How Do Observable Users Decompose D3 Code? A Qualitative Study

Melissa Lin* †
Carnegie Mellon University

Heer Patel* ‡
University of Washington

Medina Lamkin ‡
University of Washington

Hannah Bako §
University of Maryland

Leilani Battle ‡
University of Washington

ABSTRACT

Many toolkit developers seek to streamline the visualization programming process through structured support such as prescribed templates and example galleries. However, few projects examine how users organize their own visualization programs and how their coding choices may deviate from the intents of toolkit developers, impacting visualization prototyping and design. Further, is it possible to infer users' reasoning indirectly through their code, even when users copy code from other sources? We explore this question through a qualitative analysis of 715 D3 programs on Observable. We identify three levels of program organization based on how users decompose their code into smaller blocks: Program-, Chart-, and Component-Level code decomposition, with a strong preference for Component-Level reasoning. In a series of interviews, we corroborate that these levels reflect how Observable users reason about visualization programs. We compare common user-made components with those theorized in the Grammar of Graphics to assess overlap in user and toolkit developer reasoning. We find that, while the Grammar of Graphics covers basic visualizations well, it falls short in describing complex visualization types, especially those with animation, interaction, and parameterization components. Our findings highlight how user practices differ from formal grammars and reinforce ongoing efforts to rethink visualization toolkit support, including augmenting learning tools and AI assistants to better reflect real-world coding strategies.

Index Terms: Visualization toolkits, Code reuse.

1 INTRODUCTION

From designing bespoke visualizations in D3 [8] to orchestrating multi-chart interactions in Vega-Lite [28], visualization programming is a valuable skill for the data science workforce [26]. However, users may struggle to write their own customized programs [4], even when adapting existing examples [6], making it challenging to adopt more expressive toolkits like D3 [27]. One popular solution is to generate code templates [18] that users can populate with data variables [1,12]. However, templates do not necessarily reflect how users themselves may reason about their own code [4]. Broadly, we observe relatively few research projects investigating how visualization toolkit users organize their code, or how these choices impact code comprehension and reuse [6, 21]. We focus on D3 given its popularity and complexity [4, 6, 12].

In response, we analyze how D3 users organize visualization code to understand what code structures these users find intuitive. We apply qualitative methods to investigate an established measurement of code comprehension : code decomposition [9, 24, 29]. The abstraction and pattern-matching skills necessary for code decomposition make it a critical pillar of *computational thinking* that helps to distinguish novice programmers from experts [29, 34]. Further, practicing program decomposition helps students develop *metacognitive awareness*, and researchers have analyzed decomposition strategies to observe students' mental models of code in the classroom [9]. With visualization languages, users must reason about their code *and* visualization designs simultaneously [4, 6, 21], suggesting that decomposition strategies could be analyzed to gain insight into users' mental models of visualization programs. Through this analysis, we can establish frameworks for users' mental models to ground visualization programming and design tools in actual user behavior. Towards this goal, we address two core research questions: *How do D3 users (re)organize code copied from outside sources* (**RQ1**)? Further, *do D3 users really organize their code according to toolkit designers' recommendations* (**RQ2**)?

To answer these questions, we contribute a qualitative analysis of D3 code decomposition strategies across 715 Observable [19] notebooks representing 24 distinct visualization types. Given D3 users often copy from existing examples [6, 12, 14, 18], we focus our first analysis on how Observable users organize copied code (answering **RQ1**). Our findings reveal three distinct granularities for decomposing D3 programs into smaller code blocks: component, chart, and program level. Component-Level decomposition was the most prevalent strategy. We complement our analysis with interviews of 7 Observable users, who confirm that they are purposeful in how they structure their D3 code and share how their code structuring is influenced by (inferred) toolkit and community best practices.

To understand how users' mental models align with existing toolkit design paradigms (answering **RQ2**), we compare common user-made D3 components with the Grammar of Graphics (GoG). We find high component overlap for common chart types such as bar charts but diminishing coverage for more complex ones such as streamgraphs. Further, several commonly used D3 component types, such as interaction and parameterization, fall outside the GoG, suggesting gaps between theory and real-world usage. These user-driven insights validate ongoing efforts in visualization grammar development and reveal new opportunities for building D3 support tools and resources that better reflect users' code organization strategies. In summary, this paper makes four contributions:

- A qualitative analysis of 715 Observable notebooks identifying three levels of code decomposition and the impacts of code copying on decomposition strategies.
- An interview study with 7 Observable users, which corroborates our qualitative findings and clarifies user rationales for adopting different decomposition strategies.
- A comparison of user-made D3 code components with the GoG, revealing actionable gaps between theory and practice.
- Design implications for tailoring educational materials and AI-driven support to match real-word toolkit usage.

2 RELATED WORK

Visualization Code Reuse: Copying from examples is a key user strategy for creating D3 visualizations [2, 4, 6]. For example, in their analysis of 37,815 D3-related posts on Stack Overflow, Battle et al. observe that 14% of them reference just three sources:

*Both authors contributed equally to this research
†e-mail: mylin@andrew.cmu.edu
‡e-mail: [heerpate, mlamkin, leibatt]@cs.washington.edu
§e-mail: hbako@umd.edu

Observable, the D3 gallery, or Bl.ocks.org [6]. However, *D3 code often contains uncommon syntax and code structures, making it difficult to reuse* [6]. In-depth examples may over-complicate prototyping [6] or even lead to design fixation [20]. Example galleries also facilitate reuse. Yang et al. find that while users want larger galleries, creators struggle to maintain them [35]. Code reuse also has pedagogical value. Recent work showed students who adapted D3 examples created more bespoke visualizations and improved their understanding of the code [13]. Our work complements these findings by studying how example reuse impacts code organization, revealing the impact of galleries on coding structure.

Visualization Templates: Several projects aim to support code reuse through *code templates*. For example, Bako et al. contribute templates for common D3 visualization and interaction types [1]. Harper et al. propose techniques for converting existing D3 visualizations into templates [12]. Tools such as Ivy generalize these ideas to aid creation and reuse [18]. However, templates are difficult to modify beyond their defined parameters, potentially impeding users' creativity and workflows [4]. Recent work has introduced structured methods for evaluating toolkit notations, reflecting a growing need to systematize how users and developers articulate data transformation and visual mapping [17]. Similarly, we aim to strengthen the connection between toolkit usage and design.

Code Decomposition and Visualization Grammars: Code decomposition is an established measure of student comprehension and reasoning in CS education [24, 29, 34]. For example, analyzing program decomposition strategies can elucidate relationships between students' metacognitive awareness and assignment outcomes [9]. Outside the classroom, notebook decomposition strategies have been analyzed to measure how users reason about data science workflows [23, 25] and automatically reorganize computational notebooks to improve clarity [30]. Recent work also studies recurring patterns in D3 program structures across GitHub, Bl.ocks.org, and Observable [1, 4, 12]. We adopt a similar analysis approach. Note that while Observable's cell structure makes it easier to decompose code, researchers have been observing similar D3 program decomposition prior to Observable as well [1, 6].

Formalisms such as the Grammar of Graphics [33] and Layered Grammar of Graphics [31] have guided toolkit design for many years and continue to influence recent work (e.g., [7, 15, 16]). However, *it is unclear whether they align with how users reason about code.* One study found that users often reuse examples based on layout or task similarity, factors not captured by formal grammars [3]. Analyzing code structure offers a way to bridge this gap [21].

3 Data Collection & Preparation

We collected a diverse range of 240 Observable [19] notebooks spanning 24 visualization types identified in previous work [5]. We followed a strategy of searching by visualization type and screening notebooks for quality and uniqueness, detailed in supplemental materials (section 7). Similar to prior work [1, 2, 6], we observe that most Observable notebooks are *duplicates* that copy code from older notebooks and only make minor revisions such as importing a different dataset. Given the importance of code copying in creating new D3 programs (see section 2), we also collected ≈ 20 duplicates for each of our initial 240 notebooks (10 per vis. type), which expanded our dataset to 715 total notebooks. In the paper, we focus on the 240 unique notebooks. Since 475 of 715 notebooks are duplicates, one can extrapolate our results to the broader corpus.

Further, we define the following terms for our analysis: **decompose** refers to how users "organize," "structure," "break down" and otherwise separate code within a single Observable notebook using cells [10]. **Modularity** refers to the extent to which users decompose their code into separate pieces, i.e., modules such as code cells [6, 24]. **Sources** are D3 programs that provide code for other programs. It could be a notebook on Observable (e.g. from the D3 Gallery) or an external D3 program (e.g. from GitHub Gist).

4 Can We Infer User Reasoning From D3 Code?

We seek to understand how Observable users reason about D3 programs and whether this can be determined indirectly through observing users' code decomposition strategies. However, the visualization literature is unclear on how purposefully users organize reused code compared to code written from scratch. To this end, we perform a mixed methods evaluation of our corpus to **examine** notebook code structure (subsection 4.1), **inspect** code copying of sources (subsection 4.2), and **compare** decomposition strategies of corpus notebooks to their source notebooks (subsection 4.3).

To develop our codebook, a random sample of 15 notebooks from the six most popular D3 visualization types (observed in [1, 5, 6]) were collected with the procedure in section 3. *These 15 notebooks were distinct from the 715 notebooks in the main corpus.* The two lead authors independently annotated these notebooks, after which the entire author team met to discuss the codebook. The lead authors re-annotated the 15 notebooks again, achieving a Cohen's Kappa inter-rater reliability score [11] of 0.941. Finally, the lead authors coded the 240 unique notebooks collected from section 3 over 15 weeks. Discrepancies were resolved through discussions. In this section, we detail findings from our coding (the full codebook can be found in our supplemental materials).

4.1 How Do D3 Users Decompose Programs?

We examined corpus notebooks to identify the number and functionality of code cells and the relationships between cells, such as variable and function dependencies, which revealed three high-level code decomposition methods.

Component-Level decomposition was the most common strategy observed in our analysis, appearing in 83.8% of notebooks. Users create each code cell as a distinct *component*. Each component builds on previously defined components to implement a visualization (see Figure 1). Individual components also tend to fall under one step of the visualization process (e.g. importing the dataset, specifying encodings like positional scales).

Chart-Level decomposition was present in 7.1% of notebooks. Illustrated in Figure 1, it organizes code by the target output (i.e., an entire visualization) instead of visualization steps. Users create a *helper function cell* that generates the target visualization type.

Program-Level decomposition was present in only 6.3% of notebooks. Code is *not* decomposed and instead is placed all inside a single Observable cell. We acknowledge that users could be using white space to segment their code (see section 2), but the low prevalence of this strategy suggests that users are deliberately choosing to forego decomposition. We interview users about their decision-making strategies when organizing D3 code in section 5.

Only 2.9% of notebooks show both Component- and Chart-Level structure (see supplemental materials for examples of all strategies). Overall, our findings suggest that Observable users have specific ways they prefer to organize their code. Further, we may be able to **understand how users reason about their D3 code by observing how they (re)structure copied code by decomposition strategy (H1)**, which we explore in the following sections.

4.2 How do D3 Users Leverage Copied Code?

To explore our hypothesis **H1** from subsection 4.1, we analyze code reuse in Observable notebooks. Verbatim copying may indicate that Observable users are not thinking deeply about their D3 programs. However, if we observe a range of code-copying patterns, this may be indicative of deliberate reasoning processes.

We identify four code-copying strategies; two involve no direct code reuse: *"original creation,"* where users built notebooks from scratch (136 out of 240 total notebooks, 56.7%), and *"orthogonal code forking,"* where users forked a notebook but did not reuse

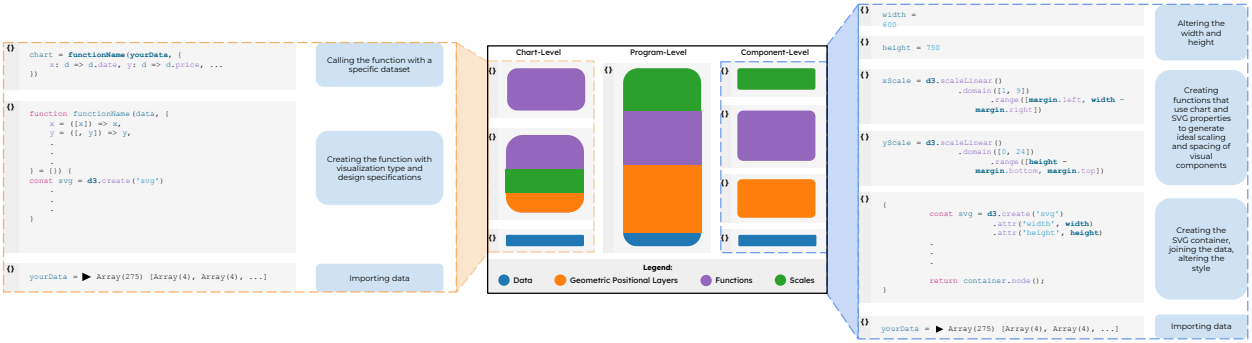

Figure 1: Centered are three abstracted notebooks with four color-coded visualization components. Program-Level has all four components in one code cell. Chart-Level has a function code cell, a function building code cell, and a data code cell. Lastly, Component-Level has all four components in different code cells. To the left is a sample of Chart-Level Decomposition. To the right is an example of Component-Level Decomposition.

any code (5/240 notebooks, 2.1%). The remaining two strategies involve code reuse: *"Observable sourced,"* where code was reused from another Observable notebook (86/240, 35.8%), and *"outside sourced,"* where code was copied from external platforms such as GitHub (13/240, 5.4%). Over half of the notebooks did not copy code; of those that did, most sourced from Observable. These percentages represent the proportion of notebooks labeled with reuse strategies (Observable- or outside-sourced) relative to the full dataset. We analyze how code copying influenced selected decomposition strategies in subsection 4.3.

4.3 How Does Code Copying Influence Decomposition?

To explore **H1** further, we refine our hypothesis: if Observable notebooks are a faithful reflection of the author's reasoning process, we posit that notebook authors will (1) write code from scratch that reflects their preferred decomposition method or (2) make substantive changes to copied code if the source clashes with how they prefer to decompose D3 code. To test this, we compared decomposition strategies in notebooks with copied code to their sources (i.e., the notebook from which the code was copied). Two key patterns emerged. First, 76.2% of notebooks *retained Component-Level decomposition* when it was already used in the source. Second, when changing strategies, *users shifted exclusively toward Component-Level decomposition*. For example, 19.8% shifted from Chart-Level to Component-Level and 9.3% from Program-Level to Component-Level. Further, zero notebooks shifted in the opposite direction.

13 Observable notebooks sourced code from outside Observable. A manual review of their code suggests that 84.62% these notebooks still employed Component-Level decomposition, similar to previous findings [1]. These findings suggest that **Component-Level decomposition is not simply a byproduct of Observable's cellular environment and may reflect how users generally reason about D3 code**. Similarly, all 7.9% of notebooks with inaccessible deprecated D3 Gallery notebook sources displayed Component-Level decomposition. Lastly, 80.9% of the Original and Orthogonal notebooks used Component-Level decomposition. Together, these results suggest that **users prefer Component-Level decomposition, even if they do not inherit any code.**

4.4 Section Summary

Our findings show that users reason about D3 code at the Component level, whether coding from scratch or reusing copied code, suggesting that **Component-Level decomposition aligns with how users reason about D3 and supporting H1**.

5 Validating Our Findings With User Interviews

Our analysis suggests that the structure of a D3 program could be used as a proxy for inferring how its author reasons about visualization code. To further understand how decomposition strategies relate to how people think about their code, we conducted an IRB-approved study with N=7 D3 users about their code organization on Observable. Participants, recruited via professional networks and Observable, varied in age, education, and occupation (details in supplemental materials).

5.1 Interview Protocol

Participants were given an overview of the study and asked to sign a consent form. They also completed an optional survey on demographics and experience with statistics, visualization, and Observable. Interviews were conducted on Zoom, lasting an average of 34 minutes each. Interviews began with participants sharing an Observable D3 notebook, explaining its purpose, code organization, and debugging strategies. They were then asked about their D3 and Observable use, influences, and reliance on sources.

5.2 Emerging Themes from the Interviews

All participants shared specific reasons for their code organization. **P4** and **P6** built functions to facilitate reuse. Participants also discussed *thoughtful deviations* from usual strategies. **P5**, who primarily uses Component-Level decomposition, used function calls for previously built charts to keep his visualization dashboard neat. **P4** sometimes puts all the code into a single cell to test new ideas or for single-use visualizations. **P7** uses multiple new cells when exploring new ideas and later streamlines into a single function.

Using certain decomposition levels also helped participants *achieve specific goals*. **P2** mentioned his team found that using a component approach to organizing the code, instead of single cells, led to better visualization rendering in their web application. **P3**, who used Component-Level decomposition, explained that building a visualization step-by-step helped him understand the code better. **P1** used new cells to enable interactions with his visualizations.

Debugging was a common challenge, in part due to JavaScript's silent failures [6]. **P2** and **P3** pointed to splitting code into distinct cells (i.e., Component-Level decomposition) as *helping identify problems*. **P1** and **P7** typically do not use Component-Level decomposition, but created new cells to debug. This aligns with prior work, which finds that Component-Level decomposition can make assignments easier to debug and faster to complete [9].

Lastly, participants shared how they learned to *structure their code from examples*. **P3** credits learning sources for shaping his Component-Level coding style. **P4** revised his function-writing style after finding D3 co-creator Mike Bostock's code organization to be clearer. **P6** discussed *"It's easy to learn [using Observable]. I can go open anybody's notebook [and see] this is how they have written it... I can say that I learn from other people's code."* **P6**'s

Table 1: Percent Coverage of Component by GoG Across Vis Types

Visualization Type	GoG Coverage	Visualization Type	GoG Coverage
Grouped Bar Chart	81.3%	Box Plot	65.3%
Line Chart	78.6%	Hexabin	65.2%
Pie Chart	75.6%	Heatmap	62.6%
Bar Chart	75.4%	Chord	61.2%
Stacked Bar Chart	74.2%	Word Cloud	59.7%
Geographic Map	74.1%	Waffle Chart	59.3%
Scatterplot	73.9%	Bubble Chart	57.1%
Area Chart	70.7%	Sunburst	56.8%
Radial Chart	68.3%	Sankey	55.8%
Parallel Coord.	66.9%	Treemap	54.5%
Graph	66.4%	Voronoi	53.9%
Donut Chart	65.8%	Streamgraph	48.6%

code structure looks very similar to Observable D3 Gallery notebooks since he frequently relies on them for inspiration.

Combining these interview findings with our results from section 4, we argue that **users deliberately structure their D3 code to match how they reason about visualization programs, with a common preference for Component-Level decomposition.** We acknowledge that future research is needed to quantitatively validate **H1**. However, we believe these results are a strong starting point for approximating user reasoning through D3 components.

6 What Can We Learn From Users' Decomposition Strategies in D3 Code?

Inspired by prior work [21], we reuse our results from section 4 to compare user-made D3 components with the Grammar of Graphics (GoG) [32] and Layered GoG [31], which formalize how toolkits and languages should ideally be structured. While established formalisms often reflect how toolkit developers think, they do not necessarily reflect how end users themselves reason about visualization code [21]. Thus, we seek to identify and characterize misalignments, which could reveal opportunities to improve toolkit usage and design. Note that we refer to the GoG and LGoG collectively as the GoG (Data and Aesthetic Mappings, Statistical Transformation, Geometric Object, Scale, and Coordinate System overlap with the LGoG; the GoG additionally includes Data Transformation).

6.1 Coverage By Visualization Type

We calculate GoG overlap based on component counts for each visualization type in Table 1. Common visualization types have greater GoG overlap. For example, Battle et al. report Geographic Map, Line Chart, Bar Chart, and Scatterplot as the most popular visualization types in their analysis of D3 usage [5], which also have high overlap with the GoG. As complexity increases, GoG overlap drops. This suggests that **the Grammar of Graphics supports common charts but is less effective for representing customized designs, where D3 is considered more useful** [12, 35], answering **RQ2**. The diminishing returns between the GoG and D3 suggest an exciting opportunity to infer higher-level abstractions from real-world toolkit usage. Also, we emphasize that these results generalize to the full dataset of 715 notebooks, including duplicates.

6.2 Isolated Components

We examined components that were often isolated in their own cell in Table 2, indicating deliberate structuring decisions. 72.9% of interactions are isolated, likely due to their complexity, which motivates their separation from other components [4, 28]. Only 54% of animations are separated, a likely side effect of how they are structured as calls to existing components, making them harder to reason about and debug. These findings provide usage-driven support for current (e.g., [28]) and future efforts to cover overlooked areas like parameterization, animation, and interaction.

Table 2: Percentage of isolated instances of each component.

Component	Percentage Isolated
Animation	54.29%
Coordinate System	70.73%
Data and Aesthetic Mappings	93.83%
Data Transformation	87.66%
Geometric Object	74.42%
Interaction	72.90%
Graph/Tree Layout	57.14%
Parameterization	86.35%
Scale	87.50%
Statistical Transformation	61.11%

7 Discussion: Implications for Future Research

Analyzing D3 Usage to Assess Existing Theory: To the best of our knowledge, our paper provides the first qualitative analysis of overlap between D3 usage and the GoG [32]. For **RQ2**, we find high alignment for simpler charts like bars and lines, while complex types such as word clouds and Sankey diagrams fall outside the GoG framework. Notably, usage of essential D3 features like animations, interactions, layouts, and parameterization is not covered by the GoG [28]. While the GoG is not expected to capture every use case, our findings provide *usage-driven evidence* for augmenting visualization grammars such as by validating recent developments in interaction grammar design (e.g., [28]).

Leveraging Decomposition Strategies for Learning: Our user study shows users purposefully adopt decomposition strategies to support their workflows both when coding from scratch and when reusing code, thus answering **RQ1**. Educators can leverage our findings by *modularizing tutorials* or assigning Program-Level code and observing how students restructure it as an *informal assessment*. AI tools could also be trained to organize and label code by components (e.g., data handling, interactions) and provide component-focused explanations, producing more intuitive, well-structured output. In preliminary tests, we observed mixed results when prompting LLMs to label and generate tutorials for individual components. We observed notable hallucinations and errors, especially in the tutorials, revealing opportunities for future research.

Developing Resources to Enhance Visualization Design Through Component Reuse: Component-Level decomposition can support efficient design by mapping common visualization components across multiple notebooks and visualization types. A resource linking reusable components across visualizations could help users quickly construct custom charts without starting from scratch [22]. In this way, users can explore D3 code along two levels of abstraction simultaneously: Component-Level semantics and the low-level D3 syntax. Such analysis may even enable automatic component detection in other languages, helping to synthesize language abstraction levels automatically (e.g., generate Vega-Lite analogs for more complex toolkits). By providing a more structured, component-based pathway to visualization design, this method could *facilitate faster design iteration and experimentation.*

Limitations. Notebooks were collected from Observable, representing a subset of D3 users. Some Observable users also use inaccessible private notebooks. While we employed multiple strategies to increase the rigor of our notebook and source collection (see section 3), there are notebooks where we were unable to locate or analyze the source. Thus, decomposition inheritance may not be fully verifiable. We encourage future studies to test our hypotheses.

Supplemental Materials

All supplemental materials are available on OSF at https://osf.io/sudb8/?view_only=

cc72bdc685804e478852a96297328eb8.

Acknowledgments

This research was funded in part by a Sloan Research Fellowship, a Mary Gates Research Scholarship, a Google gift, and NSF awards IIS-2402718, IIS-2141506, and CSGrad4US-2313998

References

[1] H. Bako, A. Varma, A. Faboro, M. Haider, F. Nerrise, B. Kenah, and L. Battle. Streamlining visualization authoring in d3 through user-driven templates. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pp. 16–20. IEEE Computer Society, Los Alamitos, CA, USA, oct 2022. doi: 10.1109/VIS54862.2022.00012 1, 2, 3

[2] H. K. Bako, X. Liu, L. Battle, and Z. Liu. Understanding how designers find and use data visualization examples. *IEEE TVCG*, 29(1):1048–1058, 2023. doi: 10.1109/TVCG.2022.3209490 1, 2

[3] H. K. Bako, X. Liu, G. Ko, H. Song, L. Battle, and Z. Liu. Unveiling how examples shape visualization design outcomes. *IEEE TVCG*, 31(1):1137–1147, Jan 2025. doi: 10.1109/TVCG.2024.3456407 2

[4] H. K. Bako, A. Varma, A. Faboro, M. Haider, F. Nerrise, B. Kenah, J. P. Dickerson, and L. Battle. User-driven support for visualization prototyping in d3. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*, IUI '23, p. 958–972. ACM, New York, NY, USA, 2023. doi: 10.1145/3581641.3584041 1, 2, 4

[5] L. Battle, P. Duan, Z. Miranda, D. Mukusheva, R. Chang, and M. Stonebraker. *Beagle: Automated Extraction and Interpretation of Visualizations from the Web*, p. 1–8. CHI '18. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3174168 2, 4

[6] L. Battle, D. Feng, and K. Webber. Exploring d3 implementation challenges on stack overflow. In *2022 IEEE Visualization and Visual Analytics (VIS)*, pp. 1–5, 2022. doi: 10.1109/VIS54862.2022.00009 1, 2, 3

[7] K. Batziakoudi, F. Cabric, S. Rey, and J.-D. Fekete. Lost in magnitudes: Exploring visualization designs for large value ranges. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25. Association for Computing Machinery, New York, NY, USA, 2025. doi: 10.1145/3706598.3713487 2

[8] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE TVCG*, 17(12):2301–2309, Dec. 2011. doi: 10.1109/TVCG.2011.185 1

[9] C. Charitsis, C. Piech, and J. C. Mitchell. Detecting the reasons for program decomposition in cs1 and evaluating their impact. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, p. 1014–1020. ACM, New York, NY, USA, 2023. doi: 10.1145/3545945.3569763 1, 2, 3

[10] C. Chen, B. Lee, Y. Wang, Y. Chang, and Z. Liu. Mystique: Deconstructing svg charts for layout reuse. *IEEE TVCG*, 30(1):447–457, 2024. doi: 10.1109/TVCG.2023.3327354 2

[11] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960. 2

[12] J. Harper and M. Agrawala. Converting basic d3 charts into reusable style templates. *IEEE TVCG*, 24(03):1274–1286, mar 2018. doi: 10.1109/TVCG.2017.2659744 1, 2, 4

[13] M. Hedayati and M. Kay. "choose-your-own" d3 labs for learning to adapt online code. In *2023 IEEE VIS Workshop on Visualization Education, Literacy, and Activities (EduVis)*, pp. 49–54, Oct 2023. doi: 10.1109/EduVis60792.2023.00014 2

[14] E. Hoque and M. Agrawala. Searching the visual style and structure of d3 visualizations. *IEEE TVCG*, 26(1):1236–1245, 2020. doi: 10.1109/TVCG.2019.2934431 1

[15] M. Kay. ggdist: Visualizations of distributions and uncertainty in the grammar of graphics. *IEEE TVCG*, 30(1):414–424, Jan 2024. doi: 10.1109/TVCG.2023.3327195 2

[16] H. Kim, Y.-S. Kim, and J. Hullman. Erie: A declarative grammar for data sonification. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24. Association for Computing Machinery, New York, NY, USA, 2024. doi: 10.1145/3613904.3642442 2

[17] N. Kruchten, A. M. McNutt, and M. J. McGuffin. Metrics-Based Evaluation and Comparison of Visualization Notations . *IEEE TVCG*, 30(01):425–435, Jan. 2024. doi: 10.1109/TVCG.2023.3326907 2

[18] A. M. McNutt and R. Chugh. Integrated visualization editing via parameterized declarative templates. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21. ACM, New York, NY, USA, 2021. doi: 10.1145/3411764.3445356 1, 2

[19] Observable, Inc. Observable. https://observablehq.com, 2024. 1, 2

[20] P. Parsons, P. Shukla, and C. Park. Fixation and creativity in data visualization design: Experiences and perspectives of practitioners. In *2021 IEEE Visualization Conference (VIS)*, pp. 76–80, 2021. doi: 10.1109/VIS49827.2021.9623297 2

[21] X. Pu and M. Kay. How data analysts use a visualization grammar in practice. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pp. 1–22. ACM, New York, NY, USA, 2023. doi: 10.1145/3544548.3580837 1, 2, 4

[22] D. Raghunandan, Z. Cui, K. Krishnan, S. Tirfe, S. Shi, T. D. Shrestha, L. Battle, and N. Elmqvist. Lodestar: Supporting rapid prototyping of data science workflows through data-driven analysis recommendations. *Information Visualization*, 23(1):21–39, 2024. doi: 10.1177/14738716231190429 4

[23] D. Raghunandan, A. Roy, S. Shi, N. Elmqvist, and L. Battle. Code code evolution: Understanding how people change data science notebooks over time. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pp. 1–12. ACM, New York, NY, USA, 2023. doi: 10.1145/3544548.3580997 2

[24] K. M. Rich, T. A. Binkowski, C. Strickland, and D. Franklin. Decomposition: A k-8 computational thinking learning trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, p. 124–132. ACM, New York, NY, USA, 2018. doi: 10.1145/3230977.3230979 1, 2

[25] A. Rule, A. Tabard, and J. D. Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, p. 1–12. ACM, New York, NY, USA, 2018. doi: 10.1145/3173574.3173606 2

[26] L. Ryan, D. Silver, R. S. Laramee, and D. Ebert. Teaching data visualization as a skill. *IEEE Computer Graphics and Applications*, 39(2):95–103, 2019. doi: 10.1109/MCG.2018.2889526 1

[27] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. Critical reflections on visualization authoring systems. *IEEE TVCG*, 26(1):461–471, 2020. doi: 10.1109/TVCG.2019.2934281 1

[28] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE TVCG*, 23(1):341–350, Jan 2017. doi: 10.1109/TVCG.2016.2599030 1, 4

[29] X. Tang, Y. Yin, Q. Lin, R. Hadad, and X. Zhai. Assessing computational thinking: A systematic review of empirical studies. *Computers & Education*, 148:103798, 2020. doi: 10.1016/j.compedu.2019.103798 1, 2

[30] S. Titov, Y. Golubev, and T. Bryksin. Resplit: Improving the structure of jupyter notebooks by re-splitting their cells. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 492–496, 2022. doi: 10.1109/SANER53432.2022.00066 2

[31] H. Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010. doi: 10.1198/jcgs.2009.07098 2, 4

[32] L. Wilkinson. *The grammar of graphics*. Springer New York, 2005. doi: 10.1007/0-387-28695-0 4

[33] L. Wilkinson, A. Anand, and R. Grossman. Graph-theoretic scagnostics. In *Information Visualization, IEEE Symposium on*, pp. 157–164. IEEE Computer Society, Los Alamitos, CA, USA, Oct 2005. doi: 10.1109/INFVIS.2005.1532142 2

[34] A. Yadav, C. Ocak, and A. Oliver. Computational thinking and metacognition. *TechTrends*, 66(3):405–411, 2022. doi: 10.1007/s11528-022-00695-z 1, 2

[35] J. Yang, A. M. McNutt, and L. Battle. Considering visualization example galleries. In *Proc. IEEE VL/HCC 2024*, pp. 329–343, 2024. doi: 10.1109/VL/HCC60511.2024.00043 2, 4