Plottery: A Code-Integrated GUI for Matplotlib

Anonymous Authors

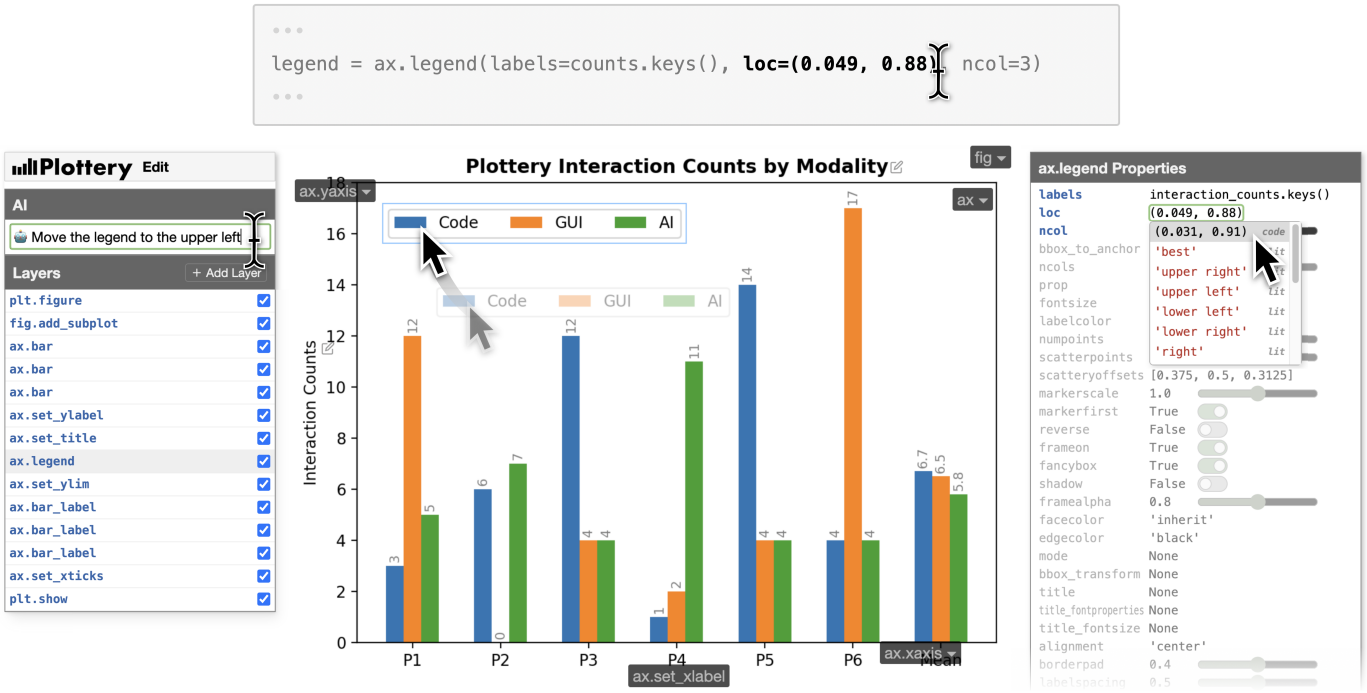


Fig. 1: PLOTTERY imitates graphical editors, offering live GUI editing of Matplotlib code in an ordinary Jupyter notebook.

***Abstract*—Scientists have many tools to author plots: graphical direct manipulation apps like a spreadsheet's chart builder, text-based programming tools like Matplotlib and ggplot2, or AI chat interfaces. Textual code is the most capable of these, but writing code, even with AI assistance, is a potentially poor fit for graphical chart design. We present PLOTTERY, a Jupyter notebook extension that leverages API type annotations to add graphical direct manipulation to the popular Matplotlib plotting library. User interactions in PLOTTERY's in-situ graphical interface are immediately reified in the user's code, so they may seamlessly mix-and-match between GUI, code, and AI interactions. In a user study, participants seamlessly alternated modalities, with the GUI used for straightforward visual changes, code for quick edits or duplicating code, and the AI for broad support. PLOTTERY showcases graphical interaction that retains code's benefits.**

***Index Terms*—GUI, Bidirectional Programming, Visualization Authoring, Jupyter**

I. INTRODUCTION

With the ubiquity of scientific data plotting, plot authors have many ways to create their plots. They might reach for a *graphical user interface (GUI)*, such as the charting features in a spreadsheet. They might write *code*, using a library such as Matplotlib, ggplot, or Vega-lite. Or, they might upload their data to a chatbot to build their plot with *AI* through natural language. Each approach has pros and cons.

GUIs are perhaps the most popular way to create plots, from commercial tools such as spreadsheet applications, Tableau, and Power BI to academic tools with innovative interfaces like Charticulator [1], Data Illustrator [2], Lyra [3], and others ([4], [5], [6], [7]). In GUIs, the available actions are discoverable in the interface, direct manipulation lets users precisely modify the visual design with their mouse cursor, and changes are displayed immediately for rapid iteration [8]. Graphical interfaces naturally fit graphical plots.

Nevertheless, code-based plotting libraries like Matplotlib [9], ggplot2 [10], or Vega-Lite [11] remain common. Although foregoing the pleasant experience of direct manipulation [12], text-based tools offer at least three advantages: code can be written *in-situ* within a data analysis framework, such as a Jupyter notebook, enabling rapid iteration between analysis and visualization ([13], [14]); code offers near-unlimited *expressivity*, allowing authors to customize the plot as desired ([14], [15], [16]); and code can *automate* plot production, for reproducibility ([17], [15]) or for drawing many similar plots [15]. Still, writing code can be tedious, for example Matplotlib, the target of this work, is disliked by some of its users ([18], [19], [20]) for its "garbage" usability [21], a problem exacerbated because, even in the AI era, using such code-based tools sometimes still requires the laborious process of looking up documentation, copy-pasting a code example, and modifying it to fit one's scenario.

To avoiding the learning curve of complicated code APIs,

plot authors may instead create their plots with natural language ([22], [23]). Current leading AI tools can draw charts based on data uploaded to the chat interface (*e.g.* [24]) or from within a spreadsheet integration ([25], [26]). Prompt-based AI interfaces offer benefits: users can describe complex charts using imprecise language and achieve their goals with little expertise in the plotting library. However, checking if AI-generated code matches the user's intent can be difficult ([27], [28]), and a GUI may be better for fine-grained graphical control and exploring design tweaks, like repositioning a legend on a plot or picking specific colors ([29], [12]).

Our goal is to augment plot code editing with a seamlessly integrated GUI, *without* losing any of code's benefits, namely, code's operation *in-situ* in the data science workflow, its *expressivity*, and its ability to *automate* plot creation. Although there are capable code-generating plotting GUIs ([3], [30], [31], [32], [33]), in our view there is still no interface that integrates GUI and code seamlessly, with *both* as powerful first-class interaction modes. To demonstrate our ideal vision of a code-integrated plotting GUI, we target a popular but difficult API: the Matplotlib library [9] within Jupyter notebooks [34]. Although Matplotlib is the most popular Python plotting library (as measured by Google search results), it is in dire need of better tooling because of its unintuitive API ([18], [21]), which earns comments from users such as *"Every time I need to use mpl for a project I die a little inside"* [19] and *"Matplotlib sucks and we all know it. It's just the only library that can do everything we need it to and has documentation detailing exactly how to do it"* [20]. To help, we implement a notebook extension called PLOTTERY. As the craft of pottery mixes clay and water to sculpt a concrete object, in PLOTTERY users mix GUI and (AI-assisted) code interactions to fluidly sculpt Matplotlib plots.

Design Goals. To simultaneously be as "GUI-like" and "code-like" as possible, PLOTTERY aims to provide:

1) **Familiarity** by imitating a traditional graphics editor.
2) **Fluid iteration** through direct manipulation ([8], [12]).
3) **Predictability** in how GUI actions become code.
4) **Compatibility** with code's benefits: in-situ operation, expressive authoring, and automation. Specifically, it should work in the notebook, support much of the Matplotlib API, and support loops and functions.
5) **Fluid mode switching**: users should be able to fluidly bounce between GUI, code, and AI interactions without the friction of switching costs ([30], [35]).

To meet these goals, PLOTTERY adopts two simple but novel design commitments:

1) **Calls = Layers.** PLOTTERY treats individual Matplotlib API calls as *layers* in a traditional graphics editor, organizing them into a layers panel for selection, toggling, and reordering (left in Fig. 1).
2) **Arguments = Properties.** The arguments in each individual call are analogs to *item properties* in a traditional editor. PLOTTERY uses Matplotlib API type annotations to present possible call arguments in a properties editor, with type-appropriate UI widgets for each property (right in Fig. 1). This type-driven approach allows PLOTTERY to support a meaningful portion of the Matplotlib API.

To show how PLOTTERY implements design goals 1-4, after discussing related work (Sec. II) we demonstrate PLOTTERY in an example walkthrough (Sec. III) followed by a description of the implementation of its key features (Sec. IV). For design goal 5, fluid mode switching, we investigate user interaction with GUI, code, and AI in PLOTTERY through an exploratory user study (Sec. V). This study provides evidence not only for PLOTTERY's usability and seamless multimodality, but also provides insights on the complementarity of GUI, code, and AI. In particular, we find (1) all modalities were used, users leaned most on AI but individual preferences varied; (2) each modality offers its own complementary strengths—GUI suits straightforward visual changes; AI helps with getting started and when unsure how to make a change; and code is used for quick, small edits and for copy-paste-modify duplication; (3) as we hoped, the interaction of all three modalities was *seamless*, study participants switched between modes after only about 3 interactions on average. We discuss these results (Sec. VI) and conclude (Sec. VII).

II. Related Work

Below we survey graphical tools for bespoke visualizations, natural language for plotting, and multimodal plot authoring.

GUIs for Custom Visualizations. GUI-based systems, dating back to at least Gold [36] and SageBrush [37], use a vector graphics editor and bind properties of primitive shapes to data to produce customized data-bound visualizations. While some systems focus on glyph (mark) customization ([38], [5]), others target better support for user interaction and data use. To support bulk edits of visualizations, some systems represent repeated elements as groups on the canvas ([3], [4], [1]), while others capture user actions as step-by-step (non-textual) programs with loops to enable reuse [39]. These systems also enable data use in various ways, from letting users design visuals before connecting with data ([2], [40]), abstracting the spatial representation of repeated data into hidden layout guides [6], enabling data binding specifications on a separate node-and-wires programming canvas (that could be further used for data transformation) [7], using a spreadsheet paradigm [41], or inferring visualizations and the relevant data transformations from user examples of data bindings [42]. We provide expressive authoring not by innovative interactions as in the above systems, but by following the expressive Matplotlib API so PLOTTERY can operate in-situ in the user's Jupyter notebook, which streamlines automated regeneration of plots by avoiding export-import to a separate application.

Natural Language for Visualizations. Users may not know how to create their desired plot with a complex GUI—a difficulty dubbed the *gulf of execution* [12]–so it would be helpful if they could describe their visualization in terms they understand. Consequently, natural language has been applied, successfully, to visualization even before the recent rise of

large language models (LLMs) ([22], [23]). Nevertheless, newer LLM-based systems can outperform classical NLP for plotting [43], and flagship LLMs are skilled at generating Matplotlib code ([44], [45]), with one user quipping *"matplotlib is actually really awful and it's hilarious that we had to invent literal AGI for me to start using it"* [46]. PLOTTERY makes no innovation here, but includes an AI prompt box to allow natural language authoring alongside GUI and code editing.

Natural Language and GUI. Although natural language can help build up a visualization without learning a GUI, Vaithilingam et al. [29] hypothesize that a GUI is better for *tweaking* plots, such as picking specific colors. Combining natural language with a GUI may provide the complementary strengths of each ([47], [29]). VisTalk [47] uses a non-LLM approach to transform keywords in the user's input (*e.g.* "red") into clickable regions that open a widget (*e.g.* a color picker). Augmenting this approach, DynaVis [29] lets users prompt an LLM to create or change their plot (represented internally as Vega-Lite [11]) and simultaneously generates a *dynamic widget* to control relevant plot properties related to the prompt (*e.g.* number inputs for editing the Y-axis limits). Users can also specifically ask the LLM to make a GUI widget. Although not targeting plotting, BISCUIT [48] similarly generates transient widgets from prompts; widgets steer AI generation of code in the following notebook cell. LLM-generated widgets can randomly vary from prompt to prompt. Because our goal is to innovate on code-integrated GUIs and not AI interaction, PLOTTERY generates its UI deterministically based on Matplotlib API type annotations rather than requiring users to prompt AI to create a GUI.

Code and GUI. Some Python-focused plugins ([49], [50], [51]) offer GUI-based chart building *in-situ* within the user's data analysis environment (*e.g.* a Jupyter notebook), generating code from the GUI. However, code generation is one-way: text edits to the code are not reflected back into the GUI. B2 [52] can reflect code edits back into the GUI but its GUI actions, like the above, only generate code rather than also edit it.

Three prior systems allow full round-trip editing between code and the GUI to truly mix the modalities ([33], [32], [30]). Pylustrator [33] takes a base plot built from Matplotlib code as input, allows users to further modify the plot from the GUI, and saves GUI edits to Matplotlib code appended to the original code. These edits are limited to moving and tweaking items—Pylustrator does not allow plotting new data within the GUI. Round-trip editing is possible but tedious: a user can edit the generated code then reopen it in Pylustrator for further GUI-based changes. CodeMend [32] combines natural language, code, and a GUI property editor, taking a natural language query from the user to highlight the most relevant calls/properties in the GUI for the user to interact with to update their code. CodeMend's GUI property editor is rather full-featured, but its overall interface was designed for demonstrating pre-LLM natural language input so it lacks the smooth editing of a traditional graphics editor that we seek. For that goal, the closest to our system is mage [30] which mixes live direct manipulation with code editing in Jupyter notebooks. Although its GUI interaction better imitates the fluid editing of a traditional graphical editor than CodeMend, mage's small plotting GUI only supports a limited subset of Vega-lite, rather than aiming for highly-customizable Python-integrated authoring.

Our goal in PLOTTERY is to demonstrate an ideal code-integrated GUI for plot authoring. PLOTTERY combines Pylustrator's direct manipulation on the plot, CodeMend's more complete treatment of the Matplotlib API, and mage's fluidity in-situ in the notebook. To this, PLOTTERY adds support for functions and loops, and presents all this like a traditional graphical editor but produces always-editable Python code.

III. PLOTTERY

PLOTTERY is a Jupyter notebook extension that augments Matplotlib code cells with a GUI, aiming to empower users to smoothly construct non-trivial plots with more pleasure and less fumbling than is usual with Matplotlib. To demonstrate PLOTTERY, we will build a bare bar chart, like  , suitable for inclusion inline in text for, *e.g.*, reporting Likert score counts. We will turn this into a function to reuse the plot with different data to produce three plots:  . Then we will add a dot for the mean, like  , to demonstrate how PLOTTERY works even after abstracting the plot into a function. We will use all three interaction modalities PLOTTERY supports: GUI, code, and AI. *Our anonymous supplement at* `tinyurl.com/plottery-rep` *includes a narrated five minute video figure recording of this walkthrough.*

Getting Started. We fill in a code cell with example data in a `counts` variable (first cell in Fig. 2). PLOTTERY provides a "New Plot" button in the Jupyter interface (not shown), we click it which adds starter code in a new cell (second cell in Fig. 2). This code creates a figure and adds one *axes*, *i.e.* cartesian coordinate box, to it. PLOTTERY's UI appears when we run the cell (Fig. 2 bottom).

```
counts = [4, 1, 3, 6, 10]
```

```
fig = plt.figure(layout='tight')
ax = fig.add_subplot(1, 1, 1)

plt.show()
```

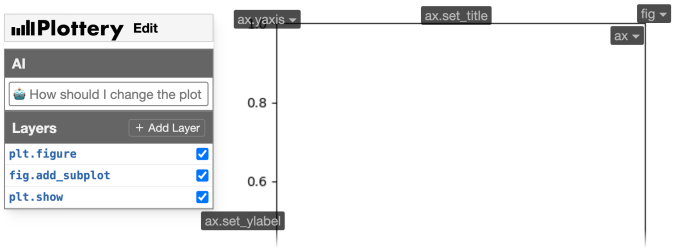


Fig. 2: Example data, starter code, and the initial PLOTTERY UI next to a new plot with an empty axes box. The left sidebar has an AI prompt box and a Layers panel listing Matplotlib calls in the code. Buttons on the plot allow adding new code.

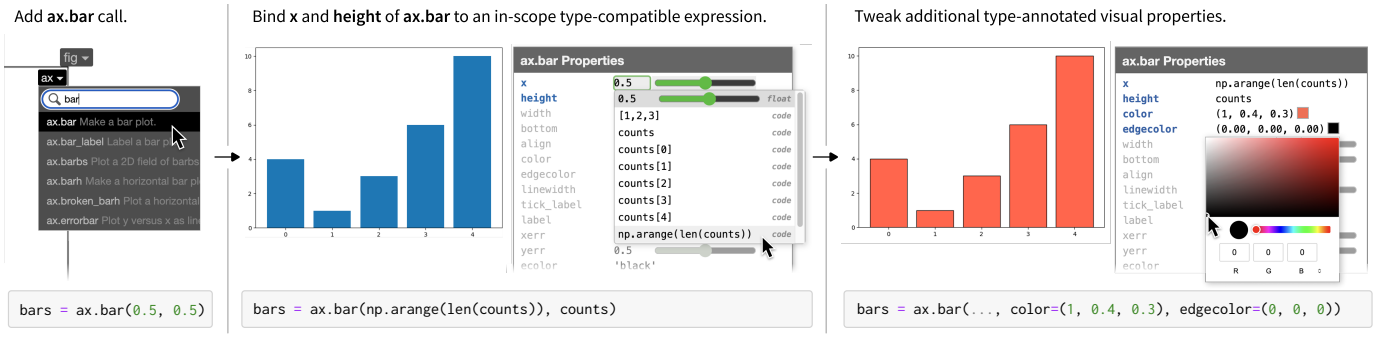


Fig. 3: Adding an `ax.bar` call from the ax ▼ dropdown, and binding and styling the bars with the Properties panel.

Layers Panel. PLOTTERY's left sidebar has an AI prompt box and a *Layers panel* that lists the code's Matplotlib calls (Fig. 2 bottom left). In the Layers panel, clicking a layer selects the call to edit its properties, drag-and-drop reorders calls, and clicking the visibility checkbox comments or uncomments the associated line of code. These latter two features are like the R data analysis tool Unravel [53], but for plotting.

On-Plot Buttons. When the mouse cursor is over the plot, *on-plot buttons* appear for adding new calls to our cell (Fig. 2 bottom right). For example the ax.set_title button would add `ax.set_title('My Plot')`. The buttons with ▼ open a menu of possible function calls, with a search box to filter the many items (Fig. 3 left). We click ax ▼ and choose `ax.bar` to add a bar plot, which adds the code `bars = ax.bar(0.5, 0.5)`.

Properties Panel. The updated code cell is automatically re-run, the new `ax.bar` call is selected in the left sidebar (not shown), and a *Properties panel* appears on the right listing both given arguments and greyed-out possible arguments for the selected call (*e.g.* Fig. 1 right). The traditional way to work with Matplotlib is to search online for a function's documentation, find a desired argument, write it in our code, and rerun. The Properties panel makes this process instant. Clicking an argument toggles it on or off, with the plot live updating to show the effect. This enables rapid exploration.

We see our bar's `x` and `height` arguments are literal numbers `0.5` and `0.5`. We want to bind these arguments to our `counts` data. We click on the number `0.5`, which lets us text-edit its value but also opens a dropdown of suggested values (Fig. 3 middle). The suggestions are based on the type annotation for the `x` argument and the variables in scope that match that type, including depth 1 indexing into those variables (*e.g.* `counts[0]`) and common expressions involving those variables (*e.g.* `np.arange(len(counts))`). The plot display live updates as we hover over each, previewing the result. We choose `np.arange(len(counts))`, which is the numbers `[0, 1, 2, 3, 4]`, as the `x` positions for our bars. For `height`, we choose `counts`, resulting in five bars (Fig. 3 middle).

Styling is similarly quick. Color arguments have a color picker widget—we use these to set the bar `color` to a shade of orange and `edgecolor` to black (Fig. 3 right). Numeric arguments can be controlled with a slider . We slide the `linewidth` to `5` to thicken their borders. Each of these changes is instantly realized in the code and on the plot.

On-Plot Direct Manipulation. We want our bars to touch each other. PLOTTERY allows certain parameters to be modified by direct manipulation on the plot itself. We grab the left or right edge of any bar and drag horizontally (Fig. 4). This adds a `width` argument to our `bar` call and modifies `width` to match our mouse movements live—we release the mouse when the bars touch each other at `width=1`. Hitting `1` exactly is possible because PLOTTERY rounds numbers to two significant figures during drags to imitate "snap to grid" in graphics editors.

Using AI. Recall our goal is to produce miniature histograms like . We now want to remove the border and numbers around our plot. Doing this through the GUI is possible but requires a few steps which we might not know. In the left sidebar's *AI panel*, we prompt "Remove the borders around the plot" and then "Remove the numbers along the axes". After each, PLOTTERY replaces the code cell with the LLM's response, highlights changed lines, and runs the cell to show the updated plot. We also ask the AI to color our bars with a gradient like .

Automation. A key reason to plot using code is for reuse: we want to make many similar bar charts. Fig. 5 shows how we set this up in our notebook: we add data for the three plots in a `counts_lists` variable, wrap all of our existing plotting code in a `plot_counts` function, and then call that function in a for-loop to plot each data series. AI could help us write this code, but here, it is simple enough to do manually. Our code now produces three plots. PLOTTERY only shows one but provides a dropdown to switch between them (Fig. 5 bottom).

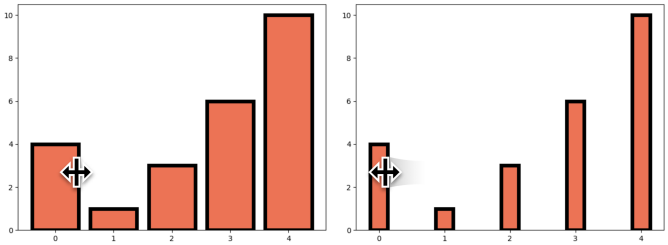


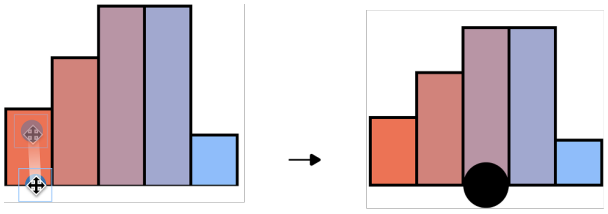
Fig. 4: Some spatial properties can be manipulated on the plot.

```
counts_lists = [
    [4,  1, 3, 6, 10],
    [15, 4, 2, 3, 0],
    [3,  5, 7, 7, 2],
]
```

```
def plot_counts(counts):
    ...
    plt.show()

for counts in counts_lists:
    plot_counts(counts)
```

Fig. 5: PLOTTERY lets us pick which of multiple plots to view.



```
def plot_counts(counts):
    mean = ...
    ...
    paths = ax.scatter(mean, 0, s=6000, c=(0.00, 0.00, 0.00))

    ax.set_ylim(bottom=-0.15*max(counts))

    plt.show()
```

Fig. 6: Final graphical changes and code.

GUI edits work even when the plotting code is inside a function or loop. Recall we want to add a dot to indicate the mean, like  . After manually writing the computation to compute the mean, we click ax ▼ on the plot to add `ax.scatter` (a scatterplot), which will, by default, draw a single circular dot. PLOTTERY inserts code relative to the code location of `plt.show()`, so `ax.scatter` is inserted *inside* our function. Direct manipulation continues to work. The dot starts out at `(0.5, 0.5)`, which we drag down to a `y` of `0` (Fig. 6 left) and then use the Properties panel to set the remaining arguments: we choose `mean` for `x`, set the color to black, and drag the `size` slider until the dot is large. The bottom half of the dot is hidden because it extends outside the axes frame, so we add an `ax.set_ylim` call and set the `bottom` of the axes to a negative value instead of 0. Fig. 6 shows the final design (right) and the code produced by these final changes. We "Save" from the menu to export PNG, and our plots are done:  !

Recap. Above we showed PLOTTERY's multimodal plot authoring: the *Layers panel* enables code reordering and selecting calls for editing, the plot area features *on-plot buttons* like ax ▼ for adding items, certain spatial properties may be tweaked by *on-plot direct manipulation*, a call's arguments are editable via the *Properties panel* with widgets and suggestions for each argument, the GUI supports functions or loops that draw multiple plots, the *AI Panel* offers edits by prompting, and *manual code edits* are always available at any time.

IV. IMPLEMENTATION

We briefly explain some of PLOTTERY's technical details. Hurried readers should only read the Properties panel section.

Notebook Extension. PLOTTERY consists of a Jupyter Notebook extension, a Python backend, and a Typescript front-end that renders the GUI. The notebook extension installs a Jupyter hook to silently rewrite code before it is sent to the Python backend, replacing calls to `plt.show()` with a PLOTTERY-specific call that tells the Python backend about the variables in scope and the code in the notebook. The call returns a Python object that renders the HTML and Javascript of PLOTTERY's interface (the object has a `_repr_html_` method for Jupyter).

Properties Panel - GUI from Type Annotations. To determine possible function arguments for the Properties panel and suggested values for each, PLOTTERY runs the MyPy [54] type checker on the notebook code to find function arguments types. Consider the type annotations in Fig. 7 for the Matplotlib `ax.bar` function, these become the possible arguments shown in the Properties panel. The types describe the possible values for each argument, as well as a possible default value (after the `=` sign). The default is used as the initial value when the user clicks to enable the argument, and the type determines the suggestions in the dropdown. For example, `bar`'s first argument `x` has type `float | ArrayLike`, meaning it may be either a `float` or an `ArrayLike` (`ArrayLike` means what NumPy considers array-like objects). For `float` and `ArrayLike`, PLOTTERY offers the suggestions `0.5` and `[1, 2, 3]` respectively. Any of the user's variables that are type-compatible with `float | ArrayLike` are also suggested, as well as depth one indexing into these variables, as shown in Fig. 3 (middle).

As a second example, consider `ax.bar`'s `align` argument. Its type is `Literal['center', 'edge']`, that is, these two strings are the *only* possible values, so the suggestions dropdown shows exactly these two (Fig. 8 left). Hovering the cursor over each previews the result on the plot: `'center'` centers the bars over the `x` values, whereas `'edge'` aligns the left edges of the bars on the `x` values. `Literal` annotations make working with such enum-like properties smooth in PLOTTERY: for example, to choose a shape for scatterplot points the user does not

```
def bar(
    self,
    x: float | ArrayLike,
    height: float | ArrayLike,
    width: float | ArrayLike = 0.8,
    bottom: float | ArrayLike = 0,
    *, # Subsequent args are keyword-only
    align: Literal["center", "edge"] = "center",
    color: ColorType | Iterable[ColorType] = ...,
    ...
    **kwargs: RectangleProps,
) -> BarContainer: ...
```

Fig. 7: A subset of type annotations for `ax.bar`

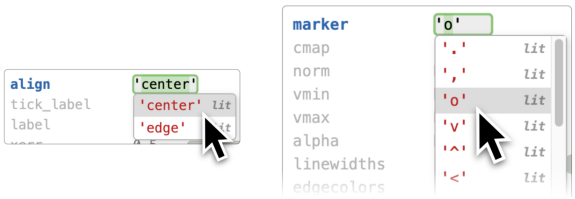


Fig. 8: The Properties panel is convenient for parameters with `Literal[...]` type annotations. Options are live-previewed.

have to look up documentation in a separate browser tab, they can just click the `marker` argument and quickly preview the suggestions by scrubbing their mouse over each (Fig. 8 right).

Certain other types are backed by UI widgets in the Properties panel. Floats and ints may be manipulated with a slider, booleans with a toggle switch, and colors show a swatch which opens an HTML5 color picker. Widgets are placed to the right of an ordinary text box so the user is also free to write arbitrary code.

PLOTTERY bundles its own Matplotlib type annotations. We started from Microsoft's Python Type Stubs project [55] and added and modified the types. For example, we might write a type `Literal['a', 'b'] | str`, even though it is the same as just `str`, because `Literal['a', 'b']` produces `'a'` and `'b'` as dropdown suggestions.

On-Plot Regions for User Calls. Internally, Matplotlib figures are a scene graph of shapes. PLOTTERY needs to know where these shapes are, both to make appropriate shapes on the output clickable and to position the on-plot buttons (like ax.set_xlabel). PLOTTERY first walks the Matplotlib scene graph and converts shapes into polygonal outline regions (Fig. 9 left), leveraging the `shapely` Python library [56] to expand each region slightly for clickability. Some shapes *should* be clickable to select one of the user's function calls (Fig. 9 right), but Matplotlib's scene graph does not ordinarily remember which calls produced the shapes. To preserve this correspondence, PLOTTERY rewrites the user's code to tag values returned from functions with location metadata: *e.g.* `text = ax.set_title(...)` becomes `text = tag(ax.set_title(...), 'ax.set_title #1')`, which sets the propery `text._came_from = 'ax.set_title #1'`. Shapes from function calls now have a `_came_from` property which PLOTTERY uses so that clicking the shape will select

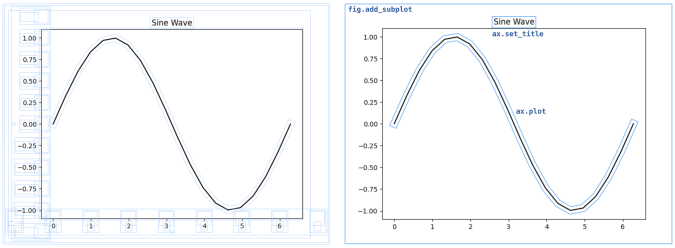


Fig. 9: Regions for all shapes in the scene graph (left, in blue) and those clickable to select a call in the user's code (right).

Fig. 10: Changing a plot title directly on the plot.

the function call in the Layers and Properties panels.

On-Plot Buttons. Similarly, PLOTTERY positions on-plot buttons for adding new function calls, *e.g.* ax.yaxis ▼ and ax.set_title. PLOTTERY has a hard-coded list of which functions should appear on which parts of the plot, *e.g.* if the `ax` variable is an `Axes` object, PLOTTERY will make `ax.set_ylim()` appear on the `ax.yaxis` shape. The challenge is figuring out that the relevant root object is named `ax`. Naively, we could walk the scene graph to determine object names, but this will produce ugly code like `fig.axes[0].set_title` instead of just `ax.set_title`. Instead, PLOTTERY walks from the *local variables* to a depth of 4. If there are multiple paths to an object, *e.g.* `axs[0]` and `fig.axes[0]`, PLOTTERY uses the shortest. The buttons on the plot, *e.g.* axs[0].set_title, use these idiomatic names when adding new code.

On-Plot Direct Manipulation. PLOTTERY lets the user directly manipulate certain shapes on the plot to move or resize them. Note that this is different from *e.g.* iPyWidgets [57] where interactions only modify the Python state behind the scenes: PLOTTERY changes the code. The parts of a shape that are draggable are determined by looking for specific argument names within the function's first ten arguments, such as "`x`" or "`y`" for the shape body, or "`xmax`", "`right`", or "`width`" for the right edge. If the argument is currently a number, *e.g.* `x=10`, then dragging the shape will change the number, otherwise dragging will add a number to the expression, *e.g.* `x=var+10`. This local code update scheme is similar to CapStudio [58], there is no backpropagation deeper into the code as in other systems ([59], [60], [61]). To determine how much a number should change per pixel of mouse movement, PLOTTERY tags shapes with the data limits of their containing axes and the axes size in pixels, to convert from pixels to the axes units.

The text for some items on the plot, *e.g.* title text and axis labels, can be edited by clicking an adjacent edit button without going to the Properties panel (Fig. 10).

V. EVALUATION

To understand the trade-offs of PLOTTERY's multi-modal approach, we conducted an IRB-approved two-part study—an exploratory lab study ("PART 1") and a longer follow-up study ("PART 2")—focusing on when and why users switch between GUI, code, and AI. Our methodology and findings are below.

We also used PLOTTERY to create charts in Fig. 14 and Fig. 15 and plots from earlier drafts of the paper (*e.g.* Fig. 1). Although not discussed here, the appendix at `tinyurl.com/plottery-rep` includes those plots and our reflections on authoring them, as well as reflections on a plot too complicated for PLOTTERY, a narrated video of the walkthrough example, and the study replication package.

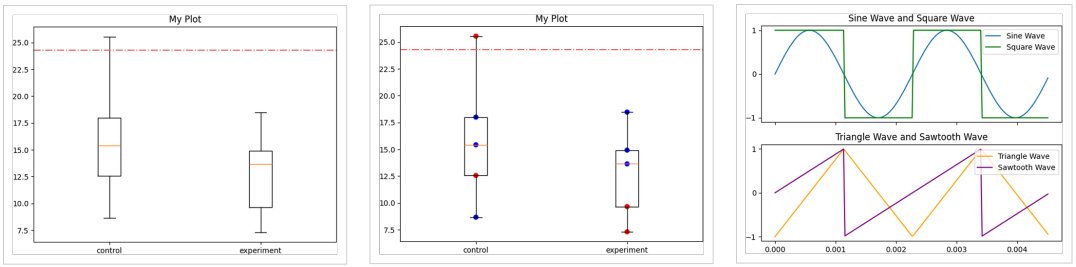


Fig. 11: PART 1 tasks. Task 2 extends the first, with GUI/AI alternately ablated. Task 3 is free use.

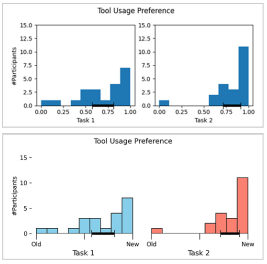


Fig. 12: PART 2 primer.

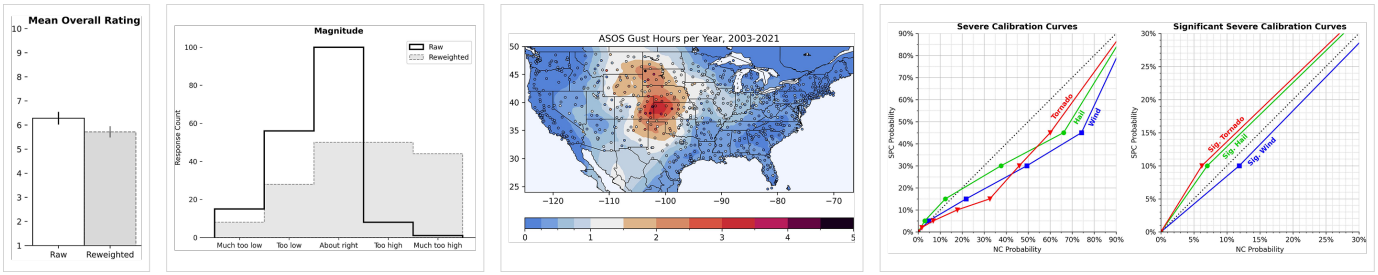


Fig. 13: The possible C3 *Mixed* tasks for PART 2, adapted from real-world plots. The tasks cover the topics of the 20 most-visited non-tutorial pages on matplotlib.org. The map task used `imshow` and a scatterplot rather than geospatial libraries.

A. User Study Methods

Studies were in person and recorded with participant consent.

Participants. We recruited 10 participants for PART 1 via institutional mailing lists and messaging platforms (5 women, 4 men, 1 undisclosed): 9 graduate students (6 in Computer Science, 1 in Data Science, 1 in Bioinformatics, and 1 in Electrical Engineering), and 1 Computer Engineering undergraduate. Based on their self-reported comfort with Matplotlib, we classified 5 as experts (P2, P4, P5, P7, P9), and 5 as non-experts (P1, P3, P6, P8, P10). Four participants further attended PART 2 (detailed below).

PART 1 ***(60min).*** Participants did a 10 minute tutorial on all PLOTTERY modalities using a bar chart like that in Sec. III, then completed three timed tasks (Fig. 11), each in one of three conditions (internet search always allowed):

(C1 *AI+Code*) No GUI use allowed, 10min;
(C2 *GUI+Code*) No AI prompting allowed, 10min;
(C3 *Mixed*) All PLOTTERY modalities allowed, 15min.

The condition order was randomly C1-C2-C3 or C2-C1-C3. Finally, participants completed a survey and a semi-structured interview to reflect.

PART 2 ***(120min).*** Four participants (P4, P7, P9, and P10, denoted for PART 2 as *P4*, *P7*, *P9*, and *P10*) attended PART 2 for more extended use of PLOTTERY to recreate real-world plots adapted from the first author's work. The plots collectively cover the topics of the 20 most-visited non-tutorial Matplotlib documentation pages [62]. Participants completed a 25-40 minute, experimenter-facilitated PLOTTERY refresher on one plot (Fig. 12), authoring half without the GUI and the other half without AI (C1 and C2, order counterbalanced), then independently attempted the remaining four plots (Fig. 13, order counterbalanced) using all modalities freely (C3 *Mixed*) limited to 40 minutes on each task. We did not expect participants to complete all four tasks given their complexity, and indeed only P7 completed a task on time. For the final 10 minutes, they participated in a semi-structured interview.

Analysis. For PART 1, we performed *quantitative* analysis as follows: (1) task success—whether a participant completed the plot within the time limit—for all tasks; (2) usage counts and durations for each modality for the C3 *Mixed* task; (3) modality switch counts and their cause (either failure from the previous interaction or just user preference) for the C3 *Mixed* task. We further did a *qualitative* thematic analysis of PLOTTERY-relevant behaviors and comments regarding modalities, benefits, breakdowns, and user desired features.

For PART 2, we *quantitatively* measured (1) modality usage counts and durations for the C3 *Mixed* tasks, and (2) which code originated from which modality, with *qualitative* analysis of the code's purpose (*e.g.* add/duplicate/tweak). We also scrutinized the videos to characterize interaction breakdowns.

B. User Study Results

Table I summarizes the takeaways from our evaluation: tasks that might suit each modality, the downsides of each, and findings on modality use. We discuss these below.

Modality Switching Was Fluid. By usage *counts* (Fig. 14), the relative usage of the three modalities was balanced overall, though preferences varied. In PART 1, three participants favored the GUI (P1, P6, P9), three favored code (P3, P5, P10), and four favored AI (P2, P4, P7, P8); two participants never touched the GUI (P2, P7) and two almost never touched the code (P4, P6). PART 2 showed similar patterns of balanced overall usage and individual preferences. Within that balanced

TABLE I: Evaluation takeaways.

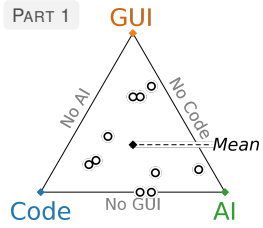
	Suited For	Downsides
GUI	· Straightforward visual changes.	· Overwhelming number of options. · Less helpful than AI. · Still requires API knowledge.
Code	· Quick and small edits. · Duplicating code.	· Requires knowing the plot code and Matplotlib API.
AI	· Starting tasks. · Generating code when unsure of API.	· Can require validation. · Sometimes need Matplotlib vocabulary.

Modality Use:
· Participants exhibited varying strategies for modality use *frequencies*, but usually spent the *most time* with AI and produced *most code* through AI.
· Participants performed ∼3 interactions in a modality before switching.
· Besides scrolling, it was easy to switch modalities.
· Live feedback is a key part of fluidity.

usage, mode switching was fluid. In PART 1, participants performed an average of 3.1 interactions in a single modality before switching, and 2.6 in PART 2. For comparison, every interaction in a random modality would average to 1.5 interactions between switches, so ∼3 interactions at a time indicates some bias to continue in the same modality, but switching modalities was nevertheless easy, a finding also supported qualitatively. Participants found the multi-modal integration *"seamless"* (P1) and *"dynamic"* (P5): *"I didn't spend too much mental effort figuring out [which to use]"* (P1), and seven participants (P2, P3, P4, P5, P8, P9, P10) highlighted the AI's in-situ responses, compared to using tools like ChatGPT in a separate tab. Because of the size of PLOTTERY's GUI, we did notice a lot of scrolling, particularly by *P7*, *P9*, and *P10*, but no participant mentioned it interfered with fluidly mode switching, perhaps because scrolling is normal in Jupyter [63].

Feelings of fluid interaction may depend on *live feedback*. In PLOTTERY, GUI changes are immediately previewed and the code cell is automatically re-run upon AI-generated code. We asked how participants perceived PLOTTERY's multi-modal interaction, and several (P3, P4, P5, P9) highlighted this liveness. Participants appreciated *"immediately seeing the effect [of AI-generated code]"* without copying or pasting (P3), *"[seeing] these things happen in real time [alongside] the underlying code"* (P9), and *"[feeling] easier to explore different options"* (P5) compared to the lack of liveness in code editing, in which one would *"have to rerun the code"* (P4, P5). As additional evidence, the first author accidentally ablated the live feedback halfway through creating one of the figures in the Appendix (from a bug not encountered in the studies), which forced them to manually re-run the cell to see the results of every GUI or AI edit. With the clunkier interface, the feeling of fluidity was lost. These results suggest that liveness is a necessary piece for fluid interaction in PLOTTERY.

The GUI may be suited for straightforward visual changes, but can overwhelm and helps less than AI. Based on their comments (PART 1) and usage (PART 2), participants found direct manipulation in the GUI useful for many kinds of edits, such as *"repositioning the legend [and] resizing the bars"* (P1, *P4*, P6), *"[changing] plot titles [and] tick labels"* (P2, *P4*, *P7*, *P9*), or *"moving things around"* in general (P3, P4). The moment P6 learned about on-plot direct manipulation in the tutorial when using it to resize bars, they exclaimed: *"Oh, my ...! The amount of times I have tried to figure out how to make the bars skinnier..."* Participants also found the GUI useful for adding/removing/restyling a visual component (P4, P6, *P7*, *P9*, *P10*), data-relevant visual changes (P1, P6), or exploring property changes—especially colors (P1, P4)—when they were *"not sure what the exact [argument or variable] would be"* (P5) and did not want to search the documentation (P2), or simply wanted to learn about *"how each thing works in Matplotlib."* Despite the GUI's support for these *straightforward* visual changes, participants struggled to complete their tasks with it: only 3/10 PART 1 participants successfully completed their task in the *GUI+Code* condition with AI ablated (P2, P5, P9), while all participants completed their *AI+Code* task without the GUI. These *GUI+Code* failures surfaced a common pattern: the sidebars show some of *what* could be changed about the visual components, but not *how* to achieve the exact desired changes. Indeed, both Matplotlib experts (P2, P4, P7) and non-experts (P3, P6, P8, P10) felt the need to search online to figure out how to use PLOTTERY to solve the task in *GUI+Code*, as the GUI (compared to AI) sometimes could not help translate user intents to PLOTTERY actions (P2, P4, P6). Additionally, some found the GUI widgets overwhelming (P2, P3, P6) with *"too many things at once"*, most of which they did not know (P3), and inefficient for larger changes when compared to AI (P2, P3, P5, P6). We classified whether mode switches in PART 1 were from goal failure or just user preference. We found no one used the GUI as their fallback when failing in code or AI, and 13 of 27 switches from the GUI are failure-driven (48%),



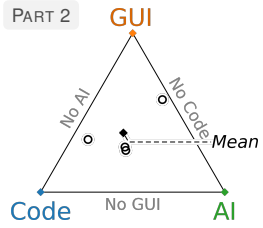
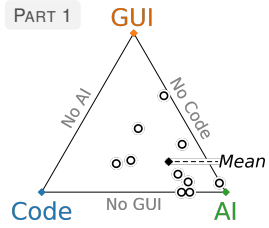


Fig. 14: Relative *number of interactions* in each modality. Individual patterns vary but overall usage is balanced.



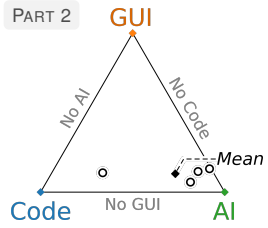


Fig. 15: Proportion of *interaction time* in each modality. For both parts, AI occupies a majority of time and GUI the least.

in contrast to 4 of 32 switches away from AI (13%) and 7 of 25 switches away from code (28%). This does not mean the GUI is unhelpful—52% of GUI interaction sequences ended in success—nor that the AI can subsume the usefulness of the GUI—the GUI can *e.g.* help you discover unfamiliar plotting functions and arguments (*P9*)—but it does reveal that the GUI often presents an overwhelming number of options which users may not know how to navigate.

Despite requiring Matplotlib fluency, direct code editing may be suitable for quick/small edits and duplication. All participants in both parts edited some code. Based on comments (PART 1) and usage (PART 2), participants found code editing suitable for quick and small changes (P1, P2, P4, *P4*, P6, P7, *P7*, P8, *P9*, P10, *P10*), like tweaking parameters previously introduced by AI or the GUI (*P4*, *P7*, *P9*, *P10*). Code editing may also be suitable for repeating existing code (P1, P2, P3, P5), which might have been previously generated by AI (*P9*, *P10*). Some in PART 1 also edited code to gain control over PLOTTERY's AI when in doubt (P3, P4, P9). Still, participants felt code editing requires familiarity with the code (P1, P2, P5, P9) and is *"most straightforward if you have already memorized the correct keywords"* (P5).

AI may be broadly helpful for starting tasks and generating code when unfamiliar with the API, but nevertheless can still require knowing Matplotlib vocabulary. Fig. 15 shows that, by time, AI was the most common modality for 6/10 participants in PART 1 and 3/4 in PART 2. AI also produced more code than other modalities for 8/10 participants in PART 1 and 3/4 participants in PART 2. Participants reached for the AI to make initial progress on a task: 3/4 PART 2 participants always started their tasks with AI, and although 6/10 PART 1 participants started in the GUI, 5/6 of those interactions failed, and it was the AI that led to initial task progress for 8/10 participants. Indeed, participants commented that AI helped initiate a task (P1, P4, P5, P6, P8), generate unfamiliar code (P1, P4, P5, P6, P7, P8, P10), accelerate typing many lines (P2, P7, P10), and replace internet searches (P3, P6). But, successful prompting may sometimes depend on knowing Matplotlib vocabulary. P8, a non-expert, timed out while failing to restyle X-axis ticks with the AI (after already failing with the GUI): some prompts had typos, which neither the AI nor they recognized, and some prompts led to unexpected changes that set back their progress. This single observation in PART 1 was exacerbated in PART 2, where the colorbar on the map task explicitly called for two Matplotlib concepts, but *P4* and *P7* still failed to progress as they repeatedly prompted the AI with descriptions of the plot's appearance rather than Matplotlib-specific terminology.

VI. DISCUSSION

PLOTTERY's goal is to seamlessly augment plot code editing with GUI (and AI) interaction to showcase an ideal plotting workflow that is both as "GUI-like" and "code-like" as possible. Specifically, our design goals (DGs) are to provide:

1) **Familiarity** by imitating a traditional graphics editor;
2) **Fluid iteration** via direct manipulation;
3) **Predictability** in how GUI actions become code;
4) **Compatibility** with code's benefits; *and*
5) **Fluid mode switching** between GUI, code, and AI.

Below we discuss the degree to which we feel we succeeded, what generalizes beyond plotting, and possible future work.

(DG1) A familiar GUI in the AI era. While users seemed to understand "Calls = Layers" and "Arguments = Properties" without trouble, study participants still spent relatively little time with the GUI (Fig. 15) and most code for most participants came from AI. AI reliance may have been exacerbated by our study design: the tasks were well-defined reproduction tasks, whereas real-world authoring may involve more exploring of visual style and more exploring of the GUI interface. Even so, most participants did use the GUI (Fig. 14), and our study does not suggest that AI prompting is a full replacement for graphical interaction. Participants used and appreciated the GUI for straightforward visual changes—it is simply nicer to drag a font size slider until it looks right than to prompt "title bigger" and "title a little smaller" over and over. Indeed, major AI coding tools are introducing direct manipulation for tweaking code-generated visual designs ([64], [65]), albeit applied to code by AI and without any interface for binding properties to existing code (Fig. 3). Nevertheless, the utility of AI in our user study highlights that graphical interfaces, despite promising discoverability [8], do still require learning in a way that AI prompting does not. PLOTTERY's long function list in the ax ▼ menu and long property lists were problems. After PART 1, we added a search box to the ax ▼ menu, but could also add search to the Properties panel or could organize related properties into tabs and groups.

A recent alternative to an overwhelming GUI is dynamic widgets [29], mini-UIs generated in response to AI prompts like in DynaVis [29] or BISCUIT [48]. Although dynamic widgets require prompting first—there is no GUI before an AI interaction—we speculate that such transient widgets might be better than a static GUI in two scenarios: (1) changing uncommon properties, or (2) changes that require multiple GUI steps or are not possible in the GUI. A possible hybrid between dynamic and static GUIs might be to treat the AI prompt box as a search box as well: as the user types it might focus the relevant part of the GUI (similar to CodeMend [32]) or after dispatching an AI prompt it might also open the relevant part of the GUI for further tweaking.

(DG2) Fluid iteration. To give pleasant feelings of direct manipulation [12], PLOTTERY aims to embody the "temporal, spatial, [and] semantic" immediacy suggested by Ungar et al. [66]. *Temporally,* GUI or AI changes are visualized instantly—even just scrubbing through a suggestions list instantly previews each item. *Spatially,* GUI interactions are close to plot elements—to move the legend, the user drags it; to edit the title, the user clicks it. *Semantically,* PLOTTERY imitates a traditional editor—reducing semantic distance to a familiar interface—while still producing idiomatic code. Further immediacy improvements are possible. PLOTTERY

lags on larger plots and multi-plot scenarios. Automatically reducing render resolution might help. Also, users often undo AI results and re-prompt: with a faster AI, PLOTTERY could instead preview the prompt result *before* the user hits enter.

(DG3&4) Predictable, compatible code generation. To make GUI actions predictable, we aimed for a straightforward linkage between code and GUI by treating the user's function calls as a "Layers panel" and their arguments as a "Properties editor". Although simple in retrospect, this choice was not obvious. The first prototype of PLOTTERY followed a different design, like Pylustrator [33], that organized shape selection and property editing around Matplotlib's internal scene graph rather than the user's code. For example, setting the X-axis label and bolding it would involve editing the internal `ax.xaxis.label` shape, producing the code:

```
ax.xaxis.label._text = 'My Label'
ax.xaxis.label._fontproperties._weight = 'bold'
```

But with the "Calls = Layers" and "Arguments = Properties" commitment, PLOTTERY instead produces the idiomatic code:

```
ax.set_xlabel('My Label', fontweight='bold')
```

Moreover, users can *write* that idiomatic code and have GUI support for editing it. PLOTTERY's compatibility with user-written code extends further, to a key reason users plot with code: producing multiple plots. For example, the plots in Fig. 14 and Fig. 15 are one function that is called four times. We can use PLOTTERY to restyle all four plots simultaneously.

(DG5) Fluid mode switching, but not a silver bullet. Adding more interaction modalities into a system, to gain the "best of all worlds", also loads a new burden on the user: they must choose which modality to use for each action [35]. Switching costs can add friction, a pitfall we hoped to avoid. Our attempt to provide *"seamless"* (P1) and fluid [30] interaction seems successful based on the study results. Besides persistent scrolling (which users seemed okay with) we didn't notice troubles switching modalities. Most participants used all the modalities, switching freely after $\sim$3 interactions on average. But, seamless multi-modality is not a panacea for creating Matplotlib plots. While each modality has advantages (Table I), these advantages do not always compensate for the others' failings. With AI misinterpreting some prompts, we observed instances where users floundered despite the presence of all three modalities. Although improvements to the GUI and the AI models (*e.g.* a vision LLM) might help, multi-modality is not (yet) a silver bullet. The Matplotlib API is complicated and reading documentation is still sometimes necessary.

Better Data Binding GUI. In Matplotlib, some kinds of data binding must be encoded manually by the user. For example, in PART 1 task 2 (Fig. 11 middle) participants needed to transform a list of successes/failures into a list of blue/red colors. Constructing such property lists is common in Matplotlib, so GUI support for it, perhaps by simple embedded spreadsheets as in Lyra [3], is the most important missing feature we would like to add to PLOTTERY.

Plotting libraries beyond Matplotlib. PLOTTERY's "Calls = Layers" and "Arguments = Properties" GUI design is well-suited for imperative visualization libraries like Matplotlib that involve a sequence of function calls, each with a potentially long list of arguments. The popular Matplotlib wrapper Seaborn [67] shares similar code structure, so PLOTTERY's design could be adapted to it given appropriate type annotations and updated function lists for the on-plot buttons. D3 [68] and Altair [69] instead involve *multiple chained* function calls to set object properties. To support this, we could treat a whole chain of function calls as a single selectable layer *group*, with the Properties panel showing a vertical stack of panes, one for each call in the chain, for editing each call's properties.

PLOTTERY targets *imperative* plotting. For *declarative*, grammar of graphics (GoG)-based plotting [70] the systems mage [30], Ivy [71], and Blace [31] all offer code editing alongside a GUI. But, these GoG GUIs operate on the visualization grammar JSON, rather than on ordinary programmatic code, which limits their compatibility with common code features like loops, functions, or even just using a variable in scope. A potential next step for these GUIs is to consider how they can more thoroughly integrate into code-based analysis environments like Jupyter notebooks.

Code-GUI inspectors beyond visualization. Organizing the GUI around code structure opens the door to applying PLOTTERY's design to other code-based domains beyond visualization. In particular, the typed-based Properties panel is not specific to Matplotlib. The panel is similar to how IDEs show a type-based autocomplete as the user begins typing an API call. But, the Properties panel instead makes the autocomplete a *persistent* widget. Ordinary IDEs for regular code could show a persistent panel of the arguments for the function call at the mouse cursor. Prong [72] has a version of this that works for JSON-based languages, but type-driven GUIs can generalize beyond JSON or Matplotlib to ordinary programming.

VII. CONCLUSION

We designed and implemented PLOTTERY, a Jupyter notebook extension that integrates code editing with GUI support for creating plots with Matplotlib. PLOTTERY imitates a traditional graphical editor by offering direct manipulation of shapes on the plot, and treating Matplotlib function calls as "layers" in a Layers panel and call arguments as "properties" in a Properties panel. Edits are live, immediately reified in the code. A prompt box lets users ask AI to modify the plot in-place. A small, two-part exploratory user study indicates suitable usages for each modality—GUI for straightforward visual changes, code for quick edits and duplication, and AI for getting started and working around API ignorance—and showed that PLOTTERY enabled seamless, interchangeable use of these modalities. We hope PLOTTERY's design sets a benchmark for future plot authoring and multi-modal programming tools that augment code's benefits—code's in-situ setting with data, flexible expressivity, and automation capabilities—with the addition of fluid GUI and AI interaction.

References

[1] D. Ren, B. Lee, and M. Brehmer, "Charticulator: Interactive Construction of Bespoke Chart Layouts," *IEEE Trans. Vis. Comput. Graph.*, vol. 25, no. 1, pp. 789–799, 2019. [Online]. Available: https://doi.org/10.1109/TVCG.2018.2865158

[2] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. T. Stasko, "Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring," in *Conference on Human Factors in Computing Systems (CHI)*, 2018. [Online]. Available: https://doi.org/10.1145/3173574.3173697

[3] A. Satyanarayan and J. Heer, "Lyra: An interactive visualization design environment," in *Computer graphics forum*, vol. 33, no. 3. Wiley Online Library, 2014, pp. 351–360.

[4] D. Ren, T. Höllerer, and X. Yuan, "IVisDesigner: Expressive Interactive Design of Information Visualizations," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2092–2101, 2014. [Online]. Available: https://doi.org/10.1109/TVCG.2014.2346291

[5] Y. Wang, H. Zhang, H. Huang, X. Chen, Q. Yin, Z. Hou, D. Zhang, Q. Luo, and H. Qu, "InfoNice: Easy Creation of Information Graphics," in *Conference on Human Factors in Computing Systems (CHI)*, 2018. [Online]. Available: https://doi.org/10.1145/3173574.3173909

[6] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister, "Data-driven guides: Supporting expressive design for information graphics," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 491–500, 2016.

[7] H. Mei, W. Chen, Y. Ma, H. Guan, and W. Hu, "VisComposer: A Visual Programmable Composition Environment for Information Visualization," *Vis. Informatics*, vol. 2, no. 1, pp. 71–81, 2018. [Online]. Available: https://doi.org/10.1016/j.visinf.2018.04.008

[8] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, August 1983.

[9] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, https://matplotlib.org/.

[10] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. [Online]. Available: https://ggplot2.tidyverse.org

[11] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-Lite: A Grammar of Interactive Graphics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 341–350, 2016.

[12] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct Manipulation Interfaces," *Hum. Comput. Interact.*, vol. 1, no. 4, pp. 311–338, 1985. [Online]. Available: https://doi.org/10.1207/s15327051hci0104_2

[13] X. Pu and M. Kay, "How Data Analysts Use a Visualization Grammar In Practice," in *Conference on Human Factors in Computing Systems (CHI)*, 2023. [Online]. Available: https://doi.org/10.1145/3544548.3580837

[14] L. Grammel, C. Bennett, M. Tory, and M. D. Storey, "A Survey of Visualization Construction User Interfaces," in *Eurographics Conference on Visualization (EuroVis)*, 2013. [Online]. Available: https://doi.org/10.2312/PE.EuroVisShort.EuroVisShort2013.019-023

[15] S. Chopra, L. Labache, E. Dhamala, E. R. Orchard, and A. J. Holmes, "A practical guide for generating reproducible and programmatic neuroimaging visualizations," *Aperture Neuro*, 2023. [Online]. Available: https://doi.org/10.52294/001c.85104

[16] archiepomchi, "Matplotlib is by the far the most versatile plotting library...you'll never get stuck trying to do something impossible." 2024, https://www.reddit.com/r/datascience/comments/19ech38/comment/kjbw3jk/, comment with 77 upvotes as of December 4, 2025.

[17] M. Ziemann, P. Poulain, and A. Bora, "The five pillars of computational reproducibility: bioinformatics and beyond," *Briefings in Bioinformatics*, vol. 24, no. 6, 10 2023. [Online]. Available: https://doi.org/10.1093/bib/bbad375

[18] jachymb, "Unpopular opinion: Matplotlib is a bad library," 2022, https://www.reddit.com/r/Python/comments/u8j6fn/unpopular_opinion_matplotlib_is_a_bad_library/, 1,100 upvotes as of August 28. 2024.

[19] (deleted), "Anyone else despises Matplotlib?" 2021, https://www.reddit.com/r/Python/comments/p59091/anyone_else_despises_matplotlib/, 711 upvotes as of August 28, 2024.

[20] inconspicuous_male, "Matplotlib sucks and we all know it..." 2022, https://www.reddit.com/r/Python/comments/u8j6fn/comment/i5lv2j1/, comment with 223 upvotes as of August 28. 2024.

[21] NFeruch, "Is it just me, or is matplotlib just a garbage fucking library?" 2024, https://www.reddit.com/r/datascience/comments/19ech38/is_it_just_me_or_is_matplotlib_just_a_garbage/, 684 upvotes as of August 28, 2024.

[22] L. Shen, E. Shen, Y. Luo, X. Yang, X. Hu, X. Zhang, Z. Tai, and J. Wang, "Towards Natural Language Interfaces for Data Visualization: A Survey," *IEEE Trans. Vis. Comput. Graph.*, vol. 29, no. 6, pp. 3121–3144, 2023. [Online]. Available: https://doi.org/10.1109/TVCG.2022.3148007

[23] H. Voigt, Ö. Alaçam, M. Meuschke, K. Lawonn, and S. Zarrieß, "The Why and The How: A Survey On Natural Language Interaction In Visualization," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, 2022. [Online]. Available: https://doi.org/10.18653/v1/2022.naacl-main.27

[24] M. Reilley, "Making Basic Charts in ChatGPT and Claude," 2025, https://www.youtube.com/watch?v=e87egRW8_rU.

[25] Catherine_Pidgeon, "Unlock the power of Copilot in Excel, now generally available," Excel Blog, 2024. [Online]. Available: https://web.archive.org/web/20241111022753/https://techcommunity.microsoft.com/blog/excelblog/unlock-the-power-of-copilot-in-excel-now-generally-available/4242810

[26] Google Workspace, "Generate charts and valuable insights using Gemini in Google Sheets," Google Workspace Updates, 2025. [Online]. Available: https://web.archive.org/web/20250129205245/https://workspaceupdates.googleblog.com/2025/01/generate-charts-and-insights-with-gemini-google-sheets.html

[27] K. Ferdowsi, R. L. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating AI-Generated Code with Live Programming," in *Conference on Human Factors in Computing Systems (CHI)*, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3642495

[28] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3641936

[29] P. Vaithilingam, E. L. Glassman, J. P. Inala, and C. Wang, "DynaVis: Dynamically Synthesized UI Widgets for Visualization Editing," in *Conference on Human Factors in Computing Systems (CHI)*, 2024. [Online]. Available: https://doi.org/10.1145/3613904.3642639

[30] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel, "mage: Fluid Moves Between Code and Graphical Work In Computational Notebooks," in *Symposium on User Interface Software and Technology (UIST)*, 2020.

[31] S. L'Yi, A. van den Brandt, E. Adams, H. N. Nguyen, and N. Gehlenborg, "Learnable and Expressive Visualization Authoring Through Blended Interfaces," *IEEE Trans. Vis. Comput. Graph.*, vol. 31, no. 1, pp. 459–469, 2025. [Online]. Available: https://doi.org/10.1109/TVCG.2024.3456598

[32] X. Rong, S. Yan, S. Oney, M. Dontcheva, and E. Adar, "CodeMend: Assisting Interactive Programming with Bimodal Embedding," in *Symposium on User Interface Software and Technology (UIST)*, 2016. [Online]. Available: https://doi.org/10.1145/2984511.2984544

[33] R. Gerum, "pylustrator: code generation for reproducible figures for publication," *Journal of Open Source Software*, vol. 5, no. 51, p. 1989, 2020. [Online]. Available: https://doi.org/10.21105/joss.01989

[34] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier, "The Jupyter/IPython architecture," in *Fall Meeting Abstracts*. American Geophysical Union, 2014.

[35] T. Lau, "Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI," *AI Magazine*, 2009. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2262

[36] B. A. Myers, J. Goldstein, and M. A. Goldberg, "Creating Charts by Demonstration," in *Conference on Human Factors in Computing Systems (CHI)*, 1994. [Online]. Available: https://doi.org/10.1145/191666.191715

[37] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein, "Interactive Graphic Design using Automatic Presentation Knowledge," in *Conference on Human Factors in Computing Systems (CHI)*, 1994. [Online]. Available: https://doi.org/10.1145/191666.191719

[38] H. Xia, N. H. Riche, F. Chevalier, B. R. D. Araújo, and D. Wigdor, "DataInk: Direct and Creative Data-Oriented Drawing," in *Conference*

on Human Factors in Computing Systems (CHI), 2018. [Online]. Available: https://doi.org/10.1145/3173574.3173797

[39] Victor, Bret, "Drawing Dynamic Visualizations," 2013, http://worrydream.com/#!/DrawingDynamicVisualizationsTalk.

[40] T. Tsandilas, "StructGraphics: Flexible Visualization Design Through Data-Agnostic and Reusable Graphical Structures," *IEEE Trans. Vis. Comput. Graph.*, vol. 27, no. 2, pp. 315–325, 2021. [Online]. Available: https://doi.org/10.1109/TVCG.2020.3030476

[41] M. Marasoiu, D. D. Nauck, and A. F. Blackwell, "Cuscus: An End User Programming Tool for Data Visualisation," in *International Symposium on End-User Development (IS-EUD)*, 2019.

[42] C. Wang, Y. Feng, R. Bodik, I. Dillig, A. Cheung, and A. J. Ko, "Falx: Synthesis-Powered Visualization Authoring," p. 15, 2021.

[43] Y. Tian, W. Cui, D. Deng, X. Yi, Y. Yang, H. Zhang, and Y. Wu, "ChartGPT: Leveraging LLMs to Generate Charts from Abstract Natural Language," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–15, 2024. [Online]. Available: https://doi.org/10.1109/TVCG.2024.3368621

[44] P. Maddigan and T. Susnjak, "Chat2VIS: Generating Data Visualizations Via Natural Language Using ChatGPT, Codex and GPT-3 Large Language Models," *IEEE Access*, vol. 11, pp. 45 181–45 193, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3274199

[45] V. Dibia, "LIDA: A Tool for Automatic Generation of Grammar-Agnostic Visualizations and Infographics using Large Language Models," in *Association for Computational Linguistics (ACL)*, 2023. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-demo.11

[46] kache, "matplotlib is actually really awful and it's hilarious that we had to invent literal AGI for me to start using it," 2024, https://x.com/yacineMTB/status/1744481910670164195, 1,100 likes as of August 29, 2024.

[47] Y. Wang, Z. Hou, L. Shen, T. Wu, J. Wang, H. Huang, H. Zhang, and D. Zhang, "Towards Natural Language-Based Visualization Authoring," *IEEE Trans. Vis. Comput. Graph.*, vol. 29, no. 1, pp. 1222–1232, 2023. [Online]. Available: https://doi.org/10.1109/TVCG.2022.3209357

[48] R. Cheng, T. Barik, A. Leung, F. Hohman, and J. Nichols, "BISCUIT: Scaffolding LLM-Generated Code with Ephemeral UIs In Computational Notebooks," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2024. [Online]. Available: https://doi.org/10.1109/VL/HCC60511.2024.00012

[49] DataBricks, "bamboolib," 2024. [Online]. Available: https://docs.databricks.com/en/notebooks/bamboolib.html

[50] J. Diamond-Reivich, "Mito: Edit a Spreadsheet. Generate Production Ready Python," in *LIVE Workshop*, 2020. [Online]. Available: https://liveprog.org/live-2020/Mito-Edit-a-Spreadsheet-Generate-Production-Ready-Python/

[51] A. Schonfeld and contributors, "dtale," 2024. [Online]. Available: https://github.com/man-group/dtale

[52] Y. Wu, J. M. Hellerstein, and A. Satyanarayan, "B2: Bridging Code and Interactive Visualization In Computational Notebooks," in *Symposium on User Interface Software and Technology (UIST)*, 2020.

[53] N. Shrestha, T. Barik, and C. Parnin, "Unravel: A Fluent Code Explorer for Data Wrangling," in *Symposium on User Interface Software and Technology (UIST)*, 2021. [Online]. Available: https://doi.org/10.1145/3472749.3474744

[54] J. Lehtosalo and Contributors, "Mypy," 2012-2024. [Online]. Available: https://mypy-lang.org/

[55] G. Wheeler, "New matplotlib stubs," 2022. [Online]. Available: https://github.com/microsoft/python-type-stubs/commit/3ae24b5d69542c83e59bf89266d68de1b3f66c45

[56] S. Gillies, C. van der Wel, J. Van den Bossche, M. W. Taves, J. Arnott, B. C. Ward, and others, "Shapely," 2024. [Online]. Available: https://github.com/shapely/shapely

[57] P. J. Contributors, "ipywidgets: Interactive html widgets," 2025. [Online]. Available: https://github.com/jupyter-widgets/ipywidgets

[58] K. Fukahori, D. Sakamoto, J. Kato, and T. Igarashi, "CapStudio: An Interactive Screencast for Visual Application Development," in *Conference on Human Factors in Computing Systems (CHI), Extended Abstracts*, 2014.

[59] R. Chugh, B. Hempel, M. Spradlin, and J. Albers, "Programmatic and Direct Manipulation, Together at Last," in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[60] M. Mayer, V. Kunčak, and R. Chugh, "Bidirectional Evaluation with Direct Manipulation," *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

[61] X. Zhang, R. Xie, G. Guo, X. He, T. Zan, and Z. Hu, "Fusing Direct Manipulations Into Functional Programs," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, pp. 1211–1238, 2024. [Online]. Available: https://doi.org/10.1145/3632883

[62] "matplotlib.org top pages," 2024. [Online]. Available: https://views.scientific-python.org/matplotlib.org/pages

[63] R. Huang, S. Ravi, M. He, B. Tian, S. Lerner, and M. Coblenz, " How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities ," in *International Conference on Software Engineering (ICSE)*, 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00232

[64] J. Ginsberg and R. Lu, "A visual editor for the Cursor Browser," Cursor Blog, 2025. [Online]. Available: https://web.archive.org/web/20251211163315/https://cursor.com/blog/browser-visual-editor

[65] Vercel, "Design mode," v0 Docs, 2025. [Online]. Available: https://web.archive.org/web/20260413034642/https://v0.app/docs/design-mode

[66] D. M. Ungar, H. Lieberman, and C. Fry, "Debugging and the Experience of Immediacy," *Commun. ACM*, vol. 40, no. 4, pp. 38–43, 1997. [Online]. Available: https://doi.org/10.1145/248448.248457

[67] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. [Online]. Available: https://seaborn.pydata.org/

[68] M. Bostock and Observable, Inc., "D3 by Observable: The JavaScript Library for Bespoke Data Visualization," 2026. [Online]. Available: https://d3js.org/

[69] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert, "Altair: Interactive statistical visualizations for python," *Journal of Open Source Software*, 2018. [Online]. Available: https://altair-viz.github.io/

[70] L. Wilkinson, *The Grammar of Graphics*. Springer, 2005, 2nd edition. [Online]. Available: https://link.springer.com/book/10.1007/0-387-28695-0

[71] A. M. McNutt and R. Chugh, "Integrated Visualization Editing Via Parameterized Declarative Templates," in *Conference on Human Factors in Computing Systems (CHI)*, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445356

[72] ——, "Projectional Editors for JSON-Based DSLs," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023. [Online]. Available: https://doi.org/10.1109/VL-HCC57772.2023.00015