Critical Reflections on Visualization Authoring Systems

Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu

Abstract—An emerging generation of visualization authoring systems support expressive information visualization without textual programming. As they vary in their visualization models, system architectures, and user interfaces, it is challenging to directly compare these systems using traditional evaluative methods. Recognizing the value of contextualizing our decisions in the broader design space, we present critical reflections on three systems we developed—Lyra, Data Illustrator, and Charticulator. This paper surfaces knowledge that would have been daunting within the constituent papers of these three systems. We compare and contrast their (previously unmentioned) limitations and trade-offs between expressivity and learnability. We also reflect on common assumptions that we made during the development of our systems, thereby informing future research directions in visualization authoring systems.

Index Terms—Critical reflection, visualization authoring, expressivity, learnability, reusability

1 INTRODUCTION

A new generation of interactive visualization authoring systems has emerged which share a common goal: enabling expressive authoring of visualizations without burdening authors with low-level concerns. They contrast with prior visualization systems that leveraged chart templates, which provide a predefined palette of chart types with only a handful of customization options, powerful libraries that require programming expertise [5], or declarative grammars that require textual specification [40, 41]. Yet, despite this shared goal, these systems make notably different design decisions as they trade off two design dimensions: expressivity, or what visualizations can be created; and learnability, or how difficult is it to author them? Given these differences, traditional evaluative methods are difficult to conduct. For instance, comparative studies are, at best, limited to a small subset of expansive capabilities and workflow permutations that these systems afford, and it is unclear how to fairly quantify and equate different interaction models.

In response, we present critical reflections of three systems from this new generation—Lyra [39], Data Illustrator [18], and Charticulator [34]. Our goal is to extract meaningful lessons about their design choices and, as their creators, we are uniquely poised to conduct the necessary in-depth comparisons of their underlying frameworks, system architectures, and interface design. Critical reflection does not fit the typical pattern for evaluation methods as it requires a completed system and the existence of other related systems; these are impossible when publishing a novel system. However, when the potential situation arises, collaborative critical reflections can reveal definitive strengths, weaknesses, and design rationales between the various systems.

Our critical reflections begin by comparing system components. We find areas of broad overlap (e.g., in the set of marks available for use), radically different approaches (e.g., the degree to which scales are exposed, and how complex layouts are interactively specified), and instances where systems have incorporated each other’s approach (e.g., Charticulator offers two distinct data binding mechanisms drawn from Lyra and Data Illustrator, respectively). Through these reflections, we distill a set of assumptions that underlie and cut-across all three systems. Reflecting on these assumptions, we realize they were necessary to scope our initial research effort, and discuss how relaxing them suggests exciting opportunities for future work.

We hope this paper can be an example of a different type of visualization contribution [17]: a constructive set of lessons learned through critical reflection of the development and public deployment of a set of different but related complex interactive systems.

2 RELATED WORK

In this section, we first present an overview of visualization design and authoring and summarize two types of approaches—programming and interactive systems—in visualization authoring. We also briefly describe the prior theoretical concepts that guided our critical reflections.

2.1 Approaches to Visualization Design

Visualization design is the general process of visualizing data. For the purpose of data analysis, it includes understanding the domain problem, the dataset, the task, the visual representation, as well as the algorithms and computer programs for realizing the design [28]. For communicative visualization, the design process includes generating findings from data, creating the visual representations, as well as annotating the visualizations and articulating them with supplemental material to clearly convey the findings. Visualization design often includes the exploration of design possibilities in an iterative process; designers make decisions on visual encodings, data content, and the overall composition [1].

We consider the narrower activity of visualization authoring, where the author already has a desired visual representation in mind, and has a dataset in the appropriate format. However, here, we discuss considerations for visualization authoring, as understanding this process was crucial to the development of our three authoring systems.

Data sketching offers an unrestricted approach to represent data by physically drawing visualizations that span in fidelity: from throw-away sketches to finished compositions [36, 44]. Related to sketching, vector graphic design applications support rapid, computer-supported design through scene graph representation, composition tools, and transformation actions. Bigelow et al. analyzed the visualization design approach of graphic designers in [1], highlighting flexibility over generative capabilities. Mendez et al. reconcile this tension by proposing a “bottom-up” approach for visualization design, as opposed to the “top-down” approach seen in other visualization design systems [24, 25].

Recently, Elijah Meeks proposed a new way to categorize design approaches based on the underlying convictions and motivations of the data visualization community. Meeks defines these modes as the “Three Waves” of data visualization [22]. The 1st wave focused on Tufte-esque
clarity, simplicity, and thoughtful composition; the 2nd wave focuses on encoding of information based on Wilkinson’s Grammar of Graphics [47] in pursuit of mass-production of visualizations; and the 3rd and current wave is defined by convergence. Meeks argues that a shift has occurred: moving away from designing individual charts and instead converging on the construction, evaluation and delivery of information products responsible for generating a valid chart by charts. In other words, this third wave coincides with a break from chasing the expanding horizon of what visualizations are capable of, and learning from other visualization design approaches to produce better products. Giorgia Lupi echos this sentiment in her “Data Humanism” manifesto [19]: “[the first] wave was ridden by many in a superficial way, as a linguistic shortcut to compensate for the natural vertigo caused by the immeasurable nature of Big Data.” In the current wave, Lupi calls for more meaningful and thoughtful visualization with a focus on the complexity, context, and imperfection of data. In this paper, we take heed of this new wave of data visualization: our reflections take a critical look at the design trade-offs made in pursuit of extending expressive capabilities.

2.2 Computer-Assisted Visualization Authoring

We distinguish between programming-based and interactive approaches to computer-assisted visualization authoring.

2.2.1 Imperative and Declarative Programming

A wide variety of frameworks, libraries, and declarative languages for creating visualizations with textual programming or specification have been created. Further, multiple widely adopted visualization programming toolkits and libraries such as Processing [32], D3 [5], and Vega [41] exist. While such toolkits and libraries achieve the ultimate expressivity, full mastery requires considerable programming expertise. The Grammar of Graphics [47] forms the basis of many declarative visualization grammars, including ggplot [46] and Vega-Lite [40]. Visualization grammars allow authors to use concise and uniform languages to represent a wide variety of visualizations. While easier to write than imperative approaches, declarative grammars still pose a challenge as authors must use text to describe interactive visualizations.

2.2.2 Interactive Systems

In 2013, Grammel et al. [9] surveyed the landscape of visualization construction user interfaces — roughly synonymous to our term interactive visualization authoring. While many new systems have emerged since this survey’s publication [16, 37], its distinction between template editors and shelf construction systems remains valid.

Template editors describe systems like Microsoft Excel, whose expressivity is limited to the set of available chart types. Some systems allow extensive customization by providing a variety of fonts, color schemes, mark types, and graphical styles. Others such as RAWGraphs [21] and Flourish [14] provide APIs for developing new templates, though as with the programming approaches surveyed above, this option is out of reach for authors lacking programming expertise.

Shelf construction is adopted in commercial systems such as Tableau (formerly Polaris [43]) and in research prototypes such as Polestar and Voyager [49]. Instead of using predefined templates, authors map data fields to encoding channels (e.g., x, y, color, shape). The system is responsible for generating a valid chart given the author’s shelf specification. However, these systems do not provide control over the underlying chart layout, nor do they allow authors to easily produce compound glyphs comprised of multiple marks.

Finally, another class of interactive systems can be described as visual builders, and our three systems are members of this class, along with systems such as iVisDesigner [33], Iolver [26], InfoNice [45], Data-Driven Guides [15], DataInk [50], and VisComposer [23]. Though their interfaces tend to be more complicated than those of template- and shelf-based systems, visual builders provide authors with fine control in specifying marks, glyphs, coordinate systems, and layouts.

2.3 Guiding Design Considerations

To guide comparison of Lyra, Data Illustrator, and Charticulator, we draw on prior theoretical concepts for evaluating interactive systems. From cognitive science, both Hutchins et al. [11] and Norman [30] define the gulf of execution and evaluation to refer to the difference between a person’s intentions and the allowable actions (execution) and the amount of effort that the person must exert to interpret the state of the system and to determine how well the expectations and intentions have been met (evaluation). To bridge these gulfs, one must cover both a semantic distance — the potential mismatch between mental models and the system model in forming one’s intention — and an articulatory distance — the effort required to express that intention [11].

Focusing on user interface construction systems, Myers et al. [29] define the threshold and ceiling as how difficult it is to learn how to use the system (threshold), and the expressive gamut of what can reasonably be achieved using the system (ceiling). Along similar lines, in the design of the Protovis [4] and D3 systems [5], Bostock et al. posit high-level design considerations of expressiveness (“Can I build it?”), efficiency (“How long will it take?”), and accessibility (“Do I know how?”). We use these same considerations in our comparisons, but use the term learnability instead of accessibility to avoid confusion.

We also employ the Cognitive Dimensions of Notation framework [3], which provides a set of heuristics for evaluating the effectiveness of notational systems such as programming languages and visual interfaces. Example cognitive dimensions include closeness of mapping (semantic distance of tool representation to domain concepts), hidden dependencies (are important links between entities visible?), hard mental operations (high demand on cognitive resources), and abstraction (types and availability of abstraction mechanisms). This framework has been used to partially evaluate visualization programming approaches such as Protovis [4] and Reactive Vega [42].

In our subsequent critical reflection, we draw on these concepts to help identify and contrast salient differences in our visualization authoring systems, as well as strengths and weaknesses shared by all three. In particular, we focus on the gamut of visualization designs that can be created with a given system, what domain constructs are exposed for expressing such designs, and the physical interactions and mental operations required for forming, as well as reusing, those expressions.

3 BACKGROUND

Our critical reflections focus on the design of three interactive systems for authoring expressive visualizations for communication purposes: Lyra [39], Data Illustrator [18], and Charticulator [34]. These systems use a vocabulary that is based on widely used visualization terms (such as marks, scales, axes, and layout) but subtle differences exist in how each system defines these terms, or introduce new ones. Here, we briefly recap the core functionality of the three systems by describing how they can each be used to recreate A Field Guide to Red and Blue America, a visualization originally published in The Wall Street Journal [31]. We do not expect readers to understand each individual detail, but rather wish to lay the foundation for our subsequent discussion on system components, which sometimes refers back to this example. Video demonstrations of the individual steps and a more comprehensive comparison of terminology are available at https://vis-tools-reflections.github.io.

The visualization uses bar charts to depict how each state’s partisan lean has shifted over time. The bar charts are positioned in a grid structure that mimics the geographic location of their corresponding states. The dataset comprises six columns: the State, GridX- and GridY-coordinates for where this state lies in the grid, Year and PVI which are annual measurements of how the state voted relative to the nation as a whole, and an Inclination field to indicate whether that measurement represents a more Democratic or Republican lean. The original visualization and an excerpt of the data are shown in Figure 1.

Lyra: We drag a rectangle mark from the toolbar to the canvas. To produce one bar chart per state, we add a “Group By” transformation and drag the State field to its property inspector. Doing so introduces a corresponding group mark and nests our rectangle mark within it. We position these groups by dragging the GridX and GridY fields to the width and height drop zones respectively. Doing so sets both the position and dimensions of the groups by producing the necessary
Data Illustrator: We begin by selecting the “Rectangle” shape, and drawing a rectangle on the canvas. We click the “Repeat” button, select the State field from the drop-down menu, and use the handle to drag out a collection of rectangles, roughly one-per-state. To produce a bar chart, we click into this collection to select a single rectangle, and then click the “Partition” button. In the dialog box, we select the Year field to generate one rectangle per-year, per-state. To correctly position these bars charts, we select the collection of rectangles and click the “Break Grid” button. Under the “Composite Position” panel, we click the data binding icon to map the x and y properties to GridX and GridY respectively. We drag the x- and y-axis handles to produce a larger visualization, and drag the peer counter slider out to display all states in the collection. To color the bars, we select an individual rectangle and bind the Inclination field to its fill color. A color legend is automatically added, and we click each legend item to customize the color palette used. To determine the bar heights, we use the “Direct Select” tool and click on the top line segment of a rectangle. In the “Segment Position” panel, we bind the y property to PVI, and drag the y-axis handle to ensure bars rise or fall as appropriate. We follow a similar set of steps to label the charts: adding a text mark to the canvas, repeating it by State, and breaking the grid. We bind the x and y positions to GridX and GridY, and choose to reuse the existing scales in the data binding menu. Finally, we bind State to the text content.

Charticulator: We begin by dragging a rectangle mark from the toolbar to the glyph editor. This step generates one rectangle per tuple in the dataset. To lay them out as a series of state bar charts, we drag GridX and GridY to the x- and y-axis drop zones of the plot segment (the central region in the chart canvas), select it, and choose the “Stack X” sub-layout option from the toolbar. In the configuration panel, we click to toggle the visibility of the x- and y-axes. Dragging PVI to the rectangle’s height drop zone in the glyph editor, and Inclination to its fill color property inspector, correctly sizes and colors the rectangles. Clicking the data binding icon reveals a configuration panel to customize the color palette. To label each state, we select the text mark and, in the glyph editor, click the top of the rectangle mark to anchor the two together. In the “Attributes” panel, we click the “Conditioned By” button, click the Year field in the drop-down menu, and choose the first year. These steps ensure there is only one label generated for each state. Finally, we drag State to the text property inspector, and make adjustments to its position.

4 Critical Reflections on Our System Components

We define critical reflections as informal discussions by system or toolkit builders to concretely define their collective objectives in support of the user community, and candidly assess the ways each system meets or falls short of these objectives. In our case, all system builders met weekly for 1 to 2 hour video conference meetings over the course of 3 months. During these meetings we directly compared our three systems: commenting on our design and implementation, reflecting on practical feedback from the user community, and addressing missed or unexplored research directions. To structure these conversations, we began by considering the eight evaluative dimensions proposed by Ren et al. [35]. We documented these discussions by recording meeting notes in a shared online folder. At times, comparing the systems began by considering the eight evaluative dimensions proposed by Ren et al. [35]. We documented these discussions by recording meeting notes in a shared online folder. At times, comparing the systems required isolated, preliminary reflection on the individual authoring systems which we documented in the shared folder. Each team carried out these isolated reflections as “take-home” tasks before the next weekly meeting. These isolated activities provided the due time to exhaustively consider the ways in which each system met or fell short of our defined objectives.

Through this process, we collectively identified expressivity and learnability as pervasive dimensions. Our process also has the benefit of drawing on our collective experience to find relevant consensus and build shared vocabulary. In this section, we detail the main components the three systems expose and how users create and manipulate them (see Table 1 for the summary).
Table 1: Summary of the main components for the three systems. 📌 denotes what features each system supports and 🈘 denotes how these components are invoked through the user interface. Video demonstrations are available at https://vis-tools-reflections.github.io.

<table>
<thead>
<tr>
<th></th>
<th>Lyra</th>
<th>Data Illustrator</th>
<th>Charticulator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mark Instantiation &amp; Customization</strong></td>
<td>📌 predefined marks only</td>
<td>🌀 custom vector shapes</td>
<td>📌 predefined marks only</td>
</tr>
<tr>
<td></td>
<td>🌟 drag from toolbar</td>
<td>🌟 draw with Pen tool; manipulate path points &amp; segments with Direct Selection tool</td>
<td>🌟 drag from toolbar; draw in glyph editor</td>
</tr>
<tr>
<td><strong>Glyph Composition</strong></td>
<td>🌟 create layers</td>
<td>🌟 group marks</td>
<td>🌟 compose in glyph editor</td>
</tr>
<tr>
<td><strong>Path Points &amp; Path Segments</strong></td>
<td>🌟 map data values to point x, y positions</td>
<td>🌟 partition line marks; draw with Pen tool</td>
<td>🌟 connect glyphs using the Linking tool</td>
</tr>
<tr>
<td><strong>Links between Glyphs</strong></td>
<td>🌟 add a force-directed layout transform</td>
<td>not supported</td>
<td>🌟 connect glyphs using the Linking tool</td>
</tr>
<tr>
<td><strong>Data Scoping for Glyphs</strong></td>
<td>🌟 one or more tuples per glyph</td>
<td>🌟 one or more tuples per glyph</td>
<td>🌟 one or more tuples per glyph</td>
</tr>
<tr>
<td></td>
<td>🌟 no user action needed for one tuple per glyph; “group by” for multiple tuples per glyph</td>
<td>🌟 repeat or partition</td>
<td>🌟 no user action needed for one tuple per glyph; “group by” for multiple tuples per glyph</td>
</tr>
<tr>
<td><strong>Mapping Data Values to Visual Properties</strong></td>
<td>🌟 drag data fields to property inspector or drop zone</td>
<td>🌟 choose data field from drop-down menu</td>
<td>🌟 perform either method</td>
</tr>
<tr>
<td><strong>Scales</strong></td>
<td>🌟 all D3/Vega scales</td>
<td>🌟 scales for categorical, temporal, and numerical data</td>
<td>🌟 scales for categorical, temporal, and numerical data</td>
</tr>
<tr>
<td></td>
<td>🌟 a scale is created when a data binding is applied; a scale can be created manually, and exists independently from data binding</td>
<td>🌟 a scale is created when a data binding is applied; users choose whether to reuse or merge a scale from a previous data binding</td>
<td>🌟 a scale is created when a data binding is applied; by default, reuses a scale from a previous data binding</td>
</tr>
<tr>
<td><strong>Axes &amp; Legends</strong></td>
<td>🌟 an axis or legend is created when a data binding is applied; they can be created independently from scales; properties are customized in configuration panel</td>
<td>🌟 an axis or a legend is created when a data binding is applied; they can be manipulated through on-canvas interaction, which changes the underlying scale</td>
<td>🌟 an axis or a legend is customized through configuration panel; a legend needs to be explicitly added through a button-click</td>
</tr>
<tr>
<td><strong>Relative Layout</strong></td>
<td>🌟 drag a target mark to a host mark’s anchor</td>
<td>🌟 specify through grouping, distribution, and alignment</td>
<td>🌟 specify through anchors, guides, handles, margins, and alignment</td>
</tr>
<tr>
<td><strong>Layout in a Collection</strong></td>
<td>🌟 stacking, force-directed, cartographic projections, and pie charts</td>
<td>🌟 grid and stacking</td>
<td>🌟 grid, stacking, and circle packing</td>
</tr>
<tr>
<td></td>
<td>🌟 add a data transformation via button-click</td>
<td>🌟 apply repeat/partition actions</td>
<td>🌟 use scaffolds and sub-layouts to position glyphs</td>
</tr>
<tr>
<td><strong>Nested Layout</strong></td>
<td>🌟 “group” marks are created when dragging data fields to “group” drop zones or applying a “group by” data transformation</td>
<td>🌟 concatenate repeat and partition actions in a flexible order</td>
<td>🌟 map categorical data columns to X &amp; Y axes and apply sub-layout; embed a Charticulator template as a nested chart</td>
</tr>
<tr>
<td><strong>Coordinate Systems</strong></td>
<td>🌟 Cartesian</td>
<td>🌟 Cartesian</td>
<td>🌟 Cartesian, polar, and arbitrary curve</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>🌟 drag from toolbar into a plot segment</td>
</tr>
</tbody>
</table>

### 4.1 Marks

Instantiating *marks* is a fundamental operation in visualization authoring systems. A mark is a primitive graphical element in a visualization, which includes rectangle, symbol, line, arc, image, and other shapes.

In Lyra, an author may instantiate a mark by dragging the corresponding icon from the marks toolbar to the canvas. Data Illustrator uses a toolbar similar to vector graphics editing applications. An author instantiates a mark by activating the desired tool, and then draw on the canvas by mouse clicks and drags (depending on the system). Charticulator supports both approaches. While the drag-and-drop approach increases directness by requiring just one single action to create a mark, the system will need to provide a reasonable default location and size for the mark. When such choice is not valid, the author may expect unexpected results. On the other hand, Data Illustrator’s interaction introduces a state of the currently activated tool. When the author loses track of this state, s/he may unexpectedly create an unwanted mark. Charticulator relieves this problem by making the activated state one-off — after creating the mark, the system deactivates the mark tool.

Authors sometimes need to create a *glyph* consisting of one or more marks. In Data Illustrator, this is done by drawing marks on the canvas and grouping them. Lyra similarly offers a “group” mark. Charticulator introduces a special canvas called the *glyph editor*, and marks can be instantiated in either the glyph editor or the chart canvas. Providing a separate glyph editor affords a larger editing region for a glyph, and makes it easier to select and manipulate marks. However, it introduces a deviation from conventional single canvas authoring systems. This design also requires the author to be aware that what is shown in the glyph editor is a prototype, and to predict the effect of an action on instances other than the currently selected one. Separating interactions between a glyph editor and a chart canvas also creates confusion, as the author will need to know the roles of the two canvases, and decide where to select and manipulate an object.

While Lyra and Charticulator rely on predefined mark types such as a rectangle, symbol, or text, Data Illustrator supports a wide spectrum of custom shapes by providing a “pen” tool — a common feature in vector graphics software. All shapes besides text and image are represented as paths. A path consists of multiple path points and path segments. An author can freely manipulate path points to achieve a desired shape. In addition, the location of path points and path segments can be bound to data. While this might be hard to learn for authors unfamiliar with vector graphics editing software, having such flexibility over mark shapes increases Data Illustrator’s expressiveness.

In Lyra and Data Illustrator, marks include not only individual elements, but also polylines or curves that connect multiple data points. Lyra supports line and area marks. The \((x, y)\) position for such marks can be bound to data to create lines or areas that connect through these points. In Data Illustrator, a line can be partitioned by a data column, resulting in a line with multiple path points. The \((x, y)\) coordinates of such path points can be bound to data. In Charticulator, however, glyphs (which may contain multiple marks) are by design individual graphical elements. Each glyph corresponds to one or a group of data tuples. To create visual links *between* glyphs, authors need to use the “linking” tool, which draws lines or bands between glyphs. This separation of glyph and link introduces a level of indirectness that impacts learnability, as even basic line charts must use the link tool. However, it increases Charticulator’s expressiveness, making it possible to create charts that have links, such as chord diagrams and arc diagrams.
4.2 Data Binding

Data binding is a core operation in authoring visualizations that involves (1) generating glyphs based on data, and (2) specifying a mapping between data fields and mark properties such as position, color, or size.

4.2.1 What Data Does a Glyph Represent?

In Lyra and Charticulator, a glyph represents one data tuple, respectively. Authors thus need to prepare the dataset to fit this assumption. For example, to create a bar chart, each row in the dataset must correspond to one bar. Based on this assumption, the generation of glyphs is automatic in these two systems. Whenever a glyph is updated in the Glyph Editor in Charticulator, the system automatically generates glyphs, one per tuple; similarly, in Lyra, glyph generation by data is not a user operation. The underlying Vega grammar follows a declarative approach, and considers explicit glyph generation to be imperative. Glyphs are thus automatically generated by Vega when authors specify a mapping between a data variable and a visual property.

Data Illustrator relaxes the requirement of one-to-one glyph-tuple mapping, and allows a glyph to represent one or more tuples through two core operations: repeat and partition. Using the Field Guide to Red and Blue America dataset as an example (Section 3), one can repeat or partition a rectangle using any of the categorical data variables. Repeating a rectangle by State will generate 51 rectangles (50 states and D.C.), and each rectangle represents all the tuples sharing the same State value. The rectangles represented by each glyph are called the data scope of the glyph. The notion of data scope applies to path points and segments too. We can partition a line using State, and each point on the result path represent the corresponding tuples as its data scope.

Data Illustrator’s approach allows authors to dynamically aggregate data via direct manipulation in a more uniform fashion than the other two systems. For instance, authors can create a bar chart where each bar represents a State and the height encodes the average PVI values. To perform the same operation in Lyra, authors need to instantiate a “Group By” data transformation followed by a “Stats” calculation via the left-hand side configuration panels. Similarly, in Charticulator, authors need to specify a “Group By” attribute on the plot segment, which aggregates the data with user-selectable aggregators.

This improved expressivity, however, introduces an additional layer of complexity as authors must verify and keep track of the glyphs’ data scopes. To address this issue, Lyra and Data Illustrator dynamically update the data table panel to reflect the data scope of selected marks.

These differences in glyph-tuple mapping have significant implications on authoring nested layouts, which we discuss in Section 4.4.3.

4.2.2 Mapping Data Values to Visual Properties

Lyra offers two ways for constructing a data binding: data fields can be dragged to a property’s inspector or to a property drop zone, a shaded region that overlays the visualization canvas. Data Illustrator eschews drag-and-drop interaction in favor of button-clicks. Properties that can be data-driven display a binding icon alongside their inspector; clicking the icon reveals a drop-down menu of data fields that the author can choose from. Charticulator adopts both approaches — fields can be dragged to drop zones or can be chosen from a drop-down menu revealed by clicking a property’s binding icon. Figure 2 illustrates these different approaches.

The differences in these data binding interactions are grounded in learnability. Lyra’s design sought to be familiar to Tableau users, while also increasing their sense of directness. Thus, dropping fields to property inspectors recalls Tableau’s “shelves” metaphor while adding additional drop zones to the visualization canvas. Overlying drop zones directly over the properties they correspond to narrows the gulf of execution. A first-use study confirmed these effects as participants described drop zones as “natural” and “intuitive” and, when compared to Tableau’s shelves, made them feel more in control. However, participants also identified two weaknesses in this interaction model: (1) drop zones provide a small active region that can be difficult to hit consistently; and (2) drop zones rely on having a mark selected, which determines how drop zones are depicted, and can be difficult to keep track of after a data binding occurs.

Data Illustrator was intentionally designed to address these shortcomings, and identified two additional concerns. Tableau and Lyra ask authors to perform dragging operations over potentially long distances, which yields a poor experience in terms of both efficiency and accessibility (e.g., for users with motor impairments). Moreover, neither system makes it clear if particular data bindings would result in inexpressive or ineffective outcomes (e.g., binding a quantitative field to an identity channel such as shape). Data Illustrator’s binding icon, in comparison, provides a single interface element that is consistently displayed regardless of the particular property being targeted. Clicking the icon, and selecting a field from the drop-down menu, is more efficient than performing a drag operation, and available fields are filtered to ensure only valid data bindings can be constructed. These gains, however, are offset by a loss of directness — data binding is the only core operation in Data Illustrator that cannot be specified by manipulating the visualization itself.

Charticulator makes the fewest compromises by adopting both the binding icon and drop zone model, and refining the latter in a few crucial ways. When one begins to drag a field, only valid drop zones are shown (and corresponding property inspectors highlighted) to ensure that improper data bindings cannot be constructed. And, the drop zones are visualized in the glyph editor, providing a single consistent place in the interface for interacting with them rather than Lyra’s dependence on the currently selected mark instance. As a result, Charticulator is able to maintain Data Illustrator’s efficiency and accessibility advantages, without sacrificing Lyra’s directness. Echoing Lyra’s evaluation, participants in Charticulator’s usability study rated drag-and-drop interaction as one of the aspects they liked most about the system.

However, there are still opportunities for further improvement. Charticulator’s drop zones, like Lyra’s, provide a relatively small active region and thus require the author to perform a fairly precise interaction. This design can yield a frustrating experience when a author drops a field close to, but not directly over, a drop zone. Inspired by the bubble cursor, an in-development version of Lyra accelerates drop zone acquisition by computing an invisible Voronoi tessellation over each possible drop zone.

Fig. 2: Data binding via dropzones in Lyra (left), via the binding icon in Data Illustrator (middle), and via either approach in Charticulator (right).
valid drop regions. Thus, the drop zone nearest to the mouse cursor is automatically chosen and the author no longer needs to drop directly over a drop zone to successfully establish a data binding.

In all three systems, the outcome of establishing a data binding is immediately reflected on the visualization canvas. This behavior tightens the feedback loop, enabling a more iterative authoring process and reducing the gulf of evaluation [30]. Lyra and Data Illustrator, however, identify a further need to bridge this gulf. They note that interactive data binding results in an accretive authoring process that displays intermediate visualization states to the author. For instance, in Lyra, binding a rectangle’s height before its x-position or width produces a set of overlapping mark instances. In Data Illustrator, a mark’s data scope (i.e., the set of tuples it is bound to) may change over time. Both systems scaffold this experience by exposing the backing data in a persistent tabular interface (Charticulator’s data table is a modal display). Data Illustrator goes a step further by only displaying records in the table that correspond to the selected mark’s data scope.

Although our discussion here has centered on how these data binding interactions are designed to support learnability, there are important expressivity concerns as well. For instance, Lyra’s two data binding mechanisms carry different semantics. When a data field is dragged to a drop zone, the system automatically infers the necessary scale functions and adds appropriate axes or legends to the visualization. This inference does not occur if the field is dropped over a property inspector, allowing for more fine-grained design choices (e.g., if the author wishes to use a scale function they have manually constructed, as described in 4.3) or for authors to bypass scale functions and have visual properties set to data values directly (e.g., when colors carry semantic resonance, as is typical with public transit data, or if data values are produced by algorithmic layouts, see 4.4). However, Lyra does not provide clear affordances communicating these differences in its interface. When an author releases a field over a property inspector, they may notice that the corresponding interface element for a scale function does not appear, but this difference is subtle and requires them to have a sufficient level of expertise to understand the purpose of scales. One could refine this design for learnability, without sacrificing the expressivity gains, by making it opt-in: by default, dropping fields on property inspectors could still trigger scale inference, but a dialog box could allow more advanced authors to bypass it for future interactions.

Data Illustrator’s data binding also carries subtly different semantics compared to the other two systems. If a data binding is removed, marks do not revert back to their previous, unbound appearance (as they do in Charticulator and Lyra). Instead, they return to being standard vector shapes that can be manipulated via drawing interactions like dragging to resize, rotate, or move. This strategy, called a lazy data binding, enables “fuzzy” layouts that approximate the original data values and supports an approach that researchers have found occurs commonly in practice, as designers seek to maintain a flexible and rich design process [1]. While powerful, it is also important to note that this technique is ripe for misuse, making it easier for people to author visualizations that look roughly accurate but with subtle changes introduced. Although it would be impossible for a system to entirely prevent such misleading visualization from being created, there are a number of ways that these outcomes could be mitigated. For instance, the system could warn authors and ask them to confirm their desire to remove a data binding. Marks that are subsequently manipulated could be identified with a persistent warning in their configuration panel, with an option to revert any changes. Or, the output visualization could allow readers to re-establish data bindings and compare the differences for themselves (e.g., via an overlay display or through animation).

4.3 Scales, Axes, and Legends

Our systems all use three constructs to operationalize data bindings: (1) scales, functions that map the data domain to a range of visual values; (2) axes, visualizations of spatial scales; and, (3) legends, visualizations of scales of non-spatial properties such as color, shape, or size. However, how visible these abstractions are to authors, and how the constituent properties are manipulated, vary significant as the three systems make different tradeoffs between expressivity and learnability.

4.3.1 Scale Visibility

When a data binding interaction is performed, all three systems automatically construct any necessary scale functions. However, they lie along a spectrum with regards to the degree these scales are exposed to the author. At one end lies Lyra, which provides scales as first-class primitives: authors are able to manually construct a scale independent of any data binding interaction. At the other end sits Charticulator, which does not distinguish a scale from its axis or legend — controls to modify the domain or range are shown on axis or legend configuration panels. And, Data Illustrator lies in-between. When an author clicks the bind icon they are prompted to create a new scale, or reuse an existing scale if the field has been previously used. Similarly, via the same interface, authors can elect to instead merge scales, producing a single scale with a domain unions across several fields.

The level of visibility has clear implications for expressivity. For example, in Data Illustrator, merging the scale functions enables authoring Gantt Charts where task start and end times are recorded as separate fields. Authors using Lyra have complete control over scale functions. Thus, they can create a scale with a domain that spans multiple distinct data fields, subsequently use it with a field outside of its domain to determine a data binding, and reuse or merge scales in a more fine-grained fashion than in Data Illustrator. This flexibility can be important even for creating simple visualizations. For instance, consider a scatterplot in which the author wants to ensure Euclidean distances are accurately plotted, or an asymmetric adjacency matrix. Both these examples require using a single scale function for both the x and y dimensions, with a domain unioned over two data fields.

However, this expressivity comes with a non-trivial complexity cost. Once an author establishes a data binding, Data Illustrator’s interface does not make clear which scale function is being used — is it a new scale, is it an existing scale, or have scales been merged? Though HCI theory cites maintaining high visibility into system components as an important dimension for reducing the gulf of evaluation [3, 11], on-going feedback for Data Illustrator and Lyra indicate that authors struggle to understand the role that scales play. In Lyra, this issue is compounded by the additional user interface elements that come with first-class scales: a separate panel that lists all available scale functions; and, a corresponding interface element for each scale that can be dragged and dropped, and appears alongside data fields as part of a data binding. Ironically, behavior that was designed to reduce interface clutter — scales, axes, and legends that were automatically created are also automatically removed when they no longer participate in a data binding — has had the knock-on effect of increasing churn in the user interface. How best to address this complexity, without losing the expressivity gains, remains unclear. An “advanced” mode is unappealing, as it introduces additional discovery costs, and would turn this complexity into a cliff rather than smoothing it out. We explore an alternate strategy Lyra might take in the subsequent subsection.

4.3.2 Manipulating Axes & Legends

Data Illustrator and Charticulator couple a scale to its axis or legend representation. Modifications of axis or legend properties (e.g., specifying an alternate color palette) map to transformations of the underlying scale function. Lyra, on the other hand, provides axes and legends as first-class primitives that can be modified independently from the corresponding scale. Lyra’s approach yields expressive gains as well as a less viscous user experience [3] — for instance, axes and legends can be created to visualize scales that do not participate in a data binding, and alternate scales can be chosen without losing any axis or legend customizations. Nevertheless, to address the complexity of scale visibility, Lyra may consider adopting a hybrid approach: scales, axes, and legends remain first-class primitives that can be manually constructed and independently customized, but are coupled if they are created automatically during a data bind.

As Figure 3 shows, authors customize axes and legends via configuration panels in Lyra and Charticulator while Data Illustrator opts for a more direct interaction model: authors can click and drag to reposition axes and legends; handles overlay axes on hover and can be dragged to shorten or lengthen an axis; and clicking individual entries in a color
The Cartesian coordinate system is a fundamental aspect of chart layouts. As such, Lyra, Data Illustrator, and Charticulator all support the Cartesian coordinate system. However, each takes a different approach to address additional coordinate systems or more advanced layouts including algorithmic ones.

Lyra’s grammatical primitives only offer compositional expressivity for the basic Cartesian coordinate system and its “reactive geometry” [41] makes it possible to anchor marks together for additional layouts (e.g., stacking, pie charts). Lyra supports more advanced layouts (e.g., treemaps) using modules that can be invoked from button presses/property inspectors, often requiring a specific data format (e.g., geographic, hierarchical, network data); people need to choose from a typology of data transformations. In other words, Lyra’s different specification options result in an inconsistent interaction model.

In contrast, Data Illustrator’s repeat and partition operators offer a consistent mechanism for basic and advanced layouts, but important implementation and conceptual limitations remain. Support for polar coordinates is conceptually well-defined though it has yet to be implemented. For instance, partitioning a circle would produce a pie chart. However, how these operators extend to support geographic, hierarchical, or network data is an open question.

Charticulator’s system of constraints offers the most uniform underlying model for layout, enabling coordinate systems such as polar and arbitrary curve coordinates, in addition to Cartesian. Advanced layouts can be implemented as a module and incorporated into Charticulator via an additional panel (as Lyra did). However, because this is not aligned with one of the core design goals (i.e., enabling people to specify a novel layout using a set of partial constraints), we acknowledge these additional algorithmic layouts as a known limitation.

5 Critical Reflections on Our Shared Assumptions

The previous section explored the ways in which our three systems differ with respect to the tradeoffs between expressivity and learnability. To complement our reflection on differences and tradeoffs, we now reflect upon our systems’ shared assumptions about the authors who use our systems, the data that authors load into our systems, the tasks that they perform, the system requirements, and the content that authors export from them. We now have greater clarity regarding these assumptions, assumptions that were not explicitly defined when we set the scope of our research and developed these systems, when our priorities were primarily expressivity and novelty. We acknowledge that these priorities likely hindered the adoption of these deployed systems to some extent. Reflecting on these simplifying assumptions provides opportunities for further refinement to drive adoption as well as directions for future research.

5.1 The Author: Literacy & Skill Transfer

A shared assumption underlying our systems is an author’s desire to visualize data without programming while still exhibiting a level of comfort with computational thinking; otherwise they would have elected to manually illustrate their charts (e.g., using pen & paper or a vector graphics application). We further assume some level of data literacy, as authors would need to understand the structure and type of data that can be loaded into the systems.

Another shared assumption pertains to familiarity with other systems. Given Data Illustrator and Charticulator’s ties to Adobe and Microsoft, respectively, they may attract authors who also use other systems produced by these organizations. For instance, experience using Adobe Illustrator may help people as they learn to use Data Illustrator, while a familiarity with configuring custom visuals for Microsoft Power BI may contribute to the process of learning to use Charticulator. However, skill transfer may occur from many other sources, and thus attaining a better understanding of skill transfer for learning our systems is an important direction for future research.

5.2 The Data: Cleaned & Pre-Processed

With regards to the data that people load into our systems, our common assumption is that the data is an appropriately formatted CSV, TSV, or JSON file. Based on the example charts showcased in the systems’ respective papers and online galleries, the size of the datasets in these files is also small: a handful or columns and dozens of rows, or a few...
hundred rows at most. There are no missing or erroneous values, and the data has already been appropriately cleaned, filtered, and aggregated using some other set of data preparation and analysis tools. Finally, the data is static, meaning that the structure or values of the data will not change at some later point in the authoring process.

All three systems exhibit an issue that we call **schematic congruency**: authoring a visualization may require the backing dataset to be structured or formatted in a particular way that may not be clear to authors a priori. In particular, all three systems expect datasets to be structured in a long (often referred to as “tidy,” and as opposed to wide) format. Data Illustrator and Chartistulator do not provide facilities to transform the backing data beyond sorting and calculating summary statistics such as the mean or median. Lyra, on the other hand, provides a palette of data transformations including operators to filter and aggregate the dataset, as well as derive new calculated fields. An additional set of layout transformations support ingesting non-tabular data (e.g., networks or geographies) but they impose their own set of structuring and formatting concerns that are invisible to the author. For example, the nodes and links of a network must be imported as two separate datasets, with each link described by properties named source and target exactly.

Although supporting a rich set of data wrangling capabilities is out of scope, how these systems should narrow schematic congruency remains an open question. As Lyra illustrates, it is not sufficient to simply extend the underlying visualization models to provide data transformation capabilities. Instead, these systems must develop higher-level scaffolding that automatically infers or suggests appropriate transformations when necessary, analogous to their existing mechanisms for data binding which automatically infer definitions for scales and guides.

### 5.3 The Task: Authoring, not Designing

Another fundamental assumption is that people want to **author** a chart, **not design** one. In other words, we assume that users do not wish to explore the visualization design space using our systems but rather come with a specific chart design in mind. Perhaps their design is something they sketched on paper, or maybe they want to emulate a design they saw elsewhere (e.g., in our systems’ associated galleries). This assumption further implies that the author is already acquainted with their dataset — we imagine the author to have already performed some exploratory data analysis and preparation prior to using the systems.

It is unclear how these assumptions hold up in practice. For authors without a specific design in mind, or for those with little or no understanding of their dataset, our systems are of little help as they begin with a blank canvas, a **tabula rasa**. Moreover, these assumptions yield largely linear interaction flows. While all systems present several different entry points to author a given design, once one starts down a particular path it can be very difficult to make a change and often involves starting from scratch (a highly viscous user experience [3]). This process stands in stark contrast to how design practice prioritizes a broad exploration of the design space (e.g., through alternating flare and focus phases [6] or parallel prototyping [8]).

Though all three systems have been released, with deployed versions were considered, what their relative strengths and weaknesses were, and may yield more valuable insights than reporting trivial usability issues or conducting contrived comparative user studies. Reflecting on a single system would involve not merely articulating the rationale for the design choices made, but discussing what alternate prototypes were considered, what their relative strengths and weaknesses were, which aspects were incorporated into the final version, or why they were ultimately discarded.

### The Devil is in the Details

We note that, even for the creators of one of the three systems, it would have been infeasible to acquire these insights by simply reading the individual papers. As there were significant inconsistencies between the abstractions each system exposed, all stakeholders needed to actively participate in extracting and mapping intricate low-level details — the heart of the critical reflections process. Detailed discussions of design rationales helped us understand the full extent of the design space and revealed points that were often new to one or more of us. Distilling this information into a series of high-level takeaways would have defeated our original motivation; we believe this because the reason any particular system favored expressivity or learnability was often the result of a series of nuanced and intertwined design choices. We believe this paper provides a new type of contribution to the visualization community [17], and demonstrates the value in making collaborative progress through in-depth critical reflection.


