An Empirical Study of Bugs in Data Visualization Libraries

WEIQI LU, The Hong Kong University of Science and Technology, China
YONGQIANG TIAN*, The Hong Kong University of Science and Technology, China
XIAOHAN ZHONG, The Hong Kong University of Science and Technology, China
HAOYANG MA, The Hong Kong University of Science and Technology, China
ZHENYANG XU, University of Waterloo, Canada
SHING-CHI CHEUNG, The Hong Kong University of Science and Technology, China
CHENGNIAN SUN, University of Waterloo, Canada

Data visualization (DataViz) libraries play a crucial role in presentation, data analysis, and application development, underscoring the importance of their accuracy in transforming data into visual representations. Incorrect visualizations can adversely impact user experience, distort information conveyance, and influence user perception and decision-making processes. Visual bugs in these libraries can be particularly insidious as they may not cause obvious errors like crashes, but instead mislead users of the underlying data graphically, resulting in wrong decision making. Consequently, a good understanding of the unique characteristics of bugs in DataViz libraries is essential for researchers and developers to detect and fix bugs in DataViz libraries.

This study presents the first comprehensive analysis of bugs in DataViz libraries, examining 564 bugs collected from five widely-used libraries. Our study systematically analyzes their symptoms and root causes, and provides a detailed taxonomy. We found that incorrect/inaccurate plots are pervasive in DataViz libraries and incorrect graphic computation is the major root cause, which necessitates further automated testing methods for DataViz libraries. Moreover, we identified eight key steps to trigger such bugs and two test oracles specific to DataViz libraries, which may inspire future research in designing effective automated testing techniques. Furthermore, with the recent advancements in Vision Language Models (VLMs), we explored the feasibility of applying these models to detect incorrect/inaccurate plots. The results show that the effectiveness of VLMs in bug detection varies from 29% to 57%, depending on the prompts, and adding more information in prompts does not necessarily increase the effectiveness. Our findings offer valuable insights into the nature and patterns of bugs in DataViz libraries, providing a foundation for developers and researchers to improve library reliability, and ultimately benefit more accurate and reliable data visualizations across various domains.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software libraries and repositories**; **Software testing and debugging**.

Additional Key Words and Phrases: Data Visualization Library, Empirical Study

ACM Reference Format:
Weiqi Lu, Yongqiang Tian, Xiaohan Zhong, Haoyang Ma, Zhenyang Xu, Shing-Chi Cheung, and Chengnian Sun. 2025. An Empirical Study of Bugs in Data Visualization Libraries. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE093 (July 2025), 24 pages. https://doi.org/10.1145/3729363

*Corresponding author.

Authors' Contact Information: Weiqi Lu, The Hong Kong University of Science and Technology, China, wluak@connect.ust.hk; Yongqiang Tian, yqtian@ust.hk, The Hong Kong University of Science and Technology, China; Xiaohan Zhong, The Hong Kong University of Science and Technology, China, elaine.zhong@connect.ust.hk; Haoyang Ma, The Hong Kong University of Science and Technology, China, haoyang.ma@connect.ust.hk; Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, Canada; Shing-Chi Cheung, The Hong Kong University of Science and Technology, China, scc@cse.ust.hk; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Canada.

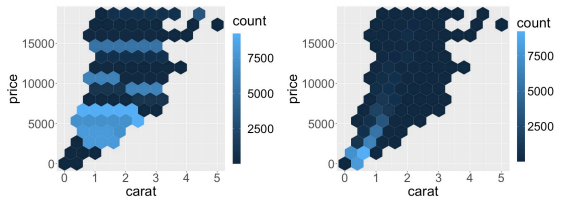
ACM 2994-970X/2025/7-ARTFSE093
https://doi.org/10.1145/3729363

1 Introduction

Data visualization (DataViz) is a representation technique employing various visual encodings, such as size, angle, color, and shape, to convey information derived from data effectively. This technique is extensively utilized across multiple domains, including presentation [59, 61], data analysis [45, 58, 92], and web applications [40, 55, 73]. To facilitate the transformation of raw data into meaningful graphic representations, various DataViz libraries have been developed, such as matplotlib [68], ggplot2 [51], and Chart.js [43]. These libraries are sophisticated visualization tools that offer a wide range of functionalities, including the capability to create different kinds of charts, 2D and 3D plots, images, and their combinations. These tools are designed to accurately depict insights, such as patterns and trends, extracted from raw data.

Despite the popularity of these DataViz libraries, their inherent complexity often leads to the emergence of bugs, which pose significant threats to both functional correctness and user experience. These bugs are generally identified by users and subsequently reported in the respective repositories of the libraries. A considerable number of these bugs can lead to incorrect functionalities, resulting in the misrepresentation of data. Such inaccuracies may foster false perceptions of the presented information, potentially leading to erroneous decision-making [70].

Figure 1 shows an example of data misrepresentation arising from a real bug in ggplot2. Users invoke the ggplot2 APIs to draw a hexagonal heatmap to show a two-dimensional distribution of 53,940 diamonds, with price in US dollars and carat weight as the dimensions of interest. This example demonstrates a typical use of DataViz libraries to visualize complex datasets that are difficult to interpret in text. However, the ggplot2 bug uses incorrect values to update the graphic parameters (*e.g.*, colors) of hexagons. The resulting plot (Figure 1a) incorrectly colors many deep blue hexagons as light blues. After the bug is fixed, the data are correctly visualized in Figure 1b, which contains only five light blue hexagons. This bug results in a misleading graphic representation of the relationship between diamond's price and carat. The `diamonds` dataset is large, comprising 53,940 entries. Users may not easily notice the anomaly, leading to incorrect data interpretation, analysis, and decision-making. For instance, diamond sellers may conclude from Figure 1a that diamonds ranging from 0.5 to 2.5 carats and priced between 5,000 and 7,500 are the most popular among consumers, suggesting a business strategy to prioritize importing these diamonds. However, the conclusion is incorrect and can lead to misguided inventory decisions and financial losses.



(a) Faulty Figure (b) Figure After Bug Fix

Fig. 1. An Illustrative Example of Data Misrepresentation (ggplot2 Issue #5037 [29]). Some hexagons in the heatmap are incorrectly colored, leading to the misrepresentation of the two-dimensional distribution.

The example above underscores the severe consequences that bugs in DataViz libraries can cause, highlighting the need for an empirical study to learn how these bugs can be better detected and prevented. DataViz libraries are distinct from other software libraries in their support of comprehensive data conversion into visual representations, requiring full integration of data processing, graphic computations, and rendering. Therefore, the insights from prior bug characterization studies in

other domains [57, 83, 84] are mostly inapplicable, necessitating a focused investigation into the specific nature of bugs in DataViz libraries. Existing research on DataViz libraries has primarily addressed data misrepresentation or misleading visualizations resulting from improper usage, with efforts focused on identifying the characteristics and impacts of such flawed visualizations [38, 85], promoting best practices [69], and enhancing the design of visualization tools to minimize syntactic errors and improve expressive flexibility [80, 88, 91]. However, *no prior studies examine the data misrepresentation caused by bugs in DataViz libraries*. Inspired by the bug characterization studies in other software domains [53, 64, 65, 83, 98], we study whether DataViz library bugs exhibit specific characteristics that facilitate their better detection and prevention by tool automation.

We gathered a dataset comprising 564 bugs reported across the five most widely used open-source DataViz libraries from September 2021 to August 2023. We carefully reviewed the associated bug reports, discussions between users and developers, patches, and test cases to identify the symptoms, root causes, and correlations between these factors, and make suggestions to prevent such bugs. Additionally, we analyzed the key steps necessary to reproduce the bugs in the reports and classify the types of test oracles employed. This comprehensive investigation aims to deepen our understanding of effective, efficient, and varied bug detection in DataViz libraries. Furthermore, leveraging recent advancements in Vision Language Models (VLMs), we explored the feasibility of applying these models to enhance testing methodologies for DataViz libraries.

Our study obtained the following **key findings**. (1) Incorrect/inaccurate plot is the *most prevalent* (39.89%) *symptom* in DataViz libraries, often involving issues with the presence, positioning, or visual properties of graphic elements like shape, color, and scaling. Detecting such bugs is critical to enhance the reliability of DataViz libraries and improve user satisfaction. (2) Incorrect graphic computation is the *major root cause*, primarily stemming from ignoring special graphic specifications, overlooking data boundaries, or writing incorrect expressions. Incorrect visual property updates and parameter mishandling are also frequent root causes. The code logic related to them should be comprehensively validated. (3) Visual property specification is the most *frequent trigger of bugs* in DataViz libraries. Automated testing techniques that systematically generate test programs with diverse visual property specifications are likely to effectively trigger bugs in DataViz libraries. (4) Two *image-related test oracles*, *i.e.*, *comparing with external images* and *comparing two figures generated by different programs* are commonly used. These oracles may facilitate future testing techniques for DataViz libraries. (5) VLMs may detect 29%∼57% of incorrect plots, depending on the prompts, and adding more information in prompts does not necessarily increase the effectiveness. Future research may carefully design effective prompts to automatically test DataViz libraries.

Contributions. This study made the following contributions.

- We conduct the first comprehensive study of DataViz library bugs by analyzing 564 bugs collected from five widely used DataViz libraries across different programming languages. The collected data are publicly available at [37].
- We provide a taxonomy of the symptoms and root causes of bugs in DataViz libraries. We identify key steps to trigger bugs and test oracles commonly used in DataViz libraries.
- Our feasibility study on applying VLMs to detect bugs in DataViz libraries demonstrates the potential for VLM-assisted bug detection, paving the way for future research in this area.
- We provide insights for developers on improving the maintenance of DataViz libraries and offer directions for future research focused on bug detection and repair.

2 DataViz Libraries

DataViz libraries aim to transform raw data into graphic representations according to the graphic specifications specified by users. While the architectures of DataViz libraries may vary based on factors such as programming languages, plotting grammar design, and supported functionalities,

their overall visualization procedures are similar. We illustrate this process with a matplotlib example in Figure 2, which visualizes the distribution of final scores in a class of 20 students, with each score interval spanning 20 units, *i.e.*, five equal-length intervals in a range from 0 to 100.

Given a set of data for visualization, users first interact with these libraries through a program consisting of plot APIs (*e.g.*, `matplotlib.pyplot` interface in matplotlib, `ggplot()` call in ggplot2) to specify how such data should be visualized. Such *graphic specifications* include the high-level graphic elements used for data visualization, along with their high-level visual properties. For example, the line with `plt.hist()` in Figure 2a instructs matplotlib to plot a histogram of `scores` with 5 `bins` in a `range` [0, 100]. In this specification, "histogram" is the high-level graphic element to be plot and `bins=5` and `range=(0, 100)` specify two high-level visual properties of the "histogram", *i.e.*, the number of bars in the "histogram" and the range of the x-axis representing scores.

Next, DataViz libraries perform *graphic computations* to transform data and high-level graphic specifications into a comprehensive set of *graphic elements* with proper visual properties. Although libraries utilize different architectures, all DataViz libraries share similarities in the core part of these computations: the mapping of data to core graphic elements that visually represent the data according to the graphic specifications. Such mapping focuses on transforming data into *visual properties* that dictate the display of these elements, particularly those reflecting data and statistics (*i.e.*, visual encodings), such as colors in a heatmap. Additionally, graphic computations encompass the creation of auxiliary graphic elements, such as titles, legends, and axes, to enhance the readability of visualized data. In Figure 2, the graphic computations are performed during the execution of the function `plt.hist()`, where the data `scores` is processed by tallying the number of students within each interval, and the result is stored in `hist_values`. Then, the processed data is mapped to graphic elements according to graphic specifications. Specifically, the student count in each interval is mapped to the corresponding bar (*i.e.*, rectangle), of which visual properties of `height` and `x` are determined by the count and interval, as in Figure 2b.

Finally, graphical *backends* are invoked by DataViz libraries to render these graphic elements on the screen, adhering to the specified visual properties from the previous step. Such backends serve as an implementation of rendering commands based on external graphics systems such as AGG [1], grid [7], PyQt [9], and so on. The majority of the backends, including AGG and grid, used by DataViz libraries are non-interactive and thus only produce static images, and some interactive backends like PyQt provide a Graphical User Interface (GUI) for user interaction. This overarching process ensures that data is effectively represented in visual form. In our example, the bars are rendered by the default backend of matplotlib according to their heights when executing `plt.show()`, resulting in a histogram that effectively depicts the score distribution in Figure 2c.

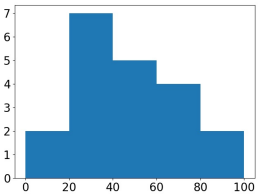
Difference from GUI libraries and graphics engines. DataViz libraries are specifically designed to visualize data using graphic representations, distinguishing them from other graphics-related libraries such as *GUI libraries* [62, 63, 72] and *2D/3D graphics engines* [47, 49, 52, 60]. While some DataViz libraries integrate *GUI* and *graphics engines* in their backends, their primary emphasis remains on mapping data to visual elements based on explicit user specifications while adhering to implicit visualization principles, such as avoiding overlap, ensuring proper layout and spacing, and applying appropriate annotations. In contrast, GUI libraries prioritize user interaction by managing the display and behavior of widgets along with event sequences, while graphics engines handle the rendering of 2D/3D objects based on their properties, such as positions, colors, and textures, ensuring graphical fidelity and physical realism under different conditions. Consequently, their bug characteristics differ: DataViz libraries are prone to errors in data-to-graphic transformation, while GUI libraries often encounter issues with handling user action sequences [62, 72], and graphics engines may face challenges in accurately rendering objects according to their visual properties [47].

```
1 import matplotlib.pyplot as plt
2 scores = [7, 14, 25, 28, 29, 32,
       33, 35, 38, 42, 49, 50, 50,
       51, 60, 61, 61, 74, 83, 96]
3 hist_values, bin_edges, patches =
       plt.hist(scores, bins=5,
       range=(0, 100))
4 plt.show()
```

(a) Graphic specification of histogram given by the code snippet

Variable	Value
hist_values	[2, 7, 5, 4, 2]
bin_edges	[0, 20, 40, 60, 80, 100]
patches	BarContainer object of 5 rectangles
Details of patches	
Rectangle 0	height = 2, x = 0, width = 20
Rectangle 1	height = 7, x = 20, width = 20
Rectangle 2	height = 5, x = 40, width = 20
Rectangle 3	height = 4, x = 60, width = 20
Rectangle 4	height = 2, x = 80, width = 20

(b) Intermediate variables and mapped graphic elements



(c) Final histogram rendered by backends

Fig. 2. An Illustrative Example of Visualization Process of DataViz Libraries.

3 Data Collection and Classification

This section introduces the steps in the collection and classification of bugs in DataViz libraries. To select the DataViz libraries for investigation, we first went through multiple lists of open-sourced DataViz libraries [6, 34], and take the following libraries into consideration: matplotlib, plotly.js, d3.js, Chart.js, ggplot2, plotly.R, plotters, Plots.jl, Vega-Lite, altair, seaborn, Vega, Ploty.py, and hGraph. Since our study aims to understand the bugs in general DataViz libraries, we excluded the libraries that are either domain-specific or built on other DataViz libraries, such as seaborn (statistic-specific library built on matplotlib) and Ploty.py (built on Ploty.js). We then grouped them according to their programming languages, and selected the one that has the most number of stars from each group. This strategy follows previous bug characterization studies [54, 84], and enhances the generalizability of this study across diverse programming languages and widely used libraries. In the end, five widely used DataViz libraries were selected for further investigation, *i.e.*, matplotlib [68] written in Python, ggplot2 [51] in R, plotters [76] in Rust, Plots.jl [75] in Julia, and Chart.js [43][1] in JavaScript.[2] Each of these libraries has a high number of stars, indicating their widespread community use. These libraries are well-documented with detailed descriptions of API usages, clear definitions and constraints of parameters, and example scripts demonstrating common use cases in their reference pages. These libraries are also actively maintained by their developers, as evidenced by a significant number of commits, the number of contributors, and their active discussions in issue-tracking systems. The comprehensive documentation and active maintenance status ensure that these libraries provide reliable and up-to-date information, making them well-suited for subsequent investigation. Table 1 shows the statistics of each selected library.

Following recent empirical studies [44, 83], we collected the bugs reported in the last two years from each selected library before our study, *i.e.*, from 1 September 2021 to 1 September 2023. The two-year window balances recency and data volume, capturing enough representative bugs while ensuring that the findings reflect current trends in bug characteristics, aiding ongoing software maintenance and future research on bug detection. Specifically, we collected 753 bugs from their issue-tracking systems and included only those accompanied by bug-fixing commits, which enables us to understand the root causes of bugs. We excluded those reports that are not actual bugs, by manually checking whether the developers explicitly stated or tagged in the issue tracking systems that the reported issue was a bug. If it was unable to be determined, we further inspected the commit messages and patches to decide whether it fixed a bug in the source code. A total of 189 issues unrelated to bugs, such as feature updates, documentation improvement, and error message improvement, were discarded. In total, 564 bugs are collected for investigation, ensuring a 95%

[1]Although D3.js has more stars than Chart.js, D3.js only has 3 fixed bugs in this duration and thus it is not selected. Among the 2,207 total issues of D3.js, only 32 are linked to pull requests, while the majority relate to inquiries about API usage.
[2]The capitalization style of their names are inconsistent among each other and we use the style in their official documentation.

confidence level with a 3.44% margin of error based on an estimated total of 1,850 reported bugs identified through keyword-based searches for the term "bug" in the issue tracking systems.

In the classification, one researcher first investigated 20% of randomly sampled bugs and proposed a draft of the taxonomy, including the categories and their definitions. Then all authors are involved in discussing and refining the proposed taxonomy. Later, based on the refined taxonomy, two researchers further individually investigated and classified all the collected bugs. During the classification process, two types of disagreements arose. Occasionally, a researcher misclassified an issue due to overlooking critical details. These disagreements were resolved through in-depth group discussions, where the associated bug reports and their resolutions were thoroughly reviewed to reach a consensus on the accurate classification. In other cases, certain issues did not align with any existing category. To address this, all researchers collaboratively discussed and defined a new category that appropriately encompassed the issue in question, as well as other potential similar issues. The final inter-rater agreement, measured using Cohen's Kappa coefficient, was 0.98, indicating a high level of consistency between the two researchers.

Table 1. The statistics of the DataViz libraries in this study (up to Sept 2024). In total, 564 bugs are studied.

Library	#Stars	#Commits	#Contributors	Language	#Collected Issues	#Studied Bugs
Chart.js [43]	64.2K	3.8K	486	JavaScript	125	112
matplotlib [68]	19.8K	18.5K	1490	Python	288	258
ggplot2 [51]	6.4K	2.1K	322	R	109	50
plotters [76]	3.7K	0.3K	99	Rust	35	19
Plots.jl [75]	1.8K	2.2K	237	Julia	196	125

4 Research Questions and Findings

Our study aims to answer the following research questions. We systematically study these questions using the collected data, conduct analyses, and make suggestions in the following subsections.

RQ1–Symptoms: *What are the symptoms of DataViz library bugs?* The symptoms of DataViz library bugs reveal how they manifest themselves in the visual output and user experience. Understanding these symptoms offers hints for the test oracle construction in test generation.

RQ2–Root Causes: *What are the root causes of DataViz library bugs?* Analyzing the root causes of DataViz library bugs reveals the underlying issues leading to incorrect or misleading visual representations. Understanding these root causes is crucial for crafting test inputs to trigger DataViz library bugs. We also investigate the correlations between symptoms and root causes, which is key to determining how to create inputs that deliberately produce a desired error symptom.

RQ3–Key Bug-Triggering Steps: *What are the key steps in bug-triggering programs to manifest bugs in DataViz libraries?* This may shed light on how to effectively trigger bugs in DataViz libraries.

RQ4–Test Oracles: *What are the test oracles used to verify if a bug is properly fixed in DataViz libraries?* Among the collected bugs, only 35.28% bugs are observed as crashes. Understanding oracles for non-crash bugs is necessary to design new testing techniques for DataViz libraries.

RQ5–VLM-aided Bug Detection: *How feasible is it to use vision language models for testing DataViz libraries?* Since VLMs have certain abilities to comprehend figures, they may be useful to aid incorrect/inaccurate plots in DataViz libraries. This research question aims to take the first step to investigate the feasibility of VLMs in this scenario.

4.1 Symptoms of Bugs in DataViz Libraries

We identify seven major bug symptoms in DataViz libraries. Among them, *Incorrect/Inaccurate Plot* is specific to DataViz libraries and the details of this symptom are discussed as follows.

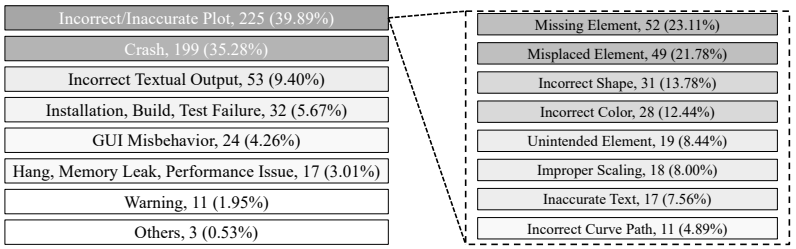


Fig. 3. Taxonomy and distribution of bug symptoms in DataViz libraries.

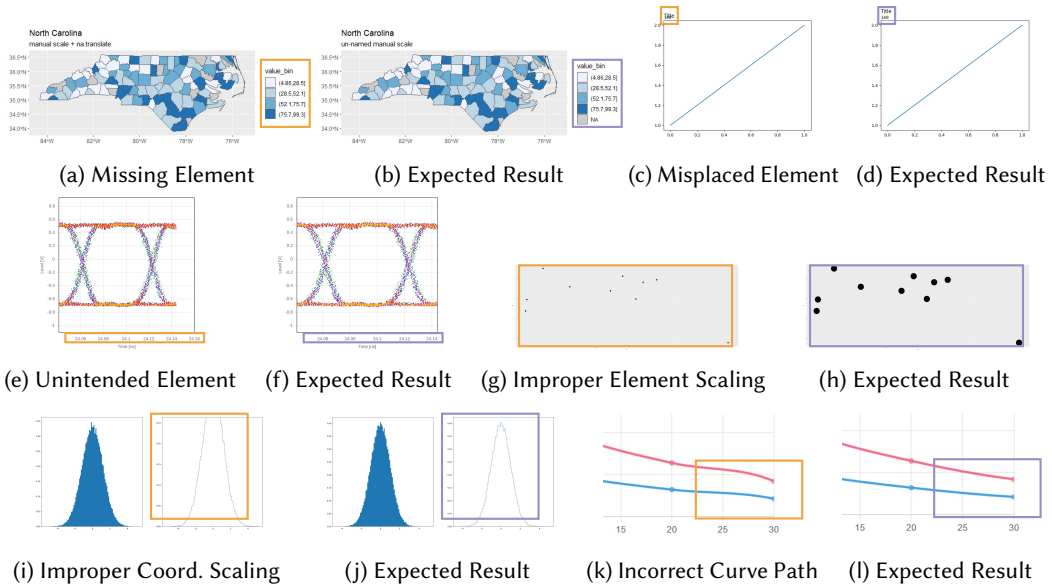
Incorrect/Inaccurate Plot. The actual graphic representation visually deviates from that specified by users or library documentation. This issue accounts for 39.89% (225/564) of collected bugs as shown in Figure 3. Our further analysis identifies eight common scenarios of this symptom:

(1) *Missing Element* (23.11% = 52/225): A graphic element, such as legend, label, or chart, is partially or completely missing. Figure 4a shows an example of the missing legend for gray areas.
(2) *Misplaced Element* (21.78% = 49/225): The position, depth, or rotation angle of a graphic element does not conform to graphic specifications or falls short of user expectations (*e.g.*, overlapping). Figure 4c shows an example that the title is misplaced and overlaps with the numeric text.
(3) *Incorrect Shape* (13.78% = 31/225): Shapes of graphic elements, such as length, width, style, and geometric form, are incorrect. For example, rectangular colorbars are plotted as triangles [12].
(4) *Incorrect Color* (12.44% = 28/225): The coloration of a graphic element does not adhere to the graphic specifications, fails to meet user expectations, or misrepresents the visualized data. For example, red lines may be incorrectly visualized as gray lines [19].
(5) *Unintended Element* (8.44% = 19/225): Elements intended to remain hidden or not explicitly requested by the user appear in the plot. Figure 4e shows an example that the axis tick of 24.16 is unintended as it is outside of the specified max limit.
(6) *Improper Scaling* (8.00% = 18/225): The coordinate scaling or visual element proportions are incorrect or deviate from user expectations. Figure 4g shows an example that the size of geometric points cannot be scaled up by specifying the size parameter. Figure 4i shows an example that the incorrect scaling of y coordinates causes a missing peak of the graph.
(7) *Inaccurate Text* (7.56% = 17/225): Poorly formatted or incorrect textual content is displayed in the plot. For example, the number +1.1e5 is incorrectly displayed as +1.1000000000e5 [13].
(8) *Incorrect Curve Path* (4.89% = 11/225): The geometric trajectory of a curve does not faithfully represent the underlying data or align with the specified axis scale. Figure 4k shows an example that the curve path at the end of the graph is incorrect.

Crash. The DataViz program exhibits unexpected termination. From Figure 3, crashes account for 35.28% (199/564) of collected bugs, making it the second most prevalent issue in DataViz libraries. In 93.97% (187/199) of crashes, programs terminate with error messages during code execution. The remaining 12 occur only after user interactions (*e.g.*, hovering, zooming). The small number of latter cases may be attributed to the fact that triggering these crashes is more challenging, as they are specifically tied to sequences of user interactions with the GUI elements.

Incorrect Textual Output. It includes inappropriate warnings, confusing error messages, *etc.*

GUI Misbehavior. The GUI window displays incorrect information or provides unexpected responses to user interactions. For example, matplotlib issue #24089 [67] reported that when using WebAgg backend to plot in Safari, dragging the corner to resize the figure does not work. Figure 3 shows that GUI misbehavior constitutes around 4.26% (24/564) of all collected bugs, most of which come from matplotlib and Chart.js that support interactive backends.



(a) Missing Element (b) Expected Result (c) Misplaced Element (d) Expected Result

(e) Unintended Element (f) Expected Result (g) Improper Element Scaling (h) Expected Result

(i) Improper Coord. Scaling (j) Expected Result (k) Incorrect Curve Path (l) Expected Result

Fig. 4. Examples of Incorrect/Inaccurate Plot. Each example shows a faulty (left) and the expected plot (right), where the orange box and purple box highlight the faulty area and the expected result, respectively.

Other Categories. 32 (5.67%) bugs are related to failures of installation, build, and testing processes. The remaining small proportion (5.50% = 31/564) of bugs fall into various categories. 3.01% (17/564) of the bugs are related to hangs, memory leaks, or performance issues, where code execution stalls, memory is not properly released, or performance is limited. 1.95% (11/564) are warnings from incorrect or deprecated API usage within the library's source code. Less than 1% of bugs are miscellaneous issues, such as no plot being displayed or misbehavior of utility functions.

Finding 1: *Incorrect/Inaccurate Plot* (39.89%) and *Crash* (35.28%) are two primary symptoms of DataViz libraries. *Incorrect/Inaccurate Plot* manifests in two distinct forms: (1) element-related issues, which involve missing, mispositioned, or redundant elements, and (2) property-related anomalies, which pertain to distortions in shape, color, and scaling.

4.2 Root Causes of Bugs in DataViz Libraries

This section presents the major root causes of DataViz library bugs and discusses the findings.

Incorrect Graphic Computation (25.71% = 145/564). This refers to the incorrect computation of attributes, intermediate variables, or high-level features (*e.g.*, layout) related to the visual properties of graphic elements. It is the major root cause of bugs in DataViz libraries, accounting for 25.71% (145/564) of all issues. The primary subcategories are outlined below, accounting for the majority of cases, with the remaining 3.45% distributed across additional minor subcategories not listed here.

(1) *Unhandled Data Boundary* (34.48% = 50/145): Special variable values are not properly handled. For example, Figure 6a shows division by zero when `this._pointLabels.length==0`. The patch replaces the length with 1 if it is `0`, preventing `angleMultiplier` from being infinite.

(2) *Unhandling of Supported Graphic Specification* (31.72% = 46/145): Special cases of the supported graphic specifications are not properly handled. In Figure 6b, `lim` specifies the range, *i.e.*, minimum and maximum values of the polar coordinate axis, but the buggy line overlooks the

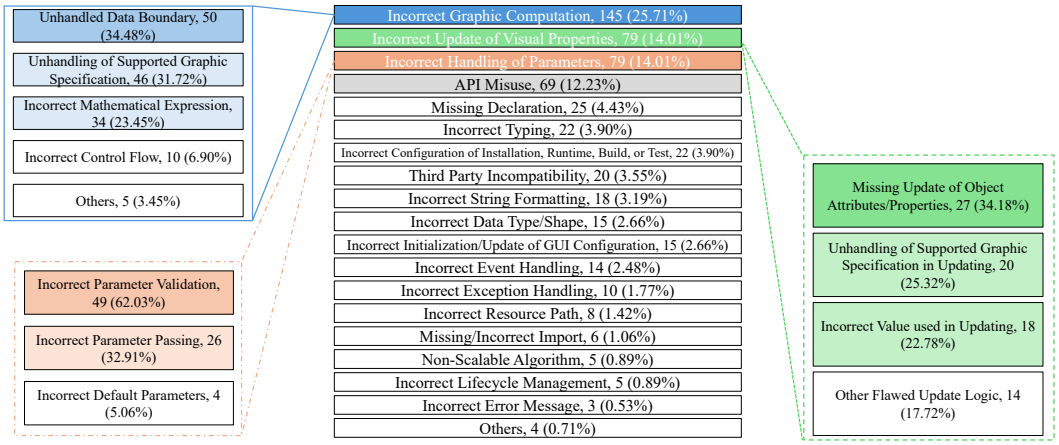


Fig. 5. Taxonomy and distribution of root causes of bugs in DataViz libraries.

minimum value when determining the axis ticks of the full circle. The overlook of the axis range specification causes the incorrect plot when the minimum value is non-zero.

(3) *Incorrect Mathematical Expression* (23.45% = 34/145): The mathematical expression for an attribute is incorrect due to rounding error, incorrect usage of math operator, wrong indexing, or variable misuse. As shown in Figure 6c, the computation of `cursor.y` does not include the variable `padding`, causing the GUI misbehavior when hovering on the legend.

(4) *Incorrect Control Flow* (6.90% = 10/145): The graphic computation is incorrect due to the wrong control flow, which is often caused by incorrect conditional expressions.

```
1 @@ -371,8 +371,9 @@ export default class
      RadialLinearScale extends LinearScaleBase
       {
2 ---  const angleMultiplier = TAU / this.
      _pointLabels.length;
3 +++  const angleMultiplier = TAU / (this.
      _pointLabels.length || 1);
```

(a) Unhandled Data Boundary (Chart.js Issue #10019 [21]). Division by zero occurs when length of point labels is zero.

```
1 @@ -298,7 +298,7 @@ def set_axis(self, axis):
2      if _is_full_circle_deg(lim[0], lim[1]):
3 ---      return np.arange(8) * 2 * np.pi / 8
4 +++      return np.deg2rad(min(lim)) + np.arange(8) *
       2 * np.pi / 8
```

(b) Unhandling of Supported Graphic Specification (matplotlib Issue #25568 [32]). The specified limit of the polar coordinate axis is not considered when determining the axis ticks of the full circle.

```
1 @@ -428,7 +428,7 @@ export class Legend extends Element {
2 ---      cursor.y += calculateLegendItemHeight(legendItem, fontLineHeight);
3 +++      cursor.y += calculateLegendItemHeight(legendItem, fontLineHeight) + padding;
```

(c) Incorrect Mathematical Expression (Chart.js Issue #11272 [33]). The expression of y position of the cursor does not include the variable padding.

Fig. 6. Examples of subcategories of *Incorrect Graphic Computation* with patches.

Incorrect Update of Visual Properties. The code that involves updating attributes or properties of visual elements or systems is incorrect due to wrong assignment, false conditional check, or others. 14.01% (79/564) of collected bugs are caused by this root cause. The reason for such prevalence is that property updates are common in the construction, combination, and modification process of graphic elements. We further categorize the fine-grained root causes as follows.

(1) *Missing Update of Object Attributes/Properties* (34.18% = 27/79): The update of attributes of a visual element is missing or called at the wrong stage. As shown in Figure 7a, creating colorbar axes will automatically create an auxiliary grid that is intended to be hidden from users, but the buggy code does not update the `visible` of grid to be `False`.
(2) *Unhandling of Supported Graphic Specification in Updating* (25.32% = 20/79): The special cases of the graphic specification are overlooked, where the update of visual properties should be handled separately. In Figure 7b, the parameter `tight` can be both `True` and `False`, but the buggy code only handles the specification of setting tight layout while overlooks disabling it.
(3) *Incorrect Value used in Updating* (22.78% = 18/79): The value used to update the object attribute is wrong or the update should not happen. In Figure 7c, the solid `facecolor` from the parent figure is also applied to the subfigures, resulting in obscuring the suptitle text. Therefore, `facecolor` of subfigures should be instead updated to `"none"`, *i.e.*, transparent.
(4) *Other Flawed Update Logic* (17.72% = 14/79): Miscellaneous issues that cause the update to be unsuccessful, wrong, or repetitive, such as incorrect update orders.

```
@@ -1143,6 +1143,7 @@ def colorbar(
       cax, kwargs = cbar.make_axes(
       ax, **kwargs)
+++    cax.grid(visible=False, which
       ='both', axis='both')
```

(a) Missing Update of Object Attributes/Properties (matplotlib Issue #21723 [16]). Visibility of grid should be `False` after colorbar is created.

```
@@ -2760,9 +2760,9 @@ def set_tight_layout(self, tight):
+++ _tight = 'tight' if bool(tight) else 'none'
    _tight_parameters = tight if isinstance(tight, dict) else {}
--- if bool(tight):
---     self.set_layout_engine(TightLayoutEngine(**_tight_parameters
      ))
+++     self.set_layout_engine(_tight, **_tight_parameters)
```

(b) Unhandling of Supported Graphic Specification in Updating (matplotlib Issue #22847 [23]). The buggy code only handles the specification of setting tight layout while overlooks disabling it.

```
@@ -2176,7 +2176,7 @@ def __init__(self, parent, subplotspec, *,
    if facecolor is None:
---    facecolor = mpl.rcParams['figure.facecolor']
+++    facecolor = "none"
```

(c) Incorrect Value used in Updating (matplotlib Issue #24910 [31]). The `facecolor` should be updated to transparent ("none") to avoid obscuring other elements.

Fig. 7. Examples of subcategories of *Incorrect Update of Visusal Properties* with patches.

Incorrect Handling of Parameters. This refers to the root causes related to input parameter handling, including incorrect default values, wrong parameter validation, and incorrect passing of parameters to the next function. 14.01% (79/564) of the bugs fall into this category.
(1) *Incorrect Parameter Validation* (62.03% = 49/79): The parameters from user input or function call are incorrectly checked for validity, which may lead to unexpected behavior in the subsequent execution of code. In Figure 8a, elements in array `stroke_size` may be missing or `NULL`, and they should be validated and replaced with zero values.
(2) *Incorrect Parameter Passing* (32.91% = 26/79): The parameters of a function are not correctly passed to the next function. In Figure 8b, the number format `this.options.ticks.format` is not passed to the function `formatNumber()`, resulting in incorrect number formatting.
(3) *Incorrect Default Parameters* (5.06% = 4/79): The default parameters of a function are incorrect. Figure 8c shows a bug that the parameters `pad` and `sep` of `OffsetBox` are incorrectly initialized as the default value `None`, which is not a valid value for subsequent use of `pad` and `sep`.

API Misuse. The usage of internal APIs of DataViz libraries or external APIs from third party libraries are incorrect. As depicted in Figure 5, API misuse accounts for 12.23% (69/564) of total bugs, and matplotlib is more severely impacted by this root cause compared with other libraries.

```
@@ -122,6 +122,8 @@ GeomPoint <-
        ggproto("GeomPoint", Geom,
      coords <- coord$transform(
       data,
      panel_params)
+     stroke_size <- coords$
       stroke
+     stroke_size[is.na(stroke_
       size)] <- 0
```

(a) Incorrect Parameter Validation (ggplot2 Issue #4624 [20]). Missing values in `stroke_size` are not properly checked and replaced.

```
@@ -312,6 +312,6 @@ export default
       class LinearScaleBase
      extends Scale {
 getLabelForValue(value) {
-    return formatNumber(value,
       this.chart.options.locale);
+    return formatNumber(value,
       this.chart.options.locale,
       this.options.ticks.format);
```

(b) Incorrect Parameter Passing (Chart.js Issue #9830 [22]). The number format of `ticks` is not passed to `formatNumber()`.

```
@@ -369,15 +369,15 @@ def draw(
      self, renderer):
 class PackerBase(OffsetBox):
-    def __init__(self, pad=None
       , sep=None, width=None,
       height=None,
+    def __init__(self, pad=0.,
       sep=0., width=None, height=
       None,
```

(c) Incorrect Default Values of Parameters (matplotlib Issue #24623 [26]). The pad and sep are incorrectly initialized.

Fig. 8. Examples of subcategories of *Incorrect Handling of Parameters* with patches.

The remaining root causes, each contributing less than 5% of the total collected bugs, form a long-tail distribution. Their definitions are introduced as follows, excluding self-explanatory causes.

(1) *Missing Declaration*: The declaration of an attribute, method, or property is absent or misplaced.
(2) *Incorrect Typing*: The variable type, class, or function header is not properly declared or defined.
(3) *Third Party Incompatibility*: Conflicts between the DataViz library and a third-party component.
(4) *Incorrect String Formatting*: The construction of strings involving variables is incorrect.
(5) *Incorrect Data Type/Shape*: It involves incorrect or lack of type conversion or data reshaping.
(6) *Incorrect Initialization/Update of GUI Configuration*: The initialization or update of various aspects of the graphical user interface (GUI) configuration is done incorrectly.
(7) *Incorrect Event Handling*: The action event signaled by users is not properly handled.
(8) *Incorrect Exception Handling*: The program fails to raise or handle exceptions properly.
(9) *Non-Scalable Algorithm*: An algorithm poorly suited for handling large or growing input sizes.
(10) *Incorrect Lifecycle Management*: Variables or memory resources are not properly managed throughout their lifecycle, leading to issues such as memory leaks or incorrect behavior.

Finding 2: DataViz library bugs are primarily manifested by four root causes: *Incorrect Graphic Computation* (25.71%), *Incorrect Update of Visual Properties* (14.01%), *Incorrect Handling of Parameters* (14.01%), and *API Misuse* (12.23%), together accounting for 65.96% of all bugs. Notably, the top three categories share a common underlying issue, *i.e.*, insufficient consideration of different combinations of graphic specifications imposed by parameter settings. Moreover, value-related issues, including unhandling of data boundary, incorrect update values, and incorrect default parameters, are frequently observed within these categories.

Correlations between Symptoms and Root Causes. We utilize parallel sets [39] to illustrate the correlations between symptoms and root causes, as in Figure 9. Each bug is represented by a (symptom, root cause) pair. We measure the occurrences of each pair and exclude those with fewer than five occurrences for clear presentation. We notice that Incorrect/Inaccurate Plot is typically caused by errors in graphic computation or updating of visual properties, and thus validating the correctness of these computation and updating steps is critical to the reliability of DataViz libraries. Meanwhile, Crash can result from ten different root causes, with the most common being Incorrect Handling of Parameters, Incorrect Graphic Computation, and API Misuse. Therefore, detecting crash bugs in DataViz libraries can be effective in revealing bugs of different root causes.

4.3 Key Steps to Trigger Bugs in DataViz Libraries

This research question aims to understand how bugs in DataViz libraries are triggered by users. Through the investigation of common patterns of plot examples in the official documentation [2, 3, 5,

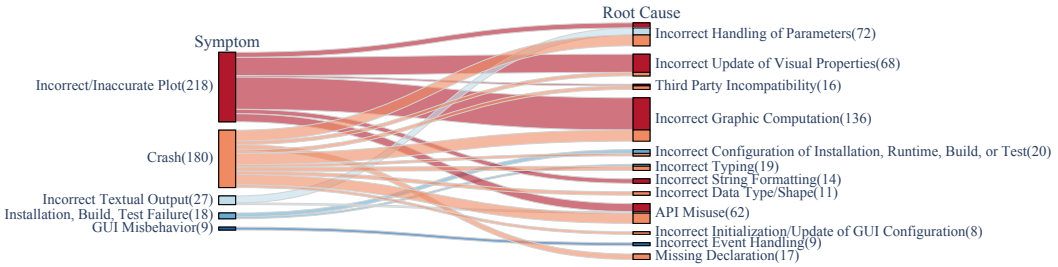


Fig. 9. The correlations between symptoms and root causes of bugs in DataViz libraries. The numbers in the parentheses are the counts of bugs in the corresponding categories.

8, 11] and frequent keywords observed in the bug reports with bug-reproducing tests, we notice that a complete procedure used by users to visualize data can be decomposed into eight steps and each step of them may trigger bugs. These steps include library and module import, backend selection, plot configuration, data preparation, text annotation, visual property specification, graphic update, and user interaction. Although their order varies depending on the API design of the specific libraries, the general pattern of these steps is similar across different DataViz libraries. Code Example 1 shows examples of each step using matplotlib, except user interaction realized by interacting with the GUI window such as zooming in and out.

We conduct a manual analysis to understand which step of the eight steps triggers bugs. Such understanding is important as it can facilitate the design of test input in automated testing. For example, if a specific step is likely to trigger a bug, automated testing techniques can focus on such a step in test generation. We first collect all bug-triggering procedures from GitHub Issues. Under the observation that Incorrect/Inaccurate Plot and Crash are the two most common symptoms, this research question only focuses on the 424 bugs under these two symptoms. Then we manually analyze their bug-triggering procedures (including programs) from GitHub Issues, to identify which step of them triggers bugs. We exclude those bugs without bug-triggering procedures. In total, 407 bugs are studied. Figure 10 shows their distribution over eight steps.

```
import matplotlib.pyplot as plt;import numpy as np # 1. Library and Module Import
plt.switch_backend('TkAgg') # 2. Backend Selection: use an interactive backend
fig, ax = plt.subplots(figsize=(8, 6)) # 3. Plot Configuration: create a figure and axes with specific size
x = np.linspace(0, 10, 100); y = np.sin(x) # 4. Data Preparation: generate some data for visualization
ax.set_title('Interactive Sine Wave Example') # 5. Text Annotation: add title
ax.plot(x, y, color='blue', linestyle='-') # 6. Visual Property Specification: draw blue solid lines
ax.set_xlim(0, 5) # 7. Graphic Update: change the x limits from [0,10] to [0,5]
plt.show() # Display the plot
```

Code Example 1. Examples of seven steps of data visualization. This matplotlib example plots a sine wave.

(1) *Visual Property Specification* (36.36% = 148/407). Function arguments specifying visual properties with specific numerical, categorical, or other forms of values cause the bug to manifest. This step is particularly error-prone as 44% (65/148) of bugs are reproduced by specifying a certain categorical attribute, while 30% (44/148) are triggered by a certain numerical value.
(2) *Data Preparation* (17.94% = 73/407). Data with particular characteristics, such as a specific value, type, size, range, and pattern, may trigger bugs. Value (29% = 21/73), type (26% = 19/73), size (15% = 11/73), and range (11% = 8/73) are the four most common characteristics of data to trigger bugs.
(3) *Graphic Update* (16.22% = 66/407). Updating certain graphic properties or elements with APIs triggers the bug. Such updates involve complex computations of elements and updates of their

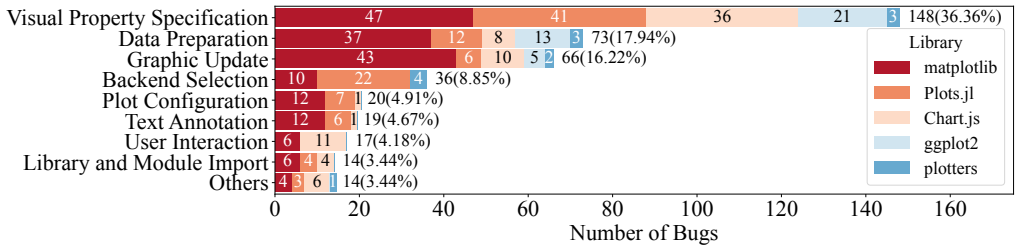


Fig. 10. The distribution of key steps for triggering bugs in DataViz libraries

properties (*e.g.*, positions and sizes). If these computations and updates do not account for existing elements in the plot and how the graphic update integrates with them, bugs are likely to be induced.

(4) *Backend Selection* (8.85% = 36/407). Different backends have diverse implementation. The choice of backends may trigger bugs, affecting how the plot is displayed or rendered.
(5) *Plot Configuration* (4.91% = 20/407). Global configurations like layout settings trigger bugs.
(6) *Text Annotation* (4.67% = 19/407). Annotation, such as labels, titles, and axis ticks, when specified with a certain length, format, content, or other patterns of texts, can trigger the bug.
(7) *User Interaction* (4.18% = 17/407). GUI actions like zooming, hovering, and clicking, trigger bugs.
(8) *Library and Module Import* (3.44% = 14/407). Bugs are triggered when using specific modules within the library or when integrating with external libraries.
(9) *Others* (3.44% = 14/407). Miscellaneous categories not explicitly covered above. For example, users manually customize graphic elements without the aid of dedicated APIs, resulting in bugs. These include cases where developers manually customize the behavior of graphical elements without the aid of dedicated APIs, as well as instances where infrequently used plotting functions are invoked.

We further examine how these key bug-triggering steps are connected to root causes using parallel sets in Figure 11. We notice that (1) Visual property specification triggers the greatest number and variety of bugs. Meanwhile, this step and data preparation are most likely to trigger bugs of the major root cause Incorrect Graphic Computation. Mutating specifications and data can be an effective way to detect DataViz library bugs, especially those related to Incorrect Graphic Computation. (2) Graphic updates can reveal bugs related to Incorrect Updates of Visual Properties and Incorrect Graphic Computation, as such updates usually invokes re-computing graphic elements.

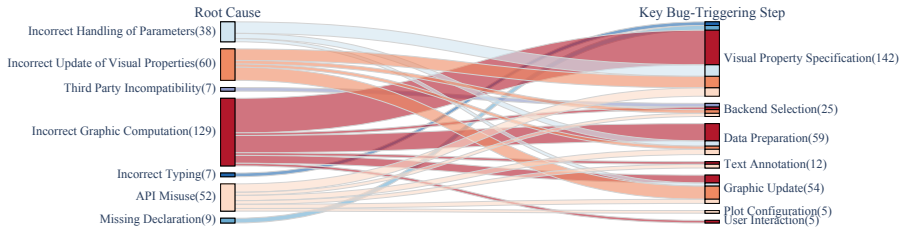


Fig. 11. The correlations between root causes and key steps to trigger bugs in DataViz libraries. The numbers in the parentheses are the counts of bugs in the corresponding categories.

Finding 3: Visual Property Specification is the most frequent trigger of bugs in DataViz libraries, with specifying certain categorical or numerical attribute values being the most common practice. Additionally, input data with specific characteristics and graphic updates are also frequent triggers.

4.4 Test Oracles

In this section, we examine the test oracles used in DataViz libraries. As discussed in §2, one key distinction of DataViz libraries is that their outputs are visual representations of data, unlike other software without graphical output, such as compilers [83, 84]. Among software with graphical output, oracles of DataViz libraries differ from others (see discussion at the end of this RQ). Validating the correctness of DataViz libraries requires test oracles capable of handling visual outputs and verifying DataViz-specific properties, which differs from oracles used in other types of software.

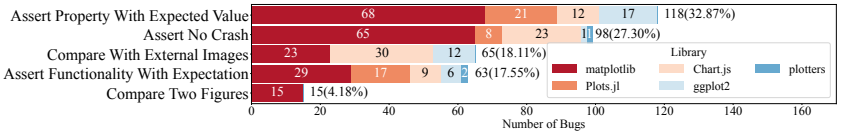


Fig. 12. The distribution of test oracles used in DataViz libraries

We manually investigate all 564 collected bugs and identify 359 fixing commits that involve test case modifications. We then analyze the test oracles of these test cases and categorize them into five groups, as shown in Figure 12. *Assert Property With Expected Value* (32.87%=118/359) is the most common oracle. This oracle asserts the value of a certain graphic property to verify the correctness of DataViz libraries. *Assert No Crash* (27.30%=98/359) is the second most common one, which ensures test programs run without errors or crashes. *Assert Functionality With Expectation* (17.55%=63/359) checks if a specific functionality aligns with the anticipated behavior. For example, function `np.testing.assert_allclose` in Figure 13a ensures that the positions of `ax` remain unchanged after removing the colorbar, as "removing colorbar" should not affect the axes location.

In addition to the aforementioned oracles, which explicitly verify the correctness of graphic properties/functionalities and ensure the absence of crashes, we identify two additional oracles that implicitly assess the correctness of properties/functionalities through image comparison.

```python
def test_remove_from_figure_cl()
    ...
    pre_position = ax.
      get_position()
    cb = fig.colorbar(sc)
    cb.remove()
    fig.draw_without_rendering()
    post_position = ax.
      get_position()
    np.testing.assert_allclose(
      pre_position.get_points(),
      post_position.get_points())
```

(a) Assert Functionality With Expectation (matplotlib Issue #20978 [15])

```python
@image_comparison(['
      colorbar_keeping_xlabel.
      png'], style='mpl20')
def test_keeping_xlabel():
    arr = np.arange(25).reshape
      ((5, 5))
    fig, ax = plt.subplots()
    im = ax.imshow(arr)
    cbar = plt.colorbar(im)
    cbar.ax.set_xlabel('Visible
       Xlabel')
    cbar.set_label('YLabel')
```

(b) Compare With External Images (matplotlib Issue #23398 [27]) using *@image_comparison*.

```python
@check_figures_equal(extensions=('
      png',))
def test_no_subslice_with_transform
      (fig_ref, fig_test):
    ax = fig_ref.add_subplot()
    x = np.arange(2000)
    ax.plot(x + 2000, x)
    ax2 = fig_test.add_subplot()
    t = mtransforms.Affine2D().
      translate(2000.0, 0.0)
    ax2.plot(x, x, transform=t+ax.
      transData)
```

(c) Compare Two Figures (matplotlib Issue #21008 [17]) using decorator *@check_figures_equal*.

Fig. 13. Examples of Test Oracle Types

Compare With External Images (18.11% = 65/359). This oracle verifies the correctness of DataViz libraries by pixel-wise comparison of generated images with the reference images of which correctness has been validated by developers. The reference images are usually generated by the fixed version of the DataViz library and saved in code repositories, along with test cases. This method is also known as *visual regression/snapshot testing* [46]. Figure 13b shows an example where the behavior is verified by comparing the generated image with an external reference image.
Compare Two Figures (4.18% = 15/359). This oracle verifies if two figures generated under different conditions are the same, which follows the concept of *metamorphic testing* [81]. Figure 13c shows a test case using this oracle. This example ensures that two generated figures are identical under equivalent transformations, thereby verifying the expected behavior in a specific scenario. Unlike *Compare With External Images*, where reference images are pre-generated by DataViz libraries before test execution, *Compare Two Figures* generates them dynamically during test execution. Therefore, this oracle only guarantees that two figures are identical, but the correctness of the reference images is not verified by developers. It is possible that both images are incorrect.

Through our analysis, we noticed the following difference between oracles used in DataViz libraries and other graphical software. The oracles in DataViz libraries primarily focus on the accurate transformation from data into visual encodings, such as position, size, color, and shape. One unique visual oracle revealed in our findings is comparing two figures under equivalent transformations, such as altering the data or changing the position of elements. In contrast, oracles in rendering engines focus on image quality (*e.g.*, sharpness, color accuracy, and fidelity), visual consistency across different devices and screen sizes, and properties like animation smoothness and frame rates during rendering [42, 66]. Oracles in VR/AR software focus on the interaction between users/environment and interface, application responsiveness, object movement, and so on [77].

Finding 4: In addition to explicit assertions regarding property values and implicit checkers for crashes, DataViz libraries leverage two image-based oracles *Compare With External Images* and *Compare Two Figures* to implicitly verify the correctness of the visual properties.

4.5 Bug Detection by Vision Language Models

In this research question, we aim to investigate to what extent VLMs can help detect incorrect/inaccurate plots. Incorrect/inaccurate plots, the most prevalent symptom, intensify the difficulties in automated testing DataViz libraries, since validating the correctness of plots requires comprehensive understanding of code intention, data characteristics, and visual representation. As we mentioned in §4.4 , in practice, developers have to manually validate the plots in regression testing [10], which is both time-consuming and prone to false positives if a new version introduces changes unrelated to the buggy component. In contrast, other failures like crashes can be easily observed. To our best knowledge, there is no automatic techniques specialized for visual bugs. Recently, the advent of Vision Language Models (VLMs) offers a potential solution for automatically detecting incorrect/inaccurate plots, since they can interpret and comprehend both code snippets and visual representations through multimodal understanding. This unique capability has demonstrated superior performance over conventional computer vision techniques in various tasks [56, 78]. These advancements motivate an investigation into whether VLMs can be utilized for automated validation of visual representation. Specifically, VLMs are multimodal large language models that process visual and textual data by treating them as input tokens, enabling the generation of coherent subsequent texts [90]. For example, GPT-4V [96] leverages this approach to handle both modalities seamlessly. However, it is unclear to what extent VLMs can facilitate detecting incorrect/inaccurate plots. To explore this, we design an experiment to assess VLMs'

ability to identify incorrect/inaccurate plots using three types of prompts, all of which include the image URL (omitted below) and can be easily integrated into automated testing pipelines.

Image. Only the generated plot is provided to investigate whether VLMs can detect incorrect/inaccurate plots. This prompt mimics a common use case where developers may leverage VLMs to identify potential anomalies in plots, assuming VLMs possess basic DataViz knowledge.

Goal Description You are provided with a plot generated by the data visualization library `[name]`. The goal is to determine whether the plot is inaccurate or incorrect based on common knowledge and visual cues, without having access to the underlying code. Your task is to identify any anomaly or obvious bug in the plot using common sense and basic understanding of data visualization principles.

Image+Test. By feeding both the generated plot and test programs into VLMs, this prompt is to investigate if additional information from test programs facilitates the bug detection of VLMs.

Goal Description You are provided with a plot generated by the data visualization library `[name]`, along with the code used to create the plot. Your task is to determine whether the generated plot is inaccurate or incorrect according to the specifications in the code. To determine the correctness: (1) Review the provided code to understand the expected features and specifications of the plot. (2) Evaluate the generated plot to determine if it matches the expectations set by the code. (3) Identify any anomaly or inconsistency in the plot that indicates potential bugs in the data visualization library.

Test Program `[Test Program Code]`

Important Notes (1) You CANNOT execute the given code to determine the expected plot. (2) Assume the given code is correct. Do not fix the code.

Image+Test+Hint. In addition to the generated plot and test program, this prompt leverages a hint, such as "Focus on the colorbar", to guide VLMs to focus on certain problematic area of the plot, *e.g.* colorbar. These hints, extracted from bug reports, assist VLMs in focusing on specific area, rather than being misled by non-problematic ones. We acknowledge that these hints may not be available before bugs are reported. However, this prompt help us to investigate whether VLMs can realize the appearance of incorrectness plots when specific hints are given. If VLMs still cannot detect incorrect plots given these hints, it suggests that their limitations are not solely due to the complexity of visual figures, but may instead stem from intrinsic limitations in comprehension.

Goal Description + Test Program + Important Notes (Details are the **same** as Image+Test)

Hint Focus on the `[Name of Problematic Visual Element]`.

(a) Any (b) Majority Voting

Fig. 14. Distribution of bugs detected by different strategies of VLM prompting. *Any* refers to bugs detected in at least 1 out of 3 responses, and *Majority Voting* refers to bugs detected in at least 2 out of 3 responses.

We utilize GPT-4o [82], a widely used VLM, as the model under investigation. From a population of 225 bugs exhibiting symptoms of incorrect or inaccurate plots, we randomly select 100 samples for analysis. This sampling approach provides a 95% confidence level with a 7.4% margin of error. These bugs are then tested using the above three prompts.

To reduce response variability in VLMs, each experiment is run three times. We apply *Majority Voting*, considering GPT-4o successful if at least two responses identify the bug [50, 74, 86]. We also report results under the *Any* criterion, where success requires only one correct response. As

shown in Figure 14, GPT-4o detects 82 bugs under *Any* criterion and 68 under *Majority Voting*. For brevity, we will focus on analyzing *Majority Voting* results, as *Any* criterion yields similar findings.

The prompt Image detects 40 bugs, achieving a detection rate of 40%. We hypothesize that VLMs leverage their basic knowledge of data visualization to recognize common visual anomalies, such as missing labels and overlapping elements. However, this prompt struggles with issues that require an understanding of graphic specifications, as it lacks access to the corresponding test code.

By incorporating the test program, Image+Test prompt identifies 13 additional bugs that are not detected by Image. These bugs often involve properties explicitly specified in the test code, such as the coexistence of multiple plot types (e.g., histogram and curve) within the same axes. *Surprisingly*, Image+Test fails to detect 24 bugs that are successfully identified by Image. We conjecture this is because Image+Test requires VLMs to both detect anomalies in plots and validate consistency between code and plots. As a result, VLMs may lose their focuses in such complex tasks, resulting in failure in bug detection. Overall, this prompt detects only 29 bugs (detection rate 29%).

Image+Test+Hint prompt detects 15 additional bugs that are missed by both Image and Image+Test, leading to the highest detection rate of 57%. We hypothesize that directing VLMs' attention to specific problematic visual elements simplifies the otherwise complex task of aligning graphic specifications with the plot. Furthermore, this approach enables VLMs to validate both explicitly specified properties and implicit visual constraints based on general domain knowledge, thus recovering 15 bugs that Image detects but Image+Test overlooks.

Despite the strengths of VLMs in detecting visualization bugs, they still exhibit notable limitations. In our evaluation, the three prompting strategies collectively identify 68 unique bugs, yet 32 remain undetected by any approach. We identify two major factors contributing to these failures.

(1) *Failure in detecting subtle symptoms*: VLMs struggle to detect visualization bugs manifesting as small deviations in the positions, shapes, or colors of graphical elements, such as minor misalignments of tick labels [25], slight shifts in points [18], minus signs shown as dashes [14], and missing coloring of the exponent label for red y-ticks [30].
(2) *Failure in interpreting data trends from visual properties*: VLMs struggle to accurately interpret data trends encoded through visual properties, such as color distributions in heatmaps [28] or patterns in line charts [24]. Detecting the anomalies in these issues requires understanding trends from two perspectives: the underlying data itself as shown in text, and the visual trends in the plot, both of which remain challenging for current VLMs.

These limitations highlight the need for further advancements in VLM capabilities, such as improved multimodal reasoning and better understanding of data-related visual properties.

Finding 5: VLMs can detect 40% of bugs with the Image prompt, leveraging their general visualization knowledge. The Image+Test+Hint prompt further improves detection rate to 57% by providing code context and directing attention to problematic elements. Surprisingly, the Image+Test prompt achieves the lowest detection rate at 29%, possibly because the combined presentation of image and code without explicit guidance increases the complexity of the task and reduces the VLMs' effectiveness in identifying critical visual discrepancies.

5 Discussion

5.1 Implications

Finding 1 highlights the prevalence of Incorrect/Inaccurate Plots and Crashes in DataViz libraries, emphasizing the need for addressing both during development and maintenance. Furthermore, Finding 1 identifies both element-level and property-level aspects of these plot errors, offering guidance for vision-based bug detection. A structured testing approach may first *verify the presence of essential elements* (*e.g.*, charts and annotations), and then *assess their visual properties* (*e.g.*, shape,

size, and color). Plot errors also vary in scope: issues like missing elements or color/shape anomalies affect only the buggy element itself, while mispositioning, redundancy, or scaling errors can disrupt surrounding elements. This suggests that *inter-element relationships* should be carefully considered during validation. Leveraging VLMs with spatial reasoning capabilities could be a promising approach to enhance automated detection and analysis of these errors.

Finding 2 identifies the three most prevalent root causes of DataViz library bugs: Incorrect Graphic Computation, Incorrect Update of Visual Properties, and Incorrect Handling of Parameters. From their subcategories, it can be observed that these root causes are partially linked to the handling of graphic specifications and parameter/variable values. This implies that during parameter validation, graphic computations, and property updates, developers should *systematically consider the corner cases of graphic specifications*, such as special layouts, non-linear transformations, and customized visual properties. Additionally, they should carefully evaluate the feasibility and validity of special or boundary values, such as the existence of zero value of the visual property `length` in Figure 6a and the choice of the transparent color represented as `"none"` in Figure 7c. We observe that such issues often involve conditional statements; therefore, leveraging pattern-based bug detection tools like SonarQube [41] and ESLint [4] may facilitate the automation of identifying and validating conditional logic related to numerical variables or categorical variables specifying visual properties.

Finding 3 and the analysis result in Figure 11 show that visual property specifications play a major role for the bugs arising from incorrect graphic computation. This suggests that an effective method for detecting such bugs is to *mutate visual properties by altering the values of parameters used in plot APIs*. We have tried a few domain-specific mutators in our prototype and successfully detected *two confirmed previously-unknown matplotlib bugs* [35, 36]. However, a key challenge in automating this approach is maintaining the *validity of the program after mutation*, ensuring that it executes without crashing and produces meaningful plots. Specifically, test programs must follow the specifications and constraints of parameters, including values and types. Such specification may be partially achieved through *program analysis techniques* [71]. For example, automatic extraction and parsing of documentation [93] may help extract such information and thus facilitate valid test case generations. Some constraints of parameters implicitly encoded in codebases, however, may require a *deeper understanding of functionality and interdependencies among functions*. This may be addressed by future research via crafting rules manually [95], or using Large Language Models to generate appropriate parameter relationships [97].

Finding 4 highlights the prevalence of test oracles that assert specific graphic property values and check for crashes, similar to other software. Interestingly, image-based test oracles are also used, including comparing with external images and comparing two figures. These correspond to *snapshot testing* and *metamorphic testing*. Snapshot testing, commonly used in graphics-related libraries, involves saving the generated figure from a fixed version as the expected image to ensure future updates do not introduce errors. Metamorphic testing, on the other hand, verifies complex functionalities by checking the equivalence of two figures generated by different but semantically equivalent code snippets. DataViz libraries currently rely heavily on snapshot testing to ensure plot correctness, which requires manual validation of reference images. Metamorphic relations, such as position addition and coordinate transformations in Figure 13c, remain underexplored and offer a promising direction for automated testing. Future work could identify *equivalent relations in data transformation and visual property mutation to enhance test automation*. This direction is motivated by the strong correlation in Figure 11, between the predominant root cause, Incorrect Graphic Computation, and two key triggering factors: Visual Property Specification and Data Preparation. Another promising direction, supported by Figure 10, is *to compare plot equivalence across different backends and libraries*. To our knowledge, no such study exists, likely due to variations in implementation across backends. This challenge might be addressed by leveraging

the data visualization understanding of VLMs in §4.5, to validate semantic equivalence over visual representations. Further, it might be worth to exploring whether property mutations lead to consistent changes in different backends/libraries; otherwise, it may imply a bug on one of them.

Finding 5 highlights the potential of VLMs for semantic understanding of data visualization and bug detection in DataViz libraries. As shown in §4.5, VLMs can detect 40% of visual bugs without code context, indicating their inherent knowledge of data visualization principles and best practices. Developers may leverage VLMs to *process image-only input for automatically detecting common anomalies*, such as missing labels and overlapping elements, without relying on explicit programmatic guidance. The significant performance drop of the Image+Test prompt compared to the Image and Image+Test+Hint prompts underscores the necessity of *guiding attention when addressing code-related bugs*, ensuring that VLMs focus on specific problematic elements rather than indiscriminately processing the entire context. Our experiments demonstrate that providing direct hints based on problematic graphic elements yields the highest detection rate of 57%. However, in practical scenarios where true buggy elements are unknown, a *systematic hinting strategy* is essential for effectively locating the buggy area. This can be achieved by progressively refining the analysis from high-level structural elements to fine-grained visual properties following prevalent visual bug characteristics discussed in Finding 1. Another potential approach, inspired by Finding 3, leverages the observation that *Visual Property Specification* is the most frequent trigger of DataViz library bugs. Following the metamorphic testing concept in *Compare Two Figures* from §4.4, rather than verifying equivalence under equivalent transformations, VLMs can be utilized to detect discrepancies under property mutations. This approach inherently directs attention to the most error-prone areas, *i.e.*, those associated with the mutated properties, thereby enhancing the precision of bug detection.

5.2 Threats to Validity

The first threat involves the subjective nature of manual labeling, as it relies on individual interpretations of code intention. To mitigate this, we defined a taxonomy through careful investigation and collaborative discussion for each research question. To minimize misclassification, two researchers independently classified all collected bugs. Discrepancies were resolved through discussion to reach a consensus. The second threat is the generalizability of our findings. To address this, we studied 564 bugs from five libraries across different programming languages and application scenarios. These libraries were developed by communities following various design philosophies. Analyzing these varied libraries helps us identify common patterns across DataViz libraries. Although our study does not cover domain-specific libraries, we noticed that many of them are built on the libraries selected in this study. For example, seaborn is built on matplotlib for statistical analysis; hGraph is based on D3.js for health data visualization. We believe that our findings are still applicable to them, and we leave future study to investigate these domain-specific ones.

6 Related Work

This section introduces two lines of research that are related to this study.

Data Visualization. Some studies have highlighted issues related to improper use of visualization tools, leading to data misrepresentation. Barcellos *et al.* [38] proposed a heuristic measurement of the quality of data visualization from the reader's perspectives of accessibility, conciseness, readability, and completeness of the visual properties and graphic elements used in the visualization. Szafir [85] conducted a comprehensive study from the user's perspective of common practices in data visualization that may cause misleading representation or ineffective presentation, such as axes beginning with non-zero, excessive use of colors, and unnecessary use of 3D graphics. Efforts have been made to establish best practices for creating effective visualizations. Midway [69] suggested ten principles for effective data visualization, among which using effective geometric objects,

representing information with proper color schemes, and including details of uncertainty are most emphasized. The development of visualization tools has been guided by the need to minimize syntactic errors and improve the expressive flexibility of graphic specifications. Wilkinson [91] proposed a layered framework named "The Grammar of Graphics", which decomposes graphic specifications of non-interactive data visualization into seven independently-customizable layers, including data, aesthetics, scale, geometric objects, statistics, facets, and coordinate system. This design has been adopted by some DataViz libraries, such as ggplot2 [87] and Vega-Lite [79]. Besides the improvement of non-interactive design, Satyanarayan*et al.* [80] introduced a declarative interaction design for interactive data visualization by modeling the interactive inputs as data streams instead of GUI events to improve the expressivity via interactions. In contrast to these studies focusing on best practices and user perspectives in data visualization, **our work** examines data misrepresentation and other bugs that arise from incorrect implementation within DataViz libraries themselves, offering a unique perspective on the internal sources of visualization errors.

Bug Characterization Studies. Bug characterization in software libraries has been extensively studied in various domains, such as deep learning (DL) libraries, GUI applications, and blockchain systems. Humbatova *et al.* [57] proposed a taxonomy of bugs in DL systems that use the most popular DL frameworks based on the manual labeling of collected artifacts from GitHub issues and Stack Overflow posts. Shen *et al.* [83] conducted an empirical study on the bugs in the DL compilers by identifying the categorization of root causes, symptoms, and error-prone stages in the DL compiler design. Liu *et al.* [65] presented their empirical study of Android bugs that cause performance issues from the perspectives of symptom, user experience, manifestation, maintenance, and root cause. Xiong *et al.* [94] conducted a comprehensive study of functional bugs in Android apps on their root causes, UI-related symptoms, test oracles, and success rate of detection by existing testing techniques. Wan *et al.* [89] performed a comprehensive study of bugs in blockchain systems and Dong *et al.* [48] studied bugs in Bash, respectively. Different from them, **our work** performs the first comprehensive study on DataViz libraries with an emphasis on the transformation process from data to graphic representations.

7 Conclusion

This paper presents an empirical analysis of 564 bugs from five widely used DataViz libraries such as matplotlib and ggplot2. We investigated the characteristics of bugs from the perspectives of symptoms, root causes, key bug-triggering steps, and oracles. The results show that incorrect/inaccurate plots are pervasive in DataViz libraries and they are majorly induced by incorrect graphic computations. Moreover, VLMs may be used in automated testing DataViz libraries with proper prompts. These findings benefit future studies to enhance the reliability of DataViz libraries.

8 Data Availability

The data of this study is publicly available at [37].

Acknowledgments

The idea of this project originated from Yongqiang Tian, and he supervised the entire project. Weiqi, Xiaohan and Haoyang are PhD students co-supervised Yongqiang Tian and Shing-Chi Cheung.

We appreciate all the insightful feedback from anonymous reviewers in FSE'25. Authors from HKUST are partially supported by the Hong Kong Research Grant Council/General Research Fund (Grant No. 16206524), the Hong Kong PhD Fellowship, and a research fund from an anonymous company. Zhenyang Xu and Chengnian Sun are partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant and CFI-JELF Project #40736.

References

[1] [n. d.]. Anti-Grain Geometry Library / SVN / [r141] /agg-2.4 — sourceforge.net. https://sourceforge.net/p/agg/svn/HEAD/tree/agg-2.4/. [Accessed 01-09-2024].

[2] [n. d.]. Chart.js Samples | Chart.js — chartjs.org. https://www.chartjs.org/docs/latest/samples/information.html. [Accessed 17-02-2025].

[3] [n. d.]. Examples; Matplotlib 3.10.0 documentation — matplotlib.org. https://matplotlib.org/stable/gallery/index.html. [Accessed 17-02-2025].

[4] [n. d.]. Find and fix problems in your JavaScript code - ESLint - Pluggable JavaScript Linter — eslint.org. https://eslint.org/. [Accessed 24-02-2025].

[5] [n. d.]. Function reference — ggplot2.tidyverse.org. https://ggplot2.tidyverse.org/reference/index.html. [Accessed 17-02-2025].

[6] [n. d.]. GitHub - hal9ai/awesome-dataviz: :chart_with_upwards_trend: A curated list of awesome data visualization libraries and resources. — github.com. https://github.com/hal9ai/awesome-dataviz. [Accessed 18-08-2024].

[7] [n. d.]. grid package - RDocumentation — rdocumentation.org. https://www.rdocumentation.org/packages/grid/versions/3.6.2. [Accessed 01-09-2024].

[8] [n. d.]. plotters - Rust — docs.rs. https://docs.rs/plotters/latest/plotters/. [Accessed 17-02-2025].

[9] [n. d.]. Riverbank Computing | Introduction — riverbankcomputing.com. https://riverbankcomputing.com/software/pyqt/intro. [Accessed 01-09-2024].

[10] [n. d.]. The Architecture of Open Source Applications (Volume 2)matplotlib — aosabook.org. https://aosabook.org/en/v2/matplotlib.html. [Accessed 26-08-2024].

[11] [n. d.]. Tutorial · Plots — docs.juliaplots.org. https://docs.juliaplots.org/stable/tutorial/. [Accessed 17-02-2025].

[12] 2021. broken colorbar when using contourf+cmap+norm+extend. https://github.com/matplotlib/matplotlib/issues/20963. [Accessed 20-08-2024].

[13] 2021. [Bug]: Additive offset with trailing zeros · Issue #22065 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/22065/. [Accessed 20-08-2024].

[14] 2021. [Bug]: cm fontset in log scale does not use Unicode minus · Issue #21540 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/21540/. [Accessed 19-02-2025].

[15] 2021. [Bug]: Error when removing colorbar in constrained layout · Issue #20978 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/20978. [Accessed 20-08-2024].

[16] 2021. [Bug]: Some styles trigger pcolormesh grid deprecation · Issue #21723 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/21723/. [Accessed 20-08-2024].

[17] 2021. [Bug]: transform keyword in ax.plot. https://github.com/matplotlib/matplotlib/issues/21008/, last accessed on Aug/19/2024.

[18] 2021. Dots misaligned in geom_dotplot · Issue #4614 · tidyverse/ggplot2 — github.com. https://github.com/tidyverse/ggplot2/issues/4614. [Accessed 19-02-2025].

[19] 2021. eps greyscale hatching of patches when lw=0. https://github.com/matplotlib/matplotlib/issues/22792. [Accessed 20-08-2024].

[20] 2021. geom_point(): when stroke=NA, cannot change size of the point · Issue #4624 · tidyverse/ggplot2 — github.com. https://github.com/tidyverse/ggplot2/issues/4624. [Accessed 20-08-2024].

[21] 2021. Last slice polar Area does not animate · Issue #10019 · chartjs/Chart.js — github.com. https://github.com/chartjs/Chart.js/issues/10019. [Accessed 20-08-2024].

[22] 2021. Tooltip doesn't utilize scale number formatting · Issue #9830 · chartjs/Chart.js — github.com. https://github.com/chartjs/Chart.js/issues/9830. [Accessed 20-08-2024].

[23] 2022. [Bug]: Cannot toggle set_tight_layout · Issue #22847 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/22847/. [Accessed 20-08-2024].

[24] 2022. [Bug]: Changing Linestyle in plot window swaps some plotted lines · Issue #22823 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/22823/. [Accessed 19-02-2025].

[25] 2022. [BUG] image tick positions drifting/ misplaced · Issue #4087 · JuliaPlots/Plots.jl — github.com. https://github.com/JuliaPlots/Plots.jl/issues/4087. [Accessed 19-02-2025].

[26] 2022. [Bug]: 'offsetbox' classes have optional arguments that are really not optional · Issue #24623 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/24623/. [Accessed 20-08-2024].

[27] 2022. [Bug]:Newer versions of matplotlib ignore xlabel on colorbar axis. https://github.com/matplotlib/matplotlib/issues/23398. [Accessed 20-08-2024].

[28] 2022. geom_hex color representation is wrong · Issue #5044 · tidyverse/ggplot2 — github.com. https://github.com/tidyverse/ggplot2/issues/5044. [Accessed 19-02-2025].

[29] 2022. hexbin is broken · Issue #5037 · tidyverse/ggplot2 — github.com. https://github.com/tidyverse/ggplot2/issues/5037. [Accessed 02-09-2024].

[30] 2023. [Bug]: offsetText is colored based on tick.color instead of tick.labelcolor · Issue #25165 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/25165/. [Accessed 19-02-2025].
[31] 2023. [Bug]: Suptitle not visible with subfigures · Issue #24910 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/24910/. [Accessed 20-08-2024].
[32] 2023. [Bug]: unexpected thetalim behavior in polar plot · Issue #25568 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/25568/. [Accessed 20-08-2024].
[33] 2023. Legend label click (& other events) use wrong label with multi-line, gets worse the more labels there are · Issue #11272 · chartjs/Chart.js — github.com. https://github.com/chartjs/Chart.js/issues/11272. [Accessed 20-08-2024].
[34] 2024. Best Data Visualization Libraries for 2024. https://www.nobledesktop.com/classes-near-me/blog/best-data-visualization-libraries. [Accessed 18-08-2024].
[35] 2024. [Bug]: Incorrect pcolormesh when shading='nearest' and only the mesh data C is provided. · Issue #29179 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/29179. [Accessed 23-02-2025].
[36] 2024. [Bug]: Poly3DCollection initialization cannot properly handle parameter verts when it is a list of nested tuples and shade is False · Issue #29156 · matplotlib/matplotlib — github.com. https://github.com/matplotlib/matplotlib/issues/29156. [Accessed 23-02-2025].
[37] 2025. GitHub - williamlus/dataviz-lib-bugs: An Empirical Study of Bugs in Data Visualization Libraries — github.com. https://github.com/williamlus/dataviz-lib-bugs. [Accessed 04-04-2025].
[38] Raissa Barcellos, Jose Viterbo, Flavia Bernardini, and Daniela Trevisan. 2018. An Instrument for Evaluating the Quality of Data Visualizations. In *2018 22nd International Conference Information Visualisation (IV)*. 169–174. doi:10.1109/iV.2018.00038
[39] Fabian Bendix, Robert Kosara, and Helwig Hauser. 2005. Parallel sets: visual analysis of categorical data. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* IEEE, 133–140.
[40] Dan Burger, Keivan G Stassun, Joshua Pepper, Robert J Siverd, Martin Paegert, Nathan M De Lee, and William H Robinson. 2013. Filtergraph: An interactive web application for visualization of astronomy datasets. *Astronomy and Computing* 2 (2013), 40–45.
[41] G Ann Campbell and Patroklos P Papapetrou. 2013. *SonarQube in action.* Manning Publications Co.
[42] W. K. Chan, S. C. Cheung, Jeffrey C.f. Ho, and T.h. Tse. 2006. Reference Models and Automatic Oracles for the Testing of Mesh Simplification Software for Graphics Rendering. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1. 429–438. doi:10.1109/COMPSAC.2006.71
[43] Chart.js. 2023. Chart.js: Simple HTML5 Charts. https://github.com/chartjs/Chart.js Accessed on July 15, 2024.
[44] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward understanding deep learning framework bugs. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–31.
[45] Antoine Clarinval and Bruno Dumas. 2022. Intra-City Traffic Data Visualization: A Systematic Literature Review. *IEEE Transactions on Intelligent Transportation Systems* 23, 7 (2022), 6298–6315. doi:10.1109/TITS.2021.3092036
[46] Victor Pezzi Gazzinelli Cruz, Henrique Rocha, and Marco Tulio Valente. 2023. Snapshot testing in practice: Benefits and drawbacks. *Journal of Systems and Software* (2023), 111797.
[47] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
[48] Yiwen Dong, Zheyang Li, Yongqiang Tian, Chengnian Sun, Michael W. Godfrey, and Meiyappan Nagappan. 2023. Bash in the Wild: Language Usage, Code Smells, and Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 8 (feb 2023), 22 pages. doi:10.1145/3517193
[49] David Eberly. 2006. *3D game engine design: a practical approach to real-time computer graphics.* CRC Press.
[50] Yoav Freund and Robert E. Schapire. 1995. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational Learning Theory*, Paul Vitányi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–37.
[51] ggplot. 2023. An implementation of the Grammar of Graphics in R. https://github.com/tidyverse/ggplot2 Accessed on July 15, 2024.
[52] Mike Goslin and Mark R Mine. 2004. The Panda3D graphics engine. *Computer* 37, 10 (2004), 112–114.
[53] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A comprehensive study of real-world bugs in machine learning model optimization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 147–158.
[54] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 147–158. doi:10.1109/ICSE48619.2023.00024
[55] Katherine J Harrison, Valérie de Crécy-Lagard, and Rémi Zallot. 2018. Gene Graphics: a genomic neighborhood data visualization web application. *Bioinformatics* 34, 8 (2018), 1406–1408.
[56] Lukas Hoyer, David Joseph Tan, Muhammad Ferjad Naeem, Luc Van Gool, and Federico Tombari. 2024. SemiVL: semi-supervised semantic segmentation with vision-language guidance. In *European Conference on Computer Vision*.

Springer, 257–275.
[57] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. doi:10.1145/3377811.3380395
[58] Johannes Kehrer and Helwig Hauser. 2013. Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey. *IEEE Transactions on Visualization and Computer Graphics* 19, 3 (2013), 495–513. doi:10.1109/TVCG.2012.110
[59] Cole Nussbaumer Knaflic. 2015. *Storytelling with data: A data visualization guide for business professionals*. John Wiley & Sons.
[60] Jonathan Knudsen. 1999. *Java 2D graphics*. " O'Reilly Media, Inc.".
[61] Robert Kosara. 2016. Presentation-Oriented Visualization Techniques. *IEEE Computer Graphics and Applications* 36, 1 (2016), 80–85. doi:10.1109/MCG.2016.2
[62] Valéria Lelli, Arnaud Blouin, and Benoit Baudry. 2015. Classifying and qualifying GUI defects. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.
[63] Kanglin Li and Mengqi Wu. 2006. *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons.
[64] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing transaction-reverting statements in ethereum smart contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 630–641.
[65] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.
[66] Joel Martin and David Levine. 2018. Property-Based Testing of Browser Rendering Engines with a Consensus Oracle. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 424–429. doi:10.1109/COMPSAC.2018.10270
[67] matplotlib. 2023. GUI Misbehavior Example. https://github.com/matplotlib/matplotlib/issues/24089 Accessed on July 15, 2024.
[68] matpltlib. 2023. matplotlib: plotting with Python. https://github.com/matplotlib/matplotlib Accessed on July 15, 2024.
[69] Stephen R Midway. 2020. Principles of effective data visualization. *Patterns* 1, 9 (2020).
[70] Vinh T Nguyen, Kwanghee Jung, and Vibhuti Gupta. 2021. Examining data visualization pitfalls in scientific publications. *Visual Computing for Industry, Biomedicine, and Art* 4 (2021), 1–15.
[71] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. springer.
[72] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. 1998. A visual test development environment for GUI systems. *ACM SIGSOFT Software Engineering Notes* 23, 2 (1998), 82–92.
[73] Dale Patterson. 2016. Interactive 3D web applications for visualization of world health organization data. In *Proceedings of the Australasian Computer Science Week Multiconference*. 1–8.
[74] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 1–18. doi:10.1145/3132747.3132785
[75] plots.jl. 2023. Powerful convenience for Julia visualizations and data analysis. https://github.com/JuliaPlots/Plots.jl Accessed on July 15, 2024.
[76] plotters. 2023. A rust drawing library. https://github.com/plotters-rs/plotters Accessed on July 15, 2024.
[77] Tahmid Rafi, Xueling Zhang, and Xiaoyin Wang. 2023. PredART: Towards Automatic Oracle Prediction of Object Placements in Augmented Reality Testing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 77, 13 pages. doi:10.1145/3551349.3561160
[78] Oindrila Saha, Grant Van Horn, and Subhransu Maji. 2024. Improved zero-shot classification by adapting vlms with text descriptions. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 17542–17552.
[79] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350.
[80] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 669–678.
[81] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.
[82] Sakib Shahriar, Brady D Lund, Nishith Reddy Mannuru, Muhammad Arbab Arshad, Kadhim Hayawi, Ravi Varma Kumar Bevara, Aashrith Mannuru, and Laiba Batool. 2024. Putting GPT-4o to the Sword: A Comprehensive Evaluation of Language, Vision, Speech, and Multimodal Proficiency. *Applied Sciences* 14, 17 (2024), 7782.

[83] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980.

[84] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. doi:10.1145/2931037.2931074

[85] Danielle Albers Szafir. 2018. The good, the bad, and the biased: Five ways visualizations can mislead (and how to fix them). *interactions* 25, 4 (2018), 26–33.

[86] Yongqiang Tian, Shiqing Ma, Ming Wen, Yepang Liu, Shing-Chi Cheung, and Xiangyu Zhang. 2021. To what extent do DNN-based image classification models make unreliable inferences? *Empir. Softw. Eng.* 26, 4 (2021), 84. doi:10.1007/S10664-021-09985-1

[87] Randle Aaron M Villanueva and Zhuo Job Chen. 2019. ggplot2: elegant graphics for data analysis.

[88] Jagoda Walny, Christian Frisson, Mieka West, Doris Kosminsky, Søren Knudsen, Sheelagh Carpendale, and Wesley Willett. 2019. Data changes everything: Challenges and opportunities in data visualization design handoff. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 12–22.

[89] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2017. Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 413–424. doi:10.1109/MSR.2017.59

[90] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, et al. 2024. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *Advances in Neural Information Processing Systems* 36 (2024).

[91] Leland Wilkinson. 2012. *The grammar of graphics*. Springer.

[92] Ziliang Wu, Shi Liu, Yuefan Zhou, Tong Xu, Jiacheng Pan, Zhiguang Zhou, Wei Chen, and Fei-Yue Wang. 2024. VIEA: A Visualization System for Industrial Economics Analysis Based on Trade Data. *IEEE Transactions on Computational Social Systems* 11, 2 (2024), 2807–2818. doi:10.1109/TCSS.2023.3288671

[93] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 176–188. doi:10.1145/3533767.3534220

[94] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. doi:10.1145/3597926.3598138

[95] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[96] Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. 2023. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421* 9, 1 (2023), 1.

[97] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 435–450. doi:10.1145/3453483.3454054

[98] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.

Received 2024-09-13; accepted 2025-04-01