VISAUDIT: EVALUATING MULTIMODAL AGENTS FOR VISUAL DIAGNOSIS AND REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Multimodal agents are increasingly used for data visualization tasks but remain limited in autonomous review. Unlike humans, they may fail to recognize when a visualization is incorrect, determine what to change, repair it without disrupting correct content, and verify whether the intervention succeeded. Existing benchmarks largely evaluate predefined individual capabilities such as chart generation, instruction-guided editing, or defect detection, and therefore do not capture this gap in autonomous review. We introduce VisAudit, a benchmark for evaluating visualization diagnosis, repair, and verification. Given a rendered chart and configurable auxiliary evidence, including its source data table, intended text summary, and visualization code, an agent iteratively diagnoses potential defects, modifies and executes visualization code, inspects execution and visual feedback, and determines when no further intervention is needed. VisAudit defines three tracks spanning diagnosed repair, autonomous repair, and open-world verification, and contains 1,900 flawed instances across 21 chart types and 10 flaw categories, together with 300 initially correct charts. We construct the benchmark through controlled perturbations of validated source visualizations, with systematic verification and human-aligned quality control to ensure that injected defects are well-defined and recoverable from the available evidence. Experiments with leading multimodal models reveal a substantial gap from reliable autonomous review: the strongest evaluated model fully recovers only 47.4% of flawed charts in the autonomous-repair setting. By exposing failures across diagnosis, intervention, preservation, and verification, VisAudit characterizes the capabilities needed to close the gap between current agents and human-like visualization review and provides actionable insights for designing more reliable autonomous multimodal systems.

1 INTRODUCTION

Multimodal agents are increasingly capable of generating, editing, and reasoning about data visualizations (Yang et al., 2025; Zhao et al., 2025; Tang et al., 2026; Rahman et al., 2026). However, autonomous visualization review requires more than executing a predefined task. When humans review a visualization, they can determine whether anything is wrong, identify the underlying problem, decide whether and how to intervene, and verify whether the result is correct. An autonomous agent should similarly be able to discover unspecified defects, determine appropriate interventions, preserve content that is already correct, and recognize when no further action is necessary. However, autonomous recognition and correction of failures remains challenging for current tool-using and multimodal agents (Liu et al., 2026; Dixit, 2026). This raises a fundamental question: *Can an agent, like a human reviewer, autonomously determine when intervention is needed, decide what to do, and verify that its intervention succeeded?*

Existing visualization benchmarks do not capture this form of autonomy. Recent benchmarks evaluate chart generation and reconstruction (Yang et al., 2025; Tang et al., 2026), instruction-guided editing (Zhao et al., 2025; Kapadnis et al., 2026; Zhang et al., 2026; Rahman et al., 2026), or misleading-visualization detection (Chen et al., 2025; Tonglet et al., 2026). Despite their different formulations, these benchmarks generally define the task to be performed in advance: the agent is asked to generate a chart, execute a specified modification, or assess a particular property of the visualization. In realistic review, however, the task itself may not be predetermined. Given a vi-

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.

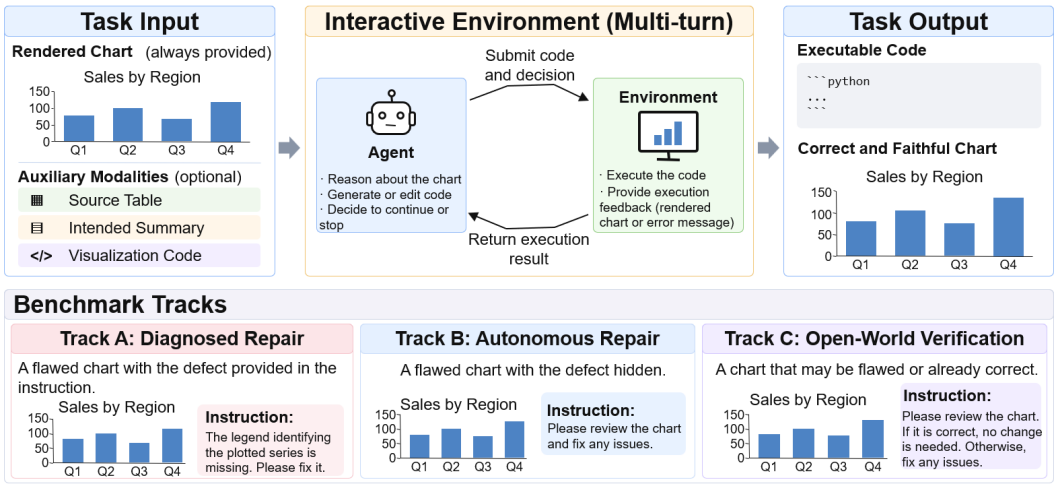


Figure 1: Overview of VisAudit. Given a chart and optional auxiliary modalities, an agent iteratively edits, executes, and verifies visualization code in a multi-turn environment. We evaluate three tracks: (A) diagnosed repair, (B) autonomous repair, and (C) open-world verification.

sualization, an agent must autonomously determine whether any problem exists, what intervention, if any, is warranted, and whether its actions have produced a satisfactory result. This shifts the requirement from completing a predefined visualization task to autonomously managing the full review process, including the possibility that no action should be taken.

To bridge this gap, we introduce **VisAudit**, a benchmark for evaluating autonomous visualization diagnosis, repair, and verification. As illustrated in Figure 1, each instance provides an initial chart together with configurable auxiliary modalities, including the source table, intended summary, and visualization code. Given this multimodal evidence, an agent determines whether and how the visualization should be modified and produces executable code that renders a correct and faithful chart. VisAudit places this task in an interactive environment, allowing the agent to propose and execute repairs, inspect the resulting visualization and execution feedback, and revise its solution when necessary. Crucially, the agent decides when to continue or stop, enabling it to test its diagnosis, observe the consequences of its interventions, and verify whether the resulting visualization is satisfactory.

We evaluate this workflow through three tracks with increasing levels of autonomy. *Track A: Diagnosed Repair* provides the defect and evaluates the corresponding repair; *Track B: Autonomous Repair* hides the defect, requiring the agent to diagnose and repair it from multimodal evidence; and *Track C: Open-World Verification* further introduces already-correct visualizations, requiring the agent to determine whether intervention is necessary. Together, these tracks isolate increasingly demanding capabilities from repair to autonomous diagnosis and open-world verification. By revealing where current agents succeed and fail across the diagnosis-action-verification loop, VisAudit provides actionable insights into the capabilities needed to develop more reliable and autonomous multimodal agents.

Our main contributions are as follows:

First, we develop a controlled and reusable visual perturbation framework that transforms correct source visualizations into flawed instances, yielding 1,900 flawed instances across 21 chart types and 10 flaw categories, together with 300 initially correct charts for open-world evaluation.

Second, we introduce three complementary evaluation tracks and a multidimensional evaluation framework that measures executability, recovery, preservation, and correct-chart recognition, capturing both repair effectiveness and the ability to avoid unnecessary interventions.

Third, we extensively evaluate leading multimodal models and conduct modality and interaction ablations, revealing substantial limitations in autonomous visualization review: agents struggle to diagnose unspecified defects, execute some repairs even when the defect is given, completely resolve multiple simultaneous flaws, and avoid unnecessary modifications to already-correct charts.

2 Related Work

Table 1: Comparison with representative visualization generation, editing, refinement, and misleading-visualization benchmarks. **Conserve** denotes explicitly preserving already-correct contents during repair; **Verify** denotes using the consequence of the model's own repair (e.g., execution and re-rendering) to assess whether it has succeeded; **Diagnose** denotes discovering what is wrong without being given the desired correction; **No-op** denotes an explicit setting in which the input may already be correct, and the model should refrain from editing; and **Self-stop** denotes allowing the agent to determine when no further intervention is required.

Work	Edit / Repair	Conserve	Verify	Diagnose	No-op	Self-stop
ChartMimic (Yang et al., 2025)	✗	✗	✗	✗	✗	✗
Chart2Code (Tang et al., 2026)	✓	✗	✗	✗	✗	✗
ChartEdit (Zhao et al., 2025)	✓	✓	✗	✗	✗	✗
ChartEditBench (Kapadnis et al., 2026)	✓	✓	✗	✗	✗	✗
RealChart2Code (Zhang et al., 2026)	✓	✓	✗	✗	✗	✗
VisEditBench (Rahman et al., 2026)	✓	✗	✓	✗	✗	✗
MisViz (Tonglet et al., 2026)	✗	✗	✗	✓	✗	✗
MisVisFix (Das & Mueller, 2025)	✓	✗	✗	✓	✗	✗
Ours	✓	✓	✓	✓	✓	✓

Chart generation and instruction-driven editing. Prior benchmarks study chart generation, reconstruction, and instruction-guided editing. ChartMimic (Yang et al., 2025) and Chart2Code (Tang et al., 2026) evaluate chart-to-code generation and reconstruction, while ChartEdit (Zhao et al., 2025), ChartE^3 (Li et al., 2026), and ChartEditVista (Chen et al., 2026) focus on modifying visualizations under explicit instructions. More recent benchmarks extend editing to multi-turn or iterative settings: ChartEditBench (Kapadnis et al., 2026) studies successive user-guided modifications, RealChart2Code (Zhang et al., 2026) evaluates instructed refinement while preserving unaffected content, and VisEditBench (Rahman et al., 2026) introduces iterative execution and render-grounded refinement. Across these settings, however, the target generation or modification is specified externally rather than autonomously determined by the agent.

Misleading visualization detection and correction. Complementary studies examine whether models can recognize deceptive or misleading chart designs. Misleading ChartQA (Chen et al., 2025) evaluates reasoning over charts containing a taxonomy of misleading patterns, while Misviz (Tonglet et al., 2026) focuses on detecting misleading visualizations and identifying violated design rules. MisVisFix (Das & Mueller, 2025) goes beyond detection by providing an interactive system for detecting, explaining, and correcting misleading visualizations.

Table 1 summarizes the distinction between these settings and ours. Existing benchmarks generally isolate individual capabilities. Generation benchmarks reconstruct visualizations, editing benchmarks provide the desired modification, and detection benchmarks identify predefined defects. Some recent works introduce preservation or iterative verification, but these capabilities remain within a specified editing objective. In contrast, VisAudit evaluates the complete autonomous loop. The agent must diagnose whether something is wrong, determine what is wrong, repair only the necessary components while preserving correct content, verify the consequences of its own intervention, and determine when no further action is required. The inclusion of initially correct charts further tests whether the agent can recognize when intervention is unnecessary rather than assuming that every input requires repair.

3 VisAudit Benchmark

VisAudit evaluates the ability of multimodal agents to autonomously review and repair data visualizations. Given a rendered chart together with an instruction and a configurable set of auxiliary modalities, an agent must determine whether the visualization contains defects and, when necessary, produce visualization code that generates a corrected chart. Unlike instruction-following chart editing, where the desired modification is explicitly specified by a user, our benchmark emphasizes *autonomous review*: depending on the benchmark track, the agent may need to autonomously iden-

tify unknown defects and determine appropriate repairs from multimodal evidence (including source data table, intended text summary, and visualization code), preserve aspects of the visualization that are already correct, and decide when no further modification is necessary.

Task Formulation. Each benchmark instance contains an initial chart V_0 and a track-dependent instruction I. The initial chart V_0 is the artifact under review and may contain one or more visualization defects. In addition, the agent may receive a configurable collection of auxiliary modalities $\mathcal{M} \subseteq \{D, S, C_0\}$, where D denotes the source data table, S denotes a natural-language summary of the intended visualization, and C_0 denotes the visualization code used to generate V_0. While V_0 may be flawed, D and S provide correct reference information about the underlying data and intended visualization, allowing the agent to identify discrepancies between the observed chart and its intended content. The initial chart is always provided, whereas the availability of the auxiliary modalities depends on the experimental configuration.

The agent is tasked with producing visualization code C^* whose rendered output V^* corrects the initial chart V_0. To do so, the agent must use the auxiliary modalities to determine whether V_0 contains a defect and, when necessary, infer the appropriate correction. When C_0 is provided, the agent can inspect and modify the implementation that produced V_0; otherwise, it must generate visualization code based on V_0 and the available reference modalities. A successful output should execute correctly, faithfully represent the underlying data and intended message, repair any defects present in V_0, and preserve aspects that are already correct.

Interaction. The agent operates iteratively in an execution-grounded environment: it observes the initial chart, instruction, and available auxiliary modalities; creates or modifies visualization code; executes the code; and inspects the resulting visualization and execution feedback. The agent may revise its solution over multiple turns and autonomously determines when no further intervention is necessary; evaluator outcomes are not exposed during interaction. Thus, the task jointly evaluates what intervention to perform and when to stop. Full details of the interactive environment, tool interface, and execution procedure are in Appendix B.

Benchmark Tracks. We define three complementary tracks that progressively increase the amount of diagnosis and verification required from the agent.

Track A: Diagnosed Repair. Track A evaluates visualization repair when the defect is already known. Every instance in this track contains a flawed visualization, and the instruction explicitly provides a textual description of the injected flaw. For example, the agent may be informed that *"the legend identifying the plotted series is missing."* The description identifies the type of defect present in the visualization, and the agent must determine how to correct it using the available modalities and produce a valid repaired visualization. This track therefore focuses on the agent's repair capability without requiring it to first discover the defect.

Track B: Autonomous Repair. Track B evaluates autonomous visualization diagnosis and repair. As in Track A, every instance contains a flawed visualization. However, the instruction does not reveal the injected defects. The agent must instead identify the defects autonomously by reasoning over the chart and the available auxiliary modalities, and then generate or modify visualization code to correct them. The distinction between Tracks A and B isolates the difficulty of autonomous diagnosis. Track A provides the defect description and therefore serves as an approximate upper-bound setting in which the diagnosis step is largely removed. Track B requires both diagnosis and repair and constitutes the primary setting of our benchmark. Comparing performance between the two tracks allows us to quantify the performance degradation caused by requiring the agent to discover visualization defects autonomously.

Track C: Open-World Verification. Tracks A and B assume that every input visualization contains at least one defect. In realistic visualization-review settings, however, an agent cannot assume that every chart requires modification. Track C therefore includes both flawed and already-correct visualizations, requiring the agent to first determine whether intervention is necessary. For a flawed visualization, the agent must autonomously diagnose and repair the defects. For an already-correct visualization, the desired behavior is to recognize that the chart is correct and make no unnecessary modifications. Track C therefore extends autonomous repair to an open-world setting by additionally evaluating the agent's ability to verify correctness and avoid false-positive repairs.

Across the three tracks, success requires producing executable visualization code that corrects genuine defects while preserving unaffected content; Track C additionally requires recognizing when no intervention is necessary. We formalize these criteria and metrics in Section 4.

Benchmark composition. We construct VisAudit from aligned source visualizations by introducing controlled defects into the plotting code while retaining the original source table and summary as reference evidence. The resulting benchmark contains 1,900 flawed instances derived from 854 source charts, spanning 21 chart types and 10 flaw categories. Table 2 summarizes the perturbation taxonomy. Each category introduces a targeted discrepancy between the rendered visualization and its reference evidence while otherwise preserving the source instance. Full details of the source data, perturbation procedure, and quality-control pipeline are provided in Appendix A.

Table 2: The ten flaw categories used to construct single-, double-, and triple-flaw instances.

Category	Injected discrepancy
Wrong title	The title contradicts the source table or summary.
Wrong axis label	An axis names a different quantity from the one plotted.
Missing legend	The mapping identifying plotted series is absent.
Swapped legend labels	Two legend labels identify each other's series.
Value perturbation	A plotted value disagrees with its source value.
Swapped series	Two series display each other's data.
Dropped series	A series present in the source evidence is omitted.
Wrong chart type	The plotting type contradicts the intended representation.
Axis truncation	An axis limit clips at least one reference data value.
Color contradiction	A visual element's color contradicts the summary.

We include instances with one, two, or three flaws to evaluate increasingly complex repair scenarios. As summarized in Table 3, the evaluation set contains 400 single-flaw, 900 double-flaw, and 600 triple-flaw instances across 106 flaw scenarios. Tracks A and B use the same 1,900 flawed tasks and differ only in whether the instruction reveals the injected flaws. Track C additionally includes 300 initially correct charts with source identities disjoint from the flawed set, yielding 2,200 instances in total.

Table 3: Evaluation-set composition. Track C contains the same flawed instances as Tracks A and B, plus 300 initially correct instances.

Task group	Scenarios	Instances per scenario	Instances
Single flaw	10	40	400
Two flaws	36	25	900
Three flaws	60	10	600
Flawed total	106	–	1,900
Additional correct charts	–	–	300

4 Evaluation Pipeline

We evaluate whether the submitted visualization code executes successfully, recovers properties altered by the injected flaws, and preserves properties that were unaffected. Evaluation is based on structured figure properties, including titles, axis labels, series, plotted values, legends, and axis configurations. Recovery and preservation capture complementary aspects of repair: an agent may successfully correct an injected defect while simultaneously introducing unintended changes elsewhere in the visualization.

Recovery. Let g, b, and s denote the flattened property maps extracted from the ground-truth chart, initial flawed chart, and submitted chart, respectively. For a figure property p, let g_p, b_p, and s_p denote its value in the corresponding property map, with a distinguished missing value used when p is absent. Let $\sim$ denote the property comparator: numerical values x and y match when $|x - y| \leq 10^{-9} \max(1, |x|, |y|)$, while text is compared after trimming and collapsing whitespace and ignoring

case, and an absent property is treated as a distinct missing value. The operator $\not\sim$ is used to denote property mismatch. We define the perturbation footprint as $F \triangleq \{p \in \mathrm{dom}(g) \cup \mathrm{dom}(b) : g_p \not\sim b_p\}$, which contains the properties changed by the benchmark perturbation and is independent of the submitted chart. For $|F| > 0$, graded and binary recovery are

$$\text{Rec-S} \triangleq \frac{1}{|F|} \sum_{p \in F} \mathbf{1}[s_p \sim g_p], \qquad \text{Rec-B} \triangleq \mathbf{1}\left[\text{Rec-S} = 1\right].$$

Thus, recovery is measured at the property level rather than the flaw-category level: a single injected flaw may alter multiple figure properties. For perturbed axis limits, a dedicated comparator checks whether the axis limits extracted from the submitted charts cover the required data extent or summary-specified range, rather than requiring them to exactly match the autoscaling padding of the ground-truth chart.

Preservation. Let $K \triangleq \big(\mathrm{dom}(g) \cup \mathrm{dom}(b)\big) \setminus F$ denote the properties unaffected by the benchmark perturbation, and let $A \triangleq \mathrm{dom}(s) \setminus \big(\mathrm{dom}(g) \cup \mathrm{dom}(b)\big)$ denote properties newly introduced by the submitted chart. We define graded and binary preservation as

$$\text{Pre-S} \triangleq \frac{\sum_{p \in K} \mathbf{1}[s_p \sim g_p]}{|K| + |A|}, \qquad \text{Pre-B} \triangleq \mathbf{1}[\text{Pre-S} = 1].$$

Thus, preservation rewards retaining properties unaffected by the injected flaw while penalizing unsupported properties introduced by the submission. For perturbations that modify plotted data, we exclude properties that change automatically as a consequence of the data modification (e.g., auto-scaled axis limits) from the preservation scope.

Executability. Executability records whether the submitted program runs successfully and produces a figure that can be retrieved by the evaluation pipeline.

Correct-Chart Recognition Rate. Correct-Chart Recognition Rate (CCR) measures whether the agent recognizes an initially correct chart and terminates without modifying its visualization code. We assess this behavior on the 300 initially correct Track C instances. CCR complements recovery and preservation by measuring whether an agent can determine that intervention is unnecessary.

Efficiency, coverage, and aggregation. We report executed tool steps and model-response counts separately, along with available token use and execution outcomes. Analyses by flaw count use the same definitions as the overall analysis. Completed unsuccessful repairs remain in the outcome statistics; technical failures and unstarted tasks are reported as coverage gaps. Each metric is averaged over its defined values with its own denominator. We report binary and graded recovery and preservation separately.

5 Experiments

Our experiments evaluate current multimodal agents along three dimensions of autonomous visualization review. We first report overall performance across the three benchmark tracks, examining how repair changes when defect information is provided, must be inferred autonomously, or when the input may already be correct. We then analyze performance by flaw category and number of simultaneous flaws to characterize where agents fail. Finally, we conduct ablations on the available input modalities and interaction budget to study the roles of multimodal evidence and iterative environment interaction.

Models and benchmark tracks. We use the following models to evaluate the benchmark: gemma-4-31B-it, Qwen3-VL-235B-A22B-Instruct, Ministral-3-8B-Instruct-2512, Mistral-Large-3-675B-Instruct-2512-NVFP4, claude-haiku-4-5-20251001, deepseek-v4.1-flash, and gemini-3.8-flash. We use shortened model names in tables for readability. Each model is evaluated on the same 1,900 flawed instances in Tracks A and B and on those flawed instances plus 300 initially correct instances in Track C.

The default input condition provides the chart, plotting code, source table, and summary. Track A provides a textual description of the injected flaw. Track B asks the agent to diagnose and repair the chart without describing the flaws. Track C additionally requires the agent to determine whether a repair is necessary.

Interaction and inference. Agents inspect files, edit code, execute it, and view rendered charts through the shared interactive environment. In the default setting, we impose no fixed limit on the number of interaction turns. Instead, the agent autonomously determines when the review is complete and terminates the interaction by calling `finish`. This self-terminating setting evaluates not only the agent's ability to diagnose and repair a visualization, but also its ability to determine when no further intervention is necessary. During inference, we use a temperature of 0 and a maximum of 8,000 output tokens per model response. Completed trajectories record the model identity, tool actions, termination reason, and available token usage.

Evaluation. We evaluate agents using executability (Exec.), binary and graded recovery (Rec-B/Rec-S), and binary and graded preservation (Pre-B/Pre-S), as defined in Section 4. Recovery and preservation metrics are reported as percentages. We additionally report interaction length as the number of executed tool actions (Turns) and inference cost as the mean input and output tokens per episode (In/Out). Token counts are summed over all model responses within an episode; input counts include cached tokens and may not be directly comparable across models due to tokenizer differences. For Track C, we additionally report correct-chart recognition (CCR) on the 300 initially correct charts. Results are further analyzed by flaw count and flaw category.

Autonomous repair remains challenging. Table 4 shows that high executability does not necessarily translate into successful repair. Most models produce executable outputs in over 96% of instances, yet binary recovery remains below 50% for all models. The gap between binary and graded recovery is also substantial, indicating that agents often correct some, but not all, properties affected by a flaw. Recovery and preservation further reveal different model behaviors. Gemini and DeepSeek achieve the highest binary recovery (47.4% and 45.6%) while also attaining strong preservation, whereas Gemma achieves a relatively high recovery score but substantially lower preservation. This suggests that successful autonomous repair requires not only identifying and correcting the target flaw, but also avoiding unintended modifications to already-correct chart properties. Interaction behavior also varies considerably: on average, DeepSeek and Gemini terminate after roughly 6–7 tool actions, while Gemma uses more than 13.

Table 4: Track B: Autonomous Repair on 1,900 flawed instances.

Model	Exec.	Rec-B	Rec-S	Pre-B	Pre-S	Turns	In	Out
Gemma	99.8	39.7	70.1	50.3	85.5	13.2	56.3	2.9
Mistral	98.4	36.8	66.0	59.2	88.2	8.5	29.0	2.3
Claude	99.9	30.8	62.5	69.0	91.7	9.0	48.5	2.6
Qwen	99.5	27.7	58.6	66.7	91.1	9.2	67.0	2.7
Ministral	87.7	25.1	52.2	37.8	70.1	10.5	73.1	3.0
DeepSeek	96.8	45.6	68.8	74.1	94.0	6.1	21.1	8.4
Gemini	90.4	47.4	65.2	79.0	96.2	6.9	28.6	7.6

Effect of diagnostic supervision and open-world verification. Having established autonomous repair performance in Track B, we next examine how performance changes when the flaw diagnosis is provided (Track A) or when the agent must operate without knowing whether the input chart is flawed (Track C). Table 5 reports performance on the same 1,900 flawed instances. Providing the diagnosis in Track A substantially improves recovery across models, indicating that identifying the underlying defect is a major bottleneck in autonomous repair. For example, Qwen reaches 60.2% binary recovery in Track A, compared with 27.7% in Track B, while Claude reaches 63.1% compared with 30.8%. Preservation generally improves as well, showing that an explicit diagnosis helps agents target their interventions and avoid unintended changes. On the flawed subset of Track C, recovery is generally comparable to Track B, while preservation also decreases for several models. The additional uncertainty over whether intervention is necessary therefore affects not only the decision to act but also the subsequent repair behavior. Notably, turn counts change relatively

Table 5: Repair performance on the 1,900 flawed instances in Tracks A and C.

Model	Track	Rec-B	Rec-S	Pre-B	Pre-S	Turns	In/Out
Gemma	A	57.1	82.2	71.6	92.6	12.2	50.0/2.8
	C	35.2	66.8	37.5	78.5	13.0	54.1/2.6
Qwen	A	60.2	83.8	84.5	96.4	8.1	43.2/2.8
	C	26.4	55.9	66.1	89.9	9.0	65.2/2.7
Ministral	A	35.1	57.2	55.9	79.2	9.2	45.0/2.9
	C	26.4	51.9	45.8	75.3	9.9	61.7/3.1
Claude	A	63.1	86.1	87.5	97.6	8.6	43.5/2.4
	C	32.9	65.1	58.3	85.5	8.7	40.4/2.2
Mistral	A	56.7	80.1	77.8	95.2	8.1	26.6/2.2
	C	41.1	69.8	52.3	85.5	8.4	26.9/2.1
DeepSeek	A	63.1	80.9	91.4	98.1	5.4	17.7/5.4
	C	47.0	72.1	73.1	92.6	6.2	20.9/7.5
Gemini	A	68.0	82.1	97.3	99.3	6.4	26.1/6.3
	C	48.2	68.2	75.2	95.0	6.9	28.7/7.1

little across tracks for most models, suggesting that the performance differences arise primarily from diagnosis and decision making rather than simply from longer interaction.

Behavior on initially correct charts. The preceding comparison considers only flawed instances in Track C. We further evaluate whether agents can recognize when no intervention is necessary using the 300 initially correct charts. As shown in Table 6, CCR varies substantially across models, ranging from 32.7% to 72.0%. Even when the input chart is already correct, agents frequently modify it rather than terminate without intervention. Moreover, binary preservation remains below 90% for all models, indicating that unnecessary interventions can alter originally correct chart properties. These findings highlight a distinct challenge in open-world visualization review: beyond diagnosing and repairing defects, agents must also determine when to leave a visualization unchanged.

Table 6: Behavior on the 300 initially correct instances in Track C.

Model	Exec.	CCR	Pre-B	Pre-S	Turns	In/Out
Gemma	99.7	34.3	50.7	87.4	12.1	45.6/2.5
Qwen	100.0	72.0	88.0	96.4	7.5	42.7/1.9
Ministral	94.7	33.0	64.7	83.9	9.9	54.8/2.5
Claude	100.0	61.0	70.0	92.7	9.4	52.0/2.8
Mistral	99.0	32.7	59.7	90.5	7.8	20.4/1.8
DeepSeek	94.0	63.3	81.3	95.9	6.0	19.5/9.3
Gemini	99.0	64.0	79.3	95.8	6.9	28.8/6.4

Table 7: Full recovery (Rec-B) by flaw category across Tracks A/B/C. Each category contains 40 single-flaw instances.

Flaw	Gemma	Qwen	Ministral	Claude	DeepSeek	Gemini	Mistral
Axis truncation	95.0/85.0/85.0	77.5/37.5/35.0	45.0/47.5/47.5	80.0/32.5/32.5	92.5/70.0/65.0	92.5/50.0/60.0	40.0/67.5/72.5
Color contradiction	62.5/52.5/37.5	62.5/30.0/17.5	22.5/12.5/10.0	72.5/57.5/45.0	65.0/65.0/62.5	72.5/52.5/57.5	47.5/25.0/37.5
Dropped series	82.5/47.5/35.0	87.5/65.0/70.0	40.0/27.5/32.5	95.0/75.0/65.0	87.5/60.0/65.0	87.5/77.5/75.0	65.0/30.0/32.5
Missing legend	75.0/30.0/30.0	75.0/15.0/20.0	40.0/30.0/30.0	77.5/27.5/40.0	77.5/55.0/60.0	75.0/45.0/47.5	52.5/25.0/45.0
Swapped legend labels	70.0/50.0/42.5	72.5/65.0/55.0	50.0/32.5/32.5	82.5/62.5/55.0	77.5/55.0/67.5	77.5/60.0/62.5	57.5/65.0/65.0
Swapped series	72.5/65.0/55.0	80.0/67.5/67.5	52.5/50.0/45.0	85.0/67.5/65.0	92.5/65.0/65.0	95.0/75.0/72.5	72.5/65.0/60.0
Value perturbation	70.0/47.5/47.5	67.5/65.0/52.5	57.5/37.5/37.5	75.0/57.5/52.5	75.0/60.0/55.0	70.0/55.0/47.5	62.5/47.5/47.5
Wrong axis label	75.0/70.0/70.0	77.5/67.5/55.0	52.5/52.5/50.0	72.5/65.0/67.5	70.0/70.0/75.0	70.0/70.0/72.5	45.0/62.5/62.5
Wrong chart type	32.5/30.0/25.0	42.5/25.0/25.0	22.5/25.0/22.5	45.0/32.5/35.0	32.5/35.0/32.5	52.5/40.0/40.0	35.0/30.0/40.0
Wrong title	100.0/92.5/95.0	100.0/92.5/92.5	77.5/80.0/85.0	97.5/97.5/97.5	90.0/90.0/95.0	92.5/92.5/85.0	77.5/95.0/100.0

Difficulty varies substantially across flaw categories. Table 7 breaks down full recovery by flaw type across all three tracks. Wrong titles are consistently among the easiest flaws to repair, whereas wrong chart types remain challenging even in Track A, where the flaw is explicitly provided. This indicates that some failures arise not from identifying the defect, but from successfully executing the required repair. Comparing Tracks A and B further reveals where autonomous diagnosis becomes a bottleneck: providing the flaw yields particularly large gains for categories such as missing legends, dropped series, and color contradictions for many models, while the gap is generally smaller for readily identifiable flaws such as wrong titles and axis labels. Performance in Tracks B and C is often closer. Although Track C adds uncertainty over whether intervention is necessary, its different open-world instruction can change agent behavior, so recovery on the flawed subset is not necessarily lower than Track B for every model–flaw combination. Together, these results expose distinct limitations at different stages of autonomous review: carrying out a known repair, identifying an unspecified defect, and deciding how to act when the need for intervention is itself uncertain.

Table 8: Recovery by number of flaws across Tracks A, B, and C. Each entry reports single/double/triple-flaw performance.

	Track A		Track B		Track C	
Model	Rec-B	Rec-S	Rec-B	Rec-S	Rec-B	Rec-S
Gemma	73.5/57.6/45.5	82.5/82.1/82.1	57.0/38.3/30.2	68.9/69.4/72.0	52.2/33.8/26.0	65.5/66.2/68.4
Qwen	74.2/63.1/46.3	82.6/84.3/83.7	53.0/23.0/17.8	62.1/57.3/58.1	49.0/22.2/17.5	56.9/55.0/56.6
Ministral	46.0/35.0/28.0	53.8/57.1/59.5	39.5/23.6/17.8	50.8/51.3/54.6	39.2/24.8/20.2	51.0/50.8/54.1
Claude	78.2/63.4/52.3	85.2/85.9/87.0	57.5/25.9/20.3	66.5/60.5/62.8	55.5/28.9/23.8	66.1/62.9/67.8
Mistral	55.5/61.6/50.2	64.1/84.2/84.7	51.3/35.6/29.0	62.4/65.4/69.1	56.2/40.3/32.2	67.2/69.1/72.7
DeepSeek	76.0/64.6/52.2	82.0/83.0/77.2	63.0/46.2/33.2	70.0/68.9/68.0	64.3/46.4/36.2	72.9/71.8/71.9
Gemini	79.0/69.2/58.8	84.1/82.6/80.2	60.5/46.8/39.7	67.0/63.9/65.9	62.0/47.7/39.8	69.8/67.4/68.3

Complete recovery degrades as flaws accumulate. Table 8 examines recovery as the number of flaws increases across all three tracks. Binary recovery generally decreases from single- to triple-flaw instances, while graded recovery remains substantially more stable. The decline in Rec-B is particularly pronounced in Tracks B and C. For example, Qwen drops from 53.0% to 17.8% in Track B and from 49.0% to 17.5% in Track C, compared with 74.2% to 46.3% in Track A. Similar gaps for several models suggest that identifying multiple simultaneous defects is an important bottleneck when the flaw information is not provided. In contrast, Rec-S changes much less with flaw count and can even increase. This is because Rec-S measures the fraction of affected properties recovered: adding flaws enlarges the recovery target but does not require every affected property to be repaired, and differences in flaw composition can make the average recovery fraction non-monotonic. The growing gap between Rec-B and Rec-S therefore indicates that agents often recover a substantial fraction of affected properties in multi-flaw instances while failing to recover all of them. Thus, increasing the number of flaws primarily reduces the likelihood of complete recovery rather than eliminating meaningful partial progress.

Different modalities play complementary roles in repair. Table 9 ablates the auxiliary inputs in Track B. Removing the source table has relatively little effect on recovery, whereas removing either the code or summary substantially reduces recovery. Without the summary, Rec-B falls to 13.1% for Gemma and 9.4% for Qwen, suggesting that the intended semantic content provides important evidence for identifying what should be corrected. Removing the code reduces recovery and collapses binary preservation to below 1%, showing its importance for targeted edits. The higher preservation observed when the table or summary is omitted should not be interpreted as improved repair: it coincides with lower recovery and is consistent with less effective interventions that leave more unaffected properties unchanged. Overall, the summary primarily helps determine *what* to change, while the original code enables controlled execution of those changes.

Table 9: Track-B modality ablation on 1,900 instances.

Model	Omitted	Rec-B	Rec-S	Pre-B	Pre-S	Turns
Gemma	None	39.7	70.1	50.3	85.5	13.2
	Code	9.4	38.9	0.9	36.7	12.5
	Table	38.8	69.7	57.5	89.2	12.0
	Summary	13.1	39.5	61.1	91.5	13.5
Qwen	None	27.7	58.6	66.7	91.1	9.2
	Code	10.2	41.8	0.8	33.8	9.3
	Table	24.5	55.9	77.7	94.2	7.8
	Summary	9.4	34.9	64.7	91.2	7.9

Interaction exhibits diminishing returns beyond a sufficient budget. The experimental results above use the default setting in which the agent determines when to stop, without a fixed interaction limit. Table 10 evaluates sensitivity to the available interaction budget. Very small budgets can prevent agents from completing the repair workflow: neither model recovers any instance within three turns, and Gemma still achieves no recovery with five. Once a sufficient budget is available, however, additional interaction yields limited gains. Gemma reaches 39.3% Rec-B with 10 turns, compared with 39.7% under unrestricted interaction, while Qwen reaches comparable performance with five turns. Preservation also tends to decrease as the budget grows, indicating that additional interaction can introduce unnecessary changes without improving recovery. Overall, interaction is necessary to complete the repair process, but allowing more turns beyond a sufficient budget provides little benefit and can increase unintended modifications.

Table 10: Track-B ablation over interaction budgets.

Model	Budget	Rec-B	Rec-S	Pre-B	Pre-S
Gemma	3	0.0	0.0	100.0	100.0
	5	0.0	0.0	100.0	100.0
	10	39.3	67.6	56.9	88.0
	Unrestricted	39.7	70.1	50.3	85.5
Qwen	3	0.0	0.0	100.0	100.0
	5	28.5	59.0	62.5	86.9
	10	27.4	58.0	65.6	90.1
	Unrestricted	27.7	58.6	66.7	91.1

6 Conclusions

We introduced VisAudit, a benchmark for evaluating autonomous visualization review through diagnosis, repair, and verification. Across three tracks spanning diagnosed repair, autonomous repair, and open-world verification, our experiments show that current multimodal agents remain far from being capable of reliable autonomous review. Further analyses reveal substantial variation across flaw types and counts. VisAudit provides a foundation for developing reliable multimodal agents that can autonomously determine when to act, what to change, and when their intervention succeeds.

REFERENCES

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 *AAAI Conference on Artificial Intelligence*, volume 40, pp. 20199–20207, 2026.

Zixin Chen, Sicheng Song, Kashun Shum, Yanna Lin, Rui Sheng, Weiqi Wang, and Huamin Qu. Unmasking deceptive visuals: Benchmarking multimodal large language models on misleading chart question answering. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 13767–13800, 2025.

Amit Kumar Das and Klaus Mueller. Misvisfix: An interactive dashboard for detecting, explaining, and correcting misleading visualizations using large language models. *IEEE Transactions on Visualization and Computer Graphics*, 2025.

Aradhya Dixit. Evaluating self-correcting vision agents through quantitative and qualitative metrics. *arXiv preprint arXiv:2601.11637*, 2026.

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. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 15922–15932, 2026.

Shuo Li, Jiajun Sun, Zhekai Wang, Xiaoran Fan, Hui Li, Dingwen Yang, Zhiheng Xi, Yijun Wang, Zifei Shan, Tao Gui, Qi Zhang, and Xuanjing Huang. Charte^3: A comprehensive benchmark for end-to-end chart editing. In *International Conference on Machine Learning*, 2026.

Zheyuan Liu, Liqiang Xiao, Yang Li, Hyokun Yun, Lihong Li, Chao Zhang, and Meng Jiang. Do LLMs catch their own mistakes? A comprehensive benchmark for reflective tool use llms. In *Findings of the Association for Computational Linguistics*, pp. 1748–1773, 2026.

Mizanur Rahman, Arshia Azimlu, Shadikur Rahman, Md Tahmid Rahman Laskar, Amran Bhuiyan, Shafiq Joty, and Enamul Hoque Prince. Viseditbench: Can vision-language models edit visualization code from multimodal feedback? *arXiv preprint arXiv:2608.10408*, 2026.

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 *Annual Meeting of the Association for Computational Linguistics*, pp. 13467–13566, 2026.

Jonathan Tonglet, Jan Zimny, Tinne Tuytelaars, and Iryna Gurevych. Is this chart lying to me? Automating the detection of misleading visualizations. In *Annual Meeting of the Association for Computational Linguistics*, pp. 8823–8844, 2026.

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, 2025.

Jiajun Zhang, Yuying Li, Zhixun Li, Xingyu Guo, Jingzhuo Wu, Leqi Zheng, Yiran Yang, Jianke Zhang, Qingbin Li, Shannan Yan, et al. RealChart2Code: Bridging the gap in real-world chart-to-code generation via multi-task evaluation. In *Annual Meeting of the Association for Computational Linguistics*, pp. 41995–42032, 2026.

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*, pp. 3616–3630, 2025.

A Dataset Construction

VisAudit pairs flawed visualizations with multimodal reference evidence that captures the intended visualization. We construct these instances by introducing controlled defects into otherwise aligned visualization examples while preserving the source evidence, enabling agents to diagnose and repair defects through cross-modal reasoning. This construction also supports controlled evaluation of both defect correction and preservation of unaffected chart properties.

Source visualizations. We use ChartNet's `core_permissive` subset (Kondic et al., 2026). Each ChartNet instance contains four aligned modalities: plotting code, its rendered chart, the underlying CSV table, and a natural-language summary describing the visualization. This cross-modal alignment is essential to our construction: before perturbation, the table and summary are semantically consistent with the reference chart and thus provide complementary evidence about its intended content. We execute the reference code (i.e., the initial code that renders ground truth chart) and retain the rendered chart and structured figure properties as the reference for evaluation. Starting from each aligned instance, we perturb the plotting code to introduce one or more controlled defects and execute the modified program to produce the flawed chart. Crucially, the original CSV table and summary remain unchanged. As a result, the perturbed chart may conflict with information preserved in these reference modalities, creating a setting in which an agent can detect and correct visualization defects by reasoning across modalities.

The flawed evaluation set contains 1,900 instances derived from 854 source charts and covers 21 chart types. A source chart can appear in several instances with different perturbations. The plotting-library labels are Seaborn (704 instances), Matplotlib (687), and Plotly (509). Reference and perturbed code contain medians of 61 and 60 lines, respectively; summaries contain a median of 285 words. The median table contains 13 data rows and four header columns. The chart-type distribution is in Appendix A.1.

Perturbation taxonomy. We define ten flaw categories (Table 2). Each introduces a specific discrepancy in the chart while leaving the source evidence unchanged. For example, value perturbation changes a plotted value while preserving the correct value in the table. Missing legend tests whether the agent recognizes that a mapping is needed to identify the plotted series, even when the text summary does not explicitly request a legend.

Generating single and combined flaws. For each flaw category, we define a category-specific prompt (in Appendix A.2) that instructs `Mistral-Medium-3.5-128B` to modify the reference plotting code so as to inject the corresponding defect while otherwise preserving the original visualization. We execute the modified code to obtain the flawed chart and compare its structured figure properties with those of the reference chart to verify the changes introduced by the perturbation. Axis truncation is handled separately using a deterministic transformation that modifies the axis limits and records the clipped reference value. To construct multi-flaw tasks, we sequentially apply two or three category-specific perturbations to the same reference instance. The combination policy retains 36 pair scenarios and 60 triple scenarios, excluding combinations with conflicting effects or effects that cannot be sufficiently attributed to their components. We retain records of the constituent edits and compare the final flawed chart with the reference to capture their combined effect.

Quality control and human alignment. Because our flawed instances are generated through language-model edits, we validate the resulting tasks to ensure that the intended defect is successfully instantiated and that it can be identified and repaired from the available reference evidence. We first apply automatic checks that require executable plotting code and a nonempty change in the extracted figure properties. Category-specific checks further verify the intended perturbation; for example, that a reference legend-label group disappears.

Beyond these structural checks, determining whether a generated task is semantically valid and solvable requires assessing whether the unchanged source evidence supports identification and repair of the injected defect. Since manually inspecting every generated instance is impractical at our scale, we use a cross-family LLM judge panel for this assessment. The panel receives the flaw description, the summary, and the CSV table (prompt in Appendix A.3) and consists of four models: `Llama-3.3-70B-Instruct`, `Qwen2.5-72B-Instruct`, `DeepSeek-V3.2`, and `gpt-oss-120b`. Each instance is evaluated by at least three models and accepted only with a majority of positive judgments. Of 10,945 generated single-flaw candidates, 8,446 (77.2%) pass all

checks. Multi-flaw instances must also preserve every component's change in the final chart and are judged per component; 3,868 of 4,370 double-flaw and 4,357 of 5,056 triple-flaw instances pass.

To validate the reliability of this LLM-based quality control, we manually inspect a random subset of 150 generated instances and compare the human assessments with the corresponding LLM-judge decisions. We manually assess whether the intended defect is present and whether the source evidence is sufficient to support its repair. We observe over 96.64% agreement between human and LLM judgments, supporting the use of the LLM panel to validate the remaining instances at scale.

Evaluation set. We sample the evaluation set to include 40 instances per single-flaw category, 25 per pair, and 10 per triple. This gives 400 single-, 900 double-, and 600 triple-flaw instances across 106 scenarios (Table 3). Tracks A and B use these same 1,900 instances and differ in whether the instruction reveals the flaw categories. Flaw count, track, available modalities, and interaction budget provide separate dimensions for studying task difficulty.

Initially correct charts and release. Track C adds 300 validated, unperturbed ChartNet instances, giving 2,200 instances. Their source identities are unique and disjoint from the 854 sources in the flawed set. Reference checks assess executability, agreement with the source table and the text summary. The correct and flawed instances are mixed with a fixed seed, and the instruction does not reveal whether a particular chart needs repair. Correct charts account for 13.6% of Track C.

Released instance identifiers and versioned manifests identify the task files and evaluation code. Reference code and perturbation metadata are reserved for the evaluator and excluded from the agent's task prompt.

A.1 Dataset statistics

Table 11 reports the source chart-type labels per task. Counts include repeated uses of a source chart across distinct perturbations. Code length counts physical lines; summary length counts whitespace-separated words; CSV length counts data records after the header. Header width is reported separately because source tables need not be rectangular.

Table 11: Instance-weighted source chart types in the flawed and additional clean populations. A zero denotes a source chart type absent from the clean selection.

Chart type	Flawed instances	Clean instances
3D Bar Chart	2	1
Area Chart	148	37
Bar Chart	182	28
Box Plot	71	0
Bubble Chart	107	14
Funnel Chart	84	24
Heatmap	22	25
Histogram	57	0
Kernel Density Estimate Plot	86	1
Line Chart	187	49
Multi-Axes Chart	162	10
Pie Chart	20	2
Radar Chart	52	19
Ring Chart	14	1
Rose Chart	44	12
Scatter Plot	157	31
Stem Plot	98	6
Swarm Plot	109	5
Tornado Chart	181	27
Treemap	21	3
Violin Plot	96	5
Total	1,900	300

A.2 FLAW-GENERATION PROMPTS

Flawed candidates are generated by `mistralai/Mistral-Medium-3.5-128B` with temperature 0, seed 42, and at most 16,000 output tokens. Every request uses the system prompt below and a user message that is shared across categories except for the category instruction and the two example values in the output contract. Axis truncation is produced by the deterministic transformation described earlier in Section A and uses no prompt. In a multi-flaw chain, each later request receives the previously perturbed script as `code.py`, and its user message starts with the sentence

```
The input already contains these intentional flaws, which must remain
unchanged:  [categories of the earlier stages].
```

followed by an empty line. The texts are reproduced from the implementation; the benchmark's working title is replaced by its published name, and line breaks and em dashes are changed for typesetting only.

System prompt.

```
You are generating benchmark instances for VisAudit, a research
benchmark on detecting and repairing flawed data visualisations.
## The data model
Every source sample consists of four mutually consistent artifacts:
  * `code.py` --- a self-contained matplotlib/seaborn script that
renders one chart
  * `chart.png` --- the rendered image
  * `table.csv` --- the underlying data table
  * `summary.txt` --- a natural-language description of the chart
Your job is to introduce EXACTLY ONE flaw into `code.py` so that the
*rendered chart starts to lie*, while `table.csv` and `summary.txt`
stay untouched and therefore stay TRUE. A downstream agent will later
be shown the flawed chart together with the truthful table and
summary, and must find and repair the flaw.  The flaw is only usable
if the truthful modalities actually contradict the new chart.
You never edit and never mention the table or the summary in the
code.  They are given to you only so that you can (a) choose a flaw
that they contradict and (b) quote the exact evidence that proves the
contradiction.
## The request
Each request gives you:  `sample_id`, `category` (the flaw type you
must inject, with its own instruction), `library`, `chart_type`, the
complete ground-truth script, `summary.txt`, the header plus the
first rows of `table.csv`, and `candidates` --- editable targets that
a static-analysis pass already located in the script.  Prefer a
target from `candidates`; those are verified to be plain, safely
replaceable literals.
## Hard constraints (violating any one of these makes the instance
unusable)
1.  RETURN THE COMPLETE MODIFIED SCRIPT --- every line, from the
first import to the last statement.  Not a diff, not a snippet, no
ellipsis, never "...  unchanged ...".
2.  CHANGE AS LITTLE AS POSSIBLE. The ideal edit touches one line;
most categories allow only a single string or number literal to
change.  Never make a second, "while I am here" improvement.
3.  NEVER REFORMAT. Preserve the original whitespace, indentation,
line breaks, blank lines, quote style (' vs "), trailing commas,
comment text, import order, variable names and statement order.
Every byte you did not have to touch must stay identical, so that
`diff gt.py flawed.py` shows only the injected flaw.
4.  NEVER TOUCH DATA THAT IS NOT THE TARGET OF THE FLAW. Only the
literal(s) named in the category instruction may change.
```

```
5.  KEEP THE SAVE CALL INTACT. The `savefig(...)` / `plt.savefig(
...)` line (filename, dpi, bbox_inches, everything) must stay
byte-identical and must remain the last rendering action.  Likewise
keep `figsize`, `dpi`, `plt.subplots(...)` geometry, `tight_layout()`
and any `plt.show()` exactly as they are --- changing the canvas
would make the automatic visibility check pass for the wrong reason.
6.  THE SCRIPT MUST STILL RUN. Same Python, same pinned matplotlib/
seaborn versions, no network, no filesystem access, no new imports,
no new helper functions, no new variables --- unless the category
explicitly asks you to insert a line.  If your edit makes a keyword
argument invalid for the new call, remove that keyword argument (and
say so in `edit_description`) rather than returning a script that
raises.
7.  THE SCRIPT MUST STAY DETERMINISTIC. Never introduce randomness,
never change or remove a random seed, never make output depend on
time or environment.
8.  NO SELF-DOCUMENTATION. Do not add comments, markers, TODOs, "#
bug injected here", or any hint that the code was modified.  The
flawed script must look like code a human wrote by mistake.
9.  THE FLAW MUST BE A PLAUSIBLE MISTAKE, not vandalism.  A tired
analyst could have made it.  And it must be repairable:  an agent
holding the table and the summary must be able to state what is wrong
and restore the original behaviour.
10.  THE FLAW MUST CHANGE THE RENDERED PIXELS. If your edit would
produce a pixel-identical or near-identical image, the instance is
worthless --- pick a different target or decline.

## Evidence contract

Together with the code you return an `evidence` object.  An automatic
solvability gate matches the quoted evidence against the untouched
`summary.txt` and `table.csv` using case-insensitive,
whitespace-normalised substring matching.  Quote VERBATIM from the
artifact you cite --- do not paraphrase, do not re-capitalise, do not
round a number, do not drop a trailing zero.  Only these keys exist;
use exactly the one your category instruction prescribes:

  * "summary":  [strings that must ALL appear in summary.txt]
  * "min_summary":  integer --- if set, only this many of the
"summary" strings need to match
  * "table":  [strings that must ALL appear in table.csv, verbatim,
including formatting such as trailing zeros]
  * "table_nums":  [numbers that must appear in table.csv; 24 also
matches "24.0"]
  * "either":  [strings that must appear in summary.txt OR in
table.csv]
  * "summary_any_color":  true --- the weakest form, it only asserts
that the summary names some colour

Evidence always describes the ORIGINAL, TRUE state that the flawed
chart now contradicts --- never the value you just injected.

## Declining

If the category genuinely does not fit this chart --- no suitable
target, the edit would be invisible, the flaw would not be
contradicted by summary or table, or the script would break --- set
`applicable=false`, give a one-sentence `decline_reason`, and leave
`perturbed_code` empty.  A clean decline is far more valuable than a
forced instance:  declines are cheap, invalid instances poison the
benchmark.

## Output

Answer only through the provided structured output schema.  `edits`
must list every contiguous text replacement you made as an (old, new)
pair, copied verbatim from the two scripts, such that applying them
to the original script reproduces `perturbed_code` exactly.  This is
machine-verified.
```

```
## V2 output-contract override
For this V2 pipeline, the JSON contract at the end of the user
request replaces
the legacy structured-output paragraph above.  Return that JSON
object directly;
there is no `edits` field.  `target` is mandatory and identifies the
exact
Figure-Object entity the postcondition must prove.  For color
contradictions,
`evidence.summary_color_mapping` is an additional allowed and
mandatory V2 key;
it maps the exact summary-named entity to its exact truth color, and
`evidence.summary_quote` must copy the complete, direct affirmative
clause that
maps exactly that one entity to exactly that one named color.
Decline when the
clause contains another chart entity or color, negation, contrast,
modality, a
condition, or a past/former-state qualifier, and also decline when
any other
clause in the full summary gives that target a different color or
blocks its
truth-color claim.
Repair truth must be reconstructable from the physical summary,
table, and
candidate code; `target` and `evidence` never supply otherwise hidden
truth.
For wrong-title, the physical summary must contain one current,
direct
affirmative title-to-exact-GT-text clause, that text must be the only
distinct
title claim in the summary, and the summary must not claim the
injected title.
For wrong-axis-label, require either a clean direct axis/
quantity-to-exact-GT-
label summary relation or an exact unique CSV column whose complete
values or
locators bind to that GT axis.  Every label claim in the active axis/
quantity
scope must agree with the GT label, and no competing injected-label
CSV
relation may exist; header presence alone, an unused code string or
an
arbitrary table cell does not count.  V2 changes exactly one
canonical axis label with replacement mechanism (a); do not use the
legacy
two-axis swap because the binding target identifies one axis.  For
wrong-chart-type,
require a clean direct current relation from the whole chart or
target series
to the exact GT kind, with no competing kind claim in that scope; a
different
non-target series may legitimately have another kind.  For
missing-legend, the
summary is a conflict check only and never reconstruction evidence.
Every one
of at least two GT labels must instead bind through at least one of
two structural
routes:  the exact full target-axis sequence of definitely-live
plotted
`label=`/`name=` literals occurs once in GT legend order and maps to
captured GT
```

```
series roots, or one unique complete wide named CSV series relation
occurs in
exact GT legend order and the captured target-axis series set/count
exactly
equals that legend group.  A subset with a hidden extra series,
comment, unused
assignment, call inside a function/branch/condition/comprehension, or
unordered
label group does not count.  For dropped-series, the complete missing
label-to-value relation must bind uniquely to the physical CSV. Its
GT mark kind
must additionally be recoverable either from a clean direct current
target-series-
to-kind summary statement (a whole-chart statement alone is
insufficient), or
from uniform same-axis GT kinds plus at least one definitely-live
surviving
labeled plot call of that kind, with every such target-axis survivor
call
resolving to that same kind.  Decline on negation, contrast,
modality, conditions,
or former-state wording.
```

User message. Square brackets mark the fields filled in per request.

```
[category instruction]
Return one JSON object and no Markdown.  It must match this contract:
{
  "applicable":  true,
  "decline_reason":  "",
  "perturbed_code":  "the complete modified Python script",
  "edit_description":  "one sentence",
  "evidence":  [category-specific evidence example],
  "target":  [category-specific target example]
}
If the category does not apply, return applicable=false, a concise
decline_reason,
empty perturbed_code, and empty evidence/target objects.  Never guess
target metadata:
it is consumed by a fail-closed semantic gate after the script
renders.
The values above are examples of the exact keys, not values to copy.
Replace every
example with the actual original/injected value and captured series
or axis identity.
--- source_id:  [source id] | category:  [category] | library:
[library] | chart_type:  [chart type] ---
## code.py
```python
[reference plotting script]
```
summary.txt (truth)
[summary]
table.csv, first rows (truth)
```
[first 12 lines of table.csv]
```
```

Category instructions. Each category instruction replaces `[category instruction]` in the user message, and the contract examples replace the two bracketed example fields.

Wrong title.

```
Replace the chart's title with a plausible but factually wrong title.
Target: the single title string literal. In a matplotlib or seaborn
script it is the one given in `candidates.title` (from `plt.title(
...)`, `fig.suptitle(...)`, `ax.set_title(...)` or `.set(title=...)`).
A plotly script carries no static candidate, so locate the literal
yourself --- it is written in one of these forms:
  * `fig.update_layout(title='...')` or `fig.update_layout(title_text=
'...')`;
  * a dict, `fig.update_layout(title=dict(text='...', x=0.5, font=
dict(size=18)))` or `title={'text': '...', 'y': 0.95, 'x': 0.5}`
--- only the `text` value is the title;
  * the express call itself, `px.<kind>(..., title='...')`, for bar,
line, scatter, pie, box, violin, treemap, funnel and the rest.
Change that literal and nothing else. Leave the `x`, `y`, `font` and
`xanchor` entries beside it as they are, and never mistake a `legend_
title_text=`, an axis title or a `subplot_titles=(...)` entry for the
chart title.
The new title must:
  * read like a real chart title --- same casing convention as the
original, roughly the same length, may keep a unit or a year range in
parentheses;
  * be WRONG for this chart: a different metric, entity, region or
time period;
  * NOT be a rewording, translation, abbreviation or near-synonym of
the true title, and must not reuse the true title's key nouns;
  * not equal any CSV column header;
  * stay at most 120 characters.
If the script sets more than one title (e.g. a suptitle plus
per-axes titles), change only the one in `candidates.title` and leave
the others untouched --- the remaining truthful title is part of what
makes the flaw findable. A plotly `title=` dict with NO `text` key
is not a second title: it only positions or styles the title that an
express call already set, so leave it alone and falsify the express
`title=` string instead.
Evidence: {"summary": ["<the complete current, direct affirmative
summary clause that states the ORIGINAL title character for
character>"]}. The clause must explicitly bind the title to that
exact text (for example, "The title is ..." or "The chart is titled
..."); a loose mention of the words is insufficient. The GT text
must be the only distinct title claim anywhere in the summary. The
clause must not negate, hedge, contrast, condition or describe a
former title, and the summary must not claim the injected or any
other competing title.
Decline if there is no title string literal, if the only `title=` in
the script is a styling dict without a `text` key, if the summary has
no qualifying direct affirmative exact-title clause, or if it
contains a second distinct title claim.
```

Contract examples: `evidence {"summary": ["The title is Original chart title."]}`; `target {"path": "axes[0].title", "truth": "original", "injected": "wrong"}`.

Wrong axis label.

```
Make an axis label describe the wrong quantity. Pick exactly ONE of
these two mechanisms:
(a) Replace ONE axis-label string literal (prefer the y label; use x
only if there is no y label) with a fluent, domain-plausible label
that names a DIFFERENT quantity, unit or dimension. It must not be a
```

```
synonym or reformulation of the true label, and must not equal any
CSV header of this table.  Keep the style of real chart labels (Title
Case, optional unit in parentheses), at most 60 characters.
(b) If distinct x and y label literals both exist, swap the two
strings between `set_xlabel`/`set_ylabel` (or `plt.xlabel`/
`plt.ylabel`).
In matplotlib and seaborn the literal sits in `set_xlabel`/`set_
ylabel`/`plt.xlabel`/`plt.ylabel`.  A plotly script carries no static
candidate; the same literal is written in one of the forms below, and
both mechanisms work on all of them:
  * `fig.update_layout(xaxis_title='...', yaxis_title='...')`, also
spelled `xaxis_title_text=` --- for (b) exchange the two strings
between the two keywords;
  * `fig.update_xaxes(title_text='...')` and `fig.update_yaxes(title_
text='...')` --- for (b) exchange the two strings between the two
calls;
  * the nested form, `fig.update_layout(xaxis=dict(title='...'),
yaxis={'title':  '...'})` --- the label is the plain string, or the
`text` of a `{'title':  {'text':  '...'}}`; a `{'title':  {'font':
...}}` holds styling only and is no target;
  * an express `labels={'<column>':  '<label>'}` mapping --- replace
the VALUE of the entry whose KEY is the column that this call passes
to `x=` or `y=`, and for (b) exchange the values of the x-column
entry and the y-column entry.  An entry for any other column renames
a legend or a hover field rather than an axis, so it is no target.
Work only with entries that are already there; never add one.
If a subplot figure writes the same axis title at several `row=`/
`col=` call sites, replace the string at EVERY one of them so the
whole figure carries the same wrong label, and use mechanism (b) only
when the figure has exactly one x title and one y title.
Change nothing else --- in particular do not touch the title, the
tick labels, the tick formatter or the data.
Evidence:
  * mechanism (a):  {"either":  ["<the original label, verbatim, from
a clean direct axis/quantity summary relation or one exact unique CSV
column whose complete values/locators bind to that GT axis>"]}
  * mechanism (b):  {"either":  ["<original x label>", "<original y
label>"]}, with each label supported by that same kind of physical
relation.
Header presence alone, an unused candidate-code string and an
arbitrary table body cell do not count.  Every label claim in the
active axis/quantity scope must agree with the exact GT label, and a
competing injected-label table relation also rejects.  A summary
clause that negates, hedges, contrasts, conditions or describes the
old label does not count either.
Decline if no axis-label string literal exists, if the script's only
`labels={}` entries rename columns that sit on neither `x=` nor `y=`,
if the exact original label has neither a qualifying summary relation
nor one exact unique, fully axis-bound CSV column, or if another
claim in the active scope gives the axis a competing label.
```

Contract examples: `evidence {"either":  ["The y-axis label is Original axis label."]}`; `target {"path":  "axes[0].xlabel", "axis_index":  0, "axis":  "x", "truth":  "original", "injected":  "wrong"}`.

Missing legend.

```
Remove the chart's legend so the plotted series can no longer be
identified, while the drawn data stays exactly the same.  Four
allowed mechanisms --- take the ONE that fits this script:
```

```
(a) Delete the ENTIRE statement that creates the legend (`plt.legend(
...)`, `ax.legend(...)`, `fig.legend(...)`) --- remove the whole line
including its trailing newline, leaving surrounding indentation and
blank lines intact.
(b) For a seaborn axes-level call that draws an automatic legend from
`hue=` (lineplot, scatterplot, barplot, boxplot, violinplot,
stripplot, swarmplot, kdeplot, histplot, pointplot, countplot),
insert `legend=False` as an additional keyword argument in that call.
Insert it as the first keyword after the opening parenthesis and keep
the rest of the call byte-identical.
(c) A plotly script that already writes `showlegend=True` inside
`fig.update_layout(...)`:  flip that ONE literal to `False`.  It is
the smallest possible edit, so prefer it whenever the script offers
it.
(d) Every other plotly script:  plotly switches the legend on by
itself as soon as the figure holds several traces or an express
grouping column (`color=`, `symbol=`, `line_dash=`, `pattern_shape=`),
so there is no legend statement to delete.  Insert exactly ONE new
line,
    fig.update_layout(showlegend=False)
immediately before the line that writes the image (`fig.write_image(
...)`, or `pio.write_image(fig, ...)`) and at the same indentation
level, spelled with the figure's own variable name.
Only that layout-level switch really removes a plotly legend.  NEVER
delete or rewrite a `legend=dict(...)` / `legend={...}` block, a
`legend_title=` or a `legend_title_text=`:  plotly goes on drawing the
legend and merely moves it back to its default position, so every
series is still named and the instance is worthless.  And never set
`showlegend=False` on the individual traces --- that is one edit per
trace, and a single trace you miss leaves half a legend standing.
`showlegend=False` also does nothing at all in two cases, and both
leave the image pixel-identical, i.e.  a silent no-op:  when the
express `color=` column is NUMERIC, because plotly then draws a
continuous colour BAR instead of a legend --- look the column up in
table.csv first and decline when it holds numbers; and on a
`treemap`, `sunburst`, `icicle` or `funnel_area`, which label their
segments inside the marks and carry no legend to switch off.
Mechanism (a) is only a real removal when nothing re-draws a legend
automatically --- this is the
single most common way this category silently fails (measured:  33 of
100 shipped instances had an
EMPTY footprint, i.e.  the rendered chart still carried a legend).
Two known re-drawers:
  * a seaborn call with `hue=` re-draws its own legend after your
deleted `plt.legend(...)`/
    `ax.legend(...)` line --- there you MUST use (b) (`legend=False`
inside the seaborn call) and
    ALSO delete the explicit legend statement if one exists; the pair
counts as one mechanism.
  * `DataFrame.plot(...)` draws a legend by default --- insert
`legend=False` into that call instead
    of only deleting a later legend line.
Before you answer, trace the final figure in your head:  if ANY code
path still produces a legend
object after your edit, the instance is a no-op --- pick the
mechanism that switches the legend off
at its source, or decline.
Do NOT delete or rename the `label=` arguments, do not remove any
series, do not change colours:  after the edit the same marks must be
drawn in the same colours --- only the key that names them is gone.
```

Never delete a legend statement that sits inside a loop or an `if` body when removing it would leave an empty block.

Evidence: {"summary": ["<series name 1>", "<series name 2>", ...], "min_summary": 2} --- list at most three exact GT names. This evidence list does not itself establish V2 recoverability: summary text is only a conflict check and is never a summary-only reconstruction route. Every removed label must instead survive through at least one of two physical structures: (1) the exact full target-axis sequence of definitely-live top-level plotted `label=`/`name=` literals appears once in exact GT legend order and maps to captured GT series roots, or (2) one unique complete wide named CSV-series relation appears in exact GT legend order and the captured target-axis series set/count exactly equals that legend group. A subset with a hidden extra series, comment, unused assignment, arbitrary table cell, call inside a function/branch/condition/comprehension, unordered or duplicate label group, negated relation or former-state claim does not count.

Decline if the chart draws only one series, if there is no legend to remove (a plotly script that already sets `showlegend=False` has none left, and a numeric `color=` column or a treemap/sunburst has none to begin with), if fewer than two exact GT labels survive through an ordered route, or if that route covers only a subset of the target-axis series.

Contract examples: evidence {"summary": ["series A", "series B"], "min_summary": 2}; target {"path": "axes[0].legend.labels", "labels": ["series A", "series B"]}.

Swapped legend labels.

Exchange the names of two series so that the legend labels the geometry wrongly.

A script names its series in one of several ways. Pick the ONE mechanism that fits this script and needs the smallest edit:

(a) Two plot calls carrying distinct `label="..."` string arguments: swap the two string values between them --- after the edit, call A carries B's name and vice versa.

(b) An explicit legend label list --- `legend(["A", "B", ...])`, `legend(handles, ["A", "B"], ...)`, `legend(labels=["A", "B"])`: swap two of its string entries.

(c) Two plotly traces carrying `name="..."` (`go.Bar(..., name='A')`, `add_trace(...)`, `add_scatter(..., name='A')`): swap the two strings.

(d) A grouping column whose values become the legend entries (`hue=`, `style=`, plotly express `color=`/`symbol=`/`line_dash=`): swap the two category names where the script spells them out --- the two keys of the dict the frame is built from, the two occurrences in the inline column literal, the two wide column names that a `melt`/`pivot` turns into levels, or the two sides of a `rename`/`replace` mapping the script already applies. The rows keep their values and their place; they only start carrying each other's name. Swap EVERY occurrence of both names in that one column --- a half-swapped column invents a third group --- and use this mechanism only when all of them fit within about three lines; a name repeated down a long column is too many edits. Then check every other place they appear --- a later `df[df['Sector'] == 'Oil & Gas']` filter, a `y1='Primary'` selection, a colour dict --- and check how colours are assigned: a palette keyed by category name follows the names by itself and stays consistent, but a POSITIONAL palette (a `palette=[...]` list, `colors[i]`) over levels that are ordered by name can move the

```
colours as well, and a mark whose fill no longer matches its own line
is a second, unwanted defect.  Decline rather than ship that.

(e) The legend labels come from a grouping column you cannot rewrite
in one or two literals:  pass the names into the EXISTING legend
call, `ax.legend(title='Region')` becomes `ax.legend(['South',
'North'], title='Region')`, listing every drawn series in drawing
order with exactly two of them exchanged.  Only do this when the
drawing order is beyond doubt --- an explicit list the script loops
over, an explicit `hue_order=`/`order=`/`category_orders=`,
alphabetically sorted pivot columns, the order of the `add_trace`
calls --- and only when the legend holds nothing but those series (
never when it also carries size, style or marker entries).  If you
would be guessing the order, decline:  a guessed order can hand the
labels back to the right geometry, and the instance is then
worthless.

Change NOTHING else.  In particular do not swap or change the
colours, the data, the markers, the argument order or the order of
the plot calls:  if you also swapped the colours in mechanism (a), (
b) or (c), the chart would become truthful again and the instance
would be worthless.  Only the naming may lie.

Prefer two series that are visually distinguishable (different
colours or clearly different shapes) and whose names both occur in
the summary or the table.

Evidence:  {"either":  ["<name A>", "<name B>"]} --- the two
ORIGINAL names, verbatim.

Decline if the chart draws fewer than two named series, if no
mechanism above applies without a second change, if the two series
are visually indistinguishable, or if the two names occur in neither
the summary nor the table.
```

Contract examples: `evidence` {"either": ["series A", "series B"]}; `target` {"path": "axes[0].legend.labels", "labels": ["series A", "series B"]}.

Value perturbation.

```
Corrupt exactly ONE number in the plotted data so that a single bar/
point is wrong while the table still holds the truth.

Pick one numeric literal from an inline data series --- a list or
tuple with at least five numeric elements that represents MEASURED
VALUES, not years, indices, x positions or axis ticks --- and replace
it with the value multiplied by one of 0.45, 1.6 or 2.25, rounded to
the same precision as the original literal (integers stay integers;
floats keep the same number of decimals).

Rules:
  * Change exactly one element.  Do not touch any other number, do
not reorder the list, do not reformat or re-wrap the list.
  * Prefer a value in the middle of the value range:  not the largest
(the axis rescales and the change can be absorbed) and never a zero.
  * The change must move the mark by a visible amount --- roughly at
least 10% of the plotted value range.
  * In `target`, `point_index` is the zero-based index of the numeric
source-list
    element you changed.  For a heatmap, that list must have a
same-length categorical
    label list and the exact source row must occur in table.csv; the
validator binds
    both through any pivot/reordering to the rendered cell.  Decline
if this mapping
    is not exact.
```

```
Evidence: {"table_nums": [<the ORIGINAL value>]} if the table
writes the number in the same form. If the table writes it with
different formatting (e.g. "0.30" vs 0.3, or with a thousands
separator), use {"table": ["<the exact token from table.csv>"]}
instead.
Decline if there is no inline numeric series of measured values, if
the chosen value does not appear in the table, or if every candidate
list is a year/index axis.
```

Contract examples: `evidence` {"table_nums": [10]}; `target` {"series_label": "series A", "point_index": 2, "truth_value": 10, "injected_value": 20}.

Swapped series.

```
Exchange the value arrays of two series so that every point of both
series lies, while the labels, colours and drawing order stay
correct.
Find two inline numeric lists of the SAME length (at least 5
elements) that belong to two different plotted series and hold
different values, and swap their contents between the two plot calls
--- either by swapping the two list literals, or by swapping the two
variable names at the point where they are passed to the plotting
calls, whichever is the smaller edit.
Copy the list literals character for character, including spacing and
number formatting: do not reformat, do not re-wrap across lines, do
not round, do not sort.
Rules:
  * Do not touch the labels, colours, markers or the order of the
plot calls.
  * The swap must be a real transposition (both series change), never
a copy of one over the other.
  * The two series must have visibly different value profiles ---
otherwise nothing changes on screen.
Evidence: {"table_nums": [<a distinctive value of series A>, <a
distinctive value of series B>]} --- e.g. the maximum of each
series, exactly as it appears in table.csv.
Decline if there are not two equal-length numeric series, if the two
series carry (almost) identical values, or if their values are not
verifiable in the table.
```

Contract examples: `evidence` {"table_nums": [10, 20]}; `target` {"labels": ["series A", "series B"]}.

Dropped series.

```
Delete one entire data series from the chart while the table and the
summary still describe it.
A script draws its series in one of several ways. Pick the ONE
mechanism that fits this script and needs the smallest edit, and drop
exactly one series with it. Leave the other series, the legend call,
the axis limits, the colours and everything else untouched.
(a) A plotting call of its own: remove the COMPLETE statement ---
`plt.plot(x, y2, label='B')`, `ax.bar(...)` --- including its
trailing newline. If the call's result is assigned (`markerline2,
stemlines2, baseline2 = ax.stem(...)`, `bar_plot = sns.barplot(...)`),
remove that whole assignment statement, together with the statements
that do nothing but style exactly those returned artists (`plt.setp(
markerline2, ...)`) --- and only when nothing else reads them.
```

```
(b) A grouping keyword that splits one call into several series ---
seaborn `hue=`/`style=`, plotly express `color=`/`symbol=`/`line_dash=
`:  filter that one level out of the frame in place, inside the
existing call --- `data=df` becomes `data=df[df['Region'] !=
'North']`, `px.line(df, ...)` becomes `px.line(df[df['Region'] !=
'North'], ...)`.  Compare against the group NAME, never against a
number.  If the call already passes `hue_order=`/`order=` (or a plotly
`category_orders=` list), delete that one entry from the list instead
--- seaborn draws only the levels it is given.

(c) A wide frame plotted with several y columns:  drop that column
from the list --- `df.plot(x='Year', y=['A', 'B', 'C'])` becomes `y=
['A', 'C']`, `df[['A', 'B']].plot()` becomes `df[['A']].plot()`.

(d) A melted or long frame:  filter that one category out where the
frame enters the plot, exactly as in (b) --- `data=df_long[df_
long['Metric'] == m]` becomes `data=df_long[(df_long['Metric'] == m) &
(df_long['Sector'] != 'Retail')]`.

(e) Series drawn by a loop over categories or columns:  narrow what
the loop iterates over, so that one series is never drawn --- `for s
in df['Sector'].unique():` becomes `for s in ['Retail', 'Energy']:`,
`for c in df_pivot.columns:` becomes `for c in df_pivot.columns.drop(
'Retail'):`.

(f) Several plotly traces:  remove one whole `fig.add_trace(...)` /
`fig.add_bar(...)` / `fig.add_scatter(...)` statement.

Rules:
  * Never remove a statement out of a loop body or a comprehension
--- that would delete every series or leave an empty block.  The one
exception:  a body that draws SEVERAL series per iteration (two
`plt.stem(...)` calls per category) may lose one of them, as long as
another drawing statement stays.
  * Never remove the last remaining series --- a blank chart is
rejected by the pipeline.
  * The series must disappear from EVERY mark that draws it.  If a
line and its `fill_between` both draw it, filter it out where the
frame is built so that both lose it; if that needs more than one
edit, decline.
  * Never introduce a number that was not in the script:  filter on
the series' name, not on an index or a threshold.
  * Check what runs after the plotting call.  Anything that indexes
the drawn artists by position (`ax.containers[1]`, `axes.flat`,
`colors[i]`) must still find them, or the script raises and the
instance is lost.
  * Do NOT remove the series' entry from an explicit `legend([...])`
list, do NOT shorten a `palette=`/`colors=` list and do NOT delete
the underlying data definition:  the leftover, now-unused data and
the stale legend entry are realistic symptoms of this bug and part of
the repair signal.

Prefer the series whose name is clearly present in the summary or in
the table.

Evidence:  {"either":  ["<label of the removed series>"]} ---
verbatim.

The complete removed label→value sequence must bind uniquely to the
physical CSV. Its GT mark kind must also remain reconstructable by
one of two routes:  either the summary directly and currently binds
the removed target series itself to that exact kind (a whole-chart
statement alone is insufficient), or all captured GT series on the
same axis have that kind and the flawed code retains at least one
definitely-live top-level surviving plotted `label=`/`name=` call of
that kind, with every such target-axis survivor call resolving to the
same kind.  Calls hidden in functions, branches, conditions or
comprehensions do not count.
```

```
Decline if the script really draws only one series, if no mechanism
above fits without a second, unrelated change, if the removed series'
complete label→value sequence is not uniquely bound to the physical
CSV, if neither kind-recovery route holds, or if the removed series'
name occurs in neither the summary nor the table.
```

Contract examples: `evidence` {"either": ["series A"]}; `target` {"series_label": "series A"}.

Wrong chart type.

```
Render the same data with the wrong chart type.

Replace the plotting function with a signature-compatible alternative
at EVERY call site of that function, keeping the positional arguments
unchanged.  Allowed swaps:
  * matplotlib (plt./ax./axs./ax1./ax2.):  bar->plot, barh->plot,
scatter->plot, plot->scatter
  * seaborn:  barplot->lineplot, lineplot->scatterplot,
scatterplot->lineplot, boxplot->violinplot, violinplot->boxplot,
stripplot->swarmplot
  * plotly express, the constructor itself:  scatter->line,
line->scatter, bar->line, area->line, histogram->box, box->violin,
violin->box, strip->box, density_contour->scatter, density_
heatmap->scatter, ecdf->line, funnel->bar, funnel_area->bar,
pie->bar, line_polar->line, scatter_polar->scatter, bar_polar->bar
  * plotly graph objects, the trace class:  Scatter->Bar,
Scattergl->Bar, Bar->Scatter, Histogram->Bar, Box->Violin,
Violin->Box, Funnel->Bar, Waterfall->Bar, Pie->Bar,
Scatterpolar->Barpolar, Barpolar->Scatterpolar

Swap ONE function or trace class and swap it at every site that calls
it.  Never two different ones in the same script, never a pair that
is not on this list, and never a second edit alongside it.

The data, labels, title, colours and axes must stay exactly as they
are.  Only the function name --- and, where unavoidable, incompatible
keyword arguments --- may change:
  * bar->plot:  remove `width=`, `bottom=`, `align=`, `yerr=`,
`capsize=`, `hatch=`, `edgecolor=`; keep `color=` and `label=`.
  * scatter->plot:  remove `s=`, `cmap=`, `norm=`, `vmin=`, `vmax=`,
`edgecolors=`; a SCALAR `c='red'` becomes `color='red'` (if `c=`
holds an array, decline instead).
  * plot->scatter:  remove `linestyle=`, `linewidth=`, `lw=`,
`markersize=`, and `ms=`.  Keep a compatible `marker=` such as
`marker='o'` byte-identical; do not remove it merely because the
line-only size keyword must go.
  * seaborn:  remove `errorbar=`, `ci=`, `capsize=`, `dodge=`,
`estimator=`, `orient=` when the target function does not accept
them.
  * plotly express --- the keywords the new constructor no longer
accepts:
      scatter->line:  `size=`, `size_max=`, `symbol=`, `symbol_map=`,
`opacity=`, `marginal_x=`, `marginal_y=`, `trendline=`, `color_
continuous_scale=`, `range_color=`;
      line->scatter:  `line_group=`, `line_shape=`, `line_dash=`,
`line_dash_map=`, `markers=`;
      bar->line:  `barmode=`, `base=`, `opacity=`, `text_auto=`,
`pattern_shape=`, `color_continuous_scale=`;
      area->line:  `groupnorm=`, `pattern_shape=`;
      histogram->box:  `nbins=`, `histfunc=`, `histnorm=`, `barmode=
`, `barnorm=`, `cumulative=`, `marginal=`, `opacity=`;
      box->violin:  `notched=`, `boxmode=`; violin->box:  `box=`,
`violinmode=`; strip->box:  `stripmode=`;
```

```
      density_contour->scatter:  `nbinsx=`, `nbinsy=`, `histfunc=`,
`histnorm=`, `z=`; ecdf->line:  `ecdfnorm=`, `ecdfmode=`, `lines=`,
`marginal=`, `opacity=`.
  * plotly graph objects --- the properties the new trace class no
longer has:
      Scatter->Bar:  `mode=`, `line=`, `line_color=`, `fill=`,
`fillcolor=`, `stackgroup=`, `connectgaps=`, `hoveron=`, and a
`marker=dict(...)` that carries `size=` or `symbol=` (a `marker_color=
` or `marker=dict(color=...)` survives);
      Bar->Scatter:  `width=`, `base=`, `offset=`, `insidetextanchor=
`, `textangle=`, `constraintext=`;
      Histogram->Bar:  `nbinsx=`, `nbinsy=`, `histfunc=`, `histnorm=
`, `cumulative=`, `xbins=`;
      Box->Violin:  `boxpoints=`, `boxmean=`, `notched=`,
`whiskerwidth=`; Violin->Box:  `box=`, `meanline=`, `points=`,
`bandwidth=`, `side=`, `scalegroup=`;
      Funnel->Bar:  `textinfo=`, `connector=`; Waterfall->Bar:  also
`measure=`, `increasing=`, `decreasing=`, `totals=`;
      Scatterpolar->Barpolar:  `mode=`, `line=`, `line_color=`,
`fill=`, `fillcolor=`, `textposition=`; Barpolar->Scatterpolar:
`width=`, `base=`, `offset=`.
  * `textposition=` does not survive a swap between a bar-like and a
line-like trace:  `'auto'`, `'inside'` and `'outside'` are bar
positions, `'top center'` and its siblings are scatter positions, and
the wrong one raises.  Drop the keyword whenever the swap crosses
that line.
  * a pie carries its data in different keywords than a bar, so
rename instead of removing:  `px.pie(..., names='Region', values=
'Share')` becomes `px.bar(..., x='Region', y='Share')`, and `go.Pie(
labels=..., values=...)` becomes `go.Bar(x=..., y=...)`.  Rename
exactly those two, then remove the pie-only rest --- `hole=`,
`textinfo=`, `insidetextorientation=`, `marker_colors=`, `rotation=`,
`sort=`, `direction=`, `pull=`.  A `domain=` is different:  it is
what puts each pie in its own tile of a grid, and dropping it stacks
every one of them into a single axes, so decline instead when the
figure draws its pies that way.
  * a polar express call likewise renames its data keywords:
`px.line_polar(df, r='Score', theta='Metric', line_close=True)`
becomes `px.line(df, x='Metric', y='Score')` --- `theta=` becomes `x=
`, `r=` becomes `y=`, and `line_close=`, `start_angle=`, `direction=`,
`range_r=`, `range_theta=`, `log_r=` go.  Never drop `r=`/`theta=`
without renaming them:  a call left without data draws nothing.

List every removed keyword in `edit_description`.  The resulting chart
must still draw visible marks --- never produce an empty plot.

Evidence:  {"summary":  ["<the complete current, direct affirmative
clause that binds the whole chart or every target series to the
original chart type>"]}.  The exact GT kind or controlled alias must
occur in that relation and must be the only kind affirmed in the
whole-chart/target-series scope.  Negation, contrast, modality,
conditions and former-state language do not count; a different
non-target series may legitimately use another kind.

Decline if there is no swappable call, if the call passes its data
via `x=`/`y=` keywords to `plt.plot`/`plt.scatter` (they reject those
keywords), if the swap needs more keyword surgery than the rules
above cover, if it would drop a data channel that the summary itself
describes (a bubble `size=`, a `pattern_shape=`) --- that would be a
second flaw on top of the wrong mark --- if the summary has no
qualifying direct affirmative chart-type relation, or if that scope
also affirms a competing kind.
```

Contract examples: evidence {"summary": ["Series A is shown as a bar chart."]}; target {"series_labels": ["series A"], "truth_kind": "bar", "injected_kind": "line"}.

Color contradiction.

```
Change the plotted colours so that they contradict a colour statement
in the summary.
The summary explicitly names colours.  Find which drawn element the
summary attributes a colour to, and repaint that element in a clearly
different colour.  Two allowed mechanisms:
(a) Swap two colour literals between two elements (`color=`, `c=`,
`edgecolor=`, `facecolor=`, or two entries of a `palette=[...]` /
`colors=[...]` list).
(b) Replace a single colour literal with a strongly contrasting one.
Rules:
  * Stay in the same notation:  a named colour is replaced by another
named colour, a hex string by another hex string.
  * The new colour must be clearly distinguishable from the old one,
both to a human eye and by RGB distance.  Good:  blue<->red,
green<->orange, purple<->yellow, brown<->teal, pink<->navy.  Bad:
blue->navy, gray->silver, '#1f77b4'->'#1f78b5'.
  * NEVER change a `cmap=` name and never change a seaborn palette
NAME (e.g.  'deep'->'muted') --- those are too subtle or may not
exist in the pinned version.
  * The literal you repaint must be one the drawn artist really uses
(measured:  9 of 100 shipped
    instances had an EMPTY footprint because the change never reached
the render).  Never repaint a
    literal that a later `palette=`/`set_palette`/style call
overrides, a variable that is never
    passed into a plot call, or a `color=` on a seaborn call that
also carries `hue=` (seaborn then
    takes colours from the palette and ignores `color=`).  If you
cannot trace the literal to a
    visible artist, decline.
  * Do not change line styles, markers, alpha, linewidth or anything
else.
Evidence:  {"summary":  ["<the colour word as written in the summary,
lowercase>"], "min_summary":  1} --- the colour the summary asserts
and the chart no longer shows.  V2 additionally requires the
output-contract `summary_quote` to be the complete verbatim direct
affirmative clause and `summary_color_mapping` to contain exactly one
target entity→truth-colour pair.  That clause may name only the
target and one truth colour, and the full summary may contain no
competing or blocked colour claim for that target.  The legacy
{"summary_any_color":  true} form is not sufficient for V2; decline
rather than use it.
Decline if the summary names no attributable direct affirmative
target→colour relation, if another clause gives that target a
different or blocked colour claim, if the named colour cannot be
traced to a colour literal in the code, or if the code contains no
plain-string colour literal.
```

Contract examples: evidence {"summary_color_mapping": {"series A": "blue"}, "summary_quote": "Series A is shown in blue."}; target {"series_label": "series A", "summary_color": "blue", "injected_color": "red"}.

A.3 QUALITY-CONTROL JUDGE PROMPT

All four judge models receive the same system prompt and user message, with temperature 0 and at most 700 output tokens. A judgment is positive when the returned `detectable` field is true; an answer without a parseable `detectable` field is requested up to three times and otherwise yields no judgment. For a multi-flaw task, each component flaw is judged separately with its own edit description. The texts are reproduced from the implementation with the same substitutions as above.

System prompt.

```
You are the quality gate of VisAudit, a benchmark of deliberately
flawed charts.
Each benchmark instance was built by taking a correct chart and
injecting exactly one flaw into
its plotting code.  The chart's underlying table (table.csv) and its
text summary (summary.txt)
were NOT touched, so they still describe the CORRECT chart.  A
benchmark agent is later shown the
flawed chart plus that truthful table and summary, and must notice
that something is wrong.
Your single question:  given only this summary and this table, could
an agent notice this flaw?
An instance is only fair if the answer is yes.  Judge the two cases
separately:
  * EXPLICIT: the evidence names the changed element literally.  If
the summary says
    'the title reads "Sales by Month"' and the title was changed,
that is explicit and
    the contradiction is direct.
  * SEMANTIC: the evidence does not name the element, but the meaning
still contradicts
    the flaw.  If the summary describes a chart of expenditure
percentages over time and
    the y-axis label was changed to "Total Enrollment", the summary
never quotes the axis
    label, yet the contradiction is unmistakable.  This still counts
as detectable.
Be strict about the opposite case:  if the summary is generic enough
that the flawed version
would read as perfectly consistent with it, the flaw is NOT
detectable and the instance must
be rejected.  Example:  a legend is removed, but the summary never
names the individual series,
so nobody could reconstruct which series is which.
Ignore whether the flaw is visually obvious.  Judge only whether the
given text and numbers
contradict it.  Quote the deciding fragment verbatim.
Respond with a single JSON object and nothing else:
{"detectable":  true or false,
 "basis":  "summary" | "table" | "both" | "none",
 "evidence_quote":  "the exact fragment that exposes the flaw, empty
if not detectable",
 "match_type":  "explicit" | "semantic",
 "reasoning":  "one sentence"}
```

User message.

```
## The injected flaw
category:  [category]
```

```
what was changed:  [edit description returned by the generator]
## summary.txt (untouched, describes the CORRECT chart)
[summary]
## table.csv, first rows (untouched)
```
[first 15 lines of table.csv]
```
Could an agent notice this flaw from the summary and table above?
```

B INTERACTIVE ENVIRONMENT

VisAudit places each agent in an execution-grounded loop. At the start of an episode, the environment copies the instance into a fresh writable workspace containing only the modalities available in that condition; evaluator metadata and reference solutions remain outside the workspace. The agent receives the track-specific instruction, a workspace listing, and six provider-neutral tools: `execute_python`, `read_file`, `write_file`, `list_files`, `view_image`, and `finish`. We disable parallel tool calls and execute one action at a time, yielding an auditable sequence of observations and interventions. The fixed system and task prompts are reported in Appendix B.1.

Stateful execution. For each episode, the runner starts a Python worker subprocess in the copied task workspace. It sends each code block to the worker and receives a structured execution result through a separate process channel. The worker reuses the same Python namespace across turns, so variables, imports, and open figures persist while the agent iteratively inspects and revises its chart. Rendering uses the headless Matplotlib `Agg` backend, and file-tool paths are checked against the episode workspace. Each execution has a timeout enforced by the worker and the parent process; if the worker becomes unresponsive, the parent restarts it and reports that its in-memory state was lost.

Multimodal feedback and stopping. An execution returns captured standard output, standard error, warnings, a traceback when applicable, and the files and images it changed. Images are not automatically returned after execution: the agent must invoke `view_image`, which returns the actual image bytes, to visually inspect its render. Execution failures are observations rather than terminal events, so recovery remains part of the task. Crucially, verifier outcomes are never shown to the agent and never trigger early stopping. Under the main `abc-uncapped-v1` protocol, there is no total tool-step or episode wall-time limit; the agent decides when to terminate by calling `finish`, while finite request, kernel-execution, and capture timeouts remain as infrastructure safeguards. A response without a valid tool call is retried twice; repeated failure terminates under a separately recorded guard. Budget ablations instead cap executed tool actions, including `finish`. For every episode, we retain the final workspace, termination origin, token usage, and complete action–observation trajectory.

B.1 AGENT PROMPTS AND TOOL INTERFACE

We use a fixed system prompt across tracks and vary only the task instruction. The following text is reproduced from the benchmark implementation; line breaks are changed for typesetting only.

System prompt.

```
You are an autonomous data-visualization agent working in a sandboxed
Python workspace.  Solve the task by calling tools, one call per
turn.
The Python kernel is persistent:  variables, imports and open figures
survive across calls, so you can build up state instead of re-running
everything.  Only files inside the workspace exist, and there is no
network access---no downloading data, no installing packages.
Look at your own output.  After rendering a chart, view the image and
check it against what the task asked for before you move on:  a
script that exits cleanly can still draw the wrong chart, mislabel an
axis, or silently drop a series.
```

```
Call finish when the task is complete, or when you are blocked and no
further tool call would help.  In that final message, say what you
did and what you verified visually---and if something is unfinished,
say plainly what and why.
```

Track B: Autonomous Repair.

```
chart.py builds a chart from the data in table.csv and saves it as
output.png.  The chart it currently produces is not faithful to that
data:  at least one thing about it contradicts what table.csv
contains or what summary.txt says the chart shows.  chart_current.png
is how the chart looks right now.
Find what is wrong, fix chart.py, and run it so that output.png shows
a chart that is faithful to the data.  Change only what is actually
wrong and leave the rest of the chart as it is.  Call finish when you
are done.
```

Track A: Diagnosed Repair. Track A uses the Track-B instruction followed by one of the following exact suffixes. For a single flaw, the suffix is

```
Diagnosis:  the flaw is that [diagnosis].  Repair exactly that.
```

For a k-flaw instance, it is

```
Diagnosis:  the chart carries k flaws:  (1) [diagnosis 1]; ...; (k)
[diagnosis k].  Repair exactly those.
```

The bracketed fields are instantiated with the category descriptions in Table 12; brackets are not included in the prompt.

Table 12: Diagnosis strings appended to the Track-A task instruction.

Flaw category	Diagnosis string
Wrong title	the chart title contradicts the provided data or summary
Wrong axis label	an axis label names a quantity other than the one plotted
Missing legend	the legend that identifies the plotted series is missing
Swapped legend labels	two legend entries are attached to each other's series
Value perturbation	one plotted value does not match its number in table.csv
Swapped series	two series plot each other's data
Dropped series	a series described in the data is missing from the chart
Wrong chart type	the chart or a series uses a chart type that contradicts summary.txt
Axis truncation	a value axis is cut so that some plotted data is not visible
Color contradiction	a plotted element's color contradicts the summary

Track C: Open-World Verification. The same instruction is used for both flawed and initially correct instances, so the prompt does not reveal their status.

```
chart.py builds a chart from the data in table.csv and saves it as
output.png; chart_current.png is how it looks right now.  The chart
may be faithful to table.csv and summary.txt, or one or more things
about it may contradict them.  Your first job is to decide which.
If the chart is faithful, do not edit chart.py:  run it once so that
output.png exists and call finish.  If something is wrong, fix
chart.py, change only what is actually wrong, run it so that
output.png shows a faithful chart, and call finish.
```

Tool interface. The opening observation also lists the available workspace files and the six tools. `execute_python` runs source code in the persistent kernel; `read_file`, `write_file`, and `list_files` expose workspace-local file operations; `view_image` returns a rendered file as a

multimodal image observation; and `finish` ends the episode. The modality ablations use the same instruction generator, which names only files actually present in the workspace. Tool schemas and generated task prompts are fixed in the released protocol and identified by checksums.

C USAGE OF LLMS

In this paper, we use LLMs to polish the writing and assist with the dataset generation. Specifically, we use a Mistral model to generate flawed data (details are in Appendix A).