing the quality of generated feedback. These studies typically focus on domains with deterministic or easily verifiable answers, such as mathematics, algorithms, or multiple-choice reasoning. Notably, Huang et al. analyze self-correction in LLMs and highlight both its potential and its limitations. Benchmarks such as RefineBench [19] evaluate refinement using domain-specific checklists and consider both guided (external feedback) and self-refinement settings. Other works, including CriticBench [24], RealCritic [38], and CriticEval [16], focus on evaluating the quality of critics, again in domains where verification can be reduced to clear, often binary outcomes. Closely related to editing, SR-Eval [44] evaluates the ability of LLMs to apply stepwise requirements in software engineering tasks, but considers cumulative sequences of different edits rather than iterative refinement from a single instruction. Similarly, Olausson et al. study self-repair in full code generation settings. Beyond deterministic domains, Sun et al. investigate refinement in text summarization, where correctness is less well-defined, but limit their analysis to single-turn prompting strategies. In the multimodal domain, VF-EVAL [34] evaluates the ability of models to generate feedback on AI-generated videos, but does not consider iterative refinement and is not related to code.

To the best of our knowledge, no existing framework captures the challenges of iterative refinement under imperfect visual verifiers. We provide the first novel empirical analysis of multi-step refinement in visual code customization, characterizing verifier behavior with regard to the quality of its classification and feedback.

9 CONCLUSION

In this paper, we explored the applicability and effectiveness of post-hoc iterative refinement with imperfect verifiers, using TikZ code customization as a case study. We introduced an evaluation framework to analyze LLM-based verifiers within an iterative refinement pipeline. Building on the VTikZ dataset, which we refined, we evaluated 20 generator–verifier pairs on 100 TikZ customization tasks, for up to five refinement steps, resulting in a total of 15,345 customized code instances. We conducted a large-scale manual evaluation of these customizations, enabling us to analyze both output quality and the impact of feedback in iterative refinement. In particular, we examined verifier performance in (i) assessing whether customization instructions are correctly applied to code, and (ii) understanding how feedback and scoring influence subsequent refinements.

Our results show that verifiers are generally capable of determining whether instructions are applied, but exhibit a bias toward classifying solutions as valid, leading to a high rate of false positives. Feedback improves performance in post-hoc iterative refinement; however, the reliability of scoring becomes more critical for stronger models. We further derived a study of feedback by systematically analyzing 138 customization trajectories of ⟨initial customization, feedback, refined customization⟩, extracting fine-grained feedback aspects and their interpretation by LLMs. Based on these findings, we provide actionable recommendations not only for TikZ but also for visual verification more broadly. These insights can support the design of more effective and reliable verifiers for further research in TikZ or other similar contexts.

Finally, our results suggest several directions for future work. The annotated trajectories collected in this study could be used to train a

learned scorer for TikZ customization, reducing manual annotation costs and potentially improving the verifier signal available during refinement. A second direction is to replicate the study on other visual languages, such as SVG or P5.js, to determine which findings transfer beyond TikZ. A third direction is to investigate imperfect visual verification in richer environments, including web interfaces, games, and 3D scenes, where the artifact to be evaluated is no longer a single rendered image but a more complex interactive environment.

ACKNOWLEDGEMENTS.

This work is supported by the Inria Défi LLM4Code.

REFERENCES

[1] Mohamad Abou Ali and Fadi Dornaika. 2025. Agentic AI: A Comprehensive Survey of Architectures, Applications, and Future Directions. *Artificial Intelligence Review* 59, 1 (Nov. 2025), 11. doi:10.1007/s10462-025-11422-4 arXiv:2510.25445 [cs].

[2] Ruhana Azam, Tamer Abuelsaad, Aditya Vempaty, and Ashish Jagmohan. 2024. Multimodal Auto Validation For Self-Refinement in Web Agents. doi:10.48550/arXiv.2410.00689 arXiv:2410.00689 [cs].

[3] Jonas Belouadi, Simone Paolo Ponzetto, and Steffen Eger. 2024. DeTikZify: Synthesizing Graphics Programs for Scientific Figures and Sketches with TikZ. doi:10.48550/arXiv.2405.15306 arXiv:2405.15306 [cs].

[4] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2024. Improving Code Generation by Training with Natural Language Feedback. doi:10.48550/arXiv.2303.16749 arXiv:2303.16749 [cs].

[5] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. doi:10.48550/arXiv.2304.05128 arXiv:2304.05128 [cs].

[6] Peitong Duan, Chin-Yi Cheng, Bjoern Hartmann, and Yang Li. 2025. Visual Prompting with Iterative Refinement for Design Critique Generation. doi:10.48550/arXiv.2412.16829 arXiv:2412.16829 [cs].

[7] Jiaxin Ge, Sanjay Subramanian, Baifeng Shi, Roei Herzig, and Trevor Darrell. 2024. Recursive Visual Programming. doi:10.48550/arXiv.2312.02249 arXiv:2312.02249 [cs].

[8] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. doi:10.48550/arXiv.2305.11738 arXiv:2305.11738 [cs].

[9] Yucheng Han, Chi Zhang, Xin Chen, Xu Yang, Zhibin Wang, Gang Yu, Bin Fu, and Hanwang Zhang. 2023. ChartLlama: A Multimodal LLM for Chart Understanding and Generation. doi:10.48550/arXiv.2311.16483 arXiv:2311.16483 [cs].

[10] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models Cannot Self-Correct Reasoning Yet. doi:10.48550/arXiv.2310.01798 arXiv:2310.01798 [cs].

[11] Lingjie Jiang, Shaohan Huang, Xun Wu, Yixia Li, Dongdong Zhang, and Furu Wei. 2025. VisCodex: Unified Multimodal Code Generation via Merging Vision and Coding Models. doi:10.48550/arXiv.2508.09945 arXiv:2508.09945 [cs].

[12] Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R. Lyu, and Xiangyu Yue. 2025. ScreenCoder: Advancing Visual-to-Code Generation for Front-End Automation via Modular Multimodal Agents. doi:10.48550/arXiv.2507.22827 arXiv:2507.22827 [cs].

[13] Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. doi:10.1162/tacl_a_00713/125177 arXiv:2406.01297 [cs].

[14] Pei Ke, Bosi Wen, Andrew Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Shengyuan Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, Jie Tang, and Minlie Huang. 2024. CritiqueLLM: Towards an Informative Critique Generation Model for Evaluation of Large Language Model Generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 13034–13054. doi:10.18653/v1/2024.acl-long.704

[15] Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. 2024. Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models. doi:10.48550/arXiv.2405.01535 arXiv:2405.01535 [cs].

[16] Tian Lan, Wenwei Zhang, Chen Xu, Heyan Huang, Dahua Lin, Kai Chen, and Xian-ling Mao. 2024. CriticEval: Evaluating Large Language Model as Critic. doi:10.48550/arXiv.2402.13764 arXiv:2402.13764 [cs].

[17] 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. doi:10.48550/arXiv.2207.01780 arXiv:2207.01780 [cs].

[18] Seongyun Lee, Sue Hyun Park, Yongrae Jo, and Minjoon Seo. 2024. Volcano: Mitigating Multimodal Hallucination through Self-Feedback Guided Revision. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Kevin Duh, Helena Gomez, and Steven Bethard (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 391–404. doi:10.18653/v1/2024.naacl-long.23

[19] Young-Jun Lee, Seungone Kim, Byung-Kwan Lee, Minkyeong Moon, Yechan Hwang, Jong Myoung Kim, Graham Neubig, Sean Welleck, and Ho-Jin Choi. 2025. RefineBench: Evaluating Refinement Capability of Language Models via Checklists. doi:10.48550/arXiv.2511.22173 arXiv:2511.22173 [cs].

[20] Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. 2025. METAL: A Multi-Agent Framework for Chart Generation with Test-Time Scaling. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 30054–30069. doi:10.18653/v1/2025.acl-long.1452

[21] Junlong Li, Shichao Sun, Weizhe Yuan, Run-Ze Fan, Hai Zhao, and Pengfei Liu. 2023. Generative Judge for Evaluating Alignment. doi:10.48550/arXiv.2310.05470 arXiv:2310.05470 [cs].

[22] Yuhang Li, Chenchen Zhang, Ruilin Lv, Ao Liu, Ken Deng, Yuanxing Zhang, Jiaheng Liu, Wiggin Zhou, and Bo Zhou. 2025. ReLook: Vision-Grounded RL with a Multimodal LLM Critic for Agentic Web Coding. doi:10.48550/arXiv.2510.11498 arXiv:2510.11498 [cs].

[23] Kevin Qinghong Lin, Yuhao Zheng, Hangyu Ran, Dantong Zhu, Dongxing Mao, Linjie Li, Philip Torr, and Alex Jinpeng Wang. 2025. VCode: a Multimodal Coding Benchmark with SVG as Symbolic Visual Representation. doi:10.48550/arXiv.2511.02778 arXiv:2511.02778 [cs].

[24] Zicheng Lin, Zhibin Gou, Tian Liang, Ruilin Luo, Haowei Liu, and Yujiu Yang. 2024. CriticBench: Benchmarking LLMs for Critique-Correct Reasoning. doi:10.48550/arXiv.2402.14809 arXiv:2402.14809 [cs].

[25] Jiaqi Liu, Kaiwen Xiong, Peng Xia, Yiyang Zhou, Haonian Ji, Lu Feng, Siwei Han, Mingyu Ding, and Huaxiu Yao. 2025. Agent0-VL: Exploring Self-Evolving Agent for Tool-Integrated Vision-Language Reasoning. doi:10.48550/arXiv.2511.19900 arXiv:2511.19900 [cs].

[26] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 46534–46594. https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html

[27] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is Self-Repair a Silver Bullet for Code Generation? doi:10.48550/arXiv.2306.09896 arXiv:2306.09896 [cs].

[28] Bo Pan, Yixiao Fu, Ke Wang, Junyu Lu, Lunke Pan, Ziyang Qian, Yuhan Chen, Guoliang Wang, Yitao Zhou, Li Zheng, Yinghao Tang, Zhen Wen, Yuchen Wu, Junhua Lu, Biao Zhu, Minfeng Zhu, Bo Zhang, and Wei Chen. 2025. VIS-Shepherd: Constructing Critic for LLM-based Data Visualization Generation. doi:10.48550/arXiv.2506.13326 arXiv:2506.13326 [cs].

[29] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2024. Automatically Correcting Large Language Models: Surveying the Landscape of Diverse Automated Correction Strategies. *Transactions of the Association for Computational Linguistics* 12 (2024), 484–506. doi:10.1162/tacl_a_00660

[30] Bowen Qu, Shangkun Sun, Xiaoyu Liang, and Wei Gao. 2025. IE-Critic-R1: Advancing the Explanatory Measurement of Text-Driven Image Editing for Human Perception Alignment. doi:10.48550/arXiv.2511.18055 arXiv:2511.18055 [cs].

[31] Charly Reux, Mathieu Acher, Djamel Eddine Khelladi, Olivier Barais, and Clément Quinton. 2025. LLM Code Customization with Visual Results: A Benchmark on TikZ. doi:10.48550/arXiv.2505.04670 arXiv:2505.04670 [cs].

[32] Juan A. Rodriguez, Abhay Puri, Shubham Agarwal, Issam H. Laradji, Pau Rodriguez, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. 2025. StarVector: Generating Scalable Vector Graphics Code from Images and Text. doi:10.48550/arXiv.2312.11556 arXiv:2312.11556 [cs].

[33] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. doi:10.48550/arXiv.2303.11366 arXiv:2303.11366 [cs].

[34] Tingyu Song, Tongyan Hu, Guo Gan, and Yilun Zhao. 2025. VF-Eval: Evaluating Multimodal LLMs for Generating Feedback on AIGC Videos. doi:10.48550/arXiv.2505.23693 arXiv:2505.23693 [cs] version: 1.

[35] Aleksandar Stanić, Sergi Caelles, and Michael Tschannen. 2024. Towards Truly Zero-shot Compositional Visual Reasoning with LLMs as Programmers. doi:10.48550/arXiv.2401.01974 arXiv:2401.01974 [cs].

[36] Shichao Sun, Ruifeng Yuan, Ziqiang Cao, Wenjie Li, and Pengfei Liu. 2024. Prompt Chaining or Stepwise Prompt? Refinement in Text Summarization. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 7551–7558. doi:10.18653/v1/2024.findings-acl.449

[37] Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. ViperGPT: Visual Inference via Python Execution for Reasoning. doi:10.48550/arXiv.2303.08128 arXiv:2303.08128 [cs].

[38] Zhengyang Tang, Ziniu Li, Zhenyang Xiao, Tian Ding, Ruoyu Sun, Benyou Wang, Dayiheng Liu, Fei Huang, Tianyu Liu, Bowen Yu, and Junyang Lin. 2025. RealCritic: Towards Effectiveness-Driven Evaluation of Language Model Critiques. doi:10.48550/arXiv.2501.14492 arXiv:2501.14492 [cs].

[39] Tianlu Wang, Ping Yu, Xiaoqing Ellen Tan, Sean O'Brien, Ramakanth Pasunuru, Jane Dwivedi-Yu, Olga Golovneva, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2023. Shepherd: A Critic for Language Model Generation. doi:10.48550/arXiv.2308.04592 arXiv:2308.04592 [cs].

[40] Jingxuan Wei, Cheng Tan, Qi Chen, Gaowei Wu, Siyuan Li, Zhangyang Gao, Linzhuang Sun, Bihui Yu, and Ruifeng Guo. 2024. From Words to Structured Visuals: A Benchmark and Framework for Text-to-Diagram Generation and Editing. doi:10.48550/arXiv.2411.11916 arXiv:2411.11916 [cs].

[41] Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. 2024. Plot2Code: A Comprehensive Benchmark for Evaluating Multi-modal Large Language Models in Code Generation from Scientific Plots. doi:10.48550/arXiv.2405.07990 arXiv:2405.07990 [cs] version: 1.

[42] Chengzhi Xu, Yuyang Wang, Lai Wei, Lichao Sun, and Weiran Huang. 2025. Improved Iterative Refinement for Chart-to-Code Generation via Structured Instruction. doi:10.48550/arXiv.2506.14837 arXiv:2506.14837 [cs].

[43] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. doi:10.48550/arXiv.2202.13169 arXiv:2202.13169 [cs].

[44] Zexun Zhan, Shuzheng Gao, Ruida Hu, and Cuiyun Gao. 2026. SR-Eval: Evaluating LLMs on Code Generation under Stepwise Requirement Refinement. doi:10.48550/arXiv.2509.18808 arXiv:2509.18808 [cs].

[45] Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, Qi Yi, Ruibin Xiong, Shihui Hu, Yue Zhang, Yuhao Jiang, Zenan Xu, Yuanxing Zhang, Wiggin Zhou, Chayse Zhou, and Fengzong Lian. 2025. ArtifactsBench: Bridging the Visual-Interactive Gap in LLM Code Generation Evaluation. doi:10.48550/arXiv.2507.04952 arXiv:2507.04952 [cs].

[46] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. doi:10.48550/arXiv.2305.04087 arXiv:2305.04087 [cs].

[47] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 54769–54784. https://proceedings.neurips.cc/paper_files/paper/2023/hash/abe1eb21ceb046209c96a0f5e7544ccc-Abstract-Conference.html

[48] Dawei Zhu, Rui Meng, Yale Song, Xiyu Wei, Sujian Li, Tomas Pfister, and Jinsung Yoon. 2026. PaperBanana: Automating Academic Illustration for AI Scientists. doi:10.48550/arXiv.2601.23265 arXiv:2601.23265 [cs].