CHARTREVISE: A DATASET AND EVALUATION PROTOCOL FOR EXACT CHART EDITING VIA CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

Chart editing requires cross-modal edit grounding, realizing a requested visual change in the code that draws it, with necessary related updates and without altering unrelated content. Existing benchmarks emphasize either code executability or chart quality, but their metrics do not clearly distinguish request completion from missed coupled updates and gratuitous changes. We introduce ChartRevise, a structured dataset and evaluation protocol for exact program-grounded chart editing. For dataset construction, we build on the grammar of graphics to systematically cover chart-editing operations, using source-program checks to verify their applicability across chart types and libraries. To improve edit exactness, our pipeline checks individual requirements and guides repair or exclusion when they are unmet. The resulting dataset contains 92,438 records covering 344 edit types across 20 chart types and three plotting libraries. For evaluation, our reference-free protocol separately measures atomic requirement completion, identifies gratuitous changes, and detects missed coupled updates. These checks are combined with successful execution and rendering to determine exact-edit success. Across five models and four external benchmarks, fine-tuning yields relative gains of 16% in mean requirement recall and 22% in mean exact-edit rate.

1 INTRODUCTION

Charts are often created from plotting programs, so editing a chart usually requires editing its code. The user, however, asks for a change to the rendered chart rather than to the code, so the requested change is defined on the chart that the program draws. Chart editing therefore requires cross-modal edit grounding, connecting a text instruction to the visual components it refers to and to the code regions that draw them. We study this program-grounded setting, in which a model receives a rendered chart, its source program, and an instruction, and must modify the right code regions so that the rendered chart reflects the requested change while unrelated chart properties remain unchanged. Errors occur in two directions. Under-editing leaves an *explicit requirement* unmet or misses a *coupled update* that the requested change implies, and over-editing introduces *gratuitous changes* to unrelated properties. Coupled updates are never stated in the instruction and gratuitous changes lie outside it, so a check that asks only whether the instruction was followed catches neither.

Figure 1 illustrates both directions on a Chart2Code-L2 task (Tang et al., 2026) requesting stacked bars with gray/green gradients and an inset boxplot in the upper right. The under-edited panel stacks and recolors the bars but omits the boxplot, leaving an explicit requirement unmet in the code (*partial completion*), and keeps the old legend colors although the legend depends on the recolored bars (*missed coupled update*). The over-edited panel completes the request, but also rewrites the title, a component the instruction never refers to (*gratuitous change*). Because references serve as both training targets and evaluation standards, such a reference can teach and reward over-editing, so exactness must be enforced per requirement in both training data and evaluation.

Prior work builds instruction-driven editing systems (Yan et al., 2024; Goswami et al., 2025) and iterative code generation for scientific plots (Yang et al., 2024), and benchmarks cover validated edits (Zhao et al., 2025a), multimodal instructions (Yang et al., 2025b), and complex modifications (Kapadnis et al., 2026; Tang et al., 2026). However, two gaps remain. First, existing resources provide limited coverage of editing operations across chart types and plotting libraries. FigEdit(Li et al., 2026) fixes a set of operations over chart specifications and ChartM^3 builds its pool around

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

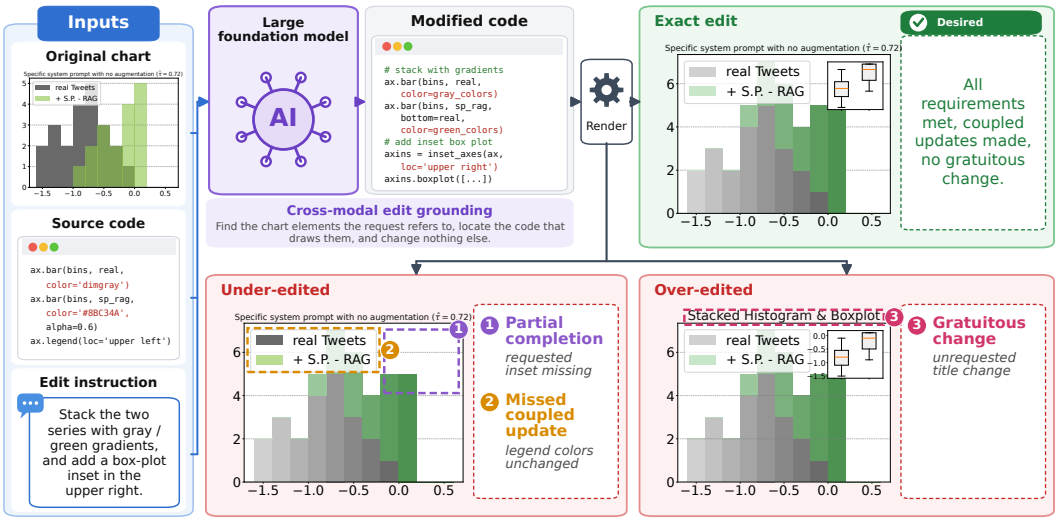


Figure 1: Failure modes in Chart2Code-L2 task 0565. Under-edited and exact panels are constructed; the over-edited panel is the released reference.

Matplotlib APIs (Yang et al., 2025b), so coverage reflects curation and does not extend systematically to new chart objects or libraries. Second, evaluation does not distinguish errors. Aggregate quality scores hide which requirements were met and what else changed, and reference-based scores can inherit flaws in the reference. Benchmarks check instruction following (Kapadnis et al., 2026), preservation(Yang et al., 2025b; Li et al., 2026), or geometric consistency (Yu et al., 2026) cover parts of the task, but none jointly scores completion, coupled updates, and gratuitous changes.

We introduce ChartRevise, a dataset and evaluation protocol for exact program-grounded chart editing, built on the idea that each edit is a set of atomic requirements that can be independently checked. For dataset construction, we derive object–action templates from the grammar of graphics and instantiate them only when their source conditions hold, yielding single and compound requests across three plotting libraries. Execution, completion, and preservation checks with atomic audits filter or repair each record. For evaluation, a reference-free protocol decomposes each edit into requirements and verifies them directly in the chart-generating program, without a ground-truth program. It separately scores *completion*, *consistency* of coupled updates, and *preservation* of unrelated content; an edit is exact only if it executes and passes all three checks. Fine-tuning on ChartRevise consistently improves completion and coupled updates on external benchmarks, but less often reduces gratuitous changes, suggesting that supervised edits teach models what to edit more reliably than what to preserve. Our contributions are summarized as follows:

- **Grammar-derived editing dataset.** Edit types are derived from the grammar of graphics, which makes coverage systematic and extensible to new objects, chart types, and libraries. The release contains 92,438 records spanning 344 edit types in twelve categories, 20 chart types, three plotting libraries, and compound requests of up to twelve actions (Sections 3.1, 3.2 and 3.5).
- **Fine-grained construction audits.** After initial validation for completion and preservation, a judge from a different model family audits coupled updates in every record, and compound records are further audited requirement by requirement rather than by one overall judge. Failed checks guide targeted repair or exclusion, and a human audit finds 84% accepted-set precision (Sections 3.4 and 3.5).
- **Reference-free exact-edit evaluation.** To our knowledge, this is the first protocol that jointly scores explicit requirements, coupled updates against a fixed per-task inventory, and gratuitous changes, without a reference edit. A human audit finds 93% agreement with the judge (Sections 4.1 and 4.4).
- **Transfer and editing-behavior analysis.** Across five models and four external benchmarks, fine-tuning on ChartRevise improves requirement recall and coupled-update recall in 19 of 20 comparisons and exact-edit rate in 17, but reduces gratuitous changes in only 10 (Section 4.2).

Table 1: Task scope, construction checks, and explicit evaluation checks of chart-editing resources.

Resource	Dataset scope			Construction checks	Evaluation checks		
	Tasks	Chart types	Instruction types		Atomic requirements	Preservation	Coupled updates
ChartEdit	1,405	19	6	Human (request, code, chart)	—	—	—
ChartM^3	1K eval. 24K train	10	—	Model (charts)	—	✓ (code-space)	—
ChartEditVista	7,964	31	6	Model (charts) Human (code, charts)	—	—	—
ChartEditBench	4,142	37	35	Model (code, charts) Program (assertions)	✓ (assertions)	—	—
Chart2Code L2	1,010	19	—	—	—	—	—
FigEdit	30,836	10	9	Program (specs)	—	✓ (image-space)	—
ChartSync	870	9	—	Model (code, charts) Human (code, charts)	—	✓ (image-space)	✓ (geometry)
ChartRevise	**92,438**	**20**	**344**	**Model(code,charts) Atomic audit+repair Human audit**	✓	✓ (code audit)	✓

2 Related Works

We compare ChartRevise with existing works, and highlight ChartRevise's combination of a large-scale editing dataset, atomic auditing and repair, and joint checks of requirement completion, preservation, and coupled updates in Table 1.

Chart-editing resources. Existing datasets study program revision and multi-turn editing (Zhao et al., 2025a; Tang et al., 2026; Kapadnis et al., 2026), multimodal instructions (Yang et al., 2025b; Chen et al., 2026), and direct image editing (Li et al., 2026; Yu et al., 2026). ChartReformer (Yan et al., 2024) edits charts from images and natural-language requests, while PlotEdit (Goswami et al., 2025) applies multimodal agents to chart editing in PDFs. These resources cover diverse interaction formats, but their instructions typically draw on predefined edits to chart objects or library-specific APIs. Such inventories offer limited support for systematic extension across plotting libraries, whereas our grammar-derived taxonomy provides shared object-action templates that can be extended with new objects and actions, and adapted across charts and libraries.

Evaluation. Current benchmarks report code or chart quality scores(Zhao et al., 2025a) and, in some cases, separate instruction-following and preservation scores (Tang et al., 2026; Yang et al., 2025b; Li et al., 2026; Yu et al., 2026; Ku et al., 2024a;b). These aggregate measures can hide individual omissions and unrequested changes. ChartEditBench (Kapadnis et al., 2026) checks individual conditions through assertions for programmatic instructions, but passing these assertions does not establish that all necessary related updates are complete. ChartSync explicitly checks text-to-geometry dependencies, but these checks cover a specific class of coupled updates rather than their broader range (Yu et al., 2026). Thus, satisfying those reported criteria does not necessarily establish that every explicit requirement and necessary related update is complete without gratuitous changes. An execution failure also leaves unresolved how much of the request was completed. Following work on evaluating LLM judges, we separately assess agreement with human judgments (Zheng et al., 2023; Liu et al., 2023b). These limitations leave cross-modal edit grounding only partially assessed: whether the requested visual changes are implemented on the intended chart components, with necessary dependent updates and no unrelated changes. Our protocol checks these outcomes in the source and edited programs at the requirement level separately, including in programs that fail to execute. Section A provides extended discussions on related works.

3 Systematic Construction and Validation of ChartRevise

We construct ChartRevise to support cross-modal edit grounding through systematic operation coverage and verified editing examples. The construction combines a grammar-derived taxonomy, source-program applicability checks, and multi-stage validation.

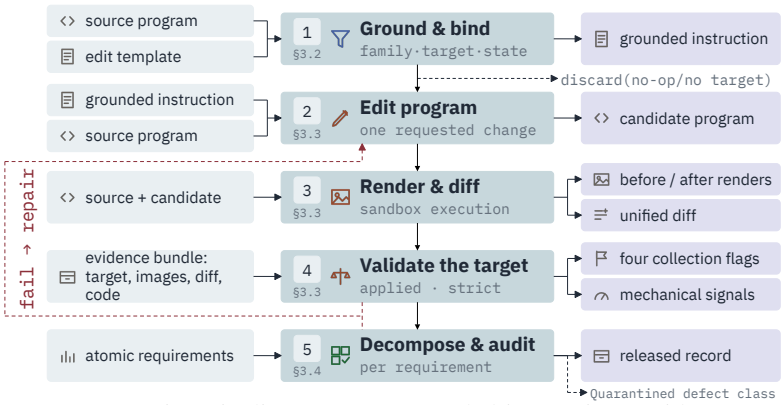


Figure 2: Dataset construction pipeline. Source-grounded instructions guide program generation. Execution and target validation produce repair feedback, and post-processing audits assess cross-modal edit grounding by checking completion, consistency, and preservation before release.

Cross-modal edit grounding. For each editing example, the model must connect the requested visual change to the relevant chart component and its implementation in code, then realize the requested state and necessary dependent updates while preserving unrelated properties. An *explicit requirement* specifies a visual property requested by the instruction; a *coupled update* is an unstated change needed to keep dependent chart elements consistent. A *gratuitous change* modifies a chart property outside the scope of the explicit requirements and necessary coupled updates. An edit is *exact* when the program executes and renders, satisfies all explicit requirements and coupled updates, and introduces no gratuitous change. Section 4.1 defines how these criteria are measured.

Our pipeline turns edit types from the taxonomy into verified training examples using existing chart programs. Figure 2 summarizes its five steps. Step 1 selects an applicable template from the taxonomy and binds its targets and arguments to a source program, producing a source-grounded instruction. Step 2 rewrites the program under this instruction. Step 3 executes and renders the source and candidate programs. Step 4 validates the target edit using code and visual evidence; execution and validation failures provide feedback for repair. Step 5 decomposes the instruction into atomic requirements and audits completion and coupled updates before release.

3.1 Grammar-derived edit taxonomy

To define the dataset's edit scope, we select twelve operation categories based on the grammar of graphics, its layered formulation, and view composition (Wilkinson, 2011; Wickham, 2010; Satyanarayan et al., 2016; 2015). These categories cover data and transforms, encodings, scales and guides, marks and layers, position, coordinates, facets, annotations, and style.

Existing resources construct edit inventories from selected object edits or library-specific operation pools (Zhao et al., 2025a; Yang et al., 2025b; Li et al., 2026). To support *systematic coverage*, we pair editable objects with applicable actions within each grammar-derived category, forming templates with placeholders. For example, legend–rename yields "Change the legend title to {`title`}." These semantic templates can be reused across compatible chart types and plotting libraries. Sections B.1 and B.3 provide the object inventory and instantiated examples.

3.2 Applicability, source grounding, and composition

An edit template is useful only when the source program supports the requested operation. We therefore check compatibility, bind valid targets, and combine instantiated actions into requests.

We first construct a compatibility table whose rows are edit templates and whose columns are chart types. A cell marks a template as applicable to a chart type when charts of that type can contain the template's target object, giving 4,316 compatible combinations (Section C.1). For each source program, we first use this table to filter templates by chart type, then inspect the code to verify that the required target object or plotting context is present. For instance, a legend-move template requires identifying an existing legend, whereas adding a reference line requires an axes object to host it. The final collection covers 344 applicable edit types.

Next, we instantiate each applicable template against the source program. Existing field names, series labels, and target objects are bound to the template, while new values such as a position or font size are selected under its constraints. We reject no-op requests and inconsistent bindings. We call the resulting instruction *source-grounded*, as its target or plotting context and arguments are resolved for that source program. Section B.3 provides more examples.

A single-action request instantiates one template. To construct compound requests, we combine instantiated actions for the same source chart, using 4–6 actions in tier 2 and 8–12 in tier 3, and check for direct conflicts among them. This produces requests that require several changes to be completed together. Section B.7 provides additional details on compound request construction.

3.3 Edited program generation, validation, and repair

Given a source-grounded instruction, the successful edit must realize the requested chart states without disturbing unrelated properties. Therefore, after generating candidate programs, we validate these outcomes and repair failures using diagnostic feedback.

Generation. We draw source programs for different chart types from ChartNet (Kondic et al., 2026). We use `gemma-4-31B-it`(Team, 2026) to obtain a complete edited program based on the source chart, source program and the source-grounded instruction. Its prompt specifies the requested change and requires unrelated chart content to be preserved.

Validation. We execute and render the source and edited programs in the same sandbox. A judge model, `gemma-4-31B-it` , then checks each candidate against the before/after renders, the unified code diff, and the complete edited program to verify that code changes affect the intended visual component and preserve unrelated chart properties. A candidate is marked applied when the requested edit reaches the named target, and strict when it also executes and makes no unrequested change, movement, or removal.

Repair. Validation failures provide the feedback for repair. Execution or rendering failures return a traceback to the editor, while an unapplied edit returns the judge's rationale. Repaired candidates are executed, rendered, and validated again before acceptance. Candidates that apply the requested edit but fail preservation are retained as diagnostic records for further processing. Detailed rules and implementation for generation and validation are described in Section B.

3.4 Post-processing audits

An overall judgment can overlook missing operations and misclassify necessary coupled updates. A second judge from a different model family, `Qwen3.8-27B` (Qwen Team, 2026b), therefore re-audits the records before release and decides the final acceptance.

Coupled updates. A requested change often forces dependent elements to change, such as the legend entry of a removed series. The audit checks these dependent elements in every record, single-action or compound. A record that leaves a dependent element in its old state is repaired with this feedback and excluded if the repair fails. Conversely, a necessary dependent change is not counted as a gratuitous change, so records rejected for such changes in first pass are restored.

Compound requirements. Among compound records with an initial judgment, 34.6% had been marked applied although at least one requested operation was not carried out. The audit decomposes each compound instruction into atomic requirements and checks each using the same validation criteria as single-action edits. Consequently, consistency and preservation are assessed across the union of atomic requirements. Unmet requirements guide targeted repair, and records that still fail are excluded. Details of post-processing audits are given in Section B.7.

3.5 Released dataset and quality assessment

We characterize its collection outcomes, then use human audit to estimate the correctness of accepted records.

ChartRevise contains 90,589 training and 1,849 test records. Figure 3 shows that editing examples span diverse chart types and all three plotting libraries, rather than concentrating on a few chart settings. Under the category mapping in Figure 4, ChartRevise contains more instructions than

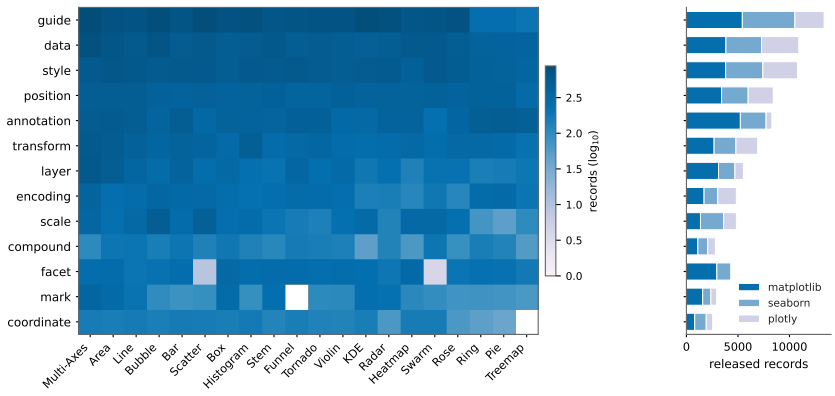


Figure 3: Record counts aggregated by category and chart type (left) and plotting library (right).

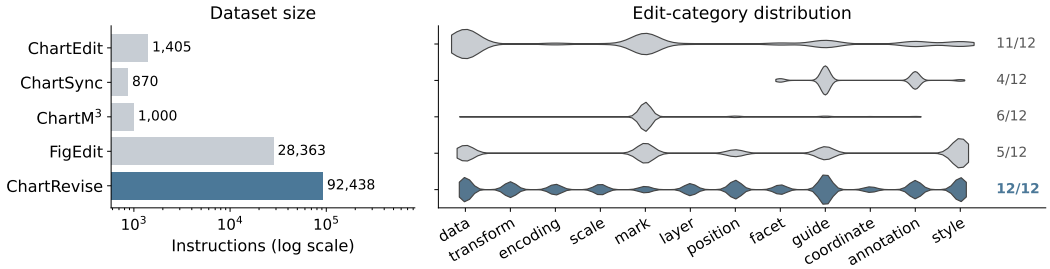


Figure 4: Dataset size and edit-category coverage under our taxonomy.

the displayed resources and spans all twelve categories more consistently, while the other resources span four to eleven. Detailed collection outcomes appear in Section C. Human audit of 500 accepted training records estimates release-weighted task-level correctness at 84.2%. Requirement-level correctness reaches 95.6% over the 1,071 requirements in the 200 audited compound tasks.

4 Evaluation

Fine-tuning on ChartRevise improves requirement completion, coupled-update consistency, and exact-edit success on external benchmarks, while preservation gains are less consistent. We describe the setup and evaluation protocol, report transfer results, analyze the remaining failures, and assess agreement with human judgments.

Setup. We evaluate Phi-3.5-Vision-Instruct (Abdin et al., 2024), InternVL2.5-8B (Chen et al., 2024b), Qwen3.5-4B and 9B (Qwen Team, 2026a), and Granite-Vision-4.1-4B (IBM Granite Vision Team, 2026) on ChartEdit (Zhao et al., 2025a), Chart2Code Level 2 (editing level only) (Tang et al., 2026), ChartM^3 (Yang et al., 2025b), and ChartSync (Yu et al., 2026). None of these benchmarks contributes source charts, instructions, or programs to training. To measure external transfer, we compare each base checkpoint with its fine-tuned counterpart on the same tasks. Models receive the rendered source chart, the complete source program, and the instruction, and generate a complete edited program. Every output is executed in a common rendering sandbox. Judge-based quantities use `gemma-4-31B-it` (Team, 2026) at temperature zero. Training details, benchmark adaptations, and complete benchmark-facing results appear in Sections D.2 and E.1.

4.1 Reference-free evaluation protocol

We measure completion, preservation, and consistency using the shared edit criteria in Section 3, without a ground-truth edited program. Each task has fixed lists of explicit requirements and necessary coupled updates, shared across models. A judge checks each required chart state in the source and edited programs, including whether the edit reaches the intended element and uses valid APIs.

The per-task *completion ratio* (CR) is the fraction of explicit requirements satisfied. *Requirement recall* (CR_μ) is the satisfied fraction pooled over all requirements; *full-completion rate* is the fraction of tasks satisfying every explicit requirement. *Coupled-update recall* is the fraction of required dependent states satisfied, pooled over inventory items. *Gratuitous changes* are neither requested nor necessary coupled updates and are reported as a mean count per task.

For task i, let s_i of K_i explicit requirements be satisfied, g_i count gratuitous changes, and c_i count missed coupled updates. Let $e_i = 1$ if the edited program executes and renders successfully, and $e_i = 0$ otherwise. Thus, completion ratio of task i is $\mathrm{CR}_i = \frac{s_i}{K_i}$, and requirement recall is $\mathrm{CR}_\mu = \frac{\sum_i s_i}{\sum_i K_i}$. Exact editing requires all four conditions,

$$\mathrm{Exact}_i = \mathbf{1}[e_i = 1]\,\mathbf{1}[s_i = K_i]\,\mathbf{1}[g_i = 0]\,\mathbf{1}[c_i = 0].$$

Exact-edit rate is averaged over all tasks. Requirement satisfaction, gratuitous changes, and missed coupled updates are made separately from execution, so correctly implemented changes can still be measured in programs that fail to run. Prompts and implementation details are given in Section D.3.

4.2 Main transfer results

Completion and coupled updates improve more consistently than preservation. Fine-tuning improves requirement recall and coupled-update recall in 19 of 20 model–benchmark comparisons and exact-edit rate in 17, but reduces gratuitous changes in only 10 (Table 2).

Table 2: External transfer under the evaluation protocol. Green numbers mark improvements over the base model, and red numbers mark declines. $rel.\ \Delta$ measures the mean change across models.

Benchmark	Family	Requirement recall (CR_μ) ↑	Full completion ↑	Gratuitous per task ↓	Coupled-update recall ↑	Exact-edit rate ↑
ChartEdit 1,405 tasks	Phi-3.5-V	**.773** +.138	**.674** +.155	**.346** -.205	**.540** +.076	**.425** +.119
	InternVL2.5	**.707** +.234	**.593** +.215	**.373** -.007	**.500** +.188	**.378** +.162
	Qwen3.5-4B	**.867** +.047	**.804** +.050	**.165** -.091	**.688** +.004	**.572** +.055
	Qwen3.5-9B	**.895** +.068	**.846** +.085	.135 +.021	**.731** +.064	**.616** +.032
	Granite	**.812** +.114	**.733** +.128	**.459** -.482	**.622** +.077	**.439** +.168
	rel. Δ of mean	*+17.4%*	*+21.0%*	*-34.1%*	*+15.3%*	*+28.3%*
Chart2Code-L2 1,010 tasks	Phi-3.5-V	**.691** +.120	**.032** +.020	1.433 +.150	**.493** +.182	**.002** +.001
	InternVL2.5	**.624** +.199	**.022** +.019	1.185 +.400	**.440** +.272	**.001** +.001
	Qwen3.5-4B	**.844** +.013	**.171** +.039	**.900** -.051	**.714** +.041	**.044** +.009
	Qwen3.5-9B	**.896** +.053	**.277** +.077	.749 +.043	**.772** +.124	**.090** +.017
	Granite	**.723** +.206	**.040** +.028	1.314 +.196	**.563** +.278	.000 -.002
	rel. Δ of mean	*+18.5%*	*+51.0%*	*+15.2%*	*+43.0%*	*+23.4%*
ChartM^3 1,000 tasks	Phi-3.5-V	**.390** +.097	**.321** +.121	**.926** -.370	**.348** +.094	**.152** +.087
	InternVL2.5	**.338** +.088	**.246** +.073	.809 +.162	**.388** +.153	**.154** +.069
	Qwen3.5-4B	**.623** +.150	**.540** +.125	**.407** -.112	**.553** +.045	**.331** +.121
	Qwen3.5-9B	**.701** +.181	**.607** +.145	**.268** -.021	**.628** +.251	**.378** +.089
	Granite	**.409** +.041	**.347** +.069	**.741** -.445	**.439** +.030	**.188** +.082
	rel. Δ of mean	*+29.3%*	*+34.9%*	*-20.0%*	*+32.1%*	*+59.3%*
ChartSync 870 tasks	Phi-3.5-V	**.819** +.056	**.832** +.041	.210 +.057	**.815** +.054	**.668** +.022
	InternVL2.5	**.838** +.119	**.849** +.141	.185 +.031	**.820** +.121	**.743** +.140
	Qwen3.5-4B	.886 -.002	.879 +.000	.062 +.048	**.825** +.002	.800 -.011
	Qwen3.5-9B	**.927** +.011	.918 -.006	.043 +.017	.853 -.010	.829 -.025
	Granite	**.819** +.036	**.810** +.017	**.252** -.674	**.768** +.022	**.644** +.228
	rel. Δ of mean	*+5.4%*	*+4.7%*	*-40.9%*	*+4.9%*	*+10.6%*
improved in		*19/20*	*18/20*	*10/20*	*19/20*	*17/20*

Using the means over the 20 comparisons, SFT yields relative gains of approximately 15.6% in requirement recall, 19.8% in coupled-update recall, and 22.4% in exact-edit rate. These gains extend beyond explicit requirements to the necessary coupled updates that maintain chart consistency, supporting the dataset's utility on external benchmarks. However, preservation improves less consistently. Although mean gratuitous changes per task decrease by 10.8% overall, they decrease in only 10 of the 20 comparisons. The decomposition exposes preservation regressions that would be obscured by reporting only the overall improvement in exact-edit rate.

Exactness filtering mainly improves coupled-update consistency. Fine-tuning on the released data raises coupled-update recall in 7 of 8 comparisons and exact-edit rate in 6 of 8 relative to

training before exactness filtering (Section 3.4). Requirement recall and gratuitous changes remain close (Section E.5).

4.3 Diagnosing exact-edit performance

Aggregate benchmark scores do not fully capture cross-modal edit grounding. We compare existing benchmark measurements after fine-tuning with the completion, consistency, and preservation outcomes measured by our protocol.

CodeScore rises while executability declines. CodeScore improves in 19 of 20 model–benchmark comparisons, with a mean gain of 8.1 points, whereas ExecRate improves in only five and decreases by 3.0 percentage points on average (Table 3). CodeScore is an LLM-based assessment of modification accuracy and code completeness, following the evaluation used by ChartEdit (Zhao et al., 2025a). The disagreement is largest on Chart2Code-L2, where CodeScore increases for every model but ExecRate falls for four. The disagreement motivates inspecting what was completed and what else changed in both executable and non-executable outputs.

Table 3: Results under the inherited benchmark-facing metrics. Each cell reports the fine-tuned value and its change from the base model.

Family	ChartEdit		Chart2Code-L2		ChartM^3		ChartSync	
	Exec.	Code.	Exec.	Code.	Exec.	Code.	Exec.	Code.
Phi-3.5-V	0.860 +0.000	**75.4** +11.6	0.334 -0.066	**38.0** +10.4	0.650 -0.010	**46.3** +11.2	0.930 -0.010	**85.5** +1.4
InternVL2.5	0.860 -0.040	**71.6** +20.4	0.380 -0.214	**32.6** +11.9	**0.740** +0.050	**43.3** +10.6	0.930 -0.030	**88.2** +8.4
Qwen3.5-4B	**0.910** +0.010	**86.7** +3.6	**0.444** +0.030	**62.1** +1.6	**0.700** +0.040	**66.8** +13.1	0.980 -0.010	**92.6** +0.3
Qwen3.5-9B	0.920 -0.020	**89.8** +6.0	0.540 -0.027	**73.4** +6.7	0.720 -0.010	**74.8** +13.6	0.980 -0.010	94.8 -0.9
Granite	0.860 -0.010	**79.7** +11.2	0.373 -0.223	**43.1** +13.4	0.630 -0.070	**48.4** +4.5	**0.900** +0.030	**83.4** +3.7

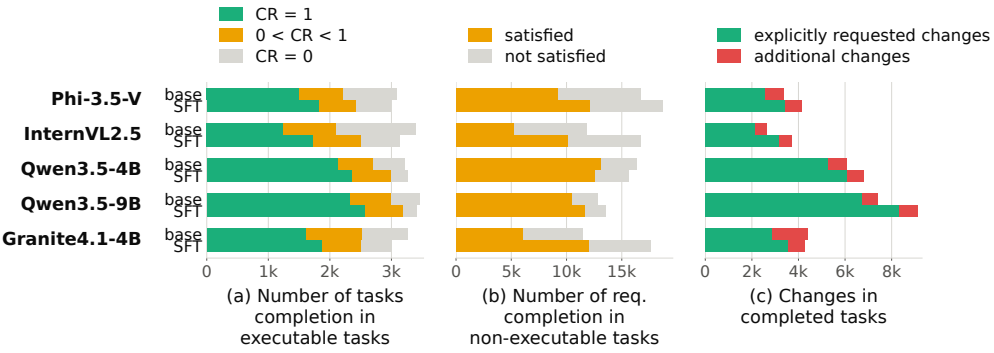


Figure 5: Editing states pooled across benchmarks. (a) Executable tasks grouped by completion ratio. (b) Requirements judged from code in non-executable outputs. (c) Explicitly requested and additional changes (coupled and gratuitous) in tasks satisfying all explicit requirements.

Completing explicit requirements is insufficient for exact editing. After SFT, all five models produce more executable outputs with full completion (CR $=$ 1) and fewer with zero completion (CR $=$ 0), although four produce fewer executable outputs overall (Figure 5(a)). For outputs that fail to run, the number of requirements judged satisfied from code also rises for four of the five models (Figure 5(b)). Even outputs that satisfy every explicit requirement can still contain additional changes, including necessary coupled updates and gratuitous changes (Figure 5(c)).

This breakdown helps interpret the divergence between CodeScore and ExecRate, since requirement completion and execution capture different aspects of an edit (Figure 5(a)), and code-based checks can reveal partial implementation even when a program fails to render (Figure 5(b)). Figure 5(c) further illustrates why evaluating only explicit requirement completion is insufficient. Exactness also depends on whether additional changes are necessary or gratuitous and whether all necessary coupled updates are satisfied. Our protocol therefore checks completion, coupled-update consistency, and preservation separately, with execution and rendering required for exact-edit success.

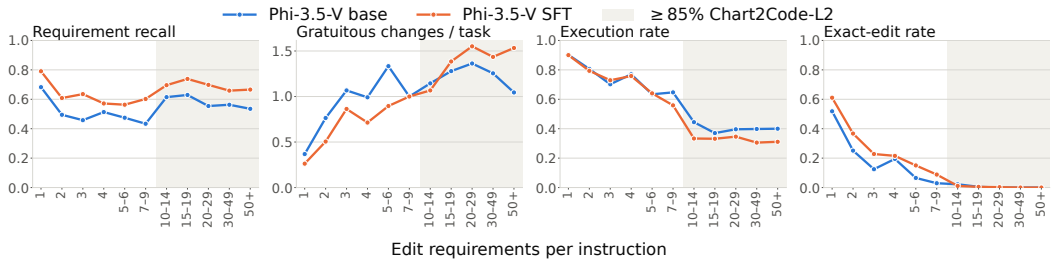


Figure 6: Phi-3.5-Vision outcomes grouped by explicit requirement count across benchmarks. In shaded bins, at least 85% of tasks come from Chart2Code-L2.

The trade-off varies with edit difficulty. Preservation remains challenging both for complex requests and in settings with little room for further improvement. On Chart2Code-L2, four of five models introduce more gratuitous changes after SFT. On ChartSync, Qwen3.5-4B and Qwen3.5-9B already achieve full-completion rates of 87.9% and 92.4% before SFT. Completion therefore changes little, while gratuitous edits increase slightly.

For Phi-3.5-Vision, higher requirement recall after SFT coexists with more gratuitous changes and lower executability on requests with many requirements. We measure *edit depth* by the number of explicit requirements. In Figure 6, SFT improves requirement recall across depth bins, but its advantage in reducing gratuitous changes reverses for requests with at least 15 requirements. Exact-edit success remains near zero at deep depths. Deeper requests can involve more code and interacting components, creating more opportunities for unrelated changes and execution errors. These results highlight a trade-off, that encouraging models to complete more changes required by complex requests can improve completion but increase execution risk, whereas conservative editing by the base model may preserve executability while leaving requirements unmet.

Execution errors. Most of the increase in execution failures after SFT falls within exception classes already observed in the base models (Section E.2).

4.4 Reliability of the evaluation

Evaluation protocol judgments agree with human assessments on 93% of the audited outputs, and both evaluations find positive fine-tuning gains on all four audited benchmark subsets. The audit covers blinded outputs from 200 paired tasks in which both programs render (Section E.4). A positive human exact-edit judgment requires every listed requirement to be satisfied and no gratuitous change. The protocol and human evaluations estimate aggregated gains of 14.0 and 14.5 percentage points, respectively, supporting agreement on the direction of the gains.

Some disagreements with human judgments arise from render-time layout effects, such as overlapping elements. These cases reveal a limitation of our code-only protocol, that code permits checks of object properties and dependencies even for non-executable outputs, but does not directly show their rendered layout. Chart evidence could complement these checks by revealing layout errors that violate explicit requirements, necessary coupled updates, or preservation. But the general aesthetic quality remains outside of the evaluation scope.

5 Conclusion

ChartRevise provides a grammar-derived dataset and a reference-free protocol for exact program-grounded chart editing. Operation templates with source-program checks give systematic coverage across chart types and libraries, and requirement-level audits catch incomplete edits during construction. The protocol separately scores explicit requirements, coupled updates, and gratuitous changes, and agrees with human exact-edit judgement on 93% of audited outputs. Across five models and four external benchmarks, fine-tuning improves requirement recall in 19 of 20 comparisons and exact-edit rate in 17 but reduces gratuitous changes in only 10, leaving preservation on complex requests as the main open challenge.

AI USE STATEMENT

We used generative AI to generate edited chart programs and assist automated validation, feedback-based repair, and auditing; LLM also serves as the judge in our evaluation protocol and benchmark-facing metrics. Additionally, we used generative AI tools to edit the manuscript for readability and to review manuscript drafts. We have reviewed all AI-assisted work: dataset quality was checked through execution checks, cross-model auditing, and human validation of sampled records (Section 3.5), the automatic judge was compared with human judgments (Section 4.4), and the authors checked all manuscript text. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The dataset is derived from ChartNet's publicly available synthetic chart programs (Kondic et al., 2026). Reuse of the source material remains subject to its license and attribution requirements. The four external benchmarks are used only for evaluation under their original terms and are not redistributed. All ChartRevise's charts are synthetic and contain no personal information. Human audits assess editing correctness on sampled records and model outputs. Since automatic editing can alter data, omit requested updates, or introduce unrelated changes, generated charts require verification before use.

REPRODUCIBILITY STATEMENT

The appendix documents the operation inventory, grounding procedure, and construction prompts (Section B), training and inference settings (Sections D.1 and D.2), and the evaluation-protocol prompts with a worked example (Sections D.3 and D.4). Complete external-benchmark results, protocol diagnostics, and the exactness-filtering ablation are provided in Section E. The release schema distinguishes distributed records from internal collection logs (Section F.1), while the human-audit descriptions specify the sampled populations and correctness criteria (Sections C.3 and E.4). We will release the dataset, the evaluation code, and the full prompts upon publication.

REFERENCES

Marah Abdin et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Tim Brooks, Aleksander Holynski, and Alexei A Efros. Instructpix2pix: Learning to follow image editing instructions. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18392–18402. IEEE, 2023.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, et al. Can it edit? evaluating the ability of large language models to follow code editing instructions. *arXiv preprint arXiv:2312.12450*, 2023.

Liangyu Chen, Yichen Xu, Jianzhe Ma, Yuqi Liu, Donglu Yang, Liang Zhang, Zihao Yue, Wenxuan Wang, and Qin Jin. Charteditor: A reinforcement learning framework for robust chart editing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 20199–20207, 2026.

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, et al. Alpagasus: Training a better alpaca with fewer data. In *International Conference on Learning Representations*, volume 2024, pp. 34767–34797, 2024a.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, et al. Internvl: Scaling up vision foundation models and aligning

for generic visual-linguistic tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 24185–24198, 2024b.

Jaemin Cho, Yushi Hu, Jason Baldridge, Roopal Garg, Peter Anderson, Ranjay Krishna, Mohit Bansal, Jordi Pont-Tuset, and Su Wang. Davidsonian scene graph: Improving reliability in fine-grained evaluation for text-to-image generation. In *International conference on learning representations*, volume 2024, pp. 15625–15645, 2024.

Kanika Goswami, Puneet Mathur, Ryan Rossi, and Franck Dernoncourt. Plotedit: Natural language-driven accessible chart editing in pdfs via multimodal llm agents. In *European Conference on Information Retrieval*, pp. 130–134. Springer, 2025.

Yucheng Han, Chi Zhang, Xin Chen, Xu Yang, Zhibin Wang, Gang Yu, Bin Fu, and Hanwang Zhang. Chartllama: A multimodal llm for chart understanding and generation. *arXiv preprint arXiv:2311.16483*, 2023.

Minggui He, Mingchen Dai, Jian Zhang, Yilun Liu, Shimin Tao, Pufan Zeng, Osamu Yoshie, and Yuya Ieiri. Chart specification: Structural representations for incentivizing vlm reasoning in chart-to-code generation. *arXiv preprint arXiv:2602.10880*, 2026.

IBM Granite Vision Team. Granite 4.1 vision, 2026. URL `https://huggingface.co/ibm-granite/granite-vision-4.1-4b`.

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? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Manav Nitin Kapadnis, Lawanya Baghel, Atharva Naik, and Carolyn Rosé. Charteditbench: Evaluating grounded multi-turn chart editing in multimodal language models. *arXiv preprint arXiv:2602.15758*, 2026.

Jovana Kondic, Pengyuan Li, Dhiraj Joshi, Isaac Sanchez, Ben Wiesel, Shafiq Abedin, Amit Alfassy, Eli Schwartz, Daniel Caraballo, Yagmur Gizem Cinar, et al. Chartnet: A million-scale, high-quality multimodal dataset for robust chart understanding. *arXiv preprint arXiv:2603.27064*, 2026.

Max Ku, Dongfu Jiang, Cong Wei, Xiang Yue, and Wenhu Chen. Viescore: Towards explainable metrics for conditional image synthesis evaluation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12268–12290, 2024a.

Max Ku, Tianle Li, Kai Zhang, Yujie Lu, Xingyu Fu, Wenwen Zhuang, and Wenhu Chen. Imagenhub: Standardizing the evaluation of conditional image generation models. In *International Conference on Learning Representations*, volume 2024, pp. 46689–46722, 2024b.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Li Li, Ryan Rossi, Sungchul Kim, Sunav Choudhary, Franck Dernoncourt, Puneet Mathur, Zhengzhong Tu, and Yue Zhao. Charts are not images: On the challenges of scientific chart editing. In *International Conference on Learning Representations*, volume 2026, pp. 11937–11966, 2026.

Fangyu Liu, Francesco Piccinno, Syrine Krichene, Chenxi Pang, Kenton Lee, Mandar Joshi, Yasemin Altun, Nigel Collier, and Julian Eisenschlos. Matcha: Enhancing visual language pretraining with math reasoning and chart derendering. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12756–12770, 2023a.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 2511–2522, 2023b.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. ChartQA: A benchmark for question answering about charts with visual and logical reasoning. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 2263–2279, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-acl.177. URL https://aclanthology.org/2022.findings-acl.177/.

Ahmed Masry, Parsa Kavehzadeh, Enamul Hoque, Shafiq Joty, et al. Unichart: A universal vision-language pretrained model for chart comprehension and reasoning. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 14662–14684, 2023.

Ahmed Masry, Megh Thakkar, Aayush Bajaj, Aaryaman Kartha, Enamul Hoque, and Shafiq Joty. Chartgemma: Visual instruction-tuning for chart reasoning in the wild. In *Proceedings of the 31st International Conference on Computational Linguistics: Industry Track*, pp. 625–643, 2025.

Arjun Panickssery, Samuel R Bowman, and Shi Feng. Llm evaluators recognize and favor their own generations. *Advances in Neural Information Processing Systems*, 37:68772–68802, 2024.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2015.

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.

Shelly Sheynin, Adam Polyak, Uriel Singer, Yuval Kirstain, Amit Zohar, Oron Ashual, Devi Parikh, and Yaniv Taigman. Emu edit: Precise image editing via recognition and generation tasks. In *2024 ieee/cvf conference on computer vision and pattern recognition (cvpr)*, pp. 8871–8879. IEEE, 2024.

Jiahao Tang, Henry Hengyuan Zhao, Lijian Wu, Zijian Zhang, Yifei Tao, Dongxing Mao, Yang Wan, Jingru Tan, Min Zeng, Min Li, et al. From charts to code: A hierarchical benchmark for multimodal models. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13467–13566, 2026.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, et al. Large language models are not fair evaluators. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 9440–9450, 2024.

Su Wang, Chitwan Saharia, Ceslee Montgomery, Jordi Pont-Tuset, Shai Noy, Stefano Pellegrini, Yasumasa Onoe, Sarah Laszlo, David J Fleet, Radu Soricut, et al. Imagen editor and editbench: Advancing and evaluating text-guided image inpainting. In *2023 ieee/cvf conference on computer vision and pattern recognition (cvpr)*, pp. 18359–18369. IEEE, 2023.

Hadley Wickham. A layered grammar of graphics. *Journal of computational and graphical statistics*, 19(1):3–28, 2010.

Leland Wilkinson. The grammar of graphics. In *Handbook of computational statistics: Concepts and methods*, pp. 375–414. Springer, 2011.

Chengyue Wu, Zhixuan Liang, Yixiao Ge, Qiushan Guo, Zeyu Lu, Jiahao Wang, Ying Shan, and Ping Luo. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3006–3028, 2025.

Renqiu Xia, Hancheng Ye, Xiangchao Yan, Qi Liu, Hongbin Zhou, Zijun Chen, Botian Shi, Junchi Yan, and Bo Zhang. Chartx & chartvlm: A versatile benchmark and foundation model for complicated chart reasoning. *IEEE Transactions on Image Processing*, 2025.

Pengyu Yan, Mahesh Bhosale, Jay Lal, Bikhyat Adhikari, and David Doermann. Chartreformer: Natural language-driven chart image editing. In *International Conference on Document Analysis and Recognition*, pp. 453–469. Springer, 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. In *International Conference on Learning Representations*, volume 2025, pp. 26590–26646, 2025a.

Donglu Yang, Liang Zhang, Zihao Yue, Liangyu Chen, Yichen Xu, Wenxuan Wang, and Qin Jin. Chartm3: Benchmarking chart editing with multimodal instructions. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pp. 5001–5009, 2025b.

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. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 11789–11804, 2024.

Jiakang Yu, Yixuan Chai, Tianci Wang, Rihui Jin, Guangkai Xu, Hongtao Deng, Xun Zhu, Wang Gao, Xinrun Guo, and Haipang Wu. Chartsync: A benchmark for visuo-logical cascading chart editing. *arXiv preprint arXiv:2607.10301*, 2026.

Kai Zhang, Lingbo Mo, Wenhu Chen, Huan Sun, and Yu Su. Magicbrush: A manually annotated dataset for instruction-guided image editing. *Advances in neural information processing systems*, 36:31428–31449, 2023.

Xuanle Zhao, Xuexin Liu, Yang Haoyue, Xianzhen Luo, Fanhu Zeng, Jianling Li, Qi Shi, and Chi Chen. Chartedit: How far are mllms from automating chart analysis? evaluating mllms' capability via chart editing. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3616–3630, 2025a.

Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. Chartcoder: Advancing multimodal large language model for chart-to-code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7333–7348, 2025b.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

A Extended Related Work

Chart understanding and chart-to-code tasks. Related work covers chart understanding and generation (Masry et al., 2022; Wu et al., 2025; Yang et al., 2025a; Liu et al., 2023a; Masry et al., 2023; Han et al., 2023; Xia et al., 2025; Masry et al., 2025; Zhao et al., 2025b; He et al., 2026), code generation and editing (Chen et al., 2021; Cassano et al., 2023; Jimenez et al., 2024), and instruction-guided image editing (Brooks et al., 2023; Zhang et al., 2023; Sheynin et al., 2024; Wang et al., 2023). Our setting requires implementing a visual edit request in an existing plotting program while preserving unrelated chart content. This requires grounding each visual request in both the chart's components and their code implementation.

Chart editing and coverage. Chart-editing resources cover program revision (Zhao et al., 2025a; Tang et al., 2026; Kapadnis et al., 2026), multimodal instructions (Yang et al., 2025b; Chen et al., 2026), and image-space edits (Li et al., 2026; Yu et al., 2026). Our evaluation provides the source chart and program and requires an edited program (Section D.2). For construction, we use the grammar of graphics (Wilkinson, 2011; Wickham, 2010; Satyanarayan et al., 2016) to organize semantic object–action templates with source-specific applicability checks across chart types and libraries. Operation categories and interaction formats describe different aspects of coverage. ChartRevise therefore emphasizes source-grounded applicability and reuse of semantic edit definitions across plotting libraries.

Evaluation. Atomic decomposition and instruction-aware visual evaluation provide related foundations (Cho et al., 2024; Ku et al., 2024b;a). In chart editing, ChartM^3 and FigEdit measure instruction compliance and preservation (Yang et al., 2025b; Li et al., 2026); ChartEditBench uses assertions for programmatic instructions (Kapadnis et al., 2026); and ChartSync checks text-to-geometry dependencies (Yu et al., 2026). Our protocol jointly checks explicit requirements, source-derived coupled states, and unrelated changes in the edited program. This separates completion, consistency, and preservation without a reference edit, with successful execution and rendering required for exact-edit success (Section 4.1).

Judge reliability. LLM judges can exhibit position and self-preference biases (Wang et al., 2024; Panickssery et al., 2024). Following work on alignment with human judgments (Liu et al., 2023b; Zheng et al., 2023), we report cross-model construction audits and human agreement (Sections 3.4 and 4.4). Our validation targets edit-specific requirement satisfaction and preservation in chart programs, rather than general response quality.

B Dataset Construction and Prompts

This section follows the construction pipeline of Section 3. Prompt descriptions are provided at the stage where they are used.

B.1 Taxonomy scope and library realizations

Table 4 lists the program objects through which each category is realized. The same object can serve several categories; the request determines the category.

Extending the inventory. A new edit type needs a target, action template, and applicability checks. A new chart type needs compatibility entries; a new library needs source-program bindings and validation. The twelve augmentation templates extend existing categories under these same requirements.

B.2 Operation inventory

The inventory contains 344 object–action templates across twelve categories. The complete main-template list and compatibility rows are stored in `edit_matrix.json`. Table 4 lists target objects, and Table 6 gives released record counts by category.

Table 4: Semantic objects of the twelve edit categories.

Category	Semantic objects
data	dataset, field, record, series, category
transform	aggregation, bin, stack, derived metric, sort, window
encoding	position, colour, size, shape, opacity, and text channels
layer	primary and overlay layers, error bars, reference line or band, secondary axis
mark	line, point, bar, area, arc, box, violin, cell
position	margins, anchor, z-order, aspect ratio, legend and annotation placement
scale	position, colour, size, and opacity scales, normalization
guide	axis, ticks, gridlines, legend, colorbar
coordinate	Cartesian axes, polar coordinates, spines
facet	subplot grid, facet layout, shared axes, panel title
annotation	text, arrow, data label, shape, image
style	figure, plot area, title, axis label, theme, palette

B.3 Grounded instruction materialization

Table 5 lists recorded template–instruction pairs. Field placeholders are filled from source-program values, while arguments such as a new grid size specify a requested state.

Table 5: Template → grounded instruction examples.

Edit	Chart	Template → realized instruction
E026	Bubble (seaborn)	`Add subgroup by {subgroup_field} within {primary_group_field}` → "For the subgroup, add subgroup by `'Crop'` within `'Region'`."
E311	Bubble (seaborn)	`Map {facet_row_field} to rows and {facet_column_field} to columns` → "For the facet layout, map `'Region'` to rows and `'Crop'` to columns."
E308	Pie (matplotlib)	`Set the multi-facet grid to {facet_rows} rows and {facet_columns} columns` → "For the subplot grid, set the multi-facet grid to 2 rows and 2 columns."
E100	Bubble (plotly)	`Remove facet mapping for {facet_field}` → "For the facet channel, remove facet mapping for `'Gender'`."

Geometry-bound arguments. Some arguments bind against the chart's rendered geometry rather than its data schema. For example, a requested figure width is floored against the length of the chart's own title, and legend-anchoring instructions name `bbox_to_anchor` explicitly.

B.4 Program generation: editor prompt

The editor and first-pass judge use `gemma-4-31B-it` served locally. The editor receives this system prompt, then the instruction and the complete source program, and returns a full-file rewrite, so that a patch that fails to apply cannot be confused with an editing failure.

```
You are a chart-editing engine. Follow the edit request exactly.

Respond with ONLY the complete edited Python program in one ```python code block.

This is a full-file rewrite, not a diff or patch. Do not omit unchanged code,
abbreviate with ..., or use comments like "# unchanged" or "# rest of code stays
the same". Write every line of the runnable program in full.

Make exactly the requested edit and no additional edit. Preserve all unrelated
chart data, encodings, labels, legends, annotations, scales, layout, styling,
chart type, and save behavior unless the instruction explicitly asks to change
```

```
them.
```

B.5 INITIAL COLLECTION VALIDATION

This first-pass collection screen uses code and image evidence. It is distinct from the code-only exact-edit audit in Section D.3. The judge receives the instruction, named target, source and edited programs, code diff, and before/after renders. The condensed prompt below describes this screen.

```
You are a meticulous chart-edit reviewer. You are given:
1. The edit instruction and its named target object
2. BEFORE and AFTER chart images
3. The SOURCE program
4. The COMPLETE AFTER program
5. The unified code diff

Anchor every judgment on the NAMED TARGET OBJECT. First establish what happened to
that object specifically -- compare its appearance, content and position between
BEFORE and AFTER, and check the diff for a change that actually targets it. Only then
consider the rest of the chart.

Judge four independent questions:
- applied: Was the exact requested edit carried out ON THAT TARGET OBJECT? Use the
  diff and the complete program to confirm the mechanism really lands (a value set but
  never used, or overwritten by a later line, is not applied; a comment describing an
  effect is not an implementation). Use the images to confirm it landed on the RIGHT
  object -- a change that hits a different object than the one named is applied:false,
  even if the code looks correct. Visually subtle changes (a linewidth, a small margin,
  a few points of font size) still count as applied when the code unambiguously
  implements them.
- no_additional_change: Was everything else left unchanged, with no extra edit?
- position_preserved: Did every object (marks, axes, legend, colorbar, annotations,
  labels, subplots/facets) keep its correct position? If the instruction itself asked
  to move something, judge only whether THAT move landed and nothing else shifted.
- no_object_removed: Did every visual object that existed in BEFORE, and that the instruction
  did NOT ask to remove, still exist in AFTER? Inventory the BEFORE image's furniture --
  legend, colorbar, title, axis labels, tick labels, pre-existing data labels,
  annotations, gridlines, error bars, reference lines, each data series -- and confirm
  each survives. A frequent failure is an edit that lands correctly while silently
  destroying something unrelated (adding a data label but dropping the legend,
  replotting a series so earlier labels vanish). If the instruction explicitly asked to
  remove that object, this stays true.

Collateral damage ELSEWHERE does not make applied false -- it belongs in the other
three fields. Damage to the TARGET ITSELF does make applied false, per the rule above.

Count changes to data, labels, legends, colorbars, annotations, marks, encodings, axes,
scales, layout, styling, sorting, filtering, faceting, chart type, or save behavior as
additional changes unless explicitly requested.

The reason MUST state what happened to the named target object specifically -- not a
generic statement that the chart changed.

Respond with ONLY one JSON object:
{"applied": true/false, "no_additional_change": true/false,
"position_preserved": true/false, "no_object_removed": true/false,
"reason": "what happened to the target object, in one or two short sentences"}
```

The main collection stores `render_ok` and the four judge flags. Applied and strict success share the all-attempt denominator, so their difference counts applied-but-not-strict attempts.

B.6 FEEDBACK REPAIR PROMPT

A repair attempt is issued when a candidate fails to render or is judged not applied, and carries the traceback or the judge's rationale. The release was cleaned in three passes, each allowing up to two further attempts.

```
You are a chart-editing engine. You previously attempted to apply an edit instruction
to a Python plotting program, but the result had a problem -- you are given the
ORIGINAL instruction, YOUR PREVIOUS attempt's code, and FEEDBACK describing exactly
what went wrong (either a render error/traceback, or a reviewer's note that the
instruction wasn't actually carried out). Produce a corrected version of the program
that still applies the original instruction and fixes the specific problem in the
feedback. Do not reintroduce the same mistake, and do not undo the original edit while
```

```
fixing the problem.
This is NOT a diff or patch task -- respond with ONLY the full corrected program, every
line of it, in a single ```python code block, ready to run as-is. Never abbreviate or
omit unchanged code with placeholders like '...', '# unchanged', or similar -- write out
every line in full, exactly as if you were submitting the complete file.
```

B.7 Compound auditing and post-processing

Audit stages and counts. The atomic audit starts from the 5,759 compound records of the preceding release. Of 710 defective records it retried, 472 were recovered. Unstable tier-3 records were excluded after a pilot on 60 of them recovered only 40%. In total 1,240 records were removed (997 tier-3, 218 tier-2, and 25 templated), leaving 4,519 compound records. The 34.6% omission rate in Section 3.4 is measured on the 4,611 tier-2 and tier-3 records that carried an initial verdict, of which 1,596 had been marked applied although at least one requirement was not met.

The quarantine audit groups defective instructions into seven recurring classes. The most common failures are cross-field: incompatible operands, generated field names that do not match the requested operation, numeric formats that do not match the field type, degenerate target choices, and fabricated values. Defective records are excluded from the release.

C Dataset Characterization and Release Validation

This section gives coverage details, collection outcomes, and the accepted-set audit summarized in Section 3.5.

C.1 Coverage notes

Each record retains its source `pool_idx`. When the source pool lacks a required object, such as an annotation arrow or reference line, an auxiliary source chart is derived from an existing program and the precondition is checked again.

A *compatibility cell* is an (edit type, chart type) pair. The initial compatibility matrix contains 4,316 eligible cells. Collection uses 25 successful records per cell as a sampling target; this is not a guarantee that every cell reaches that count, or that all collected successes survive release filtering.

The released manifest, which also includes later templates and compound records, contains 4,619 populated (template, chart type) pairs; 677 hold at least 25 records. These released counts are separate from the initial matrix and its collection target. At the coarser level used in Figure 3, the twelve categories plus the compound stratum form 258 populated (category or stratum, chart type) bins, 256 of them with at least 25 records. Coverage is therefore non-uniform, and population of a cell should not be read as attainment of the sampling target.

C.2 Released dataset composition

Table 6 reports the final training and test records by category. Compound requests combine operations from the twelve categories and are listed separately.

The 20 chart types are: Area Chart, Bar Chart, Box Plot, Bubble Chart, Funnel Chart, Heatmap, Histogram, Kernel Density Estimate Plot, Line Chart, Multi-Axes Chart, Pie Chart, Radar Chart, Ring Chart, Rose Chart, Scatter Plot, Stem Plot, Swarm Plot, Tornado Chart, Treemap, and Violin Plot.

Table 6: Per-category composition of the released dataset. Compound records are training-only.

Category	Train	Test	Total
guide	14,356	253	14,609
data	11,223	218	11,441
style	10,994	252	11,246
position	8,521	223	8,744
annotation	8,385	187	8,572
transform	7,171	177	7,348
layer	5,583	90	5,673
encoding	4,950	83	5,033
scale	4,933	120	5,053
facet	4,422	97	4,519
compound	4,519	0	4,519
mark	2,971	69	3,040
coordinate	2,561	80	2,641
All	**90,589**	**1,849**	**92,438**

C.3 Dataset characterization and human assessment

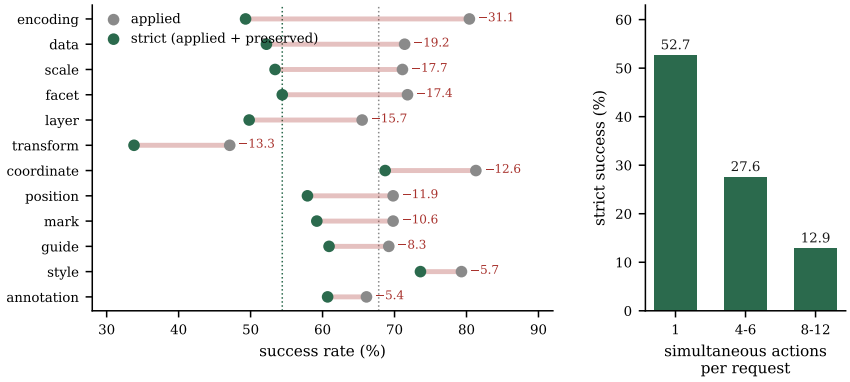


Figure 7: Preservation is a second axis of difficulty, not a proxy for the first. (a) The bar joining the two markers is the applied/strict gap; dotted lines are the earlier export means. (b) Strict success by collection subset. Tier 2 contains 4–6 actions and tier 3 contains 8–12.

Figure 7 plots (a) applied and strict success by category and (b) strict success by collection subset, from the collection-stage export.

Human assessment of accepted training records. We audit a stratified sample of 500 accepted records: 300 from the 86,070 single-action records and 200 from the 4,519 compound records. Each record is executed and inspected against its source.

Task-level correctness requires successful execution, satisfaction of every explicit requirement, and preservation of unrelated chart content. The overall estimate weights the two subsets by their proportions in the training set. **Requirement-level correctness** measures individual requested changes within the 200 audited compound records; it does not require the entire record to pass. Table 7 reports both levels.

Table 7: Human validation of accepted training records. Panel (a) reports task-level correctness by subset and the training-set-weighted estimate. Panel (b) reports correctness of individual requirements in the same 200 compound records.

Subset	Audited units	Correct	Correctness	95% CI
(a) Task level: records				
Single-action	300	255	85.0%	[80.5, 88.6]
Compound	200	139	69.5%	[62.8, 75.5]
Weighted	500	—	**84.2%**	[80.4, 88.1]
(b) Requirement level: compound requirements				
Compound	1,071	—	**95.6%**	[94.2, 96.7]

D Training and Evaluation Details

This section gives training and inference settings, metric conventions, and the code-only exact-edit protocol prompts.

D.1 Training configuration

All five fine-tuned checkpoints in Section 4 are full-parameter supervised fine-tunes of the corresponding base model. Each model is trained for one epoch on the released training split (90,589 records), starting from the public base checkpoint. Every run uses learning rate 1×10^{-5} with a cosine schedule, bfloat16 precision, weight decay 0, gradient clipping at 1.0, and gradient checkpointing. We evaluate the weights saved at the end of the epoch. Table 8 lists the settings that differ across model families.

Table 8: Per-model fine-tuning settings. Effective batch is per-device batch $\times$ gradient accumulation $\times$ number of GPUs. Settings shared by all runs are given in the text.

	Phi-3.5-V	InternVL2.5-8B	Qwen3.5-4B	Qwen3.5-9B	Granite-4.1-4B
Effective batch	$1 \times 16 \times 2 = 32$	$1 \times 8 \times 4 = 32$	$4 \times 4 \times 4 = 64$	$4 \times 2 \times 8 = 64$	$1 \times 16 \times 4 = 64$
Optimizer steps	2,830	2,805	1,416	1,416	1,416
Warmup steps	20	20	60	60	60
Optimizer	AdamW (8-bit)	AdamW (8-bit)	AdamW (8-bit)	AdamW	AdamW (8-bit)
Max sequence length	8,192	6,144	8,192	8,192	8,192
Image input	1 crop	$\leq$6 tiles + thumbnail	$\leq$1,280 visual tokens	$\leq$1,280 visual tokens	384×384
Training records	90,589	89,746[†]	90,589	90,589	90,589
Parallelism	DDP	ZeRO-3	ZeRO-3	ZeRO-3	ZeRO-2
Hardware	2×A100 40 GB	4×A100 40 GB	4×A100 40 GB	8×80 GB GPUs	4×L40S 48 GB

[†]843 records whose tokenized length exceeds the 6,144-token limit are dropped rather than truncated.

D.2 Inference and evaluation configuration

Evaluation uses the model families listed in Section 4. Each editing model receives the rendered source chart, complete source program, and text edit instruction and returns a full edited program. The automatic exact-edit judge receives code and textual task information only; rendered images are not inputs to that judge. All four benchmarks provide source code for this evaluation; metrics requiring visual pointers are not used. The evaluated task counts are 1,405 for ChartEdit, 1,010 for Chart2Code-L2, 1,000 for ChartM^3, and 870 for ChartSync.

Base and fine-tuned models are served with vLLM Kwon et al. (2023) (maximum model length 32,768) under the same settings: greedy decoding (temperature 0), at most 6,144 new tokens, and no repetition penalty. Every generated program is executed in a single rendering environment (Python 3.12, Matplotlib 3.11.1, Plotly 6.9.0) with a 60-second timeout per program. The exact-edit judge settings are given in Section D.3. Table 9 lists the metrics and their provenance.

Table 9: Metric inventory and provenance.

Metric	Source	Meaning / use
ExecRate	Mechanical	Fraction of all tasks whose generated program executes and renders.
CodeScore	LLM judge	Continuous editing-quality score over the generated program; can score non-running programs.
ChartScore	LLM judge	Overall rendered-chart quality; not target-edit success.
AppliedRate	LLM judge	Whether the requested edit reached the named target.
EditFidelity	LLM judge	Applied edit with no unrequested change, unintended reposition, or destroyed object.
Requirement recall (CR_μ)	GT-free judge	Micro-aggregated fraction of explicit requirements satisfied across all tasks.
Full completion	GT-free judge	Fraction of tasks satisfying every explicit requirement, independently of execution.
Gratuitous changes	GT-free judge	Unrequested changes that are not unavoidable consequences of requested edits, averaged per task.
Coupled-update recall	GT-free judge	Fraction of necessary updates satisfied, using one fixed source-derived inventory per task.
Missed coupled updates	GT-free judge	Unmet fixed-inventory end states: omitted or incorrectly implemented, including when the forcing requirement is unmet.
Exact-edit rate	Code judge + execution	Successful execution/rendering and a complete verdict with all requirements satisfied and neither deviation.
TESR / VLCS / BFS	ChartSync	Native text-edit, geometry, and whole-output fidelity signals.

D.3 Exact-edit protocol prompts

The four listings below summarize the code-only exact-edit protocol. They describe its inputs, decision rules, and JSON schemas. The judge receives the instruction-derived requirements, source and edited programs, and, for coupled checks, the fixed inventory of necessary updates. It receives no rendered images. Execution and rendering are checked separately to determine the executable success.

External-benchmark evaluation uses `gemma-4-31B-it` with greedy decoding (`temperature` 0, `seed` 23) and JSON output. Stages A1 and A2 run once per task before any model output is seen. The construction cross-audit uses `Qwen3.8-27B` with the same code-only protocol, decision rules, and output schemas. Changing the judge model does not introduce a separate definition of exactness. The earlier four-flag, code-and-image screen is documented in Section B.5.

Scope and correctness are separate checks. A coupled update is a required change to a source-dependent state. Its correctness is checked against the intended end state, including when the program updates it automatically. An omitted or incorrectly implemented update fails this check. A gratuitous change is an actual modification outside the scope of the explicit requirements and necessary coupled updates. Failure of an in-scope update is not, by itself, evidence of an additional out-of-scope modification. An output can fail both checks when it also changes unrelated content; the coupled inventory and extra-change list provide separate evidence for these judgments. Properties that merely need to remain unchanged belong to preservation and are not added to the coupled inventory solely because they existed in the source.

The reported coupled-update count measures unmet required end states, covering both omission and incorrect implementation. It does not distinguish these two subtypes. When the forcing requirement is not implemented, its coupled end state is also marked unmet under the fixed inventory; this score must therefore be interpreted alongside explicit requirement completion.

D.3.1 Stage A1: requirement decomposition

```
Decompose the chart-editing instruction into atomic requirements.
```

```
Each requirement is one independently checkable change, stated as a visible property
of the edited chart. Split compound requests into separate items, keep the
instruction's wording, and do not add requirements it does not state. Omit anything
that is not visible in the rendered chart (variable names, comments, file paths).

INSTRUCTION: {instruction}

Return JSON: {"requirements": ["<requirement>", ...]}
```

D.3.2 STAGE A2: COUPLED-UPDATE INVENTORY

```
List every coupled update: a visible change that the requirements do not state but that
any correct implementation must make, because an element of the SOURCE program depends
on something a requirement changes. Typical cases:
  - a legend entry, label, title, annotation or tick label that refers to changed data
  - an axis limit, tick set or colorbar range that no longer fits the changed data
  - arrays that must keep matching lengths, or style lists sized to the number of series
  - positions or offsets computed from the number of groups or the changed values
Name the source element behind each item and the requirement that forces it. List an
item even if it would update automatically. Do not restate a requirement. Return an
empty list if nothing depends on the requested changes. Do not add unrelated properties
that merely need to remain unchanged; preserving them is a preservation check, not a
necessary coupled update.

SOURCE: {source}
REQUIREMENTS: {requirements}

Return JSON: {"coupled": [{"update": "<required end state>", "element": "<source element>",
                    "forced_by": <requirement number>}, ...]}
```

D.3.3 REQUIREMENT SATISFACTION AND GRATUITOUS CHANGES

```
Audit the SOURCE and EDITED programs against the fixed requirements. No chart images
are provided. Infer chart properties from the plotting code and its data. A requirement
is satisfied only when the edited program implements the requested state on its target;
a comment, unused assignment, or overwritten setting is not sufficient evidence. Judge
these component properties independently of the program's overall execution outcome.
1. For each requirement, decide whether it is satisfied.
2. List additional chart-affecting changes evident in the programs and classify their scope:
   - "necessary": a companion change required by an explicit requirement;
   - "gratuitous": a change outside the explicit requirements and required companions.
   Whether a necessary companion was implemented correctly is checked separately. Do not
   infer a gratuitous change solely from an omitted or incorrect in-scope update. If the
   output also changes an unrelated property, list that modification as gratuitous.
   If unsure whether an additional modification is in scope, classify it as gratuitous.
Ignore code-only differences that cannot affect the chart: formatting, comments, output
file paths, and equivalent rewrites.

SOURCE: {source}
REQUIREMENTS: {requirements}
EDITED: {edited}

Return JSON: {"requirements": [{"i": 1, "satisfied": true|false}, ...],
          "extra_changes": [{"what": "<change>", "kind": "necessary"|"gratuitous"}, ...]}
```

D.3.4 COUPLED-UPDATE SATISFACTION

```
Audit the SOURCE and EDITED programs against the fixed coupled-update inventory.
No chart images are provided. Infer the required end states from code and data.
For each numbered coupled update (each forced by one requirement), decide:
  trigger_made: does the edit make, fully or partly, the change of the requirement that
             forces it?
  satisfied: does the EDITED program implement the required end state, whether through
             an explicit edit or automatic synchronization? An omitted or incorrect
             implementation is false. If the trigger was not made, answer false.

SOURCE: {source}
REQUIREMENTS: {requirements}
COUPLED UPDATES: {coupled}
EDITED: {edited}

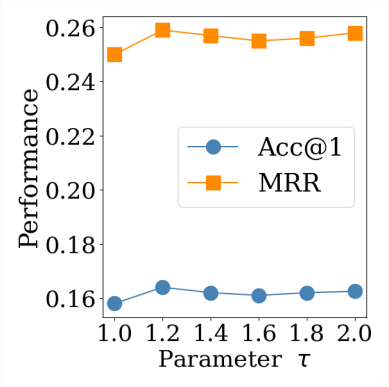
Return JSON: {"coupled": [{"k": 1, "trigger_made": true|false, "satisfied": true|false},
...]}
```

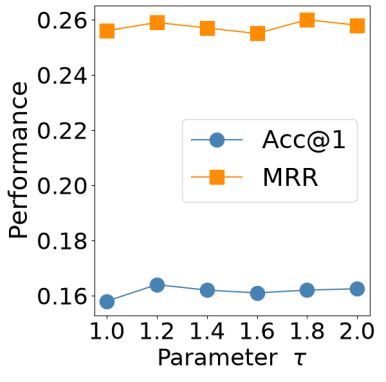
D.4 EXAMPLES OF REQUIREMENTS, COUPLED UPDATES, AND GRATUITOUS CHANGES

The following ChartEdit examples illustrate explicit requirements, coupled updates, and gratuitous changes. Task 451 is presented as a source-based illustration of a missed versus completed legend update. Tasks 1226 and 585 show fine-tuned and base Phi-3.5-V outputs, respectively.

Gratuitous change: every requirement met, plus an unrequested edit

ChartEdit task 1226. *"Modify the 1.8 value of the 'MRR' data series from 0.256 to 0.260, ensuring the marker remains square and orange."* Three requirements (the value at 1.8 is 0.260, and its marker is square and orange) define the requested edit.






```
satisfied 3/3
gratuitous 1
executes yes
```
Not exact.

source fine-tuned output

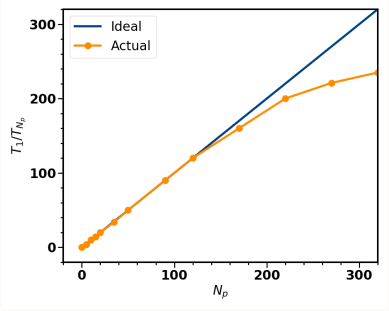
The fine-tuned model makes the requested change and also rewrites the value at 1.0:

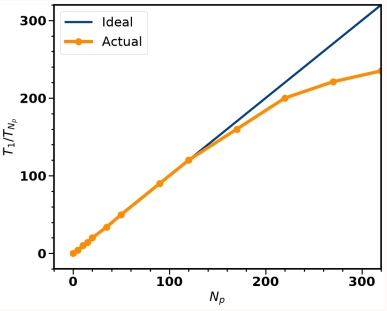
```
mrr = [0.25, 0.259, 0.257, 0.255, 0.256, 0.258]
  → [0.256, 0.259, 0.257, 0.255, 0.260, 0.258]
```

The value at 1.8 is the request, and the value at 1.0 is outside it, so the judge records one gratuitous change. The base model changes only the value at 1.8 and is an exact edit.

Partial completion: one requirement met, one left undone

ChartEdit task 585. *"Increase the width of the actual line to 6 and change its color to green to make the distinction between the actual and ideal lines more pronounced."* The instruction specifies two requirements: width 6 and color green.






```
satisfied 1/2
gratuitous 0
executes yes
```
Not exact.

source base model output

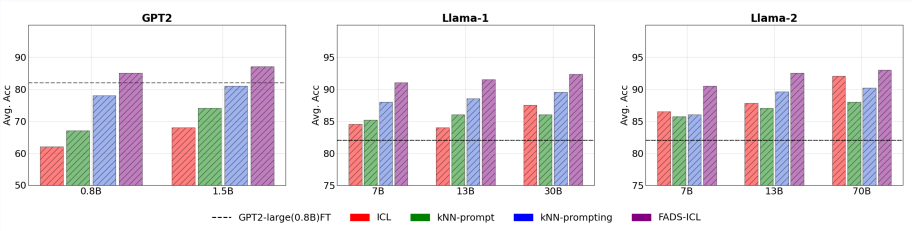
The base model changes `linewidth=4` to `linewidth=6` but keeps `color='#FF8C00'`, so the line is thicker and still orange. The judge marks the width requirement satisfied and the color requirement unmet. The fine-tuned model makes both changes and is an exact edit.

Coupled-update: a recolor whose legend must follow

ChartEdit task 451. *"Change the color of the kNN-prompting bars from royalblue to orange for better differentiation."*

The bars and the hand-built legend use separate color settings, so the legend does not update automatically:

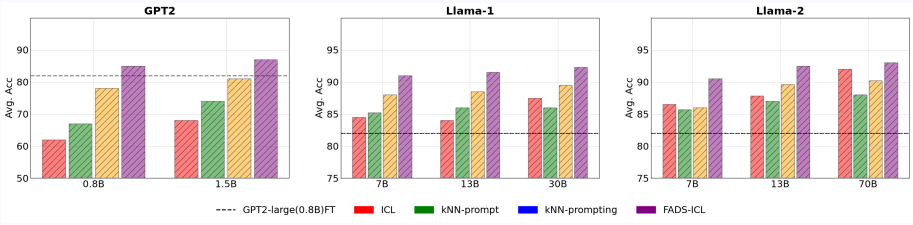
```
colors = ['red', 'green', 'royalblue', 'purple']
legend_elements = [..., Patch(color='blue', label='kNN-prompting'),
                   ...]
```



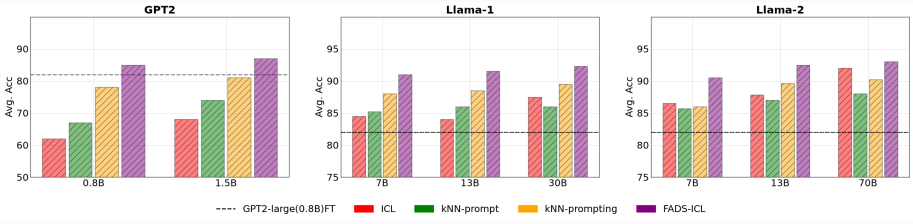
Source: blue bars and a blue legend patch.

R1 (explicit). The kNN-prompting bars are orange.
C1 (coupled). The legend patch labelled kNN-prompting is orange.

Changing `colors[2]` implements R1. C1 requires updating the separate `legend_elements[3]` patch, although the instruction does not mention the legend.



Missed coupled update (constructed illustration): orange bars, blue legend; R1 is met, C1 is unmet.



Completed coupled update (illustration): orange bars and legend; both R1 and C1 are met.

E Supplementary Evaluation Results

We present additional results for benchmark-facing metrics and our evaluation protocol.

E.1 Complete benchmark-facing transfer results

Table 10 extends Table 3 with ChartScore from ChartEdit and the native 8D F1, LLM-score, and LMM-score columns from Chart2Code. CodeScore improves in nearly every comparison, while ChartScore is mixed and ExecRate is the least stable. CodeScore also scores programs that do not execute, and the LLM and LMM columns use benchmark-specific scoring subsets, so none of these columns is a task-level success rate. Table 11 compares holistic edit verdicts with requirement completion. CR_{μ} is the pooled requirement recall defined in Section 4.1; $\mathrm{CR}_{M} = \frac{1}{N}\sum_{i=1}^{N}\mathrm{CR}_{i}$ is the mean per-task completion ratio over N tasks. Both completion measures improve in 19 of 20 comparisons after fine-tuning.

Table 10: Complete out-of-domain benchmark-facing transfer results.

Benchmark	Family	ExecRate	CodeScore	ChartScore	8D F1	LLM score	LMM score
ChartEdit	Phi-3.5-V	0.860 → 0.860	63.8 → **75.4**	91.6 → **92.4**	91.60 → **92.54**	90.91 → **91.69**	44.88 → **54.80**
	InternVL2.5	0.900 → 0.860	51.2 → **71.6**	90.2 → **92.0**	90.03 → **92.07**	90.53 → 90.29	41.82 → **51.60**
	Qwen3.5-4B	0.900 → **0.910**	83.1 → **86.7**	94.0 → **94.5**	93.69 → **94.13**	94.21 → 93.79	59.16 → **64.36**
	Qwen3.5-9B	0.940 → 0.920	83.8 → **89.8**	94.7 → **95.1**	93.81 → **94.40**	94.72 → 94.32	65.24 → **65.98**
	Granite	0.870 → 0.860	68.5 → **79.7**	90.7 → **93.6**	90.26 → **93.17**	90.09 → **92.29**	49.72 → **58.16**
Chart2Code-L2	Phi-3.5-V	0.400 → 0.334	27.6 → **38.0**	54.2 → **58.9**	23.99 → 21.17	63.77 → **67.18**	3.31 → **3.41**
	InternVL2.5	0.594 → 0.380	20.7 → **32.6**	53.7 → 52.0	31.46 → 22.17	61.84 → **65.72**	2.74 → **3.12**
	Qwen3.5-4B	0.414 → **0.444**	60.5 → **62.1**	67.6 → **69.0**	28.76 → **31.09**	77.60 → **78.33**	7.50 → **7.67**
	Qwen3.5-9B	0.567 → 0.540	66.7 → **73.4**	69.5 → **72.1**	39.50 → **39.80**	78.32 → **79.70**	7.92 → **9.01**
	Granite	0.596 → 0.373	29.7 → **43.1**	56.0 → **59.1**	33.78 → 23.91	65.23 → **70.78**	3.09 → **4.03**
ChartM^3	Phi-3.5-V	0.660 → 0.650	35.1 → **46.3**	89.9 → **90.3**	92.74 → 92.23	91.36 → **92.56**	35.45 → **43.43**
	InternVL2.5	0.690 → **0.740**	32.7 → **43.3**	91.3 → 89.6	92.50 → 92.25	92.32 → **93.40**	39.31 → **42.67**
	Qwen3.5-4B	0.660 → **0.700**	53.7 → **66.8**	92.4 → 91.8	94.54 → 93.09	94.49 → **94.98**	46.91 → **54.61**
	Qwen3.5-9B	0.730 → 0.720	61.2 → **74.8**	94.3 → 93.1	94.82 → 93.95	95.02 → **95.85**	56.03 → **57.44**
	Granite	0.700 → 0.630	43.9 → **48.4**	90.3 → 86.1	91.56 → 90.19	91.18 → **93.32**	40.33 → **45.17**
ChartSync	Phi-3.5-V	0.940 → 0.930	84.1 → **85.5**	97.9 → **98.0**	97.18 → 96.76	97.77 → **98.14**	77.57 → **81.52**
	InternVL2.5	0.960 → 0.930	79.8 → **88.2**	98.3 → 96.9	97.29 → 96.11	98.13 → **98.22**	74.08 → **86.06**
	Qwen3.5-4B	0.990 → 0.980	92.3 → **92.6**	98.9 → 98.8	97.82 → 97.61	99.24 → 97.33	87.59 → 87.42
	Qwen3.5-9B	0.990 → 0.980	95.7 → 94.8	99.5 → 99.3	97.85 → 97.81	99.45 → 99.26	92.17 → 91.06
	Granite	0.870 → **0.900**	79.7 → **83.4**	97.6 → 97.3	96.50 → **96.59**	92.00 → **97.66**	75.33 → **81.13**

Table 11: Complementary holistic outcomes.

Benchmark	Family	AppliedRate ↑	EditFidelity ↑	CR_μ ↑	CR_M ↑
ChartEdit	Phi-3.5-V	**.625** +.127	**.389** +.121	**.773** +.138	**.758** +.145
	InternVL2.5	**.573** +.224	**.345** +.143	**.707** +.234	**.703** +.241
	Qwen3.5-4B	**.711** +.024	**.470** +.052	**.867** +.047	**.871** +.049
	Qwen3.5-9B	**.743** +.059	**.498** +.022	**.895** +.068	**.899** +.073
	Granite	**.666** +.131	**.384** +.110	**.812** +.114	**.811** +.117
Chart2Code-L2	Phi-3.5-V	**.234** +.136	**.032** +.015	**.691** +.120	**.699** +.156
	InternVL2.5	**.160** +.098	**.024** +.006	**.624** +.199	**.630** +.196
	Qwen3.5-4B	**.469** +.054	**.131** +.029	**.844** +.013	**.847** +.004
	Qwen3.5-9B	**.506** +.104	**.170** +.037	**.896** +.053	**.894** +.048
	Granite	**.263** +.170	**.058** +.030	**.723** +.206	**.725** +.199
ChartM^3	Phi-3.5-V	**.243** +.047	**.105** +.031	**.390** +.097	**.427** +.120
	InternVL2.5	**.214** +.086	**.117** +.050	**.338** +.088	**.363** +.116
	Qwen3.5-4B	**.343** +.092	**.175** +.050	**.623** +.150	**.662** +.152
	Qwen3.5-9B	**.349** +.097	**.186** +.044	**.701** +.181	**.724** +.167
	Granite	**.240** +.011	**.111** +.012	**.409** +.041	**.457** +.068
ChartSync	Phi-3.5-V	**.835** +.058	**.690** +.033	**.819** +.056	**.847** +.043
	InternVL2.5	**.872** +.161	**.747** +.133	**.838** +.119	**.867** +.145
	Qwen3.5-4B	**.874** +.010	**.807** +.008	.886 -.002	**.891** +.004
	Qwen3.5-9B	.911 -.010	.840 -.021	**.927** +.011	.932 +.000
	Granite	**.828** +.024	**.680** +.093	**.819** +.036	**.825** +.019
improved in		*19/20*	*19/20*	*19/20*	*19/20*

E.2 PROTOCOL DIAGNOSTICS

Table 12 reports ChartSync's native text, geometry, and whole-output scores as complementary diagnostics. Figure 8 summarizes terminal Python exception counts in the diagnostic run. These coarse exception labels describe execution failures.

Table 12: ChartSync diagnostic metrics.

Metric	Phi-3.5-V	InternVL2.5	Qwen3.5-4B	Qwen3.5-9B	Granite
TESR	81.2 → **88.0**	74.3 → **90.7**	89.4 → **91.7**	94.7 → 94.1	83.7 → **87.9**
VLCS	64.8 → **78.5**	76.5 → **84.6**	90.0 → 87.6	87.3 → **92.3**	79.0 → **84.9**
BFS	95.4 → 93.6	96.2 → 94.7	98.1 → 97.4	98.5 → 98.3	91.1 → **94.3**
OCR F1	93.3 → 92.8	94.0 → 92.5	94.1 → 93.9	94.6 → 94.3	93.7 → 92.9
SSIM	88.4 → 87.8	88.4 → 88.3	88.8 → 88.8	88.9 → **89.0**	88.2 → 88.2

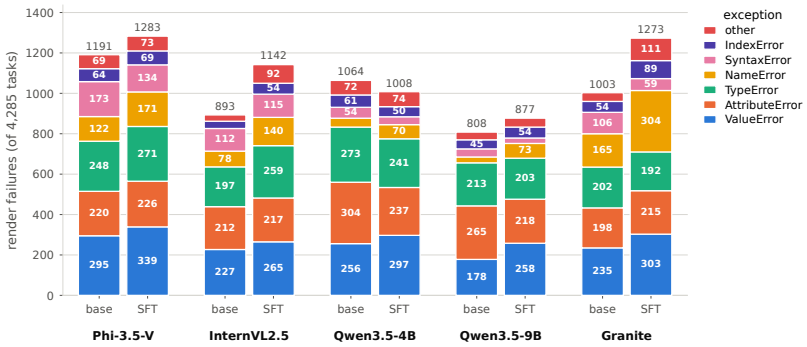


Figure 8: Terminal Python exception counts in the 4,285-task diagnostic run for all five models.

Table 13: Reference-program calibration. These calibrate the renderer–judge pipeline and are not model-specific capability ceilings.

Benchmark	ExecRate	CodeScore	ChartScore	AppliedRate	EditFidelity
ChartEdit	1.00	95.41	99.88	0.796	0.552
Chart2Code L2	0.97	98.04	95.92	0.757	0.455
ChartSync	0.99	99.79	98.58	0.956	0.901
ChartM^3	0.91	89.52	89.26	0.345	0.297

E.3 Reference-program calibration

Table 13 reports how released benchmark references score under our renderer and judge pipeline. For ChartM^3, the references obtain an AppliedRate of 0.345 despite an execution rate of 0.91 and a ChartScore of 89.26. AppliedRate checks whether the requested edit reaches its named target, whereas ChartScore assesses overall rendered-chart quality. These aggregate scores alone do not distinguish reference-edit errors from errors introduced by benchmark adaptation or automatic judging.

E.4 Human validation of exact-edit judgments

The results in Table 14 cover 200 paired external tasks: 69 from ChartEdit, 50 from ChartM^3, 64 from ChartSync, and 17 from Chart2Code-L2. Each task pairs a base output with its fine-tuned counterpart. Human judgments are made on blinded outputs using the labels described below. Agreement and gains in Table 14 use exact-edit verdicts. A positive human exact-edit verdict requires every listed atomic requirement to be satisfied and no gratuitous change to unrelated chart content. All outputs in this audit render successfully.

Table 14: Paired human–protocol exact-edit agreement and fine-tuning gains on rendered outputs. Agreement is measured per output; Δ_P and Δ_H are the gains under automatic and human judgments, respectively. Gains are absolute rate differences.

Benchmark	Pairs	Agreement	Δ_P	Δ_H
ChartEdit	69	91%	+0.232	+0.203
ChartM^3	50	91%	+0.180	+0.160
ChartSync	64	98%	+0.031	+0.078
Chart2Code-L2	17	91%	+0.059	+0.118
All	200	93%	+0.140	+0.145

Audit procedure.

- **Sampling.** The pairs come from a blinded annotation package of 370 tasks drawn with a fixed seed from two model families, InternVL2.5 and Qwen3.5-4B, chosen as the arms

with the largest and smallest automatic gains. Sampling is stratified by benchmark, and Chart2Code-L2 is further stratified into four bins of explicit-requirement count and oversampled, with per-item weights that restore population proportions. The 200 audited pairs are those in which both outputs render: all 96 such pairs from the first sampling round and 104 drawn at random. By family, ChartEdit contributes 33 InternVL2.5 and 36 Qwen3.5-4B pairs, ChartM^3 30 and 20, ChartSync 31 and 33, and Chart2Code-L2 10 and 7. Two tasks appear under both families, and no task appears twice within a family.

- **Annotators.** Two undergraduate computer science students with experience using Python plotting libraries conduct the annotation. Their labels are saved before any comparison.
- **Evidence and blinding.** Annotators see the instruction, the source program and its rendering, the fixed atomic-requirement list used by the automatic protocol, and two edited outputs, labelled A and B, each with its program and rendering. Which output is the fine-tuned one is randomized independently per item, and the mapping is kept off the annotation server. Model families appear only as pseudonyms, and judge verdicts and rationales are stored for later comparison but never shown to annotators.
- **Labels.** Annotators judge A and B separately against the same requirement list, labelling each requirement satisfied, violated, or ambiguous, and record whether the output makes any gratuitous change to unrelated content in the source chart. Necessary consequences of a requested change do not count as gratuitous. The human *exact-edit* verdict is positive only when every listed requirement is satisfied and no gratuitous change is present. The audit records requirement-level labels and a task-level gratuitous-change flag.

E.5 Effect of exactness filtering

Motivated by (Chen et al., 2024a), which explored the effects of data quantity and quality on training, Table 15 compares fine-tuning on the corpus before and after exactness filtering on all four external benchmarks. On ChartM^3, coupled-update recall improves by 3.0 and 6.9 percentage points for Phi-3.5-V and Qwen3.5-4B, respectively. On Chart2Code-L2, Qwen3.5-4B improves requirement recall and full completion by 2.5 and 3.3 percentage points, while exact-edit rate remains stable.

Table 15: Fine-tuning on the corpus before and after exactness filtering across all four external benchmarks.

Benchmark	Model	Training data	Requirement recall ↑	Full completion ↑	Gratuitous per task ↓	Coupled-update recall ↑	Exact-edit rate ↑
ChartEdit	Phi-3.5-V	Before filtering	.764	.670	.348	.527	.418
		Released	**.773**	**.674**	**.346**	**.540**	**.425**
	Qwen3.5-4B	Before filtering	**.873**	.804	**.140**	.666	.566
		Released	.867	.804	.165	**.688**	**.572**
Chart2Code-L2	Phi-3.5-V	Before filtering	.689	**.034**	**1.404**	.489	.001
		Released	**.691**	.032	1.433	**.493**	**.002**
	Qwen3.5-4B	Before filtering	.819	.138	.924	.699	**.045**
		Released	**.844**	**.171**	**.900**	**.714**	.044
ChartM^3	Phi-3.5-V	Before filtering	.382	**.325**	.941	.318	.142
		Released	**.390**	.321	**.926**	**.348**	**.152**
	Qwen3.5-4B	Before filtering	.620	.511	**.401**	.484	.304
		Released	**.623**	**.540**	.407	**.553**	**.331**
ChartSync	Phi-3.5-V	Before filtering	**.837**	**.841**	**.183**	.809	**.679**
		Released	.819	.832	.210	**.815**	.668
	Qwen3.5-4B	Before filtering	**.894**	**.885**	.082	**.839**	.793
		Released	.886	.879	**.062**	.825	**.800**

F Dataset Release and Usage

This section documents the collection-ledger fields and datasheet notes.

F.1 RECORD SCHEMA

The collection ledger stores source and edited programs, available renderings, and `evaluation.json`. Identifiers such as `task_id`, `pool_idx`, `source_chart_id`, `tmpl_id`, and `binding_id` link a candidate to its source and template; category, target object, chart type, library, and instruction describe the editing task. Timing and `attempt_log` retain generation, execution, validation, and repair history. The `preserved` field stores the first-pass judge's `no_additional_change` decision.

The mechanical signals `patch_nontrivial`, `api_touch`, `render_changed`, and `pixel_dist` provide diagnostic evidence. In particular, a nontrivial code diff does not establish that the requested chart property changed. First-pass flags and later requirement-level audits serve different roles; a positive first-pass flag alone does not certify final exactness.

F.2 DATASHEET NOTES

Collection. Source programs are drawn from ChartNet (Kondic et al., 2026); instructions and edited programs are generated by our pipeline. Coverage is non-uniform across the taxonomy.

Models. Editing and first-pass judging use `gemma-4-31B-it` served locally (vLLM, W4A16 quantized); the cross-audit uses `Qwen3.8-27B` with the same exact-edit protocol (Section D.3).

Compute. Per-sample cost is dominated by editor generation (mean 66 s) and judging (mean 17 s); rendering is under 1 s. The full collection is on the order of 1.8×10^5 editor calls and an equal number of judge calls.