

LLM Code Customization with Visual Results: A Benchmark on TikZ

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

▶ **To cite this version:**

Charly Reux, Mathieu Acher, Djamel Eddine Khelladi, Olivier Barais, Clément Quinton. LLM Code Customization with Visual Results: A Benchmark on TikZ. EASE'25 - Evaluation and Assessment in Software Engineering, Jun 2025, Istanbul, Turkey. pp.1-10, ⟨10.1145/3756681.3757003⟩. ⟨hal-05049250v2⟩

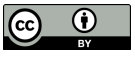
HAL Id: hal-05049250

https://hal.science/hal-05049250v2

Submitted on 4 Jun 2025

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

LLM Code Customization with Visual Results: A Benchmark 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

With the rise of AI-based code generation, customizing existing code out of natural language instructions to modify visual results – such as figures or images – has become possible, promising to reduce the need for deep programming expertise. However, even experienced developers can struggle with this task, as it requires identifying relevant code regions (feature location), generating valid code variants, and ensuring the modifications reliably align with user intent. In this paper, we introduce vTikZ, the first benchmark designed to evaluate the ability of Large Language Models (LLMs) to customize code while preserving coherent visual outcomes. Our benchmark consists of carefully curated vTikZ editing scenarios, parameterized ground truths, and a reviewing tool that leverages visual feedback to assess correctness. Empirical evaluation with state-of-the-art LLMs shows that existing solutions struggle to reliably modify code in alignment with visual intent, highlighting a gap in current AI-assisted code editing approaches. We argue that vTikZ opens new research directions for integrating LLMs with visual feedback mechanisms to improve code customization tasks in various domains beyond TikZ, including image processing, art creation, Web design, and 3D modeling.

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, Benchmark evaluation (vTikZ)

1 INTRODUCTION

Customizing code that produces visual results is a general and fundamental problem, with numerous applications in image processing (*e.g.*, SVG manipulation), digital art creation (*e.g.*, p5.js sketches), Web development (*e.g.*, HTML/CSS layouts with backend interactions), and 3D modeling (*e.g.*, Blender scripting, SCAD programs). In such contexts, modifications have to be made at the code level in order to achieve a specific visual change, such as adjusting an object's shape, changing colors, or adding elements. Rapid progress has been achieved in the field of Large Language Model (LLM)-assisted software engineering [18]. With the emergence of AI-based code generation, customizing code through natural language instructions (prompts) is more and more investigated, offering new possibilities for end-user programming and even for developers who need to onboard or interact with codebases [1, 25, 27]. However, modifying code with visual results remains a complex task, even for experienced developers, as it requires identifying the relevant code regions (feature location), generating valid code variants, and ensuring that modifications align with user intent while maintaining reliable and consistent visual results.

Users can typically prompt an LLM with high-level instructions, detailing the intended visuals supposed to be produced by a program, rather than the technical aspects of the code itself. It is useful for instance when tweaking interfaces, manipulating diagrams, or customizing images or 3D models coming from tools, scripts, or programs. To illustrate, consider a TikZ script that generates an image of a bee. A user may request an LLM to "add a third pair of wings", a seemingly simple instruction that requires multiple non-trivial steps: understanding the intent, identifying the relevant code, and applying the correct modification. Unlike humans, who can mentally simulate the visual effect of code changes, LLMs often struggle with these steps, behaving like a blind system guessing modifications without direct feedback from the rendered output. To systematically study this problem, we identify three key challenges:

- Feature location: Identifying the correct code segments that contribute to the visual elements requiring modification. Many visual programming languages, such as TikZ, lack explicit mappings between rendered elements and the corresponding source code.
- Code customization and variant synthesis: Generating alternative versions of the code that satisfy the modification request while ensuring syntactic correctness and maintaining consistency across multiple lines of code.
- Visual result validation: Ensuring that the generated output aligns with the intended change, remains visually coherent, and does not introduce unintended modifications to other elements in the figure.

Evaluating LLMs on tasks involving code customization and multimodality is promising, as it addresses both cross-modal consistency and program behavior prediction. As shown in Figure 1, LLMs can sometimes customize code correctly. However, systematically quantifying whether customized code meets an instruction remains a challenge, as no true ground truth exists for comparison. Our observation is that multiple edits can produce correct outputs, but

vision-based oracles can only approximate human preferences without certainty.

To explore these challenges, we introduce vTikZ, the first benchmark designed to evaluate the ability of LLMs to customize code while preserving coherent visual results. vTikZ consists of 100 manually curated TikZ editing scenarios, each requiring code modifications to achieve a specified visual change. Unlike existing benchmarks, which focus either on textual code edits (*e.g.*, bug fixes, refactorings) or visual-based interactions (*e.g.*, autonomous Web agents), vTikZ explicitly evaluates code customization in the presence of a visual output, bridging the gap between textual and visual modalities. We have carefully designed parameterized ground truths to account for multiple valid solutions, avoiding the unfair penalization of LLM-generated variants that still produce correct outputs. Additionally, we developed a reviewing tool to assess correctness based on the generated images. Using this tool, we have manually annotated 300 LLM-generated variants, that enabled a fine-grained validation of our ground-truths. Furthermore, we perform an evaluation of different LLMs. Results show that LLMs struggle to customize code, validating the design of vTikZ and calling to LLM-based solutions augmented with richer tools, modalities, or feedback.

To the best of our knowledge, no existing benchmark explicitly tackles LLM-driven code customization with visual validation. On the one hand, several software engineering (SE) benchmarks evaluate LLMs on tasks, such as bug fixing, refactoring, and feature addition, but these tasks focus solely on textual modifications without considering visual outcomes [2, 21, 23, 24]. On the other hand, some benchmarks assess visual-based AI agents, such as autonomous Web agents, which manipulate visual content, but they do not involve code customization. In between, there have been a few studies on LLM-based TikZ generation, where models are prompted to create figures from scratch based on textual descriptions [8, 35]. However, these approaches differ significantly from our customization scenario, where the challenge lies in modifying existing code while preserving intent. In the (grey) literature, TikZ is often used informally to evaluate LLMs with challenges like "draw a unicorn" [9, 36]. However, these evaluations lack systematic benchmarking. vTikZ extends this direction by providing a rigorous benchmark with structured evaluation metrics and a focus on modifying existing code rather than generating from scratch.

In this work, we introduce vTikZ, a benchmark designed to evaluate LLMs in customizing code with visual results. Our contributions are as follows:

- A dataset of 100 manually curated TikZ customization tasks, covering a variety of modifications (e.g., adding, removing, resizing, and repositioning elements) to assess LLMs' ability to modify existing graphical code.
- A parameterized ground truth framework, acknowledging that multiple code variants may correctly implement a given visual modification, preventing unfair penalization of LLM-generated outputs.
- A visual reviewing tool that enables automated and human-in-the-loop evaluation of generated visual results. Using this tool, we collected 300 data points, refining our evaluation framework and validating our ground-truth design.
- A systematic evaluation of LLMs on vTikZ, analyzing their strengths and limitations in feature location, code modification, and visual consistency. Our results show that existing models struggle with code customization, reinforcing the need for hybrid approaches integrating additional tools, multimodal feedback, or iterative validation mechanisms.
- An extensible benchmark that provides a structured methodology for evaluating LLM-driven code customization with visual outputs, paving the way for future research in multimodal AI-assisted programming.

The vTikZ dataset, the data resulting from the evaluation, and the manually annotated data are available online[1], along with the benchmark's code[2].

2 BACKGROUND AND MOTIVATION

In this section, we define the terminology used in the remainder of the paper and we motivate the need for a benchmark on TikZ with an illustrative example.

2.1 Background and Terminology

The work and the benchmark presented in this paper revolve around three main notions, namely TikZ, variants, and patches. *TikZ* is a tool to create graphic elements in LaTeX. It is used mainly for scientific diagrams, although users have found other use cases, notably for drawings and cartoon characters. We define a *variant* as a piece of TikZ code on which a *patch* has been applied. A patch is defined as the edits (*i.e.*, changes) between two versions of a code represented in the unidiff format, it is the same format used by Git when displaying changes between commits.

2.2 Motivating Example

To illustrate our work, we consider an example consisting in a TikZ code snippet that generates an image of a bee with two wings on each side, as shown on the left side of Figure 1. Although the bee is a simple drawing, the underlying TikZ code is not trivial to understand and manipulate, especially for developers without prior expertise. Let us suppose a developer wants to customize the bee's design by modifying the TikZ code. One way to proceed is by leveraging an LLM, providing it the existing TikZ code together with a change instruction, such as "*Add a third pair of wings to the bee*". The instruction, though simple in its intent, requires that the LLM perform multiple non-trivial sub-tasks. First, it must *understand the instruction*: What does "a third pair of wings to the bee" mean? While a human perfectly gets the intent, this is not necessarily the case for an LLM. Next, it must *identify the relevant feature*: Where are the wings defined within the TikZ code? Assuming the LLM properly interprets the instruction, it must locate the corresponding code fragment that requires modification. Finally, it must *perform the correct modification*: What changes need to be made in the code? Once the relevant lines of code are identified, the LLM must determine the appropriate additions, deletions, or modifications to reach the intended result.

Possible mistakes an LLM can make in these sub-tasks are illustrated on the right side of Figure 1. First, the worst-case scenario is generating code that fails to compile. Second, the LLM might misinterpret the instruction, leading it to modify an unintended feature, such as increasing the size of the antennas instead of adding wings. Third, it may fail to identify the relevant feature, which could result

[1] https://huggingface.co/datasets/CharlyR/vtikz
[2] https://github.com/IV2C/VTikZ

```
\tikzset{
bee/.pic={
\begin{scope}[x=3cm/480,y=3cm/480, rotate=-45, shift={(270:48)}]
\reflect[
  on left= {colors alias={body as BlueGrey800, stripes as Amber500}},
  on right={colors alias={body as BlueGrey900, stripes as Amber700}, half clip},
  lower wing/.style={fill=BlueGrey100}, upper wing/.style={fill=BlueGrey50},
  third pair of wings/.style={fill=Blue}]{
  \fill [body] (0,-160)
    .. controls ++(120:64) and ++(270:64) .. (-88, -16)
    .. controls ++( 90:64) and ++(225:64) .. (  0, 128)
    .. controls ++(315:64) and ++( 90:64) .. ( 88, -16)
    .. controls ++(270:64) and ++( 60:64) .. cycle;
  \fill [body] (0,128) ellipse [x radius=80, y radius=56];
  \fill [body] (128,222) ellipse [x radius=80, y radius=56];
  \fill [body]
    (32,160) arc (180:90:64) -- ++(6,-6) coordinate [midway] (@)
    arc (90:180:64) -- cycle;
  \fill [body] (@) circle [radius=12];
  \begin{scope}
    \clip (0,-160)
      .. controls ++(120:64) and ++(270:64) .. (-88, -16)
      .. controls ++( 90:64) and ++(225:64) .. (  0, 128)
      .. controls ++(315:64) and ++( 90:64) .. ( 88, -16)
      .. controls ++(270:64) and ++( 60:64) .. cycle;
    \foreach \i in {0,...,2}
      \fill [stripes] (-256, -160 + \i*80) rectangle ++(512, 40);
  \end{scope}
  \foreach \s [count=\i from -1] in {lower wing, upper wing}
    \fill [style=\s, shift={(16,56)}, rotate=\i*32]
      (0,0)
      .. controls ++( 30:64) and ++(180:32) .. (128,56)
      arc (90:-90:56)
      .. controls ++(180:32) and ++(330:64) .. cycle;
}
\end{scope}}
}
```

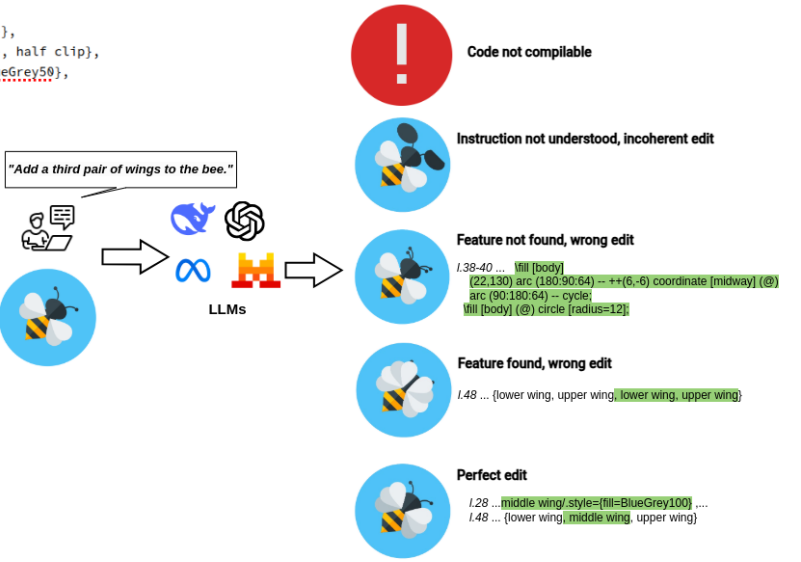


Figure 1: Contextualized example of the benchmark task.

in unintended modifications like adding an extra pair of antennas. Fourth, even when the relevant feature is correctly identified, the LLM could make incorrect modifications, for example by introducing two additional pairs of wings instead of just one. Fifth and finally, the LLM might overlook contextual subtleties; even when the edit is correct, details such as ensuring the shading of the newly added wings remains consistent with the original design must still be addressed.

These cases show how challenging customizing TikZ code is. However, even after completing these tasks, another challenge remains: the modifications made by an LLM may introduce subtle deviations from a theoretically perfect solution. For example, the newly added bee wings might exhibit slight color variations or be misaligned. Such cases complicate the assessment of modifications and make the definition of a precise oracle for evaluating correctness challenging. Addressing this issue is an objective of our work.

3 DATASET

This section details the dataset used in our benchmark, which we refer to as the *vTikZ* dataset. It consists of a total of 100 human-made TikZ variants, each derived from an original TikZ code collected from the internet, along with the corresponding instruction describing the edit used to create the variant. The following sections describe the dataset's selection, curation process, format, and content.

3.1 Selection and Curation of TikZ code

This section details the protocol we followed to select and curate the set of TikZ codes. The creation of such a dataset involved identifying TikZ code examples that exhibited both *identifiable features*

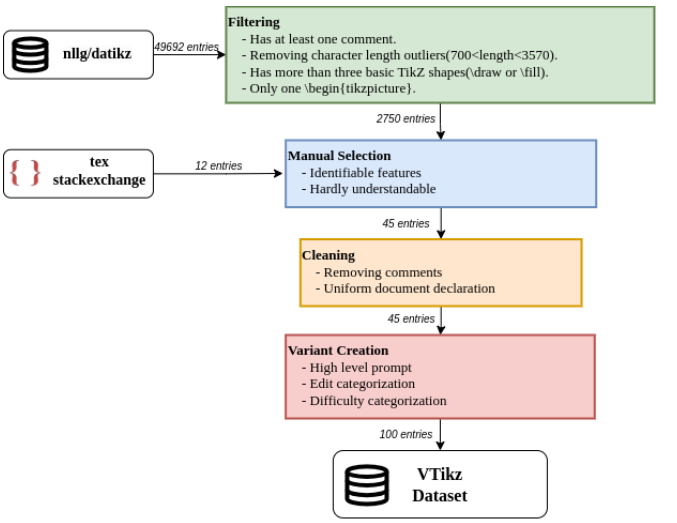


Figure 2: vTikZ dataset curation overview

and *inherent difficulty*, making them challenging for end-users to understand at first sight. Such examples illustrate the complexity of TikZ while maintaining a well-defined structure. They also support high-level prompts specifying feature edits in the diagram, *e.g.*, *"Make the eye of the dog red."*. Figure 2 shows the curation protocol we followed. We collected data from two repositories, *i.e.*, animal

drawings from a post on the LaTeX StackExchange[3], and scientific diagrams from the nllg/datikz[4] dataset. To ensure the selected TikZ code was well-suited for our benchmark's customization task, we applied the following filtering criteria to the nllg/datikz dataset:

- **At least one comment**, as comments were more likely to indicate higher-level attributes.
- **No character length outliers**, by removing codes with fewer than 700 characters or more than 3570. Computed with $[Q_1 - 1.5 \times \text{IQR},\ Q_3 + 1.5 \times \text{IQR}]$.
- **More than 3 basic TikZ shapes**, by limiting to codes primarily composed of TikZ elements (with three commands `\fill` or `\draw`).
- **Only one `\begin{tikzpicture}` command**, ensuring a single TikZ diagram is generated.

We developed these criteria through an initial manual exploration of the data, iteratively refining them based on the structural characteristics of the analyzed TikZ code. This process resulted in 12 codes from StackExchange and 2750 from the nllg/datikz dataset. The remaining dataset creation was conducted by one annotator, who was responsible for creating the customizations. To mitigate the bias induced by relying on a single annotator, the created data was reviewed and discussed with four software engineering experts. While this could introduce a new source of bias, the simplicity of the customizations, combined with the ability to visually compare the results, ensures that there is little to no ambiguity in the potential solutions, making the solutions straightforward to review. Through a random exploration of the remaining 2750 codes, we manually selected 35 codes that were more prone to being customized, and handpicked 10 codes from the 12 initial ones from stackexchange, for a total of 45 initial codes. For each TikZ code, we manually created one or more variants, with different types of changes and difficulty, leading to a total of 100 variants.

3.2 Format

The format of our dataset includes both human-made and automatically computed features, as detailed in Table 1. The human-made features for each customization consist of: the *original code* in TikZ, a list of *parameterized solution codes*, the *instruction*, the *perceived difficulty*, the *description of the resulting modification*, the *type of diagram* and the *type of modification*.

From these human-made features, we derive the following computed features:

- **Perfect Variants**: A list of theoretically perfect *variants* that fully implement the instruction.
- **Patch**: A list of string representations of the edits in Unidiff format (without context), computed by comparing the original TikZ code with its modified variants.
- **Image Input**: A rendered image of the original TikZ code.
- **Images Solution**: A list of rendered images generated from the theoretically perfect *variants*' code.
- **AST Difficulty**: The Tree Edit Distance between the original and variants TikZ codes, computed using the algorithm proposed by Zhang and Shasha [42].

[3]https://tex.stackexchange.com/a/414059
[4]https://huggingface.co/datasets/nllg/datikz

Table 1: Human-made and computed features in the vTikZ Dataset.

Human-made Features	
Original Code	Source code provided by humans.
Parameterized Solution Code	A list of code parameterized to encompass valid solutions.
Instruction	Edit instruction.
Perceived Difficulty	Subjective difficulty of the edit for each possible variant code.
Result Description	Description of the edit outcome.
Modification Type	Type of modification applied by the prompt(update, add, remove).
Type	Type of the original diagram, either "scientific" or "animal".
Computed Features	
Perfect Variants	List of theoretically perfect variants, computed from the parameterized solution code.
Patch	Unidiff patches (without context) between original and theoretically perfect variants.
Image Input	Image rendered from the original code.
Images Solution	List of images rendered from the theoretically perfect variants.
AST Difficulty	Tree Edit Distance between original and variant [42].

These features are used in the evaluation and analysis phase of our benchmark, which we both further detail in Section 4 and 5.

As mentioned earlier, multiple solutions can be valid. For example, when instructing an LLM to change a drawing element to red, various shades of red are acceptable. Similarly, when asking to move a drawing to the right, the distance should be parameterizable, as multiple shifts can be valid. To cover as many valid solutions as possible in our evaluation, we introduce four parameterization statements that can be added to TikZ code:

- `§range(lower, higher, default)`: Specifies a valid numerical range between *lower* and *higher*, with a *default* value. Multiple valid radius sizes can thus be defined as follows:
  `\fill [Red600] (56, 0) circle [radius=§range(10, 30, 20)];`
- `§rangei(value, interval)`: Functions similarly to `§range` but defines a range using an interval around a central value, equivalent to `§range(interval-value, interval+value, value)`.
  The previous radius can be parameterized as follows:
  `\fill [Red600] (56, 0) circle [radius=§rangei(20,10)];`
- `§choice([A,B,C,...], default)`: Specifies a list of valid values $A, B, C, \ldots$.
  `\fill [Red§choice([600,700],600)] (56, 0) circle [radius=20];`
- `§def(value)`: Defines a variable name. Since different TikZ codes may use distinct variable names while remaining functionally equivalent, any code with a different variable name at `§def` is considered identical as long as the variable is used in the same locations. A variable name can be defined as follows:
  `\definecolor{§def(myred)}{rgb}{0.9,0.2,0.2}`
  `\fill [myred] (56, 0) circle [radius=20];`

During the creation of the dataset, the default values defined above are used to generate the theoretically perfect variant.

However, parameterized code alone cannot encompass every solution. To address this, we provide multiple parameterized variants that implement the same solution in different alternatives. For instance, instead of simply changing a color to red, TikZ allows defining a custom color using `\definecolor` and reusing it later. Since a single

Table 2: Dataset Summary

Edit Categories		**Difficulty Levels**	
Scientific Edits	50	Easy Edits	41
Animals Edits	50	Medium Edits	36
		Hard Edits	23
		Avg. AST Difficulty	20.42 (1–160)
Dataset Metrics		**Code Metrics**	
Number of Codes	45	Avg. Lines of Code	63 (25–104)
Avg. Edits per Code	2.22	Avg. Characters	2167 (918–3323)
Edit Types			
Add	25		
Remove	13		
Update	62		

parameterization cannot cover both approaches, we can include multiple variants: one applying the red color and another defining and referencing a custom color.

3.3 Descriptive Analysis

We now present an analysis of the dataset (Table 2), demonstrating its scope and diversity through statistics, including the number of edits, lines of code, difficulty levels, and types of modifications.

Each customization in the dataset falls into one of two main categories-*Scientific* or *Animals*-depending on the source of the input code, with 50 customizations in each category. The 45 initial codes in the vTikZ dataset contain, on average, 2.22 customizations each. These customizations are classified into three types, *add, remove, or update*, with respective counts of 25, 13, and 62. These customizations are mainly structural, involving layout, colors, sizes, and shapes.

In total, the dataset contains in 41 easy, 36 medium, and 23 hard customizations, classified by the "perceived difficulty" feature. The computed AST difficulty averages 20.42, ranging from a minimum of 1 to a maximum of 160. The input codes have an average of 63 lines of code and 2,167 characters, with lengths varying between 25 and 104 lines of code and character counts ranging from 918 to 3,323. Although most customizations may be considered straightforward for an experienced user, editing TikZ code presents two major challenges. First, the model must accurately *identify relevant feature(s)* based on the user's instructions, a non-trivial task given the complexity and abstract nature of TikZ. Second, after correctly locating relevant lines of code, the LLM must *perform the correct modification*, ensuring that changes do not disrupt the interdependent structure of the code. As we will demonstrate, LLMs struggle even with seemingly simple customizations.

4 BENCHMARK

Evaluating the code-editing capabilities of LLMs plays a major role in the Software Engineering community, ensuring among other things, the reliability, quality, and compliance of generated code. In this section, we present the *vTikZ* benchmark, in which LLMs are prompted to generate TikZ code variants based on a given instruction. Through this task, we assess the reliability of LLMs and evaluate to which extent the generated code variants comply with the given instruction.

In the remainder of the section, we first present the evaluation metrics, followed by an explanation of the task, and finally, we provide technical details regarding the benchmark.

4.1 Evaluation metrics

We employ five metrics to comprehensively assess the quality of a customized code variant compared to the reference solution.

Primary Metrics.

- **CompileMetric**: A binary metric (0 or 100), indicating whether the generated code compiles successfully.
- **LocationMetric**: A binary metric (0 or 100), indicating whether 100% of the lines of a patch was edited.
- **SuccessCustomizationMetric**: A binary metric (0 or 100), evaluating whether the generated code matches a parameterized solution or produces an exact image match.

Complementary Metrics.

- **SimilarityMetric**: Measures similarity between patches using CrystalBleu [13].
- **LineMetric**: Computes the percentage of correctly edited lines, defined as:

$$\text{LineMetric} = \frac{100 \times \text{Number of correctly edited lines}}{\text{Total number of edited lines in the reference solution}}$$

4.2 Task

Given an input TikZ code c, an instruction p, and a rendered image $R(c)$ from the original code c (optionally provided as input to the model), the *LLM* generates a variant v

$$v = LLM(c, p, R(c)),$$

Because of the probabilistic nature of Large Language Models, we evaluate their performance using Best-Of-N sampling, generating N variants for each code-instruction pair c, p while varying the temperature t:

$$v_i = LLM^t(c, p, R(c)) \quad \text{for } i = 1, \ldots, N.$$

Given a single generated variant v_a, we compare it against a set of m reference solutions, each represented as a tuple of a template t_i and a theoretically perfect variant r_i: $\{(t_i, r_i)\}_{i=0}^{m}$. To determine the most relevant reference tuple for comparison, we compute each metric M_d as follows:

$$M_d^i = \text{metric}(v_a, (t_i, r_i)) \quad \text{for} \quad i = 0, 1, 2, \ldots, m$$

We then rank the reference solutions in descending order based on their metric scores (see Section 4.1), selecting the tuple that yields the highest scores. The ranking relies on the following order of priority applied to the primary metrics: SuccessCustomizationMetric, LocationMetric, LineMetric, SimilarityMetric, CompileMetric. For this variant v_a, we then only consider the metrics' scores for the tuple that yields the best scores. Finally, for each variant v_i in the N tries, we apply the same ranking process to select the best-performing variant.

Following this process, we selected among the N generated variants the one that yields the best scores by comparing each with each possible solution.

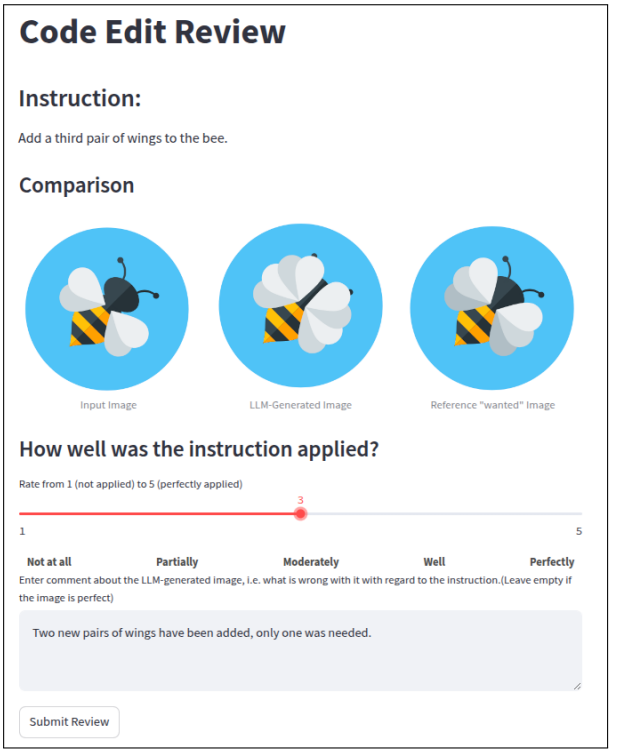


Figure 3: Annotation interface

4.3 Technical Details

The benchmark is designed to evaluate both LLMs and Large Multimodal Models (LMMs) in a comparable manner. In addition to textual inputs, LMMs can also process images, making them suitable for TikZ-based image modifications. Both model types leverage system prompts, ensuring a better alignment with the benchmark objectives. LLM and LMM system prompts are structured as follows:

> *You are an expert coding assistant specialized in modifying file contents based on instructions.*
> *Given an instruction and file content, respond only with the updated file's full content, ensuring it is entirely enclosed between code tags like this*
> *“‘*
> *content*
> *”’*
> *Provide no additional text or explanations beyond the code tags.*

For LMMs, when the image is also provided, the system prompt is modified to include information about the provided image:

> *[...] Given an instruction, file content, and the image that the current file creates, [...]*

All models were evaluated through API inference providers, using the temperature setting provided in Table 3 and Figure 3.

4.4 Benchmark Refinement and Validation

We updated our benchmark in two phases: first by refining the evaluation metrics, then by validating and expanding the set of ground truths through human feedback.

The initial version focused on a limited set of text-to-text and image-to-image metrics applied to a reduced Large Language Model set. LLM-generated variants were scored on a five-point scale based on visual inspection via an early version of our annotation interface (see Figure 3). This led to the classification of 745 variants and revealed weaknesses in metric coverage—particularly in identifying valid solutions overlooked by automated measures. These findings motivated the introduction of a *parameterized solution framework*, allowing multiple valid solutions per task and better reflecting the structured nature of code edits.

Building on this framework, we conducted a second evaluation using a redesigned annotation interface (also shown in Figure 3). One annotator provided both numerical scores and qualitative feedback for each variant. Out of all generated outputs—selected and unselected across multiple attempts—2,525 were compilable. From these, we created a new dataset with 300 annotated instances.

Each entry includes two new fields: `human_score`, reflecting correctness and completeness, and `human_comment`, capturing specific strengths or errors. This annotation process proved instrumental in validating and refining the benchmark, as it surfaced 13 additional correct solutions that were previously missed.

5 RESULTS AND ANALYSIS

In this section, we evaluate state-of-the-art LLMs on our constructed dataset using the defined metrics, producing a benchmark that we then analyze. Our benchmark includes five recently released open-source LLMs, namely DeepSeek-R1-Distill-Llama-70B, Llama-3.1-8B, Llama-3.3-70B, Llama-3-70B, all in their instruct versions, and one closed source model, *i.e.*, GPT-4o(2024-08-06), evaluated both with and without an image input. To ensure a fair comparison, we selected a temperature of 0.7 for most evaluations, as this is a commonly used default parameter among inference providers, mimicking the typical user experience. Our analysis aims to answer the following research question: *To what extent can LLMs generate successful code customizations?*

5.1 Variant Classification

To assess how well LLMs can generate successful code customizations, we classified the variants using the Compile, Location, and SuccessCustomization metrics, as shown in Figure 4.

The Compile Metric gives promising results, especially for the GPT-4o at $k = 5$, the only one with an average score of 100. Other models are close behind, the furthest being llama-3.1-8b, achieving 82. For the Location Metric, GPT-4o manages to edit the right lines on average 60% of the time, while the other models lack behind with 45% for DeepSeek-R1-Distill-Llama-70B at $k = 5$ with a temperature of 1.5. The SuccessCustomizationMetric only goes up to an average of 28 with GPT-4o, the Llama-3.3-70B and DeepSeek-R1-Distill-Llama-70B models achieve closer results going from 13 to 16.

5.2 Variants Similarity To Solution

The classification from the previous section defines which variant constitutes a solution. In this section, we evaluate how close a variant is to a solution by presenting and analyzing the similarity and line scores.

The scores are presented for each model in Table 3, along with the number of k attempts and temperature settings. The Modalities used

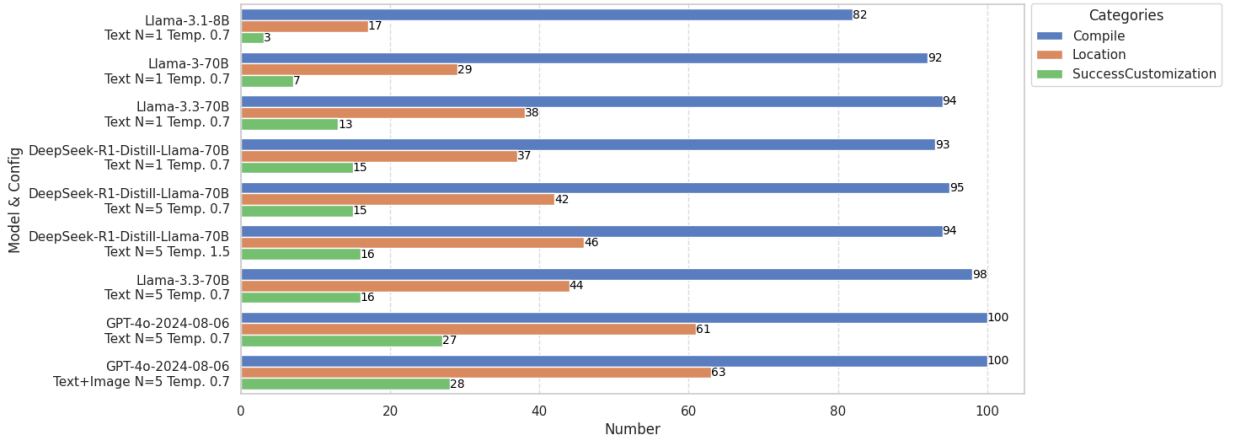


Figure 4: Classification of Variants using the Compile, Location, and SuccessCustomization metrics

Table 3: Average similarity and line scores by configuration

Model	Modality	N	Temp.	Similarity	Line
Llama-3.1-8B	Text	1	0.7	18.2	31.4
Mixtral-8x7B	Text	1	0.7	17.4	30.7
Llama-3-70B	Text	1	0.7	29.3	45.6
Llama-3.3-70B	Text	1	0.7	37.9	54.5
DeepSeek-R1-Distill-Llama-70B	Text	1	0.7	35.8	48.1
DeepSeek-R1-Distill-Llama-70B	Text	5	0.7	39.0	58.4
DeepSeek-R1-Distill-Llama-70B	Text	5	1.5	36.9	57.8
Llama-3.3-70B	Text	5	0.7	41.7	60.6
GPT-4o-2024-08-06	Text	5	0.7	55.7	75.0
GPT-4o-2024-08-06	Text+Image	5	0.7	57.0	75.3

are either Text only(T) or Text and Image(T+I), which only applies to GPT-4o. We observe that the Similarity Metric –computed using the CrystalBLEU score between the two patches– exhibits a similarity of 57% for GPT-4o, significantly outperforming all other models. On the lower end, the Mixtral-8x7B and Llama-3.1-8B models achieve similarity scores around 18%.

Regarding the Line Metric, which measures the percentage of correctly edited lines, GPT-4o achieves an average accuracy of 75.3%, while other models under-perform, with the closest being Llama-3.3-70B at 60.6%.

5.3 Observations and Analysis

Although the exact size of the GPT-4o model is not publicly available, the scores measured with our benchmark show an upward trend with the number of parameters of the models. Larger models tend to perform better, particularly newer models like Llama-3.3-70B which achieves double the SuccessCustomization metric score of Llama-3-70B despite having the same number of parameters.

Surprisingly, the reasoning-focused model DeepSeek-R1-Distill-Llama-70B exhibits similar results to its non-reasoning counterpart, suggesting that reasoning capabilities do not significantly enhance TikZ editing performance.

Giving the input image to the GPT-4o model resulted in better scores overall, and the number of successful edits decreased marginally (by only one) when solely relying on code inputs. This suggests a potential lack of multimodal consistency, at least in the context of TikZ diagrams.

We observe that the SuccessCustomization metric unfortunately varies from 3% to 28% of the solutions, which emphasizes the current limits of LLMs in customizing TikZ code. Regarding the Location and Compile metrics, they vary respectively from 17% to 63% and from 82% to 100%. This shows nonetheless the existing potential in succeeding in such as task, but will require a more innovative approach to ensure a more reliable and successful customization. Overall, even with the best-performing models evaluated at $N = 5$, the results remain disappointing. The models correctly edit the proper code only 28% of the time, which is insufficient for real-world usage.

6 DISCUSSION

6.1 Benchmark Extension using Human-in-the-loop Feedback

The creation of the reviewing tool (see Section 4.4 and Figure 3) not only refined our evaluation pipeline but also extended the benchmark's capabilities by introducing human-centric insights and annotations. By systematically adding `human_score` and `human_comment` fields, the dataset now goes beyond metric-based evaluation to include nuanced qualitative assessments. These additions can enhance the interpretability and reliability of evaluation results.

Importantly, this human-augmented extension is *optional* and not part of the evaluation pipeline described earlier; rather, it provides additional data that can be leveraged for further research and analysis. In particular, this enriched dataset augments the original vTikZ dataset with both metric outputs and structured human feedback, making it a valuable resource for fine-tuning models to better align with human judgment; conducting error analysis and failure mode discovery; exploring the space of valid code transformations.

Moreover, the reviewing tool we developed can be reused to augment our datasets with further fine-grained human feedback. Beyond static evaluation, it also enables the investigation of interactive scenarios where LLMs iteratively incorporate feedback to guide their code generation process.

6.2 LLM failure mode

In the following section we list reasons why LLMs fail to make the right edit to the code using the already compileable 300 annotated variants. Among these annotated variants, 75 were right edits, for the remaining 225 ones, the major failure reasons were:

- *Feature found, wrong edit*(65) - The LLM finds what to edit but does not manage to make the right edit.
- *Too many features edited*(19) - The LLM correctly edits the target but also alters unrelated parts, likely to maximize chances of success, but ending up making an unwanted customization.
- *Not all feature found*(41) - The LLM only partially applies the edit; e.g., it removes a container but does not remove all the content within the container.
- *Feature not found*(67) - The LLM does not find the feature, for example, editing the color of the wrong feature.
- *instruction/code not understood*(20) – Cases where the LLM fails to understand the instruction or the code. It's unclear whether the model is unsure or simply cannot apply the instruction. For example, when asked to "make the shark blue", it sometimes colors all features blue—either from misunderstanding what to color or from literal interpretation.
- *No modifications*(13) - Happening with smaller and less performing models, that sometimes did not change anything in the original code.

6.3 Limitations

Best-Of-N measurements. Due to the inherent randomness of LLMs, measurements with $N = 1$ at a nonzero temperature may not accurately reflect the model's average performance. To mitigate this, we also provide evaluations using $N = 5$ for the Llama-3.3-70B, DeepSeek-R1-Distill-Llama-70B, and GPT-4o models.

Parameterized evaluation. The parameterization matching in Section 3.2 uses four commands, which may not cover cases where certain lines are optional or line placement does not affect correctness. These commands were defined during an initial data exploration phase, where we analyzed variants classified as perfect by human evaluators, despite differences from the reference perfect solution. When these commands could not capture all variations, we duplicated the solution and manually created alternative variants.

vTikZ dataset. The vTikZ dataset consists of 100 manually curated examples. While this is relatively small compared to datasets such as nllg/datikz, it provides high-quality, handcrafted examples representative of customization scenarios. Our evaluation shows that LLMs can successfully solve up to 28% of these examples with $k = 5$, highlighting fundamental limitations in their capabilities and underscoring the relevance of our benchmark.

TikZ scope. The benchmark targets TikZ which arguably does not require full software engineering (SE) expertise. But it provides a testbed for evaluating cross-modal consistency in code customization – a key concern in, e.g., front-end development or end-user programming. Although the task does not capture the full complexity of SE projects spanning multiple files and artifacts, strong performance on vTikZ would offer an encouraging signal for broader SE challenges.

You are an image classification agent. Your role is to evaluate whether a given instruction has been correctly applied to an image. You are given the original image, the modified image, and an instruction.
Response Format:
- Provide a step-by-step analysis of the image in relation to the instruction.
- Conclude your response with either <YES> or <NO> on a new line, depending on whether the instruction was applied.
- Ensure that <YES> or <NO> is enclosed within less than (<) and greater than (>) signs and appears on a separate line at the end of the response.
- Ensure the less than (<) and greater than (>) signs are only used at the end of the response and nowhere else.
Was the instruction "{instruction}" applied to the image?

Figure 5: Prompt used for the evaluation of VLM/Multimodal models to serve as oracles

6.4 Limits and Challenges of Integrating Vision into LLM-based Code Customization

Integrating vision modules in LLM-based code customization is a promising direction to enhance performances of LLMs in the dataset's task. This section discusses the potential benefits and challenges associated with such an approach.

To integrate vision into LLM-based customization, several approaches can be leveraged, including Vision-Language Models (VLMs), multimodal LLMs, and object detection techniques. These methods can help assessing whether an image created from a generated code is a solution to the given instruction. In the context of an agentic system, a generated solution could be evaluated in a self-refinement protocol, however, each approach presents distinct limitations, which we examine in this section.

Using our benchmark solutions as ground truth, we conducted two preliminary tests to evaluate the potential of VLMs and multimodal models as oracles. The goal was to determine whether these models could correctly classify a reference solution as one that applies the prompt. The prompt illustrated in Figure 5 was used for both tests.

VLM limitations in TikZ code verification tasks. In our initial test, we provided Llama3.2-Vision with correct image solutions and asked if the instruction was applied. The model incorrectly flagged 51 out of 100 instructions as not applied. However, some instructions, like "Make the cat wider," require the original image for assessment, so we conducted a second test using GPT-4o-mini with both original and edited images, in which 28 out of 100 instructions were misclassified. A similar evaluation on incorrect solutions showed that LLaMA-90B-Vision misclassified 48 out of 232, while GPT-4o-mini misclassified 20. Although these models perform better in identifying incorrect solutions, they still exhibit limitations, showing they are unreliable for TikZ code verification.

Challenges in using object detection for code customization. Another approach involves leveraging object detection solutions to identify regions that should be edited. However, this approach exhibits several limitations. First, many object detection models operate either with a predefined and limited vocabulary, or through zero-shot object detection. The former cannot identify specific features, like "eyes," while the latter requires dynamic vocabulary definition, adding complexity. Using zero-shot detection as an oracle also presents challenges:

even if regions are identified, verifying changes is difficult. Image alterations may lead to false positives, and edits outside the defined region can result in false negatives. Solutions such as Gemini Spatial[5] and OmniParser [28] offer similar capabilities but are subject to the same limitations. Finally, even if a perfect oracle were to exist, another challenge arises in mapping image features to the code itself. While languages like HTML allow inspecting elements to find their corresponding definitions in the code, this feature is not available in TikZ and cannot be assumed in all languages.

7 RELATED WORK

Repository-level code edit Benchmark. Several benchmarks evaluate LLMs on end-to-end software development at the repository level, often leveraging existing GitHub issues, such as SWE-Bench [21], SWE-Bench+ [2], REPOCOD [24], and RES-Q [23]. While related to our work, these benchmarks primarily assess autonomous agents executing technical, code-related instructions and do not focus on the visual output of code.

Code generation benchmarks from text instruction. LLMs are widely evaluated on code generation tasks, exemplified by MBPP [4], HumanEval [10], and many others [15, 17, 20, 34, 45]. These benchmarks primarily assess full-code generation for programming challenges. Some works extend this evaluation to codes with graphical outputs, such as web frameworks [12], mobile/desktop applications [43], plots [14, 45] and SVG [29, 38]. However, they focus on generating complete code rather than editing existing code.

Code generation benchmark from multimodal instruction. Certain benchmarks evaluate LLMs on code generation from multimodal inputs. HumanEval-V [41] introduces images alongside textual instructions for coding challenges, Design2Code [30] evaluates website generation from images, and Detikzify [8] assesses TikZ code generation from hand-drawn sketches. Additionally, Wei et al. [35] evaluates TikZ code generation from textual or visual inputs. While closely related to our work, these benchmarks assess code generation capabilities, whereas we focus on code editing.

Graphic code generation solutions. Some approaches generate or edit graphical code based on high-level prompts, particularly for animations or creative coding, using custom LLM-based interfaces [3, 32] or specialized frameworks [26]. Many works have also been conducted in the context of SVG[6, 29, 37–39]. Other works, such as Automatikz [7] and Diagrammer [40], incorporate planning and self-refinement loops for structured diagram generation. These approaches however focused on full-code generation out of textual instructions. The customization task demands a precise understanding of both code structure and user intent to make targeted edits without breaking the original diagram. For instance, Belouadi et al. [7] did fine-tune a model on TikZ code and created their own model CLiMA, but our preliminary experiments revealed they did not work for code edition, generating completely different codes than the one provided.

Visual and editing code benchmarks. Image generation is a well-established field, with numerous benchmarks assessing text-to-image generation [5, 11, 16, 19]. However, these benchmarks are unrelated to code generation or editing, which is the focus of our contribution. To the best of our knowledge, Wei et al. [35] is the only work that partially evaluates code editing, but it does so by regenerating code from an image before applying edits, rather than directly modifying existing code. The area of code customization remains under-explored, particularly for TikZ code, which our contribution is the first to address to the best of our knowledge.

Web agents code benchmarks. The notion of autonomous web agents is an active research area aiming to develop AIs capable of navigating and acting on the web to accomplish complex tasks in natural language [22]. Realistic testing environments and rigorous benchmarks like WebArena [44] and WebGames [31] are essential for evaluating their progress and identifying the challenges that need to be overcome. Platforms like OpenHands (formerly OpenDevin) [33] facilitate the creation and experimentation of these web agents. Our benchmark on TikZ provides another code-centric task that one might find on the Web for vector image creation applications. It provides a mean of evaluating the ability of future WebAgents to work on these kinds of applications.

8 CONCLUSION

In this paper, we challenged LLMs' capabilities in customizing TikZ code from end-users instructions in natural language. This task requires accurately identifying the relevant parts of the source code to customize, and generating a syntactically correct variant that aligns with the user's intended visual outcome. We first presented a dataset of TikZ customization scenarios, covering a wide range of edit types and complexities. This dataset serves as the backbone of the vTikZ Benchmark, which enables a systematic evaluation of state-of-the-art LLMs. As part of this benchmark, we introduced a parameterized ground-truth framework to rigorously assess the quality of generated TikZ code. Additionally, we contributed with a reviewing tool and a collection of 300 annotated LLM-generated outputs. Our evaluation shows that current LLMs perform poorly in TikZ customization, achieving the desired output in only 13% of cases in one-shot and 28% of cases despite having 5 tries to succeed. These results highlight limitations of existing models and the need for more advanced self-refining solutions, possibly leveraging vision models and agent-based approaches. This work thus opens room for multimodal LLM-based solutions for code.

Benchmark evolution and policy. The results presented rely on version 1.0 of the benchmark, which is subject to future revisions. Planned updates include expanding the range of TikZ code and editing scenarios, potentially incorporating missing correct solutions, and establishing a leaderboard. A split between public and private benchmark instances is also under consideration.

Future work. Multiple perspectives can be explored in the context of graphical language editing. While TikZ serves as a strong testbed, extending the evaluation to additional languages or libraries—such as SVG, P5.js, Turtle, or Matplotlib—would enhance the generalizability of our findings. Beyond standalone graphical languages, future research could investigate code editing capabilities in domains such as game development, mobile applications, or web interfaces, that typically involve large code bases. These applications may require evaluating more complex systems, including autonomous agents, rather than solely LLMs. Our findings indicate that existing solutions may be insufficient to guarantee correct visual outputs. Future work could focus on developing more advanced approaches, such as agent-based systems or iterative refinement strategies, to improve the accuracy and reliability of code edits.

Acknowledgements. This work is supported by the Inria Défi LLM4Code.

[5] https://github.com/google-gemini/starter-applets/tree/main

REFERENCES

[1] Mathieu Acher, José Galindo Duarte, and Jean-Marc Jézéquel. 2023. On Programming Variability with Large Language Model-based Assistant *(SPLC '23, Vol. A)*. 8–14. https://dl.acm.org/doi/10.1145/3579027.3608972

[2] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992 version: 1.

[3] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts *(UIST '23)*. New York, NY, USA, 1–22.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

[5] Eslam Mohamed Bakr, Pengzhan Sun, Xiaoqian Shen, Faizan Farooq Khan, Li Erran Li, and Mohamed Elhoseiny. 2023. HRS-Bench: Holistic, Reliable and Scalable Benchmark for Text-to-Image Models. arXiv:2304.05390.

[6] Ayan Banerjee, Nityanand Mathur, Josep Lladós, Umapada Pal, and Anjan Dutta. 2024. SVGCraft: Beyond Single Object Text-to-SVG Synthesis with Comprehensive Canvas Layout. arXiv:2404.00412.

[7] Jonas Belouadi, Anne Lauscher, and Steffen Eger. 2024. AutomaTikZ: Text-Guided Synthesis of Scientific Vector Graphics with TikZ. arXiv:2310.00367 [cs].

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

[9] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712.

[10] Mark Chen and al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

[11] Jaemin Cho, Abhay Zala, and Mohit Bansal. 2023. Visual Programming for Text-to-Image Generation and Evaluation. arXiv:2305.15328.

[12] Yi Cui. 2024. WebApp1K: A Practical Code-Generation Benchmark for Web App Development. arXiv:2408.00019 version: 1.

[13] Aryaz Eghbali and Michael Pradel. 2023. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code *(ASE '22)*. New York, NY, USA, 1–12.

[14] Kanika Goswami, Puneet Mathur, Ryan Rossi, and Franck Dernoncourt. 2025. PlotGen: Multi-Agent LLM-based Scientific Data Visualization via Multimodal Feedback. arXiv:2502.00988 version: 1.

[15] Patrick Haller, Jonas Golde, and Alan Akbik. 2024. PECC: Problem Extraction and Coding Challenges. arXiv:2404.18766.

[16] Shuhao Han, Haotian Fan, Jiachen Fu, Liang Li, Tao Li, Junhui Cui, Yunqiu Wang, Yang Tai, Jingwei Sun, Chunle Guo, and Chongyi Li. 2024. EvalMuse-40K: A Reliable and Fine-Grained Benchmark with Comprehensive Human Annotations for Text-to-Image Generation Model Evaluation. arXiv:2412.18150.

[17] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. arXiv:2105.09938.

[18] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (Dec. 2024), 220:1–220:79. https://dl.acm.org/doi/10.1145/3695988

[19] Kaiyi Huang, Kaiyue Sun, Enze Xie, Zhenguo Li, and Xihui Liu. 2023. T2I-CompBench: A Comprehensive Benchmark for Open-world Compositional Text-to-image Generation. arXiv:2307.06350.

[20] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974.

[21] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770.

[22] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. AI Agents That Matter. arXiv:2407.01502 [cs].

[23] Beck LaBash, August Rosedale, Alex Reents, Lucas Negritto, and Colin Wiel. 2024. RES-Q: Evaluating Code-Editing Large Language Model Systems at the Repository Scale. arXiv:2406.16801.

[24] Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. 2024. Can Language Models Replace Programmers? REPOCOD Says 'Not Yet'. arXiv:2410.21647.

[25] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv:2409.02977.

[26] Vivian Liu, Rubaiat Habib Kazi, Li-Yi Wei, Matthew Fisher, Timothy Langlois, Seth Walker, and Lydia Chilton. 2024. LogoMotion: Visually Grounded Code Generation for Content-Aware Animation. arXiv:2405.07065.

[27] Yongkun Liu, Jiachi Chen, Tingting Bi, John Grundy, Yanlin Wang, Jianxing Yu, Ting Chen, Yutian Tang, and Zibin Zheng. 2024. An Empirical Study on Low Code Programming using Traditional vs Large Language Model Support. arXiv:2402.01156.

[28] Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. OmniParser for Pure Vision Based GUI Agent. arXiv:2408.00203 [cs.CV] arXiv:2408.00203.

[29] Juan A. Rodriguez, Abhay Puri, Shubham Agarwal, Issam H. Laradji, Pau Rodriguez, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. 2024. StarVector: Generating Scalable Vector Graphics Code from Images and Text. arXiv:2312.11556.

[30] Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2Code: How Far Are We From Automating Front-End Engineering? arXiv:2403.03163.

[31] George Thomas, Alex J. Chan, Jikun Kang, Wenqi Wu, Filippos Christianos, Fraser Greenlee, Andy Toulis, and Marvin Purtorab. 2025. WebGames: Challenging General-Purpose Web-Browsing AI Agents. arXiv:2502.18356 [cs].

[32] Tiffany Tseng, Ruijia Cheng, and Jeffrey Nichols. 2024. Keyframer: Empowering Animation Design using Large Language Models. arXiv:2402.06071.

[33] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs].

[34] Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. 2023. MCoNaLa: A Benchmark for Code Generation from Multiple Natural Languages. arXiv:2203.08388.

[35] 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. arXiv:2411.11916.

[36] Simon Willison. 2025. Notes on Google's Gemma 3. https://simonwillison.net/2025/Mar/12/gemma-3/

[37] Ronghuan Wu, Wanchao Su, Kede Ma, and Jing Liao. 2023. IconShop: Text-Guided Vector Icon Synthesis with Autoregressive Transformers. *ACM Trans. Graph.* 42, 6 (Dec. 2023), 230:1–230:14.

[38] Ximing Xing, Juncheng Hu, Jing Zhang, Dong Xu, and Qian Yu. 2024. SVGFusion: Scalable Text-to-SVG Generation via Vector Space Diffusion. arXiv:2412.10437.

[39] Ximing Xing, Haitao Zhou, Chuang Wang, Jing Zhang, Dong Xu, and Qian Yu. 2024. SVGDreamer: Text Guided SVG Generation with Diffusion Model. arXiv:2312.16476.

[40] Abhay Zala, Han Lin, Jaemin Cho, and Mohit Bansal. 2024. DiagrammerGPT: Generating Open-Domain, Open-Platform Diagrams via LLM Planning. arXiv:2310.12128.

[41] Fengji Zhang, Linquan Wu, Huiyu Bai, Guancheng Lin, Xiao Li, Xiao Yu, Yue Wang, Bei Chen, and Jacky Keung. 2024. HumanEval-V: Evaluating Visual Understanding and Reasoning Abilities of Large Multimodal Models Through Coding Tasks. arXiv:2410.12381.

[42] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (Dec. 1989), 1245–1262. Publisher: Society for Industrial and Applied Mathematics.

[43] Dewu Zheng, Yanlin Wang, Ensheng Shi, Hongyu Zhang, and Zibin Zheng. 2024. How Well Do LLMs Generate Code for Different Application Domains? Benchmark and Evaluation. arXiv:2412.18573 version: 1.

[44] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. 2023. WebArena: A Realistic Web Environment for Building Autonomous Agents. *arXiv preprint arXiv:2307.13854* (2023). https://webarena.dev

[45] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. arXiv:2406.15877.