Anonymous authors

000

001

003 004 005

006

011

012

013

014

015

016

017

018

021

025

026 027

028

029

031

032

034

039

040

041

042

043 044

045 046

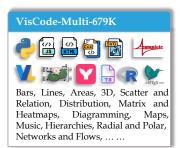
047

048

051

052

Paper under double-blind review



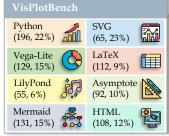




Figure 1: Overview of VisCoder2. We present three components: 1) a dataset of 679K executable code–visualization pairs with multi-round correction dialogues across 12 programming languages; 2) a benchmark spanning 8 languages with natural language instructions, executable code, and rendered outputs; 3) a family of visualization coding agents that iteratively execute, render, and self-debug, approaching the performance of proprietary models.

ABSTRACT

Large language models (LLMs) have recently enabled coding agents capable of generating, executing, and revising visualization code. However, existing models often fail in practical workflows due to limited language coverage, unreliable execution, and lack of iterative correction mechanisms. Progress has been constrained by narrow datasets and benchmarks that emphasize single-round generation and single-language tasks. To address these challenges, we introduce three complementary resources for advancing visualization coding agents. VisCode-Multi-679K is a large-scale, supervised dataset containing 679K validated and executable visualization samples with multi-turn correction dialogues across 12 programming languages. VisPlotBench is a benchmark for systematic evaluation, featuring executable tasks, rendered outputs, and protocols for both initial generation and multi-round self-debug. Finally, we present **VisCoder2**, a family of multi-language visualization models trained on VisCode-Multi-679K. Experiments show that VisCoder2 significantly outperforms strong open-source baselines and approaches the performance of proprietary models like GPT-4.1, with further gains from iterative self-debug, reaching 82.4% overall execution pass rate at the 32B scale, particularly in symbolic or compiler-dependent languages.

1 Introduction

Recent advances in large language models (LLMs) have enabled coding agents Jimenez et al. (2023); Yang et al. (2024b) that can generate visualization code, execute it, and even revise their outputs in response to feedback (Robeyns et al., 2025; Li et al., 2025). These agents are increasingly applied to data analysis and reporting workflows, where producing plots and diagrams is a central task (Galimzyanov et al., 2024).

While existing models can attempt these steps, they often fail in practice: generating code that crashes, produces incorrect visuals, or lacks flexibility across programming languages and libraries (Goswami et al., 2025). Building more reliable visualization coding agents requires resources

that go beyond single-round generation, supporting multi-language coverage, runtime validation, and iterative correction through execution feedback (Yang et al., 2023). However, current datasets and benchmarks lack these capabilities, limiting progress toward agents that can effectively assist in real-world visualization workflows (Ni et al., 2025).

Visualization presents a uniquely valuable setting for advancing these agents. Unlike general-purpose code generation (Li et al., 2022), visualization tasks produce clear and interpretable outputs: the execution process and rendered figure provide an immediate signal of whether the code executed successfully and whether the output aligns with the intended result (Ni et al., 2025). Moreover, visualization requires cross-domain reasoning, combining knowledge of data handling, plotting syntax, and design conventions (Satyanarayan et al., 2016). Crucially, real-world workflows are inherently iterative — analysts rarely produce perfect visualizations on the first attempt, instead refining their code based on runtime behavior and visual inspection (Goswami et al., 2025). This natural feedback loop makes visualization tasks especially well-suited for developing agents that can generate and self-correct code (Chen et al., 2023).

Despite this potential, existing resources for visualization code generation remain narrow in scope. Most datasets focus on single languages, such as Python or Vega-Lite (Galimzyanov et al., 2024; Luo et al., 2021), and include many snippets that cannot be executed reliably (Ni et al., 2025). They lack validated, executable samples, and they do not provide the multi-turn interactions needed to train models for iterative debugging (Ni et al., 2025). Existing benchmarks also have significant gaps: they emphasize single-round generation and do not support systematic evaluation across languages or multi-round repair scenarios (Yang et al., 2023). As a result, current models are tested in settings that fail to capture the complexity of real-world visualization development (Goswami et al., 2025).

To address these limitations, we introduce two complementary resources. First, we present **VisCode-Multi-679K**, a large-scale supervised instruction-tuning dataset comprising 679K executable visualization and code-correction samples across twelve programming languages. VisCode-Multi-679K combines validated visualization code extracted from diverse open-source repositories (Lozhkov et al., 2024; Yang et al., 2025a; Rodriguez et al., 2025) with multi-turn dialogues that teach models to revise faulty code based on execution feedback (Zheng et al., 2024). Second, we propose **VisPlotBench**, a benchmark for evaluating visualization coding agents across eight languages. VisPlotBench provides carefully curated, executable tasks with natural language instructions and rendered outputs, along with a standardized evaluation protocol for both initial generation and multi-round self-debug (Galimzyanov et al., 2024).

Finally, we train **VisCoder2**, a family of multi-language visualization models built on VisCode-Multi-679K. VisCoder2 substantially outperforms size-matched open-source baselines (Hui et al., 2024; Guo et al., 2024a; Ni et al., 2025) and closes much of the performance gap with proprietary models such as GPT-4.1 (Fachada et al., 2025). Experiments show that iterative self-debug yields further improvements, reaching **82.4**% at the 32B scale, *on par with GPT-4.1 and surpassing GPT-4.1-mini*, particularly benefiting symbolic or compiler-dependent languages like LilyPond, LaTeX, and Asymptote. Together, **VisCode-Multi-679K**, **VisPlotBench**, and **VisCoder2** establish a foundation for building and evaluating visualization coding agents that can operate reliably across diverse programming languages and real-world visualization tasks.

2 VISCODE-MULTI-679K: AN INSTRUCTION TUNING DATASET FOR VISUALIZATION ACROSS TWELVE PROGRAMMING LANGUAGES

We present **VisCode-Multi-679K**, a supervised instruction tuning dataset for visualization code generation and feedback-driven correction across twelve programming languages. The dataset supports robust multi-language code generation and enables iterative refinement through multi-turn supervision, aligning with the needs of interactive visualization workflows.

VisCode-Multi-679K unifies two complementary sources of supervision. The first is a large collection of executable visualization code extracted from open source repositories across twelve programming languages, spanning diverse chart types, libraries, and real-world usage patterns. Each sample is validated for runtime execution and paired with its rendered output, ensuring reliable supervision for multi-language code generation. The second source is 66K multi-turn dialogues from the Code Feedback dataset (Zheng et al., 2024), which provide training signals for revising faulty code based

on execution feedback. Although these dialogues are not exclusively visualization-oriented, they are essential for modeling realistic self-correction behaviors in iterative workflows.

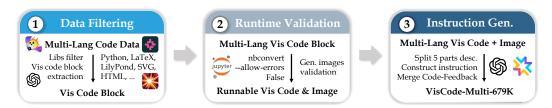


Figure 2: Data construction pipeline for VisCode-Multi-679K. We collect code blocks across twelve programming languages from open-source repositories, including large-scale code corpora, synthetic visualization datasets, and domain-specific diagram collections. We validate executability and render outputs through Jupyter-based runtime checks, yielding instructions paired with images. We integrate multi-turn dialogues from *Code-Feedback* to provide iterative correction supervision.

Figure 2 summarizes the construction pipeline of VisCode-Multi-679K, forming the raw material for a four-stage process: library-based filtering, code block extraction, runtime validation, and instruction generation. The following subsections detail each stage.

2.1 CODE EXTRACTION FROM PUBLIC REPOSITORIES

We construct VisCode-Multi-679K by drawing on three complementary open source corpora: *the-stack-v2*¹ (Lozhkov et al., 2024), *svg-diagrams*² (Rodriguez et al., 2025), and *CoSyn-400K*³ (Yang et al., 2025b; Deitke et al., 2024). These sources are complementary: *the-stack-v2* provides large-scale, diverse code across many languages, capturing realistic visualization embedded in general programs; *svg-diagrams* contributes domain-specific SVG samples focused on diagram rendering; and *CoSyn-400K* offers synthetic but cleanly structured visualization code spanning multiple languages. Together, they cover both natural and synthetic usage across a wide range of languages and visualization styles. From each corpus, we extract code that invokes widely used visualization libraries to capture real-world plotting practices. These sources provide the raw material for a pipeline with four stages: library-based filtering for each language, code block extraction, runtime validation, and instruction generation.

Filtering and Code Block Extraction. For the-stack-v2 (Lozhkov et al., 2024), which contains approximately 900B tokens of code, we restrict our selection to two filtered subsets: stack-edu⁴ (Allal et al., 2025) and the-stack-v2-train-smol-ids⁵. stack-edu was curated from the-stack-v2 using a classifier-based filtering strategy that retains only high-quality educational programming content. the-stack-v2-train-smol-ids is a near-deduplicated subset further filtered with heuristics and spanning 17 programming languages. We first apply library-based filters on these subsets to identify approximately 5.3M visualization code candidates in Python, JavaScript, C++, TypeScript, HTML, and R. Because most examples are embedded in broader program contexts rather than self-contained plotting examples, we use GPT-4.1-mini (OpenAI, 2025) to extract standalone plotting blocks for each language. When the original code does not include data, we inject mock inputs so that each block can execute in isolation. This structural cleaning preserves realistic visualization usage while remaining compatible with our runtime pipeline. After filtering and reconstruction, we obtain roughly 900K candidate blocks.

For *svg-diagrams*, which contains 182K domain-specific SVG samples focused on diagrams from *star-vector* (Rodriguez et al., 2025), we apply regular-expression filtering to remove noisy data that lack width, height, or other essential components. This step retains about 79K candidate blocks.

For *CoSyn-400K*, we select 408K visualization snippets across eight languages, including Python, HTML, LaTeX, SVG, Asymptote, Mermaid, LilyPond, and Vega-Lite. *CoSyn-400K* pro-

¹hf.co/datasets/bigcode/the-stack-v2

²hf.co/datasets/starvector/svg-diagrams

³hf.co/datasets/allenai/CoSyn-400K

⁴hf.co/datasets/HuggingFaceTB/stack-edu

⁵hf.co/datasets/bigcode/the-stack-v2-train-smol-ids

vides synthetic but cleanly structured code spanning a wide range of styles, with well-rendered outputs and consistent structure. Unlike *the-stack-v2*, its Python and HTML code store logic and data separately, which requires reconstruction for runtime execution. For languages requiring reconstruction, we rebuild runnable scripts by inserting lightweight annotations such as column headers and a data row to emulate realistic data loading. When necessary, we append missing plotting function calls to ensure that each language can execute within a Jupyter notebook environment.

Runtime Validation. To ensure executability, we run each candidate block in isolated Jupyter environments. C++, JavaScript, and R are executed in dedicated kernels, while all other languages share the Python kernel. Each block is run with nbconvert using allow-error=False to enforce strict filtering. We apply a fixed timeout and terminate runs that hang or enter infinite loops via a simulated keyboard interrupt. Only samples that execute successfully and generate valid image files that are non-monochrome and larger than 10KB are retained. This step produces 245K validated plotting scripts from *the-stack-v2*, 43K from *svg-diagrams*, and 322K from *CoSyn-400K*, each paired with its rendered output. The detailed distribution is shown in Table 5.

Instruction Generation. To enable models to learn from both structural code features and rendered visual outputs, we generate natural language instructions for each validated example using GPT-4.1 (OpenAI, 2025). This process ensures that supervision captures not only code syntax but also the semantics of the corresponding visualization.

To capture both data semantics and visual design, each instruction is structured into five components: (1) a brief setup description specifying the programming language and visualization libraries used; (2) a description of either the underlying data (for data-driven code) or the visible elements of the figure (for non-data-driven code); (3) a data block that either contains a copied data-generation line or a two-row preview, left empty for non-data-driven cases; (4) a high-level output description that conveys the intended visualization conceptually; and (5) a style description capturing colors, grid layout, and other visual properties. These components are assembled into a fixed template:

```
[Output Description]
[Setup]
[Data/Visual Description]
"The data is shown below:" or None
[Data] or None
[Style Description]
```

This format enforces a consistent prompt structure across sources and languages, ensuring that models receive a unified description of the visualization target, its data, and its stylistic attributes.

2.2 MULTI-TURN INSTRUCTION-FOLLOWING DIALOGUES WITH EXECUTION FEEDBACK

VisCode-Multi-679K further includes over 66K multi-turn dialogues from the *Code-Feedback*⁶ dataset (Zheng et al., 2024). These dialogues cover programming tasks in Python, HTML, JavaScript,R, and other languages, with user instructions, model-generated code, and follow-up turns carrying execution feedback or revision prompts.

Although not tailored to visualization, they provide essential supervision for teaching models to revise faulty code based on runtime signals and to reason over iterative interactions. We incorporate these dialogues into the instruction tuning corpus alongside single-turn samples from *stack-edu*, *the-stack-v2*, *svg-diagrams*, and *CoSyn-400K*. This integration allows models to practice both initial code generation and multi-turn refinement strategies.

3 VISPLOTBENCH: MULTI-LANGUAGE BENCHMARK FOR VISUALIZATION CODING AGENTS

VisPlotBench is a benchmark for evaluating visualization coding agents across eight languages. Unlike prior efforts that focus on a single language or specification style, VisPlotBench spans imperative libraries, declarative grammars, markup-based formats, and symbolic notations, providing a standardized protocol for assessing both initial code generation and multi-round self-debug.

⁶hf.co/datasets/m-a-p/Code-Feedback

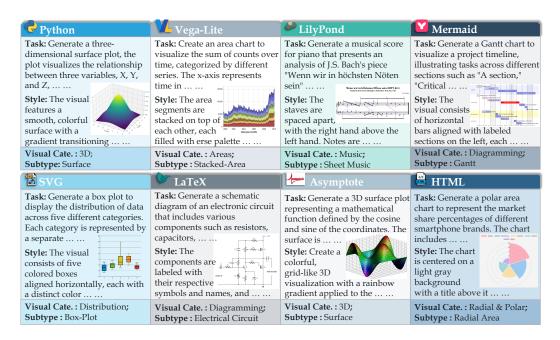


Figure 3: Overview of VisPlotBench. The benchmark covers eight visualization languages and contains 888 diverse visualization tasks, each combining a natural language instruction and a rendered visual. Tasks are annotated with a Visual category and a Subtype, spanning 13 categories in total.

3.1 Overview

Existing visualization benchmarks are narrow in scope: most cover a single language, few chart families, and no iterative debugging. **VisPlotBench** fills these gaps with 888 tasks across eight languages and 13 Visual categories (Figure 4). The taxonomy spans common families such as Bars, Lines, and Scatter, while adding rarely represented ones like Hierarchies, Music, and Networks & Flows. Each task combines a natural language instruction, executable code, and a rendered output, enabling execution-grounded evaluation. With its execute—render—score protocol and multi-round self-debug loop, VisPlotBench provides the first systematic benchmark for assessing visualization coding agents across languages and task types.

Table 1: Comparison with existing benchmarks. VisPlotBench provides executable, multi-language tasks with natural language instructions, rendered outputs, and a standardized protocol for both initial code generation and multi-round self-debugging.

Benchmark	Coverage	Self-debug	Visual Category	Num
VisEval (Chen et al., 2024)	Python	X	4	2,524
MatPlotBench (Yang et al., 2024c)	Python	X	11	100
nvBench (Luo et al., 2021)	Vega-Lite	X	4	25,750
nvBench 2.0 (Luo et al., 2025)	Vega-Lite	X	5	7,878
Text2Vis (Rahman et al., 2025)	Python	X	10	1,985
PandasPlotBench (Galimzyanov et al., 2024)	Python	X	10	175
PandasPlotBench-Enhanced (Ni et al., 2025)	Python	~	10	175
VisPlotBench (ours)	8 languages	V	13	888

Table 1 positions VisPlotBench among representative benchmarks across four dimensions: language coverage, visual categories, self-debug support, and dataset size. Earlier resources remain narrow—focusing on Python or Vega-Lite, with limited chart types and no iterative debugging. VisCoder introduced self-debugging for PandasPlotBench, while VisPlotBench generalizes this to eight languages, expands coverage to 13 categories, including Hierarchies, Music, and Networks & Flows, and standardizes evaluation for systematic cross-language assessment.

3.2 DATA COLLECTION AND CURATION

We assemble 888 executable tasks from publicly available examples, library documentation, and high-quality code snippets across eight programming languages. The tasks span 13 Visual categories and 116 Subtypes, covering common families such as Bars, Lines, and Scatter, as well as underrepresented ones including Hierarchies, Music, and Networks & Flows.

Each candidate script is executed in an isolated runtime with language-specific kernels or headless renderers. Tasks are retained only if execution succeeds and a valid image is produced. We discard visually trivial outputs (e.g., near-monochrome images) and remove duplicates by hashing rendered outputs and normalizing code. This process yields a pool of verified code–image pairs compatible with our evaluation pipeline.

Annotators then review verified pairs, removing low-quality items such as unreadable or degenerate plots. Each remaining task is annotated with a Visual category and Subtype from the shared taxonomy shown in Appendix G, with library-specific idioms added when appropriate. A double-pass review with conflict resolution ensures consistency across languages.

3.3 TASK CONSTRUCTION

Each VisPlotBench task extends the verified code-image pair with a structured natural language instruction. To ensure consistency across languages, we adopt a five-part schema: $Setup \rightarrow Plot$ $Instruct \rightarrow Data\ Instruct \rightarrow Task\ Description \rightarrow Style\ Description$. This schema provides a unified template that reflects both the semantic intent and the stylistic requirements of each visualization.

Setup, Plot Instruct, and Data Instruct are authored separately for each language so that tasks capture real usage, including syntax constraints, runtime notes, and data access conventions. Task Description and Style Description are generated with GPT-4.1 conditioned on the verified code and its rendered visual. The Task Description specifies the semantic intent and structural elements required for correctness, while the Style Description summarizes perceptual attributes such as layout, annotations, label formatting, and color usage. Detailed authoring templates and generation prompts are provided in Appendix F.2 and F.3.

The final instruction is the concatenation of the five components, producing a unified input format across languages. This design enables coding agents to condition on natural language instructions paired with minimal data previews and generate executable code that satisfies both the semantic and stylistic requirements of the task.

3.4 EVALUATION PROTOCOL

VisPlotBench adopts a standardized execute—render—score pipeline. Each submission is executed in an isolated runtime with language-specific kernels or headless renderers, subject to strict timeouts and log capture. The process outputs three artifacts—a rendered image, an execution log, and a metadata record—supporting execution-grounded and judgment-based evaluation.

Evaluation metrics extend those of PandasPlotBench and VisCoder. **Execution Pass Rate** checks whether the code runs without error and produces a valid visualization. **Task Score** measures instruction compliance using an LLM judge guided by semantic and structural rubrics, and **Visual Score** assesses perceptual similarity between generated and reference outputs. Both follow the GPT-based judging protocol of PandasPlotBench.

To assess iterative refinement, VisPlotBench includes a multi-round self-debug protocol. Unresolved tasks are revisited for up to three rounds, where the model receives the instruction, its prior code, and an excerpt of the execution log before producing a revision. The final score reflects the best attempt, mirroring real-world correction loops and enabling systematic evaluation of both baseline generation and feedback-driven recovery.

4 MAIN RESULTS

We evaluate both proprietary and open-source models on VisPlotBench to compare execution reliability across parameter scales, programming languages, and evaluation modes. Proprietary refer-

ences include GPT-4.1 (OpenAI, 2025) and its lighter variant GPT-4.1-mini (OpenAI, 2025), while open-source baselines include DeepSeek-Coder (Guo et al., 2024b), DeepSeek-CoderV2 (Zhu et al., 2024), Qwen2.5-Coder (Hui et al., 2024), and VisCoder (Ni et al., 2025). Our VisCoder2 models are trained on VisCode-Multi-679K using Qwen2.5-Coder backbones at 3B, 7B, 14B, and 32B scales, a full training and evaluation setup provided in Appendix B. A controlled ablation of training data sources is presented in Appendix D.

Table 2: Overall execution pass rate (%) of selected models on the VisPlotBench benchmark. The best-performing model in each scale is shown in **bold**, and the second best is underlined.

Model	Exec Pass Overall	Python (196)	Vega-Lite (129)	LilyPond (55)	Mermaid (131)	SVG (65)	LaTeX (112)	Asymptote (92)	HTML (108)
GPT-4.1	63.4	64.3	84.5	43.6	68.7	95.4	31.3	21.7	89.8
GPT-4.1 + Self Debug	82.4	84.2	96.1	63.6	93.9	96.9	66.1	46.7	97.2
GPT-4.1-mini	58.9	64.8	84.5	16.4	51.9	95.4	29.5	23.9	86.1
GPT-4.1-mini + Self Debug	81.1	80.6	96.9	<u>56.4</u>	94.7	96.9	<u>58.9</u>	48.9	100.0
			\sim 3B	Scale					
DeepSeek-Coder-1.3B-Ins.	32.3	29.1	53.5	30.9	63.4	7.7	4.5	13.0	36.1
Qwen2.5-Coder-3B-Ins.	45.8	34.2	68.2	3.6	74.1	75.4	17.9	18.5	62.0
VisCoder-3B	56.1	45.4	83.7	21.8	<u>75.6</u>	76.9	23.2	30.4	79.6
VisCoder2-3B	67.7	56.1	83.0	50.9	76.3	87.7	36.6	62.0	93.5
VisCoder2-3B + Self Debug	70.0	63.3	84.5	52.7	76.3	87.7	38.4	63.0	94.4
			~ 7B	Scale					
DeepSeek-Coder-6.7B-Ins.	46.4	39.3	79.8	7.3	91.6	96.9	18.8	0.0	22.2
Qwen2.5-Coder-7B-Ins.	51.2	41.3	76.0	5.5	77.9	92.3	25.9	13.0	64.8
VisCoder-7B	57.2	58.2	71.3	23.6	77.1	93.9	25.9	17.4	75.9
VisCoder2-7B	70.9	64.8	83.0	69.1	78.6	96.9	39.3	64.1	82.4
VisCoder2-7B + Self Debug	76.4	77.0	84.5	72.7	84.7	96.9	42.9	70.7	84.3
			∼ 14B	Scale					
DeepSeek-Coder-V2-Lite-Ins.	55.3	47.5	75.2	49.1	69.5	93.9	29.5	20.7	64.8
Qwen2.5-Coder-14B-Ins.	59.5	50.0	83.0	25.5	74.8	98.5	30.4	25.0	83.3
VisCoder2-14B	72.1	65.3	93.0	54.6	81.7	89.2	42.0	56.5	90.7
VisCoder2-14B + Self Debug	78.4	78.1	94.6	63.6	86.3	90.8	45.5	66.3	94.4
			~ 32B	Scale					
DeepSeek-Coder-33B-Ins.	54.3	58.2	90.7	30.9	87.0	92.3	24.1	21.7	12.0
Qwen2.5-Coder-32B-Ins.	57.5	50.5	83.0	30.9	71.0	93.9	29.5	17.4	78.7
VisCoder2-32B	73.1	65.3	94.6	56.4	87.0	81.5	42.9	58.7	91.7
VisCoder2-32B + Self Debug	$\overline{82.4}$	81.6	96.1	69.1	90.1	86.2	61.6	$\overline{71.7}$	93.5

4.1 OVERALL COMPARISON

 Table 2 summarizes execution pass rates for all models across eight visualization languages and overall averages. The following analysis examines differences between proprietary and open-source models, variation across languages, and the relative advantages of VisCoder2 under both default and self-debug evaluation modes.

Proprietary Models Remain Stronger. GPT-4.1 achieves 63.4% overall, the highest among reference models, and GPT-4.1-mini follows closely. Both perform strongly on standardized declarative or markup languages such as Vega-Lite, SVG, and HTML, all above 84%. In contrast, instruction-tuned open-source models remain far behind. At the 7B scale, Qwen2.5-Coder reaches only 51.2% overall, with fewer than 30% on LaTeX and just 5.5% on LilyPond. Previous VisCoder variants improve Python performance but fail to generalize across languages. These results underline the substantial gap between proprietary and open-source models.

Cross-Language Variation. Performance differs sharply across visualization languages. Vega-Lite and HTML are close to saturation for most models, while Python shows steady gains with scale. By contrast, symbolic and compiler-dependent languages remain the most difficult. Even GPT-4.1 achieves less than 45% on LilyPond and under 25% on Asymptote, and open-source baselines fall much lower. This uneven landscape highlights that progress on symbolic grammars is the key bottleneck for reliable multi-language visualization.

VisCoder2 Advantage. Across all scales, VisCoder2 consistently outperforms size-matched open-source baselines. At 32B, it improves overall execution pass rate by approximately 15 points compared with Qwen2.5-Coder and reaches parity with GPT-4.1. The only consistent shortfall is on SVG, where VisCoder2 trails the strongest baseline by over 10 points. Overall, VisCoder2 is the first open-source model to match proprietary reliability on executable visualization tasks.

Effect of Self-Debug. Iterative correction consistently improves execution reliability across model families and scales. Proprietary models benefit strongly, and VisCoder2 follows the same trend: at larger scales, overall execution rises by nearly ten points when self-debugging is enabled. The effect is especially pronounced for symbolic and compiler-dependent languages such as LilyPond, LaTeX, and Asymptote, where fragile syntax or compilation errors dominate. Self-debugging enables the model to repair these shallow but frequent failures, allowing models to resolve previously intractable failures into valid outputs. This demonstrates that feedback-driven refinement is not just a marginal improvement but a critical mechanism for tackling the hardest visualization languages.

4.2 TASK AND VISUAL SCORE ANALYSIS

We analyze Task Score and Visual Score on three representative languages that highlight different behaviors: LaTeX illustrates execution—semantics mismatch, LilyPond shows the largest gains on symbolic grammars, and SVG exposes model—library sensitivity where semantic and perceptual signals diverge. Results for all languages and scales are provided in Appendix H.

Table 3: Performance of selected languages on the VisPlotbench benchmark. For each model, we report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**). The best-performing model in each scale is shown in **bold**, and the second best is underlined.

		La	TeX (1	12)			Lil	yPond	(55)		SVG (65)				
Model	Exec	M	ean	Good	(≥75)	Exec	M	ean	Good	d(≥75)	Exec	M	ean	Good	(≥75)
	Pass	vis	task	vis	task	Pass	vis	task	vis	task	Pass	vis	task	vis	task
GPT-4.1	31.3	18	26	13%	25%	43.6	14	38	5%	36%	95.4	45	92	14%	94%
GPT-4.1 + Self Debug	66.1	38	56	25%	51%	63.6	17	54	5%	53%	96.9	45	93	14%	95%
GPT-4.1-mini	29.5	21	25	18%	25%	16.4	2	12	0%	11%	95.4	41	86	11%	86%
GPT-4.1-mini + Self Debug	58.9	35	50	23%	49%	56.4	14	42	0%	35%	96.9	42	88	11%	88%
Qwen2.5-Coder-7B-Instruct	25.9	11	15	6%	8%	5.5	0	3	0%	4%	92.3	23	58	0%	40%
VisCoder2-7B	39.3	15	23	6%	15%	69.1	16	52	2%	45%	96.9	34	73	3%	62%
VisCoder2-7B + Self Debug	42.9	16	24	6%	15%	72.7	17	55	2%	45%	96.9	34	73	3%	62%
Qwen2.5-Coder-32B-Instruct	29.5	14	25	9%	27%	30.9	5	22	2%	18%	93.9	34	81	3%	75%
VisCoder2-32B	42.9	20	35	11%	34%	56.4	14	39	2%	27%	81.5	33	68	11%	63%
VisCoder2-32B + Self Debug	61.6	28	45	14%	42%	69.1	16	48	2%	35%	86.2	34	71	11%	66%

LaTeX: Execution–Semantics Mismatch. Models often capture the intended structure of a figure but fail to compile reliably. For example, GPT-4.1 improves from 31.3% to 66.1% execution pass rate with Self-Debug, while task scores remain around 50 even when execution fails. VisCoder2 raises execution and task scores compared with baselines, but compilation errors remain frequent. This pattern indicates that semantic alignment does not always translate into successful rendering.

LilyPond: Symbolic Grammar Gains. VisCoder2 delivers the clearest advantage on symbolic languages. At 7B, Qwen2.5-Coder executes only 5.5% of tasks, while VisCoder2 reaches 69.1% and further improves with Self-Debug. The proportion of examples with task scores above 75 also increases by more than tenfold. These results show that targeted coverage of symbolic grammars in VisCode-Multi-679K translates directly into reliable generation and semantic adherence.

SVG: Sensitivity to Rendering Libraries. Execution success is high across most models, yet visual scores lag behind task scores. For instance, GPT-4.1 with Self-Debug achieves 95.4% execution and a task score near 90, but the average visual score is below 50. VisCoder2 performs competitively but trails Qwen2.5 on execution at larger scales (81.5% versus 93.9% at 32B). These discrepancies suggest that evaluation on SVG is strongly influenced by library-specific rendering details rather than semantic understanding alone.

4.3 ERROR ANALYSIS

To better understand failure modes across languages, we analyze execution errors before and after self-debug. Many language-specific exceptions, such as FunctionSignatureError in Asymptote or MarkupError in LilyPond, were merged into four broader categories for clarity: Structural Errors (syntax or parsing), Type & Interface Errors (invalid calls or arguments), Semantic / Data Errors (mismatched variables or values), and Runtime / Environment Errors (renderer or package issues). Representative results for VisCoder2-32B are shown in Table 4, with full breakdowns in Appendix J.

Table 4: Representative error transitions for VisCoder2-32B across eight visualization languages. Each cell shows error counts from initial failure to the final self-debug round $(X \to Y)$. A dash indicates the error type does not occur for that language.

Error Category	Python	Vega-Lite	LilyPond	Mermaid	SVG	LaTeX	Asymptote	HTML
Structural Errors	$1 \rightarrow 1$	$2 \rightarrow 1$	$14 \rightarrow 10$	$12 \rightarrow 9$	$8 \rightarrow 7$	$10 \rightarrow 4$	$9 \rightarrow 3$	-
Type & Interface	$13 \rightarrow 3$	$2 \rightarrow 1$	$5 \rightarrow 2$	-	-	-	-	-
Semantic / Data	$19 \rightarrow 8$	-	-	-	-	$28 \rightarrow 23$	$15 \rightarrow 11$	-
Runtime / Env.	-	$2 \rightarrow 2$	-	-	-	$27 \rightarrow 6$	$8 \rightarrow 6$	$3 \rightarrow 2$

Effective recovery on structural and interface errors. Self-debug reduces shallow errors such as missing tokens or invalid arguments across multiple languages. For example, Python interface errors fall from 13 to 3 (Figure 6), and structural errors in LilyPond decrease from 14 to 10 (Figure 12). Mermaid and Asymptote show the same trend, with syntax and function signature errors shrinking after correction (Figure 15). These cases benefit from explicit diagnostic traces, making them relatively easy to fix through iterative feedback.

Persistent failures in semantic and runtime errors. Errors involving semantics or execution environments remain difficult to resolve. In LaTeX, undefined variables decrease only slightly (28 to 23), and Asymptote variable mismatches improve only marginally (15 to 11) (Figure 24). Renderer failures such as Vega-Lite rendering errors (2 to 2) and HTML request failures (3 to 2) often persist across all rounds (Figure 28). These errors require deeper reasoning over symbolic grammars and runtime contexts, which current self-debug protocols cannot fully capture. Symbolic languages and renderer-sensitive environments therefore remain the dominant bottlenecks, pointing to the need for grammar-aware training objectives and more robust runtime integration.

5 CONCLUSION

Reliable visualization coding goes beyond single-pass generation: it requires competence across diverse languages and the ability to refine outputs iteratively in response to execution feedback. Existing datasets and benchmarks lack these capabilities, limiting progress toward practical agents for real-world workflows.

We addressed these gaps through three contributions. First, we introduced VisCode-Multi-679K, a large-scale instruction tuning dataset that unifies executable visualization code across twelve languages with multi-turn feedback dialogues. Second, we built VisPlotBench, a benchmark covering eight visualization languages under a standardized execute—render—score protocol, with tasks spanning 13 categories and 116 subtypes. Third, we trained the VisCoder2 model family on these resources, showing that it consistently outperforms open-source baselines and approaches proprietary models in execution reliability.

Our experiments highlight two insights. Broad multi-language coverage is essential: symbolic and compiler-dependent languages such as LaTeX, LilyPond, and Asymptote remain challenging, yet progress on them is decisive for true generalization. Iterative refinement further proves indispensable: self-debug delivers large gains across models, especially on languages where structural and semantic errors are common.

Taken together, VisCode-Multi-679K, VisPlotBench, and VisCoder2 establish the first systematic framework for building and evaluating visualization coding agents. We believe these resources can accelerate the development of agents that are not only multilingual but also capable of realistic correction loops, pushing toward reliable coding assistants for data analysis, reporting, and beyond.

ETHICS STATEMENT

This work complies with the ICLR Code of Ethics. In this study, no human subjects or animal experimentation were involved. All datasets used were accessed and used under their licenses/terms of use, in line with applicable usage guidelines and with no infringement of privacy. We took steps to minimize unfair bias or discriminatory effects throughout the research process. No personally identifiable information was processed, and no experiments were conducted that could raise privacy or security risks. We remain committed to transparency and research integrity across the entire project lifecycle.

REPRODUCIBILITY STATEMENT

We have made every effort to ensure that the results presented in this paper are reproducible. We release all prompts used to construct the training datasets in Appendix F.1, the full processing pipeline applied to raw sources, and the prompts and instructions used in building the *VisPlotBench* benchmark in Appendix F.2 and subsection F.3. We also provide a detailed description of the experimental setup in Appendix B.

In addition, all datasets used (stack-edu, the-stack-v2-train-smol-ids, svg-diagrams, and Code-Feedback) in constructing *VisCode-Multi-679K* are publicly available, as described in subsection 2.1. We will open-source our complete training and evaluation codebase, along with all trained model weights after publication. This release will include comprehensive documentation and usage examples to support future research and enable direct comparison with our results.

We believe these measures will enable other researchers to reproduce our findings, trace our dataset construction pipeline, and extend our work in future studies.

REFERENCES

- Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big data-centric training of a small language model, 2025. URL https://arxiv.org/abs/2502.02737.
- Nan Chen, Yuge Zhang, Jiahang Xu, Kan Ren, and Yuqing Yang. Viseval: A benchmark for data visualization in the era of large language models. *IEEE Transactions on Visualization and Computer Graphics*, 2024.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
- Matt Deitke, Christopher Clark, Sangho Lee, Rohun Tripathi, Yue Yang, Jae Sung Park, Mohammadreza Salehi, Niklas Muennighoff, Kyle Lo, Luca Soldaini, et al. Molmo and pixmo: Open weights and open data for state-of-the-art multimodal models. *ArXiv preprint*, abs/2409.17146, 2024. URL https://arxiv.org/abs/2409.17146.
- Victor Dibia. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. *ArXiv preprint*, abs/2303.02927, 2023. URL https://arxiv.org/abs/2303.02927.
- Nuno Fachada, Daniel Fernandes, Carlos M. Fernandes, Bruno D. Ferreira-Saraiva, and João P. Matos-Carvalho. Gpt-4.1 sets the standard in automated experiment design using novel python libraries. *arXiv preprint arXiv:2508.00033*, 2025.
- Timur Galimzyanov, Sergey Titov, Yaroslav Golubev, and Egor Bogomolov. Drawing pandas: A benchmark for llms in generating plotting code. *ArXiv preprint*, abs/2412.02764, 2024. URL https://arxiv.org/abs/2412.02764.
- Kanika Goswami, Puneet Mathur, Ryan Rossi, and Franck Dernoncourt. Plotgen: Multi-agent Ilm-based scientific data visualization via multimodal feedback. *arXiv preprint arXiv:2502.00988*, 2025.
- Anastasiia Grishina, Vadim Liventsev, Aki Härmä, and Leon Moonen. Fully autonomous programming using iterative multi-agent debugging with large language models. *ACM Transactions on Evolutionary Learning*, 5(1):1–37, 2025.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *ArXiv preprint*, abs/2401.14196, 2024a. URL https://arxiv.org/abs/2401.14196.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024b.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *ArXiv preprint*, abs/2409.12186, 2024. URL https://arxiv.org/abs/2409.12186.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- Hao Li, Haoxiang Zhang, and Ahmed E Hassan. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering. *arXiv* preprint *arXiv*:2507.15003, 2025.

- Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. *ArXiv preprint*, abs/2411.04329, 2024. URL https://arxiv.org/abs/2411.04329.
 - Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
 - Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.
 - Tianqi Luo, Chuhan Huang, Leixian Shen, Boyan Li, Shuyu Shen, Wei Zeng, Nan Tang, and Yuyu Luo. nvbench 2.0: A benchmark for natural language to visualization under ambiguity. *arXiv* preprint arXiv:2503.12880, 2025.
 - Yuyu Luo, Jiawei Tang, and Guoliang Li. nvbench: A large-scale synthesized dataset for cross-domain natural language to visualization task. *arXiv preprint arXiv:2112.12926*, 2021.
 - Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.
 - Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.
 - Yuansheng Ni, Ping Nie, Kai Zou, Xiang Yue, and Wenhu Chen. Viscoder: Fine-tuning Ilms for executable python visualization code generation. *arXiv preprint arXiv:2506.03930*, 2025.
 - OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, 2025. URL https://openai.com/index/gpt-4-1/.
 - Mizanur Rahman, Md Tahmid Rahman Laskar, Shafiq Joty, and Enamul Hoque. Text2vis: A challenging and diverse benchmark for generating multimodal visualizations from text. *arXiv preprint arXiv:2507.19969*, 2025.
 - Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv* preprint arXiv:2504.15228, 2025.
 - Juan A. Rodriguez, Abhay Puri, Shubham Agarwal, Issam H. Laradji, Pau Rodriguez, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. Starvector: generating scalable vector graphics code from images and text. *arXiv* preprint arXiv:2312.11556, 2025.
 - Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23 (1):341–350, 2016.
 - Wonduk Seo, Seungyong Lee, Daye Kang, Zonghao Yuan, and Seunghyun Lee. Vispath: Automated visualization code synthesis via multi-path reasoning and feedback-driven optimization. *ArXiv* preprint, abs/2502.11140, 2025. URL https://arxiv.org/abs/2502.11140.
 - Chengyue Wu, Yixiao Ge, Qiushan Guo, Jiahao Wang, Zhixuan Liang, Zeyu Lu, Ying Shan, and Ping Luo. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. *ArXiv preprint*, abs/2405.07990, 2024. URL https://arxiv.org/abs/2405.07990.
 - Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. Waitgpt: Monitoring and steering conversational llm agent in data analysis with on-the-fly code visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pp. 1–14, 2024.

- Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, et al. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv* preprint arXiv:2406.09961, 2024a.
 - John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. Advances in Neural Information Processing Systems, 36:23826–23854, 2023.
 - John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. Advances in Neural Information Processing Systems, 37:50528–50652, 2024b.
 - Yue Yang, Ajay Patel, Matt Deitke, Tanmay Gupta, Luca Weihs, Andrew Head, Mark Yatskar, Chris Callison-Burch, Ranjay Krishna, Aniruddha Kembhavi, and Christopher Clark. Scaling textrich image understanding via code-guided synthetic multimodal data generation. *arXiv* preprint arXiv:2502.14846, 2025a.
 - Yue Yang, Ajay Patel, Matt Deitke, Tanmay Gupta, Luca Weihs, Andrew Head, Mark Yatskar, Chris Callison-Burch, Ranjay Krishna, Aniruddha Kembhavi, et al. Scaling text-rich image understanding via code-guided synthetic multimodal data generation. *ArXiv preprint*, abs/2502.14846, 2025b. URL https://arxiv.org/abs/2502.14846.
 - Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, et al. Matplotagent: Method and evaluation for llm-based agentic scientific data visualization. *ArXiv preprint*, abs/2402.11453, 2024c. URL https://arxiv.org/abs/2402.11453.
 - Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv preprint*, abs/2502.01718, 2025. URL https://arxiv.org/abs/2502.01718.
 - Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. Swift:a scalable lightweight infrastructure for fine-tuning, 2024. URL https://arxiv.org/abs/2408.05517.
 - Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *ArXiv preprint*, abs/2402.14658, 2024. URL https://arxiv.org/abs/2402.14658.
 - Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

A	Rela	ted Work
В	Exp	eriment Setup
C	Trai	ning Data Distribution
D	Abla	ation Study
E	Lim	itation
F	Pror	npt Used and Instruct Design
	F.1	Prompt Used in VisCode-Multi-679K
	F.2	Prompt Used in VisPlotBench
	F.3	Instruct Design in VisPlotBench Evaluation
G	Taxo	onomy of Visualization Types and Subtypes
Н	Brea	kdown Main Results
	H.1	Python, Vega-Lite & Lilypond
	H.2	Mermaid, SVG & LaTeX
	H.3	Asymptote & HTML
I	Brea	kdown Self-Debug Results
	I.1	Python & Vega-Lite
	I.2	LilyPond & Mermaid
	I.3	SVG & LaTeX
	I.4	Asymptote & HTML
J	Brea	kdown Error Type Results
	J.1	Python
	J.2	Vega-Lite
	J.3	Lilypond
	J.4	Mermaid
	J.5	SVG
	J.6	LaTeX
	J.7	Asymptote
	J.8	HTML
K	Case	Study
	K.1	Python: Successful Generation
	K.2	Python: Self-Debug Recovery

756	K.3	Python: Self-Debug Failed	42
757 758	K.4	Vega-Lite: Successful Generation	43
759	K.5	Vega-Lite: Self-Debug Recovery	44
760			
761	K.6	Vega-Lite: Self-Debug Failed	45
762	K.7	Lilypond: Successful Generation	46
763	K.8	Lilypond: Self-Debug Recovery	47
764 765	K.9	Lilypond: Self-Debug Failed	48
766			
767	K.10	Mermaid: Successful Generation	49
768	K.11	Mermaid: Self-Debug Recovery	50
769	K.12	2 Mermaid: Self-Debug Failed	51
770		S SVG: Successful Generation	52
771			
772	K.14	SVG: Self-Debug Recovery	53
773 774	K.15	S SVG: Self-Debug Failed	54
775	K.16	LaTeX: Successful Generation	55
776	K.17	LaTeX: Self-Debug Recovery	56
777		B LaTeX: Self-Debug Failed	57
778 779		Asymptote: Successful Generation	58
780			
781	K.20	Asymptote: Self-Debug Recovery	59
782	K.21	Asymptote: Self-Debug Failed	60
783	K.22	2 HTML: Successful Generation	61
784	К 23	B HTML: Self-Debug Recovery	62
785		•	
786 787	K.24	HTML: Self-Debug Failed	63
788		A XI	
789	L LLN	1 Usage	64

A RELATED WORK

LLMs for Visualization Code Generation Large language models have shown promising results in generating visualization code from natural language descriptions (Yang et al., 2024c; Chen et al., 2024; Galimzyanov et al., 2024). Most existing approaches focus on single languages, particularly Python with matplotlib or plotly (Wu et al., 2024; Yang et al., 2024a), while some explore specification-based methods using Vega-Lite (Xie et al., 2024). However, these systems face significant limitations: they typically support only one or two programming languages, lack systematic execution validation, and often generate code that fails to run reliably (Ni et al., 2025). Multilanguage code generation efforts in broader domains (Lozhkov et al., 2024; Muennighoff et al., 2023) provide extensive language coverage but lack the specialized knowledge required for visualization tasks, particularly for domain-specific languages like LaTeX for mathematical plots or LilyPond for musical notation. Our VisCode-Multi-679K dataset addresses these limitations by providing validated, executable visualization samples across twelve programming languages, enabling robust multi-language visualization code generation with systematic quality control and execution verification.

Self-Debug and Coding Agents Recent advances in coding agents have emphasized iterative development capabilities, where models can generate, execute, and refine code through multiple rounds of feedback (Jimenez et al., 2023; Yang et al., 2024b). Self-debug approaches leverage execution traces, error messages, and runtime outcomes to guide automatic code correction (Chen et al., 2023; Madaan et al., 2023; Zheng et al., 2024; Zeng et al., 2025). Agent-based systems further extend these capabilities by incorporating planning, tool use, and collaborative debugging workflows (Grishina et al., 2025; Li et al., 2024). While these methods show promise in general programming tasks, their application to visualization remains underexplored. Existing visualization systems like LIDA (Dibia, 2023) incorporate some feedback mechanisms, but lack the systematic multi-turn correction capabilities needed for reliable cross-language deployment. Our work uniquely combines multi-language visualization generation with systematic self-debug, enabling VisCoder2 to iteratively refine code across diverse programming environments, particularly excelling in symbolic languages where execution validation is essential.

Visualization Benchmark Existing visualization benchmarks focus predominantly on Python (Galimzyanov et al., 2024; Chen et al., 2024; Yang et al., 2024c; Rahman et al., 2025) or declarative specifications like Vega-Lite (Luo et al., 2021; 2025), limiting their applicability across diverse programming environments used in real-world data analysis. While general code datasets like *the-stack-v2* (Lozhkov et al., 2024) provide broad language coverage, they lack visualization-specific content and execution validation. Most visualization benchmarks evaluate only single-turn generation, failing to capture the iterative debugging workflows that characterize practical visualization development (Ni et al., 2025; Seo et al., 2025). Without multi-language support and multi-round evaluation, existing benchmarks cannot assess whether models can handle the diverse toolchains and iterative workflows essential for real-world visualization tasks. VisCode-Multi-679K addresses these limitations by providing the first large-scale dataset with execution-validated visualization code across twelve programming languages, while VisPlotBench enables a systematic evaluation of both initial generation and multi-turn self-debug capabilities across visualization tasks in multiple programming languages.

B EXPERIMENT SETUP

Training Setup. We fine-tune Qwen2.5-Coder-Instruct (Hui et al., 2024) at four parameter scales: 3B, 7B, 14B, and 32B. This setup allows us to assess the generalizability of VisCode-Multi-679K across capacities. All models are trained for 3 epochs with a learning rate of 5×10^{-6} , a warm-up ratio of 0.05, and a cosine scheduler. We perform full-parameter tuning in bfloat16 precision on $8 \times H100$ GPUs with a total batch size of 64, using the SWIFT infrastructure (Zhao et al., 2024).

Evaluation Setup. All evaluations are conducted on VisPlotBench using the standardized protocol in Section 3.4. We report three metrics: Execution Pass Rate, Task Score, and Visual Score, capturing executability, semantic alignment, and perceptual similarity. Models are also tested under the self-debug protocol with up to three rounds of correction based on execution feedback, assessing both baseline generation and recovery through iterative refinement.

C TRAINING DATA DISTRIBUTION

Table 5: Distribution of visualization code samples across languages and sources. The final column reports per-language totals, and the final row reports per-source totals.

[Back to Appendix Contents]

Language	CoSyn-400K	the-stack-v2	svg-diagrams	Total
Python	66,052	120,902	-	186,954
HTML	75,315	59,915	-	135,230
LaTeX	124,039	_	_	124,039
SVG	2,693	-	43,928	46,621
JavaScript	_	28,807	_	28,807
Asymptote	22,539	- -	_	22,539
C++	- -	16,776	_	16,776
R	_	13,437	_	13,437
Mermaid	13,381	- -	_	13,381
LilyPond	12,093	_	_	12,093
Vega-Lite	6,790	_	_	6,790
TypeScript	•	6,315	-	6,315
Total	322,902	246,152	43,928	612,982

D ABLATION STUDY

To disentangle the contribution of each data source, we conduct a controlled ablation study using Qwen2.5-Coder-7B as the base model. Separate models are fine-tuned on individual subsets of The-Stack-V2, CoSyn, StarVector, and Code-Feedback, under the same instruction-tuning setup as the full configuration. We report execution pass rates on VisPlotBench in both default and self-debug modes, with comparisons to the untuned Qwen2.5-Coder-7B baseline and the full VisCode-Multi-679K model (Table 6).

Table 6: Execution pass rates of Qwen2.5-Coder-7B models trained on individual subsets of VisCode-Multi-679K. Each model is evaluated under both default (✗) and self-debug (✔) modes.

[Back to Appendix Contents]

Model	Self-Debug	Overall	Python	Vega-Lite	LilyPond	Mermaid	SVG	LaTeX	Asymptote	HTML
Qwen2.5-Coder-7B-Ins.	×	51.2 59.0	41.3 61.7	76.0 77.5	5.5 5.5	77.9 79.4	92.3 92.3	25.9 30.4	13.0 20.7	64.8 76.9
+ The-Stack-V2-246K	×	49.0 56.5	47.5 58.2	81.4 83.7	7.3 10.9	69.5 73.3	84.6 84.6	0.9 31.3	17.4 18.5	64.8 65.7
+ CoSyn-323K	×	59.2 62.2	25.5 31.1	83.7 84.5	65.5 69.1	57.3 61.1	100.0 100.0	36.6 38.4	56.5 62.0	91.7 91.7
+ StarVector-44K	×	40.1 44.5	43.4 53.6	72.1 73.6	5.5 7.3	67.9 70.2	16.9 18.5	10.7 13.4	13.0 19.6	47.2 50.0
+ Code-Feedback-66K	×	55.2 63.1	47.5 62.2	78.3 80.6	20.0	81.7 81.7	92.3 92.3	27.7 38.4	17.4 23.9	65.7 83.3
+ Full VisCode-Multi-679K	×	70.9 76.4	64.8 77.0	83.0 84.5	69.1 72.7	78.6 84.7	96.9 96.9	39.3 42.9	64.1 70.7	82.4 84.3

Natural vs. Synthetic. Training on The-Stack-V2 alone yields limited improvements and even degrades symbolic languages such as LaTeX, reflecting the sparsity of clean visualization signals in general-purpose code. By contrast, CoSyn delivers large gains on symbolic and grammar-sensitive languages, with execution rates on LilyPond and Asymptote rising by over 60 points compared to the baseline. This contrast shows that large-scale synthetic data provides valuable structural coverage that complements natural code.

Domain vs. Multi-turn. The StarVector subset contributes primarily to SVG but is too small to improve overall performance. In contrast, Code-Feedback does not drastically shift baseline pass rates but produces consistent gains under self-debug, lifting overall execution from 55.2% to 63.1%. This demonstrates that multi-turn dialogue data provides critical supervision for recovery through iterative correction, rather than improving one-shot generation.

Full Dataset Synergy. Combining all subsets yields the strongest model. With VisCode-Multi-679K, the overall pass rate reaches 70.9% in default mode and 76.4% with self-debug, substantially surpassing both the untuned baseline and any single-source variant. These results confirm that the dataset's diverse composition—balancing natural, synthetic, domain-specific, and iterative data—is essential for building robust multi-language visualization coding agents.

E LIMITATION

While VisCode-Multi-679K and VisPlotBench represent a significant step toward building reliable multi-language visualization coding agents, several limitations remain. First, the training corpus, although spanning twelve languages, is still imbalanced: common ecosystems such as Python and Vega-Lite are well represented, whereas symbolic and domain-specific languages have far fewer examples. This uneven distribution may bias models toward dominant languages and limit generalization in low-resource cases. Second, VisPlotBench currently supports eight languages; expanding benchmark coverage to a broader set of visualization frameworks would enable more comprehensive evaluation and reduce the risk of overfitting to the existing taxonomy. Addressing these limitations will be crucial for advancing toward fully general-purpose visualization coding agents.

F PROMPT USED AND INSTRUCT DESIGN

In this section, we present the prompts used during the construction of VisCode-Multi-679K and VisPlotBench.

F.1 PROMPT USED IN VISCODE-MULTI-679K

Code Extraction Prompt

Model: GPT-4.1-mini

```
# LANGUAGE= [Python, JavaScript, TypeScript, C++, R, HTML]
# LANG_BULLET = {

Puthon: Write a single by file No external files or internet access. Use helpe
```

Python: Write a single .py file. No external files or internet access. Use helper libraries only if truly required for the chart.

JavaScript: Write a single . js file. Add necessary imports for any required libraries.

TypeScript: Write a single .ts file. Use ES6 module syntax (import/export). The code should not require a module bundler.

C++: Write a single .cpp file with main(). Program must exit automatically. Do not load external assets. If linking is needed, add one build command as a comment.

R: Write a single .R file. Use library (Library). Create mock data if needed. No external files. HTML: Write a single HTML file. Include a <script> tag and the needed DOM element for the chart. All code runs in the browser.

You are a {LANGUAGE} visualization code extraction agent.

Given a {LANGUAGE} code snippet and the used library, your task is to extract a **minimal yet runnable** {LANGUAGE} snippet that reflects how the library is actually used **for visual output**.

Guidelines: - Keep only the logic needed for the **visual output**; remove unrelated code.

- If the library is **not used for rendering/drawing/plotting** in Code, return "null".
- If inputs/assets are missing, create semantically relevant mock data that makes the output meaningful.
- Preserve the **main intent**, API pattern, key parameters, and style** from the original code; simplify when it improves clarity, and avoid adding new wrappers or layers that are not essential.
- Make the **visual output** clear and professional: use appropriate visual cues (titles/labels/legends when applicable); keep layout readable.
- Ensure the snippet runs standalone in a minimal environment and **terminates automatically** (no user input required).
- {LANG_BULLET}
- If the library is unused or information is insufficient, return "null".

```
Used Library: {used_libs}
Code: {code}
```

1080 Instruct Generation Prompt: the-stack-v2& svg-diagrams 1081 1082 Model: GPT-4.1 # LANGUAGE=[Python, JavaScript, TypeScript, C++, R, HTML, SVG] 1084 1085 You are given a {LANGUAGE} code snippet that renders an image. A rendered image of the resulting 1086 output is provided at the end. Your task is to infer and clearly describe the purpose, structure, and style 1087 of this image. 1088 Break your response into the following five parts: 1089 1. Setup (state the language and rendering context, including any tools or libs implied). 1090 2. Data/Visual Description 1091 - If the code is data-driven: summarize the inputs the code relies on and any shaping operations. - If the code is not data-driven: summarize the visible content of the image. 3. Data Generation (the data-generation lines copied verbatim, or "None" if not applicable). 1093 4. Output Description (omit language constructs; start with "Generate..." or "Create...", and describe 1094 the final image conceptually). 1095 5. Style Description (describe appearance and layout without naming language constructs). 1096 Each part must start on a new line, numbered 1 through 5. Use plain text only; no markdown. 1098 1099 Code: 1100 {code} 1101 1102 Image: 1103 1104 Instruct Generation Prompt: CoSyn-400K 1105 1106 Model: GPT-4.1 1107 1108 # FOR DATA-DRIVEN LANGUAGES 1109 # LANGUAGE= [Python, Vega-Lite, HTML, LilyPond, Mermaid] 1110 You are given a {LANGUAGE} code snippet that produces a rendered visual. A rendered image of 1111 the resulting output is provided at the end. Your task is to infer and clearly describe the purpose and 1112 structure of this visual. 1113 1114 Break your response into the following four parts: 1. Setup (state the {LANGUAGE} and its rendering context, including any tools or specification 1115 frameworks implied). 1116 2. Data/Content Description (summarize the input fields, entities, or content the code relies on, 1117 including any shaping or transformation operations). 1118 3. Output Description (omit library, directive, or element names; start with "Generate..." or "Create...", 1119 and describe the visual conceptually). 4. Style Description (describe appearance and layout without naming language constructs). 1120 1121 Each part must start on a new line, numbered 1 through 4. 1122 Use plain text only; no markdown. 1123 1124 Code: 1125 {code} 1126 Image:

Instruct Generation Prompt: CoSyn-400K Model: GPT-4.1 #FOR NONE DATA-DRIVEN LANGUAGES # LANGUAGE= [Asymptote, SVG] You are given a {LANGUAGE} code snippet that renders an image. A rendered image of the resulting output is provided at the end. Your task is to infer and clearly describe the purpose and structure of this image. Break your response into the following four parts: 1. Setup (state the {LANGUAGE} and its rendering context). 2. Visual Elements (summarize the visible components of the image). 3. Output Description (omit language constructs; start with "Generate..." or "Create...", and describe the image conceptually). 4. Style Description (describe appearance and layout without naming language constructs). Each part must start on a new line, numbered 1 through 4. Use plain text only; no markdown. Code: {code} Image:

1188 F.2 PROMPT USED IN VISPLOTBENCH 1189 1190 Task & Style Description Generation Prompt: 1191 Model: GPT-4.1 1192 # LANGUAGE= [Python, Vega-Lite, HTML, LilyPond, Mermaid, Asymptote, HTML, SVG] 1193 1194 You are given a {LANGUAGE} code that produces a rendered visual. A rendered image of the resulting 1195 output is provided at the end. Your task is to infer and clearly describe the purpose and structure of this visual. 1196 Break your response into the following two parts: 1. Task Description (omit libraries and specific 1197 function names at this part, start with "Generate..." or "Create..."). 2. Style Description (describe 1198 appearance and layout without using specification keywords). 1199 Each part must start on a new line, numbered 1 through 2. Use plain text only; no markdown. CODE: code 1201 IMAGE: 1202 Vis & Task Judge Prompt 1203 1204 #Visual Judge 1205 You are an excellent judge at evaluating visualization plots between a model generated plot and the ground truth. 1207 You will be giving scores on how well it matches the ground truth plot. 1208 The generated plot will be given to you as the first figure. Another plot will be given to you as the second figure, which is the desired outcome of the user query, 1209 meaning it is the ground truth for you to reference. 1210 Please compare the two figures head to head and rate them. 1211 Suppose the second figure has a score of 100, rate the first figure on a scale from 0 to 100. 1212 Scoring should be carried out in the following aspect: 1213 Plot correctness: compare closely between the generated plot and the ground truth, the more resemblance the generated plot has compared to the ground truth, the higher the score. The score 1214 should be proportionate to the resemblance between the two plots. 1215 Ignore color matching. If the plots present the same information but are made in different colors, 1216 consider them matching. Capture the resemblance of the main idea of the plot. 1217 Only rate the first figure, the second figure is only for reference. 1218 After scoring from the above aspect, please give a final score. Do not write anything else. The final score is preceded by the [FINAL SCORE] token. 1219 For example [FINAL SCORE]: 40 1220 1222 #Task Judge 1223 You are an excellent judge at evaluating visualization plot according to the given task. You will be giving scores on how well plot image matches the task. 1224 The generated plot will be given to you as an image. 1225 Please score how well plot matches the task. Score it on a scale from 0 to 100. 1226 Scoring should be carried out in the following aspect: 1227 Task adherence: how the plot corresponds to the task given below (begins from [PLOT TASK] token). After scoring from the above aspect, please give a final score. Do not write anything else. The final 1228 score is preceded by the [FINAL SCORE] token. 1229 For example [FINAL SCORE]: 40 1230 1231 1232 1233

1237

F.3 INSTRUCT DESIGN IN VISPLOTBENCH EVALUATION

Python Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in Python. All answers must be enclosed in a block "python SOME CODE" containing one complete Python code. Example minimal spec: "python

```
print('hello world')
""
```

SETUP INSTRUCT:

Use Python programming language. Import essential libraries. The essential libraries needed are pandas for managing dataframes and [USED_LIB] with its subsidiary libraries for plotting. Ensure importing numpy as np and scipy if they are used in program. DO NOT use or import other visualization libraries.

#PLOT INSTRUCT:

Write a code to build a plot of dataframe according to following instructions. Write a code that returns plot, not just function declaration. Do not write explanations, just a code enclosed in codeblock. Important reasoning write in comments to the code. Make sure that all used libraries and functions are imported.

#DATA INSTRUCT:

Load df dataframe by single line df = pd.read_csv("data.csv"). DO NOT alter df dataframe columns or add columns. This df dataframe should remain intact. The metadata of the dataframe is following:

Vega-Lite Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in Vega-Lite. All answers must be enclosed in a block "vegalite SOME CODE" containing one complete Vega-Lite specification. Example minimal spec: "vegalite

```
{
    "$schema":"https://vega.github.io/schema/vega-lite/v6.json",
    "data":{"values":[{"hello":"world"}]},
    "mark":"text",
    "encoding":{"text":{"field":"hello","type":"nominal"}}
}
```

#SETUP INSTRUCT:

Setup. Use the Vega-Lite v6 JSON schema and produce exactly one valid Vega-Lite specification as a single top-level JSON object that MUST include the \$schema property. Do not output raw Vega specifications, imperative code, language-specific wrappers, or references to other plotting libraries; use only Vega-Lite encodings, transforms, and configuration required by the plot.

#PLOT INSTRUCT:

Write a code to build a plot of the dataset according to the following instructions. Return one complete Vega-Lite JSON specification enclosed in a single code block, and do not include explanations or comments. Use only the constructs permitted by the setup, ensure that all referenced field names exactly match the dataset metadata, and do not rename or drop columns; only use non-destructive Vega-Lite transforms if required.

DATA INSTRUCT:

Data description. Load the dataset by setting "data": "url": "data.csv". Do not create synthetic data or load inline data values. The metadata of the dataset is following:

Mermaid Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in Mermaid diagrams. All answers must be enclosed in a block "mermaid SOME CODE" containing one complete Mermaid specification. Example minimal spec: "mermaid

```
graph TD
   A[Start] -\rightarrow B{Condition}
   B -\rightarrow|Yes| C[Do something]
   B -\rightarrow|No| D[Stop]
...
```

SETUP INSTRUCT:

Setup. Use Mermaid syntax only and produce exactly one valid Mermaid diagram definition. Do not output explanations, comments outside the code block, or code in other languages or formats. Do not split the diagram into multiple blocks. Ensure the code can be rendered directly by mermaid-cli (mmdc).

#PLOT INSTRUCT:

Write a diagram according to the following instructions. Do not include explanations or natural language outside the code block. Ensure that the diagram is self-contained, syntactically correct Mermaid code, and does not rely on external data or libraries. Use node and edge labels exactly as provided in the instructions.

#DATA INSTRUCT:

Data description. The diagram is constructed only from the provided instructions. Do not load external files or datasets.

LilyPond Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in LilyPond. Always use the version statement 'version "2.22.1". All answers must be enclosed in a block "'lilypond SOME CODE" containing one complete LilyPond score. Example minimal spec: "'lilypond

```
\version "2.22.1"
\score {
     \new Staff { c' d' e' f' }
     \layout { }
}
```

SETUP INSTRUCT:

Setup. Use LilyPond syntax only and produce exactly one valid LilyPond music notation definition. Do not output explanations, comments outside the code block, or code in other languages or formats. Do not split the notation into multiple blocks. Ensure the code can be rendered directly by LilyPond.

#PLOT INSTRUCT:

Write a music notation according to the following instructions. Do not include explanations or natural language outside the code block. Ensure that the notation is self-contained, syntactically correct LilyPond code, and does not rely on external data or libraries. Use note names and other musical symbols exactly as provided in the instructions.

#DATA INSTRUCT:

Data description. The music notation is constructed only from the provided instructions. Do not load external files or datasets.

SVG Instruct

SYSTEM PROMPT:

You are a helpful programming assistant proficient in SVG. All answers must be enclosed in a block "svg SOME CODE" containing one complete SVG specification. Example minimal spec: "svg

```
<svg width="100" height="100" xmlns="http://www.w3.org/2000/svg">
    <circle cx="50" cy="50" r="40" stroke="black"
        stroke-width="2" fill="red" />
    </svg>
```

SETUP INSTRUCT:

Setup. Use SVG syntax only and produce exactly one valid SVG definition. Do not output explanations, comments outside the code block, or code in other languages or formats. Do not split the diagram into multiple blocks. Ensure the code can be rendered directly by SVG viewers.

#PLOT INSTRUCT:

Write an SVG according to the following instructions. Do not include explanations or natural language outside the code block. Ensure that the SVG is self-contained, syntactically correct SVG code, and does not rely on external data or libraries. Use shapes and attributes exactly as provided in the instructions.

#DATA INSTRUCT:

Data description. The SVG is constructed only from the provided instructions. Do not load external files or datasets.

Asymptote Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in Asymptote. All answers must be enclosed in a block "'asymptote SOME CODE" containing one complete Asymptote specification. Example minimal spec: "'asymptote

```
import graph;
size(100);
draw((0,0)--(1,1));
...
```

SETUP INSTRUCT:

Setup. Use Asymptote syntax only and produce exactly one valid Asymptote definition. Do not output explanations, comments outside the code block, or code in other languages or formats. Do not split the diagram into multiple blocks. Ensure the code can be rendered directly by Asymptote.

#PLOT INSTRUCT:

Write an Asymptote according to the following instructions. Do not include explanations or natural language outside the code block. Ensure that the Asymptote is self-contained, syntactically correct Asymptote code, and does not rely on external data or libraries. Use shapes and attributes exactly as provided in the instructions.

#DATA INSTRUCT:

Data description. Please use the provided data definition code to construct the plot. Do not modify this code or create data by yourself. Use the variables defined in this code directly when building the plot. The data definition code is as follows:

LaTeX Instruct

SYSTEM PROMPT:

You are a helpful programming assistant proficient in LaTeX. All answers must be enclosed in a block "latex SOME CODE" containing one complete LaTeX document. Example minimal spec: "latex

\documentclass{standalone}

1410 \begin{document}

Hello

1411 \end{document}

1414 # SETUP INSTRUCT:

Setup. Use LaTeX syntax only and produce exactly one valid LaTeX document as a single code block. Do not output explanations, comments outside the code block, or code in other languages or formats. Ensure the document can be rendered directly by LaTeX compilers.

1418 # PLOT INSTRUCT:

Write a LaTeX code to build a plot of the dataset according to the following instructions. Return exactly one complete LaTeX document enclosed in a single code block. Include all required packages. Do not include explanations or comments. Do not create synthetic data or modify the dataset.

#DATA INSTRUCT:

Data description. Load the dataset by adding \pgfplotstablereadlatex.csv\datatable. Do not create synthetic data or modify the dataset. The metadata of the dataset is following:

HTML Instruct

#SYSTEM PROMPT:

You are a helpful programming assistant proficient in HTML. All answers must be enclosed in a block "html SOME CODE" containing one complete HTML document Example minimal spec: "html

```
<!DOCTYPE html>
<html>
<body>Hello</body>
</html>
""
```

#SETUP INSTRUCT:

Setup. Use HTML syntax only and produce exactly one valid HTML document as a single code block. Do not output explanations, comments outside the code block, or code in other languages or formats. Ensure the document can be rendered directly by web browsers.

#PLOT INSTRUCT:

Write an HTML code to build a plot of the dataset according to the following instructions. Return exactly one complete HTML document enclosed in a single code block. Include all required libraries and scripts. Do not include explanations or comments. Do not create synthetic data or modify the dataset.

#DATA INSTRUCT:

Data description. Load the dataset by defining: const data = [html.csv]; [html.csv] is a placeholder for the parsed CSV rows. Assume that the placeholder will be replaced at runtime by the CSV content converted into a JavaScript array of objects (i.e., a list of dicts), where each object represents one row with column names as keys and cell values as values. Write the code as if "data" is already such a valid JavaScript array of objects. Do not create synthetic data or modify the dataset. The metadata of the dataset is following:

G TAXONOMY OF VISUALIZATION TYPES AND SUBTYPES

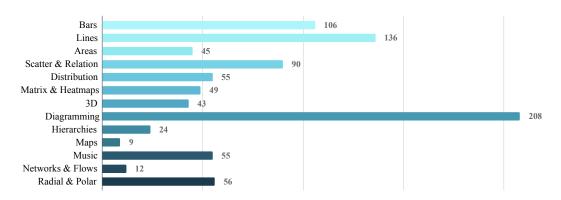


Figure 4: Distribution of fine-grained visualization types in VisPlotBench. Tasks are organized into 13 Visual categories and 116 Subtypes, ensuring broad coverage of both common and underexplored visualization families.

[Back to Appendix Contents]

Table 7: Taxonomy of Visual Categories and Subtypes [Back to Appendix Contents]

Visual Category	Subtype	Count	Visual Category	Subtype	Count
	vertical-bar	31		area	17
	horizontal-bar	23		stacked-area	14
	grouped-bar	15		normalized-stacked-area	4
	normalized-stacked-bar stacked-bar	8 7	Areas	difference-area missing-data-matrix	4 3
	diverging-bar	5		ternary-area	1
	dot-plot	5		streamgraph	1
Bars	lollipop	3		ridgeline	1
Dais	sorted-bar	2			
	waterfall	1		bubble	25
	polar-bar bullet	1 1		scatter	24 20
	funnel	1		color-scatter regression-ci	4
	combo-chart	1		ternary-line	4
	missing-bar	1	Scatter & Relation	quadrant-chart	3
	marimekko	1		ellipse-scatter	3
				polar-line-scatter	3
	single-line	45		splom	2
	multi-line	39		connected-scatter dumbbell chart	1 1
	function-line step-line	26 10		dumbbell chart	1
Lines	gapped-line	6		box-plot	17
	band-line	4		histogram	13
	slope-chart	3		density-contours	5
	candlestick	3		violin	5
				kde-1d	6
	surface	21	Distribution	hexbin-2d	2 2
	multi-line scatter	3 4		qq-plot rug-plot	2
	point-cloud	3		ridgeline	1
	solid	3		prediction-interval	1
3D	single-line	2		spectrum	1
	vector-field-map	2		_	
	3d-density-contours	2		heatmap	40
	connected-scatter	1	Matrin 0 Haatman	calendar-heatmap	5
	isosurface slices	1 1	Matrix & Heatmaps	missing-corr-heatmap adjacency-matrix	2 1
	sinces	1		correlation-heatmap	1
	sequence-diagram	37		correlation nearmap	•
	flowchart	25		treemap	10
	geometric-figure	20		sunburst	4
	electrical-circuit-diagram	16	Hierarchies	circle-packing	3
	state-machine	16		missing-dendrogram	3
	table uml-class-diagram	15 12		tidy-tree indented-tree	3 1
	gantt	11		macmed-tree	1
	timeline	11		choropleth	4
	simple-figure	10	Maps	vector-field-map	2
Diagramming	concept-illustration	10	iviaps	dot-map	2
Diagramming	icon	10		proportional-symbol-map	1
	block-diagram physics-diagram	3 2		sankey	5
	venn	2		chord	2
	word-cloud	2	NT 4 1 0 TM	dependency-graph	2
	mind-map	2	Networks & Flows	arc-diagram	1
	color-palette	1		dag-layered	1
	arrow-annotations	1		force-directed	1
	Chemical graph	1		nia.	17
	sankey	1		pie radar	17 10
Music	sheet-music	55		polar-line-scatter	10
1.14010	onest masie	23	Radial & Polar	donut	7
				radial-bar	7
				radial-area	3
				wind-rose	2

H Breakdown Main Results

In this section, we provide a breakdown of model performance in VisPlotBench. For each visualization language,e report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**).

H.1 PYTHON, VEGA-LITE & LILYPOND

Table 8: Performance of selected languages on the VisPlotbench benchmark. For each model, we report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**). The best-performing model in each scale is shown in **bold**, and the second best is underlined.

		Pv	thon (1	196)			Vega-Lite (129)					LilyPond (55)			
Model	Exec		ean		l(>75)	Exec		ean		(>75)	Exec		ean		d(>75)
	Pass	vis	task	vis	task	Pass	vis	task	vis	task	Pass	vis	task	vis	task
GPT-4.1	64.3	53	61	51%	61%	84.5	60	68	56%	66%	43.6	14	38	5%	36%
GPT-4.1 + Self Debug	84.2	66	76	64%	76%	96.1	64	74	60%	72%	63.6	17	54	5%	53%
GPT-4.1-mini	64.8	53	61	47%	59%	84.5	53	63	45%	60%	16.4	2	12	0%	11%
GPT-4.1-mini + Self Debug	80.6	61	71	56%	67%	96.9	60	71	51%	68%	<u>56.4</u>	14	42	0%	35%
					$\sim 3B$	Scale									
DeepSeek-Coder-1.3B-Instruct	29.1	16	19	10%	11%	53.5	1	2	0%	0%	30.9	2	1	0%	0%
Qwen2.5-Coder-3B-Instruct	34.2	23	28	17%	24%	68.2	25	34	13%	22%	3.6	1	2	0%	0%
VisCoder-3B	45.4	32	39	26%	35%	83.7	31	37	20%	26%	21.8	3	7	0%	2%
VisCoder2-3B	56.1	39	45	33%	38%	83.0	41	49	33%	40%	50.9	10	31	2%	15%
VisCoder2-3B + Self Debug	63.3	42	49	35%	40%	84.5	43	50	34%	41%	52.7	10	32	2%	15%
					~ 7B	Scale									
DeepSeek-Coder-6.7B-Instruct	39.3	25	29	19%	23%	79.8	37	47	24%	37%	7.3	0	3	0%	4%
Qwen2.5-Coder-7B-Instruct	41.3	29	37	24%	32%	76.0	40	50	29%	40%	5.5	0	3	0%	4%
VisCoder-7B	58.2	40	48	33%	42%	71.3	39	49	31%	43%	23.6	4	11	2%	4%
VisCoder2-7B	64.8	44	54	37%	49%	83.0	49	58	43%	51%	69.1	16	52	2%	45%
VisCoder2-7B + Self Debug	77.0	50	61	41%	54%	84.5	49	59	43%	52%	72.7	17	55	2%	45%
					~ 14B	Scale									
DeepSeek-Coder-V2-Lite-Instruct	47.5	32	40	28%	36%	75.2	36	43	27%	33%	49.1	9	28	0%	13%
Qwen2.5-Coder-14B-Instruct	50.0	35	43	28%	39%	83.0	52	61	42%	53%	25.5	5	12	2%	4%
VisCoder2-14B	65.3	47	56	39%	52%	93.0	55	63	47%	58%	54.6	11	44	0%	40%
VisCoder2-14B + Self Debug	78.1	55	64	46%	58%	94.6	56	64	47%	60%	63.6	12	47	0%	40%
					~ 32B	Scale									
DeepSeek-Coder-33B-Instruct	58.2	40	48	34%	41%	90.7	52	61	40%	51%	30.9	3	11	0%	4%
Qwen2.5-Coder-32B-Instruct	50.5	36	43	30%	41%	83.0	48	57	39%	49%	30.9	5	22	2%	18%
VisCoder2-32B	65.3	49	56	42%	54%	94.6	60	70	53%	65%	56.4	14	39	2%	27%
VisCoder2-32B + Self Debug	81.6	58	68	46%	62%	96.1	62	72	54%	67%	69.1	16	48	2%	35%

H.2 MERMAID, SVG & LATEX

Table 9: Performance of selected languages on the VisPlotbench benchmark. For each model, we report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**). The best-performing model in each scale is shown in **bold**, and the second best is underlined.

		Mei	rmaid	(131)			S	VG (6	5)			La	TeX (1	.12)	
Model	Exec	M	ean	Good	(≥75)	Exec	M	ean	Good	(≥75)	Exec	M	ean	Good	l(≥75)
	Pass	vis	task	vis	task	Pass	vis	task	vis	task	Pass	vis	task	vis	task
GPT-4.1	68.7	41	57	22%	56%	95.4	45	92	14%	94%	31.3	18	26	13%	25%
GPT-4.1 + Self Debug	93.9	56	77	32%	73%	96.9	45	93	14%	95%	66.1	38	56	25%	51%
GPT-4.1-mini	51.9	33	45	18%	43%	95.4	41	86	11%	86%	29.5	21	25	18%	25%
GPT-4.1-mini + Self Debug	94.7	58	79	26%	74%	96.9	42	88	11%	88%	<u>58.9</u>	35	50	23%	49%
					\sim 3B	Scale									
DeepSeek-Coder-1.3B-Instruct	63.4	19	25	2%	8%	7.7	1	1	0%	0%	4.5	2	1	2%	1%
Qwen2.5-Coder-3B-Instruct	74.1	30	38	9%	21%	75.4	18	39	2%	28%	17.9	6	9	3%	5%
VisCoder-3B	<u>75.6</u>	32	40	12%	21%	76.9	13	31	0%	12%	23.2	9	12	7%	9%
VisCoder2-3B	76.3	43	59	23%	50%	87.7	25	59	3%	48%	36.6	14	21	3%	12%
VisCoder2-3B + Self Debug	76.3	43	59	23%	50%	87.7	25	59	3%	48%	38.4	14	23	3%	13%
					~ 7B	Scale									
DeepSeek-Coder-6.7B-Instruct	91.6	40	50	11%	28%	96.9	19	46	0%	22%	18.8	6	11	3%	8%
Qwen2.5-Coder-7B-Instruct	77.9	39	53	13%	38%	92.3	23	58	0%	40%	25.9	11	15	6%	8%
VisCoder-7B	77.1	41	54	17%	43%	93.9	23	53	2%	32%	25.9	10	15	6%	12%
VisCoder2-7B	78.6	43	59	20%	53%	96.9	34	73	3%	62%	39.3	15	23	6%	15%
VisCoder2-7B + Self Debug	84.7	45	62	21%	54%	96.9	34	73	3%	62%	42.9	16	24	6%	15%
					~ 14B	Scale									
DeepSeek-Coder-V2-Lite-Instruct	69.5	34	46	12%	34%	93.9	23	55	2%	34%	29.5	10	16	4%	10%
Qwen2.5-Coder-14B-Instruct	74.8	39	56	15%	48%	98.5	33	80	5%	77%	30.4	15	22	6%	15%
VisCoder2-14B	81.7	53	67	32%	62%	89.2	34	72	8%	65%	42.0	22	33	12%	27%
VisCoder2-14B + Self Debug	86.3	55	70	33%	64%	90.8	34	72	8%	65%	45.5	24	35	12%	28%
					~ 32B	Scale									
DeepSeek-Coder-33B-Instruct	87.0	44	57	15%	40%	92.3	23	58	0%	43%	24.1	8	14	4%	11%
Qwen2.5-Coder-32B-Instruct	71.0	41	56	21%	53%	93.9	34	81	3%	75%	29.5	14	25	9%	27%
VisCoder2-32B	87.0	51	67	31%	62%	81.5	33	68	11%	63%	42.9	20	35	11%	34%
VisCoder2-32B + Self Debug	90.1	54	69	34%	63%	86.2	34	71	11%	66%	61.6	28	45	14%	42%

H.3 ASYMPTOTE & HTML

Table 10: Performance of selected languages on the VisPlotbench benchmark. For each model, we report (1) execution pass rate (**Exec Pass**), (2) mean visual and task scores (**Mean**), and (3) the proportion of samples scoring at least 75 (**Good**). The best-performing model in each scale is shown in **bold**, and the second best is underlined.

		Asy	mptote	(92)			\mathbf{H}^{\prime}	ΓML (1	(80	
Model	Exec	M	ean	Good	(>75)	Exec	M	lean	Good	l(> 75)
	Pass	vis	task	vis	task	Pass	vis	task	vis	task
GPT-4.1	21.7	12	20	7%	20%	89.8	48	64	21%	50%
GPT-4.1 + Self Debug	46.7	22	41	9%	39%	97.2	51	68	22%	52%
GPT-4.1-mini	23.9	13	22	7%	21%	86.1	36	53	11%	34%
GPT-4.1-mini + Self Debug	48.9	21	40	9%	36%	100	42	62	12%	42%
		~	3B Sca	ale						
DeepSeek-Coder-1.3B-Instruct	13.0	0	0	0%	0%	36.1	2	3	1%	0%
Qwen2.5-Coder-3B-Instruct	18.5	8	11	4%	9%	62.0	16	19	6%	7%
VisCoder-3B	30.4	7	12	3%	8%	79.6	21	29	9%	179
VisCoder2-3B	62.0	23	36	7%	26%	93.5	34	 47	8%	239
VisCoder2-3B + Self Debug	63.0	23	37	7%	27%	94.4	34	47	8%	239
		~	7B Sca	ale						
DeepSeek-Coder-6.7B-Instruct	0	0	0	0%	0%	22.2	5	8	1%	3%
Qwen2.5-Coder-7B-Instruct	13.0	7	10	5%	9%	64.8	20	31	6%	139
VisCoder-7B	17.4	7	11	3%	9%	75.9	20	32	5%	169
VisCoder2-7B	64.1	27	43	11%	33%	82.4	30	46	7%	199
VisCoder2-7B + Self Debug	70.7	29	47	11%	35%	84.3	31	47	7%	219
		~	14B Sc	ale						
DeepSeek-Coder-V2-Lite-Instruct	20.7	5	10	1%	9%	64.8	21	32	4%	189
Qwen2.5-Coder-14B-Instruct	25.0	12	17	9%	16%	83.3	34	50	9%	319
VisCoder2-14B	56.5	_ _ _ _	45	15%	41%	90.7	41	₋ -	12%	369
VisCoder2-14B + Self Debug	66.3	31	50	16%	45%	94.4	42	60	13%	379
		~	32B Sc	ale						
DeepSeek-Coder-33B-Instruct	21.7	8	14	2%	9%	12.0	4	6	0%	0%
Qwen2.5-Coder-32B-Instruct	17.4	9	13	5%	12%	78.7	33	49	11%	329
VisCoder2-32B	58.7	⁻ 27	46	10%	39%	91.7	43	61	18%	489
VisCoder2-32B + Self Debug	$\overline{71.7}$	31	53	10%	41%	93.5	44	62	18%	499

I Breakdown Self-Debug Results

In this section, we provide a breakdown of model performance under the self-debug setting. For each language, we report execution pass rates across up to three rounds of automatic correction, grouped by model series

I.1 PYTHON & VEGA-LITE

Table 11: Execution pass rates (%) in Python and Vega-Lite under the normal and self-debug settings. Models that fail initially are allowed up to three rounds of automatic correction. Left columns show Python results, right columns show Vega-Lite results.

[Back to Appendix Contents]

Model	Normal	Python Self Debug			Normal	Vega-Lite Self Debug			
		Round 1	Round 2	Round 3		Round 1	Round 2	Round 3	
GPT-4.1	64.3	75.0	81.6	84.2	84.5	95.3	96.1	96.1	
GPT-4.1-mini	64.8	73.5	79.1	80.6	84.5	95.3	96.9	96.9	
∼ 3B Scale									
DeepSeek-Coder-1.3B-Instruct	29.1	35.7	35.7	35.7	53.5	53.5	53.5	53.5	
Qwen2.5-Coder-3B-Instruct	34.2	39.8	41.8	42.9	68.2	68.2	69.0	69.0	
VisCoder-3B	45.4	51.0	52.6	52.6	83.7	83.7	83.7	83.7	
VisCoder2-3B	56.1	61.7	62.8	63.3	83.0	84.5	84.5	84.5	
		^	~ 7B Scale						
DeepSeek-Coder-6.7B-Instruct	39.3	46.9	49.5	53.1	79.8	81.4	81.4	81.4	
Qwen2.5-Coder-7B-Instruct	41.3	53.6	60.2	61.7	76.0	77.5	77.5	77.5	
VisCoder-7B	58.2	66.8	68.9	71.9	71.3	76.0	77.5	77.5	
VisCoder2-7B	64.8	72.5	76.0	77.0	83.0	84.5	84.5	84.5	
∼ 14B Scale									
DeepSeek-Coder-V2-Lite-Instruct	47.5	54.6	55.6	58.7	75.2	78.3	79.8	79.8	
Qwen2.5-Coder-14B-Instruct	50.0	65.3	72.5	76.0	83.0	86.8	86.8	86.8	
VisCoder2-14B	65.3	76.5	78.1	78.1	93.0	93.8	94.6	94.6	
\sim 32B Scale									
DeepSeek-Coder-33B-Instruct	58.2	67.9	71.4	73.0	90.7	92.3	92.3	92.3	
Qwen2.5-Coder-32B-Instruct	50.5	70.9	78.1	79.1	83.0	87.6	89.9	89.9	
VisCoder2-32B	65.3	76.0	80.1	81.6	94.6	96.1	96.1	96.1	

I.2 LILYPOND & MERMAID

Table 12: Execution pass rates (%) in LilyPond and Mermaid under the normal and self-debug settings. Models that fail initially are allowed up to three rounds of automatic correction. Left columns show LilyPond results, right columns show Mermaid results. s]

M. 1.1	LilyPond Self Debug				N I I	Mermaid Self Debug		
Model	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
GPT-4.1	43.6	54.5	63.6	63.6	68.7	84.7	93.0	93.9
GPT-4.1-mini	16.4	30.9	47.3	56.4	51.9	81.7	90.1	94.7
		^	~ 3B Scale					
DeepSeek-Coder-1.3B-Instruct	30.9	32.7	32.7	32.7	63.4	76.3	77.9	78.6
Qwen2.5-Coder-3B-Instruct	3.6	5.5	5.5	5.5	74.1	76.3	76.3	76.3
VisCoder-3B	21.8	21.8	21.8	21.8	75.6	76.3	76.3	76.3
VisCoder2-3B	50.9	52.7	52.7	52.7	76.3	76.3	76.3	76.3
		^	~ 7B Scale					
DeepSeek-Coder-6.7B-Instruct	7.3	9.1	10.9	10.9	91.6	93.9	94.7	94.7
Qwen2.5-Coder-7B-Instruct	5.5	5.5	5.5	5.5	77.9	79.4	79.4	79.4
VisCoder-7B	23.6	27.3	30.9	30.9	77.1	80.9	80.9	80.9
VisCoder2-7B	69.1	72.7	72.7	72.7	78.6	84.0	84.7	84.7
		~	14B Scale					
DeepSeek-Coder-V2-Lite-Instruct	49.1	52.7	52.7	52.7	69.5	69.5	69.5	71.0
Qwen2.5-Coder-14B-Instruct	50.0	65.3	72.5	76.0	83.0	86.8	86.8	86.8
VisCoder2-14B	54.6	63.6	63.6	63.6	81.7	86.3	86.3	86.3
		~	32B Scale					
DeepSeek-Coder-33B-Instruct	30.9	40.0	41.8	41.8	87.0	87.0	87.8	88.6
Qwen2.5-Coder-32B-Instruct	30.9	40.0	43.6	43.6	71.0	74.8	75.6	76.3
VisCoder2-32B	56.4	61.8	69.1	69.1	87.0	89.3	90.1	90.1

I.3 SVG & LATEX

Table 13: Execution pass rates (%) in SVG and LaTeX under the normal and self-debug settings. Models that fail initially are allowed up to three rounds of automatic correction. Left columns show SVG results, right columns show LaTeX results.

Model	SVG Self Debug					LaTeX Self Debug		
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
GPT-4.1	95.4	96.9	96.9	96.9	31.3	53.6	59.8	66.1
GPT-4.1-mini	95.4	96.9	96.9	96.9	29.5	50.9	55.4	58.9
		~ 3B 5	Scale					
DeepSeek-Coder-1.3B-Instruct	7.7	95.4	95.4	95.4	4.5	5.4	5.4	5.4
Qwen2.5-Coder-3B-Instruct	75.4	75.4	75.4	75.4	17.9	17.9	17.9	17.9
VisCoder-3B	76.9	76.9	76.9	76.9	23.2	25.9	25.9	25.9
VisCoder2-3B	87.7	87.7	87.7	87.7	36.6	38.4	38.4	38.4
		^	~ 7B Scale					
DeepSeek-Coder-6.7B-Instruct	96.9	98.5	98.5	98.5	18.8	19.6	22.3	22.3
Qwen2.5-Coder-7B-Instruct	92.3	92.3	92.3	92.3	25.9	28.6	30.4	30.4
VisCoder-7B	93.9	93.9	93.9	93.9	25.9	38.4	42.0	43.8
VisCoder2-7B	96.9	96.9	96.9	96.9	39.3	42.9	42.9	42.9
		~	14B Scale					
DeepSeek-Coder-V2-Lite-Instruct	93.9	93.9	93.9	93.9	29.5	33.9	35.7	35.7
Qwen2.5-Coder-14B-Instruct	98.5	98.5	98.5	98.5	30.4	37.5	38.4	38.4
VisCoder2-14B	89.2	90.8	90.8	90.8	42.0	43.8	45.5	45.5
		~	32B Scale					
DeepSeek-Coder-33B-Instruct	92.3	92.3	92.3	92.3	24.1	28.6	31.3	31.3
Qwen2.5-Coder-32B-Instruct	93.9	93.9	93.9	93.9	29.5	42.9	50.0	51.8
VisCoder2-32B	81.5	84.6	86.2	86.2	42.9	55.4	59.8	61.6

I.4 ASYMPTOTE & HTML

Table 14: Execution pass rates (%) in Asymptote and HTML under the normal and self-debug settings. Models that fail initially are allowed up to three rounds of automatic correction. Left columns show Asymptote results, right columns show HTML results.

[Back to Appendix Contents]

Model	Normal	Asymptote Self Debug			M 1	HTML Self Debug		
	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3
GPT-4.1	21.7	35.9	43.5	46.7	89.8	96.3	97.2	97.2
GPT-4.1-mini	23.9	37.0	42.4	48.9	86.1	99.1	99.1	100
		~ 3B S	Scale					
DeepSeek-Coder-1.3B-Instruct	13.0	17.4	17.4	17.4	36.1	36.1	36.1	36.1
Qwen2.5-Coder-3B-Instruct	18.5	18.5	18.5	18.5	62.0	65.7	70.4	70.4
VisCoder-3B	30.4	31.5	32.6	32.6	79.6	83.3	83.3	83.3
VisCoder2-3B	62.0	63.0	63.0	63.0	93.5	94.4	94.4	94.4
		^	7B Scale					
DeepSeek-Coder-6.7B-Instruct	0.0	1.1	2.2	2.2	22.2	25.0	25.0	25.0
Qwen2.5-Coder-7B-Instruct	13.0	16.3	20.7	20.7	64.8	75.9	76.9	76.9
VisCoder-7B	17.4	26.1	26.1	26.1	75.9	81.5	82.4	82.4
VisCoder2-7B	64.1	68.5	70.7	70.7	82.4	84.3	84.3	84.3
		~	14B Scale					
DeepSeek-Coder-V2-Lite-Instruct	20.7	23.9	26.1	26.1	64.8	76.9	79.6	79.6
Qwen2.5-Coder-14B-Instruct	25.0	32.6	39.1	40.2	83.3	89.8	89.8	89.8
VisCoder2-14B	56.5	64.1	66.3	66.3	90.7	94.4	94.4	94.4
		~	32B Scale					
DeepSeek-Coder-33B-Instruct	21.7	26.1	28.3	29.4	12.0	14.8	14.8	14.8
Qwen2.5-Coder-32B-Instruct	17.4	25.0	31.5	33.7	78.7	88.9	89.8	89.8
VisCoder2-32B	58.7	68.5	71.7	71.7	91.7	92.6	93.5	93.5

J Breakdown Error Type Results

J.1 PYTHON

Table 15: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in Python. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Euron Tono		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1		Round 3	
AttributeError	17	12	9	8	15	12	12	10	
FileNotFoundError	-	-	-	-	1	1	1	1	
ImportError	2	2	1	0	2	1	1	1	
SchemaValidationError	1	1	1	1	-	-	-	-	
KeyError	-	-	-	-	3	2	1	0	
KeyboardInterrupt	7	7	6	6	9	9	9	9	
CellSizeError	-	-	-	-	1	1	1	1	
DataError	-	-	-	-	2	2	2	2	
NameError	-	-	-	-	2	1	0	0	
RuntimeError	2	0	0	0	-	-	-	-	
SyntaxError	1	1	1	1	1	1	1	1	
TypeError	20	16	14	14	13	7	3	3	
ValueError	20	10	4	1	19	10	8	8	
Total Errors	70	49	36	31	68	47	39	36	

J.2 VEGA-LITE

Table 16: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in Vega-Lite. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

E-man Toma		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	
JSONDecodeError	1	0	0	0	-	-	-	-	
KeyboardInterrupt	1	0	0	0	1	1	1	1	
ParseError	8	2	2	2	2	1	1	1	
TypeError	9	4	2	2	2	1	1	1	
RenderingError	1	0	0	0	2	2	2	2	
Total Errors	20	6	4	4	7	5	5	5	

J.3 LILYPOND

Table 17: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in Lilypond. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Eurou Trono		G]	PT-4.1			VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3		
FileNotFoundError	1	1	1	1	1	1	1	1		
MarkupError	3	2	2	2	4	4	4	4		
SyntaxError	25	17	12	12	14	12	10	10		
TypeError	2	2	2	2	5	4	2	2		
Total Errors	31	25	20	20	24	21	17	17		

J.4 MERMAID

Table 18: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in Mermaid. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Ermon Tuno		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2 0 9 - 1 - 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	Round 3	
StructureError	2	2	0	0	1	0	0	0	
SyntaxError	32	16	7	7	12	10	9	9	
TypeError	2	1	1	1	-	-	-	-	
UnknownDiagramError	4	1	0	0	1	1	1	1	
YAMLException	1	0	0	0	-	-	-	-	
LogicError	-	-	-	-	1	1	1	1	
DiagramLimitError	-	-	-	-	1	1	1	1	
KeyboardInterrupt	-	-	-	-	1	1	1	1	
Total Errors	41	20	8	8	17	14	13	13	

J.5 SVG

Table 19: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in SVG. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Ennon Tymo		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	
SyntaxError	1	1	1	1	4	2	2	2	
UnclosedError	2	1	1	1	8	8	7	7	
Total Errors	3	2	2	2	12	10	9	9	

J.6 LATEX

Table 20: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in LaTeX. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

E		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	
KeyboardInterrupt	16	16	16	15	5	5	5	2	
PackageError	2	2	1	1	-	-	-	-	
RuntimeError	17	9	7	7	27	12	9	6	
StructureError	3	2	2	1	6	3	3	3	
SyntaxError	5	4	4	4	10	6	4	4	
UndefinedError	21	17	15	15	28	26	24	23	
Total Errors	64	50	45	43	77	52	45	38	

J.7 ASYMPTOTE

Table 21: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in Asymptote. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

E		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	
AmbiguousFunctionCall	-	_	_	-	1	1	1	1	
AmbiguousUsageError	1	1	1	1	1	1	1	1	
CastError	2	1	1	1	-	-	-	-	
FunctionSignatureError	28	20	18	16	9	4	3	3	
ModuleLoadError	16	15	13	13	2	2	2	2	
RuntimeError	1	1	1	1	8	7	6	6	
SyntaxError	3	2	1	1	-	-	-	-	
VariableError	21	19	17	16	15	12	11	11	
KeyboardInterrupt	-	-	-	-	2	2	2	2	
Total Errors	72	59	52	49	38	29	26	26	

J.8 HTML

Table 22: Distribution of execution errors for GPT-4.1 and VisCoder2-32B in HTML. Each column shows error counts at different self-debugging rounds after initial failure.

[Back to Appendix Contents]

Euman Tema		G	PT-4.1		VisCoder2-32B				
Error Type	Normal	Round 1	Round 2	Round 3	Normal	Round 1	Round 2	Round 3	
ConsoleError	1	1	1	1	3	2	2	2	
PageError	9	2	1	1	3	3	2	2	
RequestFailed	1	1	1	1	3	3	3	3	
Total Errors	11	4	3	3	9	8	7	7	

K CASE STUDY

In this section, we present a set of representative examples from VisCoder2-32B to illustrate model behavior across the eight visualization languages.

K.1 PYTHON: SUCCESSFUL GENERATION



VisCoder2-32B: Python

ID:1

TASK: Generate a scatter plot to visualize the relationship between sepal width and sepal length for different species of flowers. The data points are colored according to species, with the size of each point representing petal length. Additional information about petal width is available when hovering over the points.

STYLE: The plot features a grid background with evenly spaced horizontal and vertical lines. Data points are represented as circles of varying sizes, each filled with a distinct color corresponding to a species. A legend on the right side identifies the species by color. The axes are labeled with sepal width and sepal length, and the overall layout is clean and organized for easy interpretation.



Figure 5: Example of a successful generation in **Python** (ID: 1). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.2 PYTHON: SELF-DEBUG RECOVERY



VisCoder2-32B: Python

ID:69

TASK: Generate a scatter plot to visualize the relationship between two variables, labeled as 'x' and 'y', from a dataset. The plot is designed to display individual data points, allowing for the examination of potential correlations or patterns between the measurements. The y-axis values are formatted to represent millions, providing a clear understanding of the scale.

STYLE: The visual features a title at the top, with axes labeled horizontally and vertically. Data points are depicted as blue dots, evenly distributed across the plot area. A dashed grid is present, enhancing readability, and a hover tool is available for interactive data exploration. The overall layout is clean, with a focus on clarity and accessibility.

Scatter Plot of x vs y M0.0000008 No Image Rendered 4000000.0M \$4000000.0M Normal Rendered Debug Rendered Ground Truth Normal Generated Code Self-Debug Generated Code from holoviews import opts df = pd.read csv("... ...") scatter = hv.Scatter(df, 'x', 'y').opts(title="Scatter Plot of x vs y", show_grid=True, gridstyle={'line_dash': 'dashed', 'line_color': 'gray', 'dashed', 'line_co 'line_width': 0.5} grid_style={'line_dash': 'dashed', 'line_color': 'gray', hv.savescatter.opts(yformatter ='%.1fM') scatter.opts(yformatter='%.1fM') ValueError: Unexpected option 'grid_style' for Scatter type across all extensions.

Figure 6: Example of a failed generation in **Python** (ID: 69), where the initial code raises a **ValueError** and is resolved in the **first** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.3 PYTHON: SELF-DEBUG FAILED

X

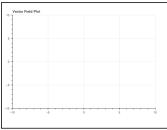
VisCoder2-32B: Python

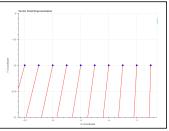
ID: 115

TASK: Generate a visual representation of a vector field using data from a CSV file. Display points on a coordinate system where each point has an associated vector. The vectors are represented as arrows originating from each point, indicating direction and magnitude.

STYLE: The plot features a rectangular layout with a title at the top. The background is white with grid lines. Points are marked with blue dots, and arrows are red with a filled triangular head. The axes are labeled with numeric scales and titles.

No Image Rendered





Normal Rendered

Debug Rendered

Ground Truth



AttributeError: unexpected attribute 'quiver' to figure.

Figure 7: Example of a failed generation in **Python** (ID: 115), where the initial code raises a **AttributeError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

K.4 VEGA-LITE: SUCCESSFUL GENERATION



VisCoder2-32B: Vega-Lite

ID:18

TASK: Create a population pyramid for the US in the year 2000, displaying the distribution of population by age and gender. The data is filtered to include only the year 2000, and gender is categorized into 'Male' and 'Female'. The population numbers are adjusted to be negative for females to allow for a mirrored bar chart effect.

STYLE: The visual consists of horizontal bars stacked in a mirrored fashion, with age groups sorted in descending order from top to bottom. The left side represents females with purple bars, while the right side represents males with orange bars. The x-axis shows population numbers in millions, formatted with an 'M' suffix. A legend at the top indicates the color coding for gender, and the overall layout is clean with no grid lines.



Figure 8: Example of a successful generation in **Vega-Lite** (ID: 18). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.5 VEGA-LITE: SELF-DEBUG RECOVERY

X

2322

23232324

232523262327

2328

2329

2330

2331

2332

2333

233423352336

2337

23382339

2340

234123422343

2344

23452346

2347

23482349

2350

2351

2352 2353

2354

23552356

2357

2358235923602361

236223632364

2365

2366

2367

VisCoder2-32B: Vega-Lite

ID:14

TASK: Create a bar chart that displays the production budget of the movie with the highest US Gross within each major genre. The x-axis represents the production budget in quantitative terms, while the y-axis lists the major genres in nominal terms.

STYLE: The chart is oriented horizontally with bars extending from the left to the right. Each bar corresponds to a different genre, labeled on the left side. The production budget is marked along the bottom axis, with gridlines aiding in visual comparison. The bars are uniformly colored in blue, providing a clear contrast against the white background.

Action Black Comedy No Image Rendered Horro Production Budget for max US Gross Normal Rendered Debug Rendered Ground Truth Normal Generated Code Self-Debug Generated Code "aggregate": [{ "op": "max", "field": "US Gross", "as": "max_US_Gross" }], "max_US_Gross" }], "groupby": ["Major Genre"], "sort": [{ "field": "US Gross", "order": groupby": ["Major Genre"] "lookup": "max US Gross", "data": { "url": "data.csv" }, "fields": ["US Gross"], "frame": [null, 0] "values": ["Production Budget"] "filter": "datum['US Gross'] datum.max_US_Gross" as": "Production Budget" TypeError: Cannot read properties of undefined (reading 'signal').

Figure 9: Example of a failed generation in **Vega-Lite** (ID: 14), where the initial code raises a **TypeError** and is resolved in the **second** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.6 VEGA-LITE: SELF-DEBUG FAILED



VisCoder2-32B: Vega-Lite

ID:50

TASK: Create a grid of point plots that represents different animals categorized by country. Each row corresponds to a country, and within each row, different animals are represented by distinct shapes. The animals are visually encoded using both shape and color to differentiate between categories such as "cattle," "pigs," and "sheep." The plot does not display axes, focusing solely on the visual arrangement of shapes to convey the data.

STYLE: The visual consists of a series of rows, each labeled with a country name on the left. Within each row, animal shapes are arranged horizontally, with each shape filled with a specific color corresponding to the animal type. The shapes are evenly spaced and sized uniformly, creating a clean and organized appearance. The absence of axes and legends emphasizes the visual comparison of animal types across different countries.

No Image Rendered No Image Rendered Normal Rendered Debug Rendered Self-Debug Generated Code { "\$schema": "https://vega.github.io/schema/vegalite/v6.json ", "data": { "url": "data.csv" " }, " " config": { "stroke": null }, "axis": { "grid": false }, "legend": null } TypeError: Cannot read properties of undefined (reading 'disable') at Lp.parseLegends

Figure 10: Example of a failed generation in **Vega-Lite** (ID: 50), where the initial code raises a **TypeError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

K.7 LILYPOND: SUCCESSFUL GENERATION



VisCoder2-32B: LilyPond

ID:15

Task: Generate a musical score with two sequences of notes. The first sequence is transposed from C major to D major and uses traditional noteheads. The second sequence uses a different set of notehead styles, including shapes like crosses, triangles, and diamonds. Both sequences consist of the notes C, D, E, F, G, A, B, and C, arranged in two measures each.

Style: Create a visual layout with two staves, each containing a series of musical notes. The first staff uses standard noteheads, while the second staff employs varied geometric shapes for noteheads. The staves are aligned horizontally, with a treble clef and key signature at the beginning of each. The measures are evenly spaced, and the overall appearance is clean and organized, with a small footer crediting the engraving software.

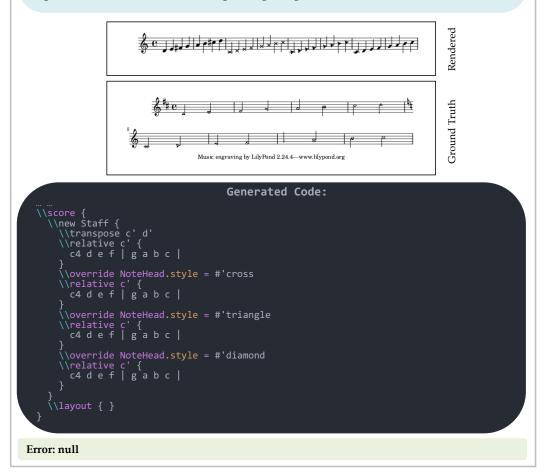


Figure 11: Example of a successful generation in **Lilypond** (ID: 15). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.8 LILYPOND: SELF-DEBUG RECOVERY



VisCoder2-32B: LilyPond

ID · 13

Task: Generate a musical score with three staves, each containing a sequence of notes. The notes are arranged in measures, with a time signature indicating a slow tempo. The score includes rehearsal marks and tempo indications at the beginning of each staff.

Style: Create a vertically aligned layout with three evenly spaced staves. Each staff begins with a treble clef and includes a tempo marking labeled "Largo" with a metronome setting. Rehearsal marks are placed prominently. The notes are evenly spaced, and the overall appearance is clean and organized, with a footer crediting the music engraving software.

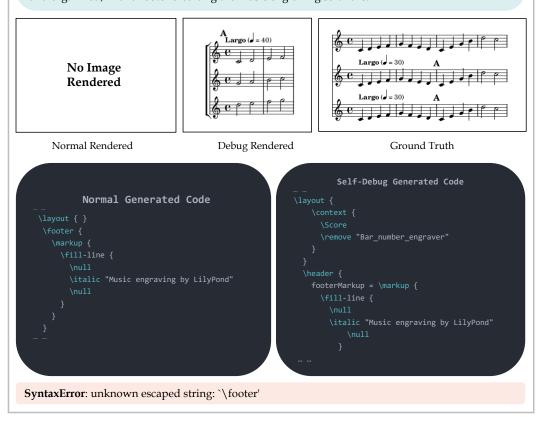


Figure 12: Example of a failed generation in **Lilypond** (ID: 13), where the initial code raises a **SyntaxError** and is resolved in the **first** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.9 LILYPOND: SELF-DEBUG FAILED

X

VisCoder2-32B: LilyPond

ID:48

Task: Generate a musical score with repeated sequences of notes in varying time signatures. The score is divided into two sections: the first section contains lines with four measures each, and the second section contains lines with six measures each. Each section is labeled accordingly to indicate the number of measures per line.

Style: Create a clean and organized layout with a treble clef at the beginning of each staff. The time signatures change within the repeated sequences, alternating between 5/4, 3/4, and 3/2. The labels "4 measures per line:" and "6 measures per line:" are italicized and placed at the beginning of each respective section. The notes are evenly spaced, and the overall appearance is neat, with clear separation between the sections.

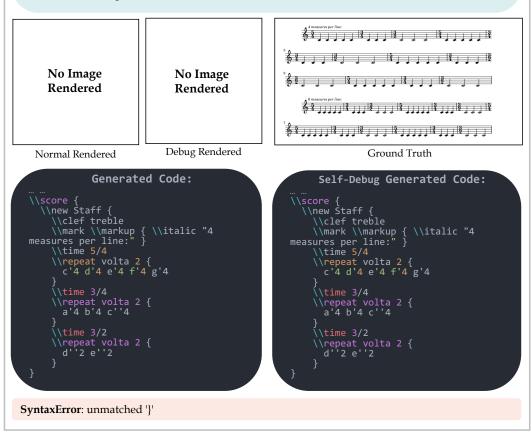


Figure 13: Example of a failed generation in **Lilypond** (ID: 48), where the initial code raises a **TypeError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

K.10 MERMAID: SUCCESSFUL GENERATION

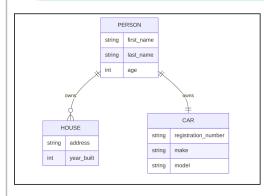


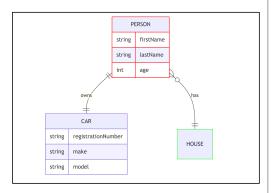
VisCoder2-32B: Mermaid

ID:19

Task: Generate an entity-relationship diagram illustrating the relationships between three entities: PERSON, CAR, and HOUSE. The PERSON entity includes attributes for first name, last name, and age. The CAR entity includes attributes for registration number, make, and model. The diagram shows a one-to-one relationship between PERSON and CAR, indicating that a person owns a car. Additionally, there is a one-to-many relationship between PERSON and HOUSE, indicating that a person can have multiple houses.

Style: The diagram features rectangular boxes for each entity, with the PERSON entity outlined in red, the CAR entity in purple, and the HOUSE entity in green. Each box contains rows listing the attributes, with data types specified. Lines connect the entities, with labels indicating the nature of the relationships. The lines have different end markers to represent the type of relationship, and the overall layout is clean and organized with a clear visual hierarchy.





Generated Code:

erDiagram
PERSON|--o{ HOUSE : owns
PERSON|--|CAR : owns
PERSON {
 string first_name
 string last_name
 int age
}
CAR {
 string registration_number
 string make
 string model
}
HOUSE {
 string address
 int year_built
}
Error: null

Figure 14: Example of a successful generation in **Mermaid** (ID: 19). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.11 Mermaid: Self-Debug Recovery



VisCoder2-32B: Mermaid

ID:88

Task: Generate a state diagram illustrating the transitions between three states: "SomeOtherState," "Your state with spaces in it," and "YetAnotherState." The diagram begins with an initial state that transitions to both "SomeOtherState" and "Your state with spaces in it." From "SomeOtherState," the flow continues to "YetAnotherState." Similarly, "Your state with spaces in it" also transitions to "YetAnotherState." Finally, "YetAnotherState" leads to a final state.

Style: Create a visual with rounded rectangles representing states, connected by arrows indicating transitions. The initial and final states are depicted as filled circles. "Your state with spaces in it" is styled with bold and italic text, while other states have regular text. The layout is organized with a top-down flow, and the states are outlined with a subtle color.

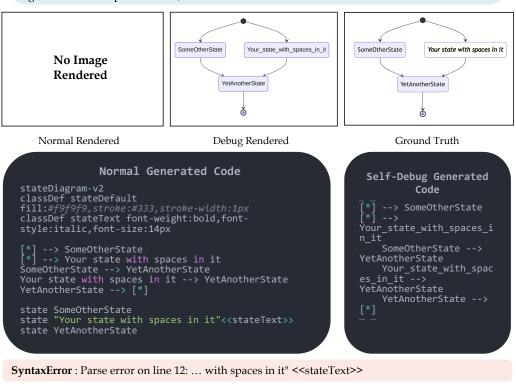


Figure 15: Example of a failed generation in **Mermaid** (ID: 88), where the initial code raises a **SyntaxError** and is resolved in the **second** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.12 MERMAID: SELF-DEBUG FAILED



VisCoder2-32B: Mermaid

ID:80

Task: Generate a sequence diagram illustrating interactions between participants named Alice, Bob, Carl, and an actor named Donald. The diagram depicts a conversation where Alice greets Bob, and they exchange pleasantries. Alice then introduces Carl, who subsequently greets Donald. The diagram also shows the destruction of Carl and Bob, with Alice commenting on the number of participants and Bob agreeing.

Style: The visual consists of horizontal arrows connecting labeled rectangular boxes representing participants, with a stick figure symbolizing the actor. The boxes and arrows are aligned vertically, showing the flow of communication. Messages are displayed as text along the arrows, and the destruction of participants is marked by an 'x' on the arrows. The layout is clean, with a light color scheme and clear separation between interactions.

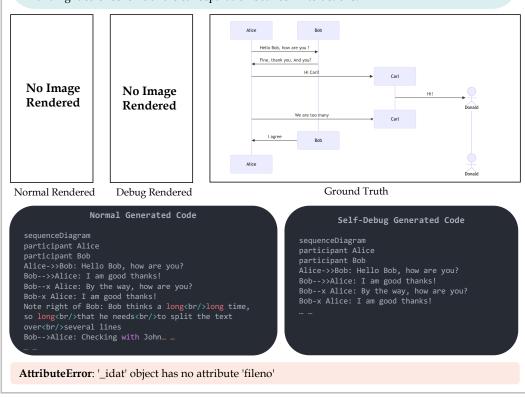


Figure 16: Example of a failed generation in **Mermaid** (ID: 80), where the initial code raises a **AttributeError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

K.13 SVG: SUCCESSFUL GENERATION

$\langle \! \rangle$

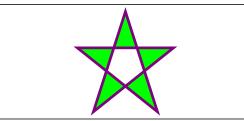
VisCoder2-32B: SVG

ID:52

TASK: Generate a five-pointed star shape using a polygon. The star is defined by a series of connected points that form its structure.

STYLE: The star is filled with a bright green color and outlined with a thick purple border. The background is black, making the star stand out prominently. The overall appearance is bold and vibrant, with a strong contrast between the star and its outline.





Rendered

Ground Truth

Generated Code: <svg width="200" height="200" xmlns="http://www.w3.org/2000/svg"> <polygon points="100,10 40,198 190,78 10,78 160,198" fill="lime" stroke="purple" stroke-width="10" /> </svg>

Error: null

Figure 17: Example of a successful generation in **SVG** (ID: 52). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.14 SVG: SELF-DEBUG RECOVERY



VisCoder2-32B: SVG

ID: 42

TASK: Generate a flowchart illustrating the process for producing a W3C Recommendation. The flowchart includes stages such as Working Draft (WD), Candidate Recommendation (CR), Proposed Recommendation (PR), and W3C Recommendation (REC). It also includes decision points and possible transitions between these stages, such as advancing to the next stage, returning to a previous stage, or publishing as a note. Each stage and decision point is labeled with descriptive text.

STYLE: Create a visual with elliptical nodes representing different stages, connected by arrows indicating the flow of the process. The nodes are labeled with abbreviations like WD, CR, PR, and REC, and are filled with white. Text annotations provide additional context for each transition, using a small font size. Arrows are solid or dashed, with varying colors to indicate different types of transitions. Some elements are partially transparent to indicate optional paths. The layout is horizontal, with nodes aligned in a sequence from left to right.

No Image Rendered

Normal Rendered | Wid Decision to Ballow Information | Wide Decision to Ballow Information | Wi

Debug Rendered Ground Truth



ExpatError: no element found: line 362, column 18.

Figure 18: Example of a failed generation in **SVG** (ID: 42), where the initial code raises a **ExPatError** and is resolved in the **first** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.15 SVG: SELF-DEBUG FAILED



VisCoder2-32B: SVG

ID:12

TASK: Create a horizontal bar chart to compare the usage of different media types by gender. The chart displays five categories: Internet, TV, Newspaper, Magazine, and Radio. Each category has two bars representing data for females and males. The x-axis shows numerical values from 0 to 3, indicating the level of usage or preference, while the y-axis lists the media types.

STYLE: The chart has a white background with a light gray area for the graph. The bars are colored with semi-transparent red for females and blue for males. Dashed vertical lines serve as grid lines for reference. Labels for the x-axis are centered below the axis, and y-axis labels are rotated vertically. Data values are displayed on the bars. A legend on the right identifies the colors for female and male data. The title "Question 7" is centered at the top.

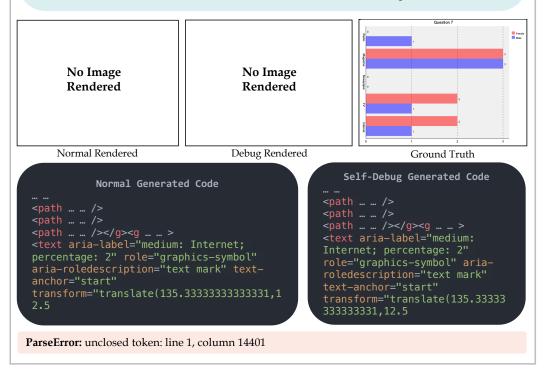


Figure 19: Example of a failed generation in **SVG** (ID: 12), where the initial code raises a **ParseError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

K.16 LATEX: SUCCESSFUL GENERATION



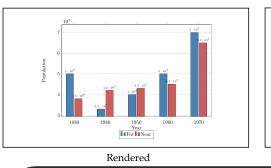
Error: null

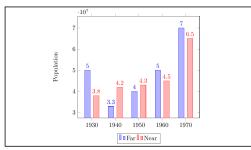
VisCoder2-32B: LaTeX

ID:33

Task: Generate a bar chart comparing the population of two groups labeled 'Far' and 'Near' over a series of years. The x-axis represents the years, and the y-axis represents the population size. Each year has two bars, one for each group, showing their respective population values.

Style: Create a vertical bar chart with distinct colors for each group: blue for 'Far' and red for 'Near'. The bars are slightly separated to enhance clarity. The y-axis label is rotated vertically on the left side. Data values are displayed above each bar in corresponding colors. A legend is positioned below the chart, centered, indicating which color corresponds to each group. The chart has a clean, simple layout with no gridlines.





Ground Truth

Generated Code:

"""

\begin\{\textit{tikzpicture}\}
\begin\{\textit{axis}\}[
\textit{width=12cm,}
\textit{height=8cm,}
\textit{xlabel=Year,}
\textit{ylabel=Population,}
\textit{ybar,}
\textit{bar width=15pt,}
\textit{ymajorgrids=true,}
\textit{xtick=data,}
\textit{xtick=data,}
\textit{xticklabels from table={\datatable}{Year},}
\textit{legend style={at={(0.5,-0.15)}, anchor=north, legend columns=-1},}
\textit{nodes near coords,}
\textit{nodes near coords align={vertical},}
\text{every node near coord/.append style={font=\textit{tiny}},}

\textit{]}

"" ""

Figure 20: Example of a successful generation in **LaTeX** (ID: 33). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.17 LATEX: SELF-DEBUG RECOVERY



VisCoder2-32B: LaTeX

ID:37

Task: Generate a scatter plot using data from a CSV file, where the x-axis represents sample indices and the y-axis represents random values. The plot displays individual data points as marks without connecting lines.

Style: The plot features blue circular markers distributed across the graph. The axes are labeled with numerical values, and the grid is not visible. The layout is compact, with data points densely packed, especially along several horizontal bands.

0.6 No Image 0.4 Rendered 0.9 Sample Index Normal Rendered Debug Rendered Ground Truth Self-Debug Generated Code Normal Generated Code \usepackage{pgfplots}\pgfplotstableread[col sample_index,random_value ylabel=Random Value, only marks, mark=*, mark size=1pt, mark options={blue}, grid=major, \documentclass[*border=10pt*]{standalone} \usepackage{pgfplots} style={dashed,gray!30} \usepackage{pgfplotstable}

Figure 21: Example of a failed generation in **LaTeX** (ID: 37), where the initial code raises a **NameError** and is resolved in the **second** round of self-debug, resulting in a corrected plot that matches the intended semantics.

NameError: LaTeX rendering failed: ! LaTeX Error: \usepackage before \documentclass.

K.18 LATEX: SELF-DEBUG FAILED



VisCoder2-32B: LaTeX

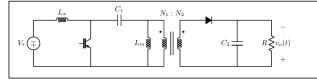
ID:97

Task: Generate a schematic diagram of an electrical circuit featuring a voltage source labeled V_s, an inductor labeled L_x, a capacitor labeled C_1, and another inductor labeled L_m on the primary side. The circuit includes a transformer with a turns ratio $N_1:N_2$, a diode, a capacitor labeled C_2, and a resistor labeled R on the secondary side. The output voltage is denoted as $v_0(t)$.

Style: Create a clean and organized layout with components aligned horizontally. The circuit elements are connected with straight lines, and labels are placed near each component for clarity. The transformer is depicted with coils and a turns ratio label. The diagram uses a simple, monochrome style with a consistent line thickness and small font size for labels.

No Image Rendered

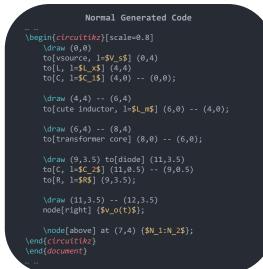
No Image Rendered



Normal Rendered D

Debug Rendered

Ground Truth



NameError: ! Package circuitikz Error: Unknown circuitikz element `vsource'.

Figure 22: Example of a failed generation in **LaTeX** (ID: 97), where the initial code raises a **NameError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

ASYMPTOTE: SUCCESSFUL GENERATION

VisCoder2-32B: Asymptote

ID: 31

Task: Generate a line graph with data points marked along the line. The graph plots a series of points connected by straight lines, with markers at each data point. The x-axis is labeled with "x" and the y-axis with "y", both featuring tick marks for reference.

Style: Create a simple and clean visual with a rectangular frame. The data points are marked with circles, some filled and some outlined, depending on their position. The axes have evenly spaced tick marks, and the labels are positioned outside the frame. The overall layout is balanced and minimalistic, emphasizing clarity and readability.

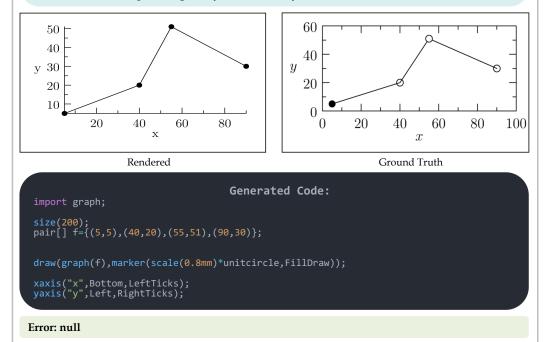


Figure 23: Example of a successful generation in **Asymptote** (ID: 31). The model generates code that executes successfully and produces a plot consistent with the ground truth. [Back to Appendix Contents]

K.20 ASYMPTOTE: SELF-DEBUG RECOVERY

X

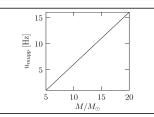
VisCoder2-32B: Asymptote

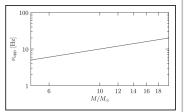
ID:53

Task: Generate a logarithmic plot of the identity function over the range of x-values from 5 to 20. The x-axis represents mass in solar units, labeled as "\$M\/M_\odot\$", and the y-axis represents frequency in Hertz, labeled as "\$\nu_{\rm upp}\$ [Hz]". The plot displays a straight line indicating a direct proportionality between the two variables on a logarithmic scale.

Style: Create a rectangular plot with a horizontal orientation. The x-axis is marked with ticks at intervals of 6, 10, 12, 14, 16, and 18, while the y-axis is marked with ticks ranging from 1 to 100. The plot uses a clean, minimalist style with a single line and clear labels for both axes. The layout is simple, focusing on the relationship between the two variables without additional embellishments.

No Image Rendered





Normal Rendered

Debug Rendered

Ground Truth

Normal Generated Code

```
import graph;
size(200,150);
real[]
x={5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
real[]
y={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
draw(graph(x,y),LogScale,LogScale);
xaxis("$M/M_\odot$",BottomTop,LeftTicks);
yaxis("$\nu_{\rmupp}$[Hz]",LeftRight,RightTicks);
```



NameError: draw(graph(x,y),LogScale,LogScale): no matching variable 'LogScale'

Figure 24: Example of a failed generation in **Asymptote** (ID: 53), where the initial code raises a **NameError** and is resolved in the **third** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.21 ASYMPTOTE: SELF-DEBUG FAILED



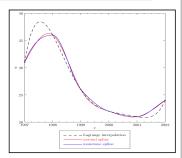
VisCoder2-32B: Asymptote

ID:79

Task: Generate a visual comparison of interpolation methods over a specified range. The plot includes three curves: a Lagrange interpolation represented by a dashed line, a natural spline shown in red, and a monotone spline depicted in blue. The x-axis is labeled with values from 1997 to 2002, while the y-axis ranges from 20 to 40. Data points are marked along the curves. **Style:** Create a clean and organized layout featuring a large square plot area. The x-axis and y-axis are clearly labeled with regular tick marks. The legend is positioned below the plot, providing clear differentiation between the interpolation methods using distinct colors and line styles. Data points are indicated with small gray dots, enhancing the visual clarity of the curves.

No Image Rendered

No Image Rendered



Normal Rendered

Debug Rendered

Ground Truth

```
Normal Generated Code

real a=1997, b=2002;
int n=5;
real[] xpt=a+sequence(n+1)*(b-a)/n;
real[] ypt=(31,36,26,22,21,24);
horner h=diffdiv(xpt,ypt);
fhorner L=fhorner(h);

guide gL=graph(L,a,b,500);
draw(gL,dashed);
dot(gL,red);

real[] ypt2={31,36,26,22,21,24};
spline s2=spline(xpt,ypt2,Spline(natural));
guide gs2=graph(s2,a,b,500);
draw(gs2,red);

real[] ypt3={31,36,26,22,21,24};
spline s3=spline(xpt,ypt3,Spline(monotone));
guide gs3=graph(s3,a,b,500);
draw(gs3,blue);
```

```
Self-Debug Generated Code
...
real a=1997, b=2002;
int n=5;
real[] xpt=a+sequence(n+1)*(b-a)/n;
real[]
ypt={31,36,26,22,21,24};
guide gL=graph(new real(real x) {
    return ypt[0] + (x-xpt[0])*(ypt[1]-ypt[0])/(xpt[1]-xpt[0]);
}, a, b, 500);
draw(gL,dashed);
dot(gL,red);
....
```

TypeError: horner h=diffdiv(xpt,ypt): no type of name 'horner'

Figure 25: Example of a failed generation in **Asymptote** (ID: 79), where the initial code raises a **TypeError** and is still failed after three rounds self-debug.

K.22 HTML: SUCCESSFUL GENERATION

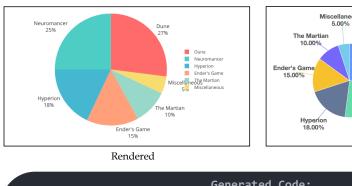
\otimes

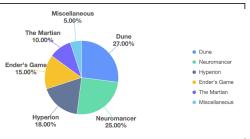
VisCoder2-32B: HTML

ID:6

Task: Generate a pie chart to display the distribution of various science fiction categories. The chart segments represent different categories, such as "Dune," "Neuromancer," "Hyperion," "Ender's Game," "The Martian," and "Miscellaneous," with each segment sized according to its value. The chart includes labels showing the category name and its percentage of the total.

STYLE: Create a centered layout with a light gray background. The pie chart is placed within a white, rounded rectangle with a subtle shadow, providing a clean and modern look. The title, "Science Fiction Categories Distribution," is centered above the chart in a bold, dark gray font. The legend is positioned to the right of the pie chart, with category names in a smaller font size, each colored to match its corresponding pie segment. Labels on the pie chart are bold and positioned outside the segments for clarity.





Ground Truth

Generated Code:

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<tittle>Plotly Pie Chart</title>
<script src=""></script>
<style>... ...</style>
</head>
<body>
<div id="chart">
</div>
<script>... ...</script>
</body>
</html>

Error: null

Figure 26: Example of a successful generation in **HTML** (ID: 6). The model generates code that executes successfully and produces a plot consistent with the ground truth.

[Back to Appendix Contents]

K.23 HTML: SELF-DEBUG RECOVERY



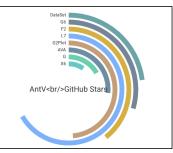
VisCoder2-32B: HTML

ID:9

Task: Generate a radial bar chart to visualize the number of GitHub stars for various AntV projects. Each bar represents a project, with the length corresponding to the star count. The bars are color-coded based on the number of stars, with different colors indicating different ranges of star counts. The chart includes a central annotation displaying the total number of stars across all projects.

Style: Create a circular chart with a white background and a subtle shadow effect. The bars are arranged in a radial pattern, with rounded ends and varying colors. The chart is centered on the page, with a title above it. Inside the chart, there is a central text area with a GitHub icon, the word "AntV," and the total star count in blue. The overall layout is clean and balanced, with a modern aesthetic.

No Image Rendered





Normal Rendered

Debug Rendered

Ground Truth



```
Self-Debug Generated Code
......
< annotations: [
{
    type: 'text',
    position: ['50%', '50%'],
    content: 'AntV<br/>GitHub
Stars',
    style: {
    fontSize: 24,
    fill: '#333',
    textAlign: 'center',
    textBaseline: 'middle'
    }
}
.......
</script>
</body>
</html>
```

ImportError: HTML Request failed: https://.....7a.svg (404 Not Found).

Figure 27: Example of a failed generation in **HTML** (ID: 9), where the initial code raises a **ImportError** and is resolved in the **first** round of self-debug, resulting in a corrected plot that matches the intended semantics.

K.24 HTML: Self-Debug Failed



VisCoder2-32B: HTML

ID:85

Task: Generate a calendar heatmap to visualize daily website traffic analytics for the year 2023. Each day is represented by a square, with the color intensity indicating the volume of unique visitors. The data includes variations based on weekly patterns, monthly multipliers, and special events or holidays that affect traffic levels.

Style: Create a centered layout with a light background and a white container featuring rounded corners and a shadow effect. The title and subtitle are displayed at the top in bold and smaller text, respectively. The calendar heatmap is positioned below, with months labeled at the top and days of the week indicated. A color gradient bar at the bottom illustrates the traffic intensity, with darker colors representing higher traffic. Additional text explains the color coding.

Daily Website Traffic Analytics No Image No Image Rendered Rendered Debug Rendered Ground Truth Normal Rendered Normal Generated Code Self-Debug Generated Code constctx=document.getElementById('calendarHeatmap').getContext('2d data: calendarData, backgroundColor: 'rgba(54, 162, 235, 0.7)', elivr.net/npm/chartjselivr.net/npm/chartjsmax: 1000, calendar@1.0.0/dist/chartjs **TypeError**: HTML rendering error: "calendar" is not a registered controller.

Figure 28: Example of a failed generation in **HTML** (ID: 85), where the initial code raises a **TypeError** and is still failed after three rounds self-debug.

[Back to Appendix Contents]

L LLM USAGE

Large Language Models (LLMs) were used to assist with language editing and polishing of the manuscript. Specifically, we used an LLM to refine wording, improve readability, and enhance clarity in several sections.

The LLM was not involved in research ideation, methodology, data analysis, or experimental design. All technical ideas, analyses, and conclusions were developed and validated by the authors; the tool was limited to editorial support.

The authors take full responsibility for the content of the manuscript, including any text edited with LLM assistance. We verified suggested edits for accuracy and originality, ensured compliance with ethical guidelines, and noted that LLMs are not listed as authors. We disclose this usage in accordance with ICLR's policies on LLM assistance and attribution.