GlyphWeaver: Unlocking Glyph Design Creativity with Uniform GDSL and AI

Can Liu, Shiwei Chen, Zhibang Jiang, Yong Wang

***Abstract*—Glyph-based visualizations depict data using collections of glyphs whose visual channels represent data attributes. Authoring such visualizations can require substantial programming effort while making it difficult to retain user control during iterative design. We present *GlyphWeaver*, an interactive prototype for authoring glyph-based visualizations while retaining user control over their structures and data mappings. *GlyphWeaver* comprises three key components: a glyph domain-specific language (GDSL), a GDSL operation management mechanism, and a multimodal interaction interface. The GDSL is a hierarchical container model that uses *basic*, *repeater*, and *compositor* containers to represent the structures of glyph-based visualizations. The operation management mechanism restricts modifications of the GDSL to five atomic operations, enabling inspectable editing of GDSL instances. The multimodal interaction interface supports direct manipulation, natural-language commands, and parameter adjustments. A multimodal large language model translates the natural-language commands into GDSL operations. We evaluate *GlyphWeaver* through two case studies, a quantitative translation benchmark, and a user study with 13 participants. The results demonstrate the effectiveness and good usability of *GlyphWeaver* in authoring glyph-based visualizations through the benchmark instructions.**

***Index Terms*—Glyph design, domain-specific language, large language model, visualization generation**

I. Introduction

Glyph-based visualization depicts a dataset using a collection of visual objects, referred to as glyphs, whose visual attributes encode one or more attributes of data records [3], [49]. Examples appear in award-winning designs from *Information is Beautiful* [18] (Figure 1), which were originally crafted manually using graphics software and later reproduced using our approach. For instance, the OECD example in Figure 1a maps eleven quality-of-life attributes to eleven petals arranged radially around a center [42]. Its structure combines repeated petals with other visual elements in a nested composition. Such structures fall outside the bounded chart design space of high-level grammars such as Vega-Lite [38], which targets concise specifications of common statistical graphics.

Despite their value, glyph-based visualizations can be challenging and time-consuming to implement. The challenges arise from two primary factors: **the structural complexity and diversity of their glyphs** (**C1**) and **the need for human control during the design process** (**C2**). The examples in Figure 1 use different structures and data mappings; their construction may involve parameterized repetition, nested arrangements, and compositions of heterogeneous elements. Designers often construct such glyphs manually in graphics software such as Adobe Illustrator, Figma, and Sketch. During this process, designers must specify data-to-visual mappings, organize repeated elements, and refine visual forms and styles.

Prior research [55], [56] has attempted to lower the authoring barrier through automated approaches. MetaGlyph [55] maps tabular data to elements decomposed from a metaphoric image, but does not support nested or hierarchical structures. GlyphCreator [56] focuses on circular glyph designs, limiting the range ofstructures it can represent. Thus, these approaches reduce manual effort within their respective design spaces but do not cover the heterogeneous and nested structures considered in this work. Low-level programming libraries such as D3.js [4] provide the flexibility to implement such structures, but require designers to translate visual concepts into code. Therefore, a gap still exists between flexible low-level implementation and accessible authoring support for glyph-based visualizations.

To address this gap, we propose *GlyphWeaver*, an interactive prototype for authoring glyph-based visualizations with reduced manual implementation. Given user-created graphical elements and instructions expressed through natural language or direct manipulation, *GlyphWeaver* constructs or updates an editable Glyph Domain-Specific Language (GDSL) specification and renders the corresponding glyph-based visualization. *GlyphWeaver* integrates three core components (Fig. 2): (1) a **GDSL** with composable containers, (2) a **GDSL operation management mechanism** based on atomic operations and multimodal large language models (MLLMs), and (3) a **multimodal interaction interface** supporting both natural language and direct manipulation. At the core of *GlyphWeaver* is the GDSL, a structured representation **(C1)** that uses three container types: basic, repeater, and compositor. A basic container holds a visual mark (e.g., a bar in a bar chart), a repeater container holds multiple repeated instances of contents based on user-defined parameters, and a compositor container arranges multiple different parts. For example, the OECD glyph in Figure 1a can be represented by a basic container for one petal, a repeater container that creates and radially arranges eleven petals, and a compositor container that combines the repeated petals with the center. We derived these container types by analyzing 50 glyph-based visualizations

C. Liu is with the School of Information and the Institute for the Community for the Chinese Nation at the Central University of Finance and Economics, and with Nanyang Technological University. This work was conducted primarily while he was a research fellow supervised by Yong Wang at Nanyang Technological University. Email: can.liu@cufe.edu.cn.

S. Chen, and Y. Wang are with Nanyang Technological University. Email: can.liu@ntu.edu.sg, shiwei004@e.ntu.edu.sg, yong-wang@ntu.edu.sg.

Y. Wang is the corresponding author.

Z. Jiang is with Providence Health and Services. Email: zhibang.jiang@newschool.edu.

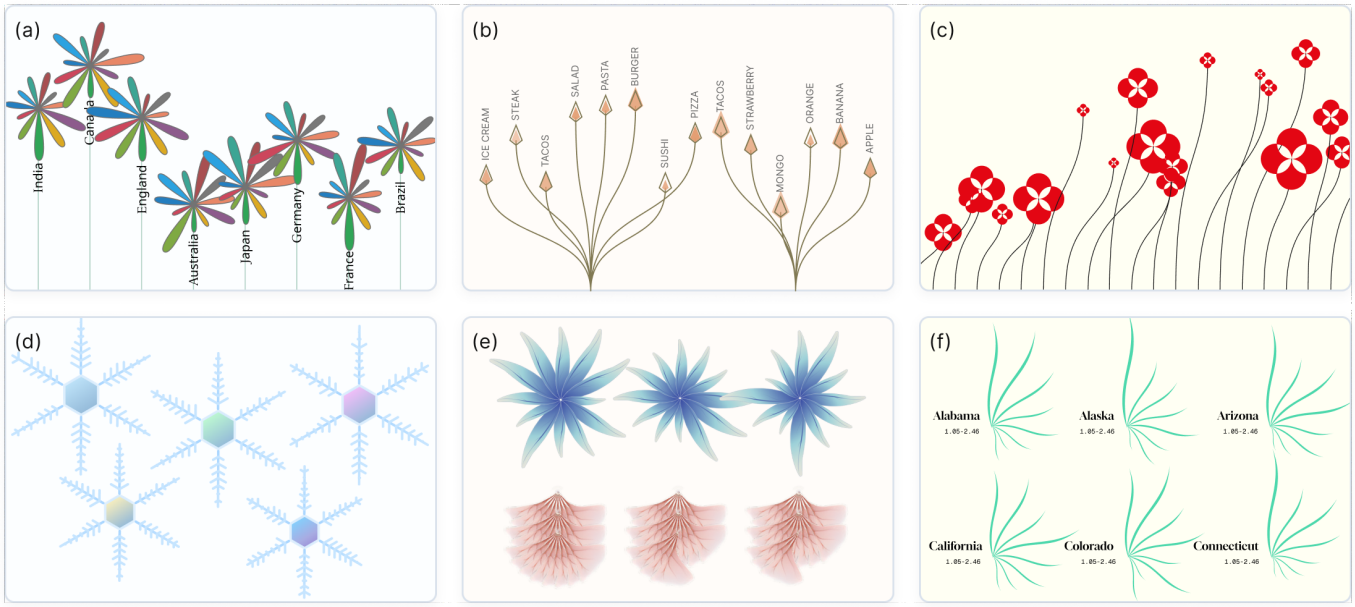


Fig. 1: Glyph-based visualization examples recreated with *GlyphWeaver*. All the original glyph-based visualizations are award-winning designs from '*Information is beautiful*'[18] (a [42], b [9], c [7], d [51], e [12], f [2]).

from the *Information is Beautiful* [18] winning list and the D3 Gallery [1]; all analyzed structures could be represented by combinations of the three container types. Then, the GDSL operation management mechanism defines five atomic GDSL operations: creating basic containers, creating repeater containers, creating compositor containers, modifying parameters, and encoding data. By nesting these atomic operations, different structures can be constructed. The GDSL operation management mechanism leverages MLLMs to interpret user intentions and translate them into corresponding GDSL operations **(C2)**. The multimodal interaction interface accepts natural-language input through the dialogue panel (Figure 6(c)) and direct manipulation on the Figma canvas (Figure 6(a)), providing two ways for designers to specify and revise operations **(C2)**.

These three components work together in a coupled workflow: users interact with the prototype through the multimodal interaction interface, expressing their design intentions via natural language or direct manipulation; the operation management mechanism interprets these inputs and translates them into predefined GDSL operations; these operations are then applied to the GDSL to construct or modify the resulting structures. This workflow keeps MLLM output within the GDSL operation space and exposes inferred parameters for user adjustment. Compared with low-level implementation, *GlyphWeaver* is designed to reduce the manual effort required to create glyph-based visualizations while retaining human control. We conducted two case studies, a translation benchmark, and a user study to evaluate *GlyphWeaver*'s support for interactively authoring glyph-based visualizations. Our contributions are as follows:

- We introduce a hierarchical GDSL for representing the structures of glyph-based visualizations through basic, repeater, and compositor containers, together with five atomic operations for modifying GDSL instances.
- We propose *GlyphWeaver*, an interactive prototype that incorporates GDSL with a GDSL operation management mechanism to translate natural-language and direct-manipulation input into inspectable operations. We further extensively evaluate it through case studies, a translation benchmark, and a user study with 13 participants.

We have released glyph-based visualizations created with *GlyphWeaver* here: https://glyphweaver.github.io/.

II. Related Work

Our work is related to glyph-based visualization, domain-specific language, and AI-powered visualization generation.

A. Glyph-Based Visualization

Glyph-based visualizations are compact visual representations that map multiple data variables onto visual channels such as color, shape, orientation, and size [49]. They have long been employed in multivariate visualization [40], [35], vector field visualization [50], and domain-specific applications such as biological systems [28] and human-automation collaboration [32]. Foundational surveys synthesize the evolution of glyph-based techniques, design principles, and evaluation methodologies [3], [13]. Design principles and conceptual models have been proposed to guide effective glyph creation, including perceptual uniformity, redundant encoding, importance ordering, and spatial arrangement [49], [3]. Prior glyph literature further distinguishes one-to-one mappings, in which individual data dimensions are assigned to distinct visual channels, from many-to-one mappings, in which multiple dimensions jointly affect the same visual channel [49], [3]. Prior studies have examined how glyph form and semantic relevance affect interpretation. Flury and Riedwyl [11] introduced asymmetric facial glyphs for multivariate data. Siirtola [41] found that data-related glyphs were preferred and produced

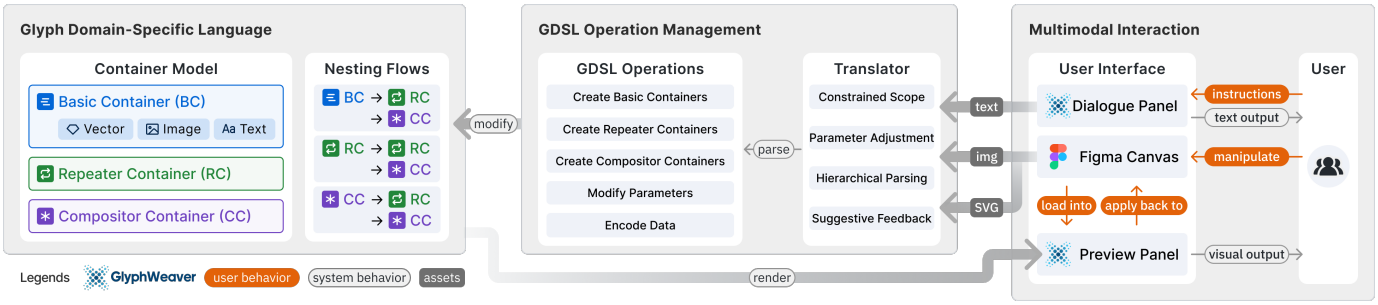


Fig. 2: *GlyphWeaver* consists of three components: The glyph domain-specific language is the theoretical foundation; the GDSL operation management mechanism serves as an intermediary layer between GDSL and the multimodal interaction interface; the multimodal interaction interface allows users to interact with the user interface using natural language and direct manipulation.

higher accuracy than unrelated glyphs in the tasks examined. Li et al. [22] further found that metaphoric associations could facilitate interpretation when they were compatible with the represented concept, but could interfere with interpretation when they were incompatible. Together, these findings show that glyph form and semantic relevance can influence interpretation and task performance, although their effects depend on the particular design and task. Despite their capacity to encode multiple variables in compact forms, glyph-based visualizations with heterogeneous structures remain challenging to design. Traditional approaches often require manual manipulation or low-level programming [36], which imposes a steep technical barrier compared to charting tools like Vega-lite [38] or ECharts [19]. Recent approaches have attempted to automate glyph creation within more specific design spaces. GlyphCreator generates circular glyphs from examples [56], whereas MetaGlyph generates metaphoric glyphs by decomposing a user-selected metaphoric image and mapping tabular data to its visual elements [55]. These method-specific assumptions limit their support for heterogeneous and nested structures.

B. Domain-Specific Language in Visualization

Domain-Specific Languages (DSLs) [29] provide structured abstractions for visualization design. Existing systems like Lyra [37] and Data Illustrator [25] combine direct manipulation with programmatic control, demonstrating how DSLs integrate usability with semantic precision. Both general-purpose and specialized DSLs have been proposed. General-purpose DSLs such as Vega-Lite [38] provide concise declarative specifications for a bounded chart design space, whereas specialized DSLs target structures such as trees [20], networks [39], [15], or multidimensional data [43]. Visualization DSLs span different levels of abstraction: low-level representations expose flexible construction primitives, whereas higher-level declarative grammars provide more concise specifications for a bounded design space [29], [4], [38]. For instance, D3.js [4] exposes low-level primitives for custom implementations; Vega-Lite [38] sacrifices some flexibility for ease of use via high-level declarative specifications. Recent research on visualization DSLs has tried to address this tension. For instance, Bluefish [34] promotes cognitive alignment with declarative abstractions. Mascot [24] categorizes operations into generative and modificative primitives to preserve interactivity while retaining access to low-level operations. DataGarden [33] further attempts to bridge informal sketches with structured templates. DataSway [54] generates and coordinates animation clips for SVG-based metaphoric visualizations.

Inspired by the above research, we propose GDSL, which enables the creation of glyph-based visualizations through composition. GDSL is paired with a set of operations and uses an MLLM to translate user inputs into GDSL operations, embodying the principle of agency plus automation [16] and augmenting human control with AI assistance when creating glyph-based visualizations.

C. AI-powered Visualization Generation

AI-powered visualization generation spans multiple output representations and levels of user control [48], [47], ranging from chart recommendation and specification generation to the direct generation of visual artifacts and code.

Natural-language interfaces such as Articulate, DataTone, NL4DV, FlowSense, and Iris support conversational data analysis or translate user queries into visualizations [44], [14], [31], [57], [10]. Learning-based approaches similarly generate specifications or recommend chart designs [8], [23], [27], [26], [17]. Complementing these learned approaches, Draco formalizes visualization design knowledge as constraints [30]. KG4Vis, AdaVis, and LLM4Vis further improve recommendation transparency through structured knowledge or natural-language explanations [21], [58], [46]. These methods primarily produce chart-level specifications or recommendations rather than editable representations of such structures.

Other systems directly generate infographics, semantically enriched charts, visualization code, or adapted SVG examples [6], [52], [45], [5], [53]. Although these approaches broaden the range of generated outputs, artifact or code generation alone does not expose the domain-specific operations needed to construct and revise nested glyph components.

GlyphWeaver therefore uses an MLLM as a translator rather than as an unconstrained end-to-end generator. Its output is restricted to atomic operations over an explicit GDSL, while users can combine natural-language input with direct manipulation and inspect the resulting intermediate states. This design connects AI assistance to a hierarchical representation specialized for the interactive and controllable authoring of glyph-based visualizations.

III. Overview

GlyphWeaver is an interactive glyph-authoring prototype with three components (Fig. 2): the glyph domain-specific language (GDSL) represents heterogeneous and nested structures in glyph-based visualizations and their data mappings to address C1; the operation management mechanism applies five supported GDSL operations to GDSL instances; and the multimodal interaction interface captures user input through natural language and direct manipulation while keeping inferred parameters available for user inspection and adjustment, addressing C2. This overview provides a roadmap of the architecture; the following three sections describe the GDSL, operation management mechanism, and multimodal interaction interface in detail.

- **Glyph Domain-Specific Language.** The GDSL (section IV) is a container-based and structure-driven representation for these structures. At its core, GDSL treats every visual element as part of a container hierarchy, where containers can hold primitives, repeated elements, or composite assemblies. This approach allows users to build complex structures by nesting and composing containers.
- **GDSL Operation Management Mechanism.** Directly authoring a GDSL instance requires knowledge of its syntax and container hierarchy. To address this, we introduce an operation management mechanism (section V) that restricts modifications of GDSL instances to five atomic GDSL operations. While each operation is simple and well-defined, they can be composed to construct complex structures. Such a mechanism translates user intentions into these operations, reducing the learning curve.
- **Multimodal Interaction Interface.** *GlyphWeaver* provides a multimodal interaction interface (section VI) that supports direct manipulation via the Figma interface, natural language (NL) input through a dialogue panel, and parameter tuning through interactive controls. Leveraging an MLLM, the framework translates these inputs into GDSL operations.

IV. Glyph Domain-Specific Language

This section introduces the GDSL, a hierarchical container-based model for representing the structures of glyph-based visualizations, as illustrated in Figure 3. It is comprised of the containers (subsection IV-A) and the container relationships (subsection IV-B). We also use a GDSL instance to explain its usage (subsection IV-C).

A. Containers

We analyzed 39 award-winning visualizations from *Information is Beautiful* and 11 D3 Gallery examples, totaling 50 cases. To derive the structural abstractions used in GDSL, we manually decomposed each glyph-based visualization into its construction process and recorded the recurring structural operations involved in assembling visual elements. Through iterative comparison across cases, we observed that most construction processes could be consistently interpreted as combinations of three recurring patterns: (1) constructing individual visual primitives, (2) duplicating elements with parameterized variation, and (3) spatially composing heterogeneous structures. These patterns eventually informed the design of the three container abstractions in GDSL: basic containers, repeater containers, and compositor containers.

Not every glyph uses all three containers, but their combinations represented all 50 analyzed structures.

All analyzed visualizations employed basic and repeater containers, while 64% additionally involved compositor structures. Across the 233 recorded container operations, we identified 89 basic operations, 105 repeater operations, and 39 compositor operations. This decomposition is also consistent with common visualization construction paradigms such as D3.js, where visual primitives are first instantiated and encoded through data binding, repeated through enter mechanisms, and further organized through hierarchical grouping and composition. While the analyzed examples are not exhaustive, they cover a broad range of structures found in infographic and visualization practice, including radial layouts, repeated structures, and nested assemblies.

- **Basic containers (BC)** encapsulate basic elements (polygons, lines, images, and text). A **BC** maintains an independent coordinate system. It enables precise control over visual attributes such as position, scale, rotation, and color, making it a fundamental building block in glyph design. Basic containers typically encode data in the form of a dictionary, where each data value is directly mapped to the visual attributes of an element. For example, a number value can be mapped to the width and height of a rectangle.
- **Repeater containers (RC)** replicate a unit multiple times using either the Cartesian or polar coordinate systems. Cartesian coordinates are for grid-based and linear arrangements, and polar coordinates are for circular arrangements. A **RC** contains a child container and defines both the repetition count and arrangement pattern, replicating the child container into multiple instances according to this specified arrangement. An **RC** encodes a list of data values, where each item is mapped to an instance of its child container. For example, if a petal is rotated and duplicated 12 times to form a flower, and different petals are expected to have varying sizes, each entry in the data array would contain a unique scale value.
- **Compositor containers (CC)** integrate multiple different child containers. **CCs** also support both Cartesian and polar coordinate systems. Unlike **RCs** that manage duplications of a child container, **CCs** coordinate the positioning of different child containers. A **CC** typically use dictionary-structured data, where different keys correspond to different child containers.

Spatial layout of containers. The positions of containers are influenced by their inner coordinate system, the transformation relative to the parent, and the spatial relationship between sibling containers.

- **Inner coordinate system.** Each container (**BC**, **RC**, and **CC**) possesses an independent coordinate system for positioning its internal elements with an origin point. These coordinate systems can be either Cartesian  or polar .
- **Parent-relative transformation.** This property defines the transformation of a container relative to its parent container,

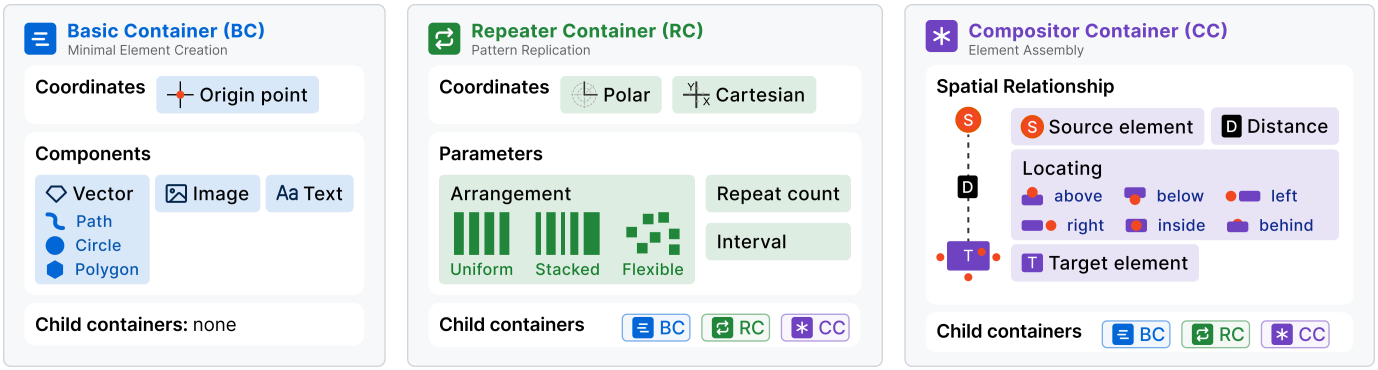


Fig. 3: GDSL has three containers: Basic containers encapsulate individual visual elements with an independent coordinate. Repeater containers replicate a child container using Cartesian or polar coordinates, supporting three arrangements (uniform, stacked, and flexible). Compositor containers coordinate spatial relationships between child containers.

including three types: translation, rotation, and scaling. The transformation is assigned by the parent container or independently mapped using the container's own encoded data. In RCs (Figure 3), for instance, the parent can assign different transformers to its child repeat instances, as shown in Figure 4 (b), where the red flower assigns varying rotation angles to four circles. There are three repetition arrangements: uniform, stacked, and flexible, as shown in Figure 3: Repetitive instances in uniform arrangement are placed at uniform intervals, ensuring consistent spacing. In a Cartesian system, intervals are defined by x and y values; in a polar system, by radius and angle. Repetitive instances in stacked repetition are recursively positioned, where each element's position is determined by previous instances. Repetitive instances in flexible repetition are arranged with variable spacing or transformations, allowing for more organic and dynamic layouts. A container in a flexible arrangement can determine its own transformer based on data, as illustrated in Figure 4 (d), where green stems scale in the y-direction according to their individual data, resulting in flowers of diverse heights.

- **Sibling container relationship.** Within CCs, spatial relationships may exist between sibling containers. Spatial relationships in *GlyphWeaver* are defined as distances between anchor points on source S and target T (Figure 3 CC). Each spatial relationship can define anchor points as reference points, which may correspond to geometric features of the container (such as center, edge, or corner points). The distance D (as a 2D vector) between the anchor points of the source and target containers is also given. As shown in Figure 4 (c), a spatial relationship exists between the red flower and the green stem, with the red flower anchored to the top point of the green stem with a distance of 0. When calculating positions, these spatial relationships are translated into constraints (equations based on anchor points and distances), which are solved to determine the final (x, y) coordinates of all containers.

B. Hierarchical Container Relationships

GDSL employs a hierarchical container system that enables nested structure while maintaining manageable complexity.

Cognitive alignment. A hierarchical structure inherently aligns with users' cognitive processes. Users typically begin with basic visual marks and progressively develop complexity by applying repetition and composition. Complex visual structures can be derived through the orchestration of three container types. This hierarchical architectural framework not only facilitates extensibility but also promotes the development of reusable components that can be leveraged in subsequent design iterations.

Nested construction. The construction process in GDSL leverages this hierarchical approach through a nesting mechanism. BCs serve as the foundation, which can then be duplicated to form RCs, while multiple different containers can be combined into a compositor container. This process can continue recursively: a compositor container itself can be a child of a repeater container.

Complexity management. To manage the potential complexity that arises from such hierarchical nesting, GDSL implements several control mechanisms. GDSL encapsulates parameters and transformations at each level of the container hierarchy. Each container operates within its independent coordinate system, preventing spatial transformations from interfering across different levels. Additionally, GDSL maintains independent data management at each hierarchical level, preserving a clear data flow throughout the structure. Each level also maintains a separate management system that governs parameter mapping rules specific to its scope. This hierarchical and modular approach allows GDSL to maintain clarity and control while enabling users to create complex visualizations through element composition and repetition.

C. GDSL Instance Example

Figure 4 showcases four levels of GDSL instances. The BC ● represents the fundamental building unit, as illustrated in Figure 4 (a). It defines a red circle in a Cartesian coordinate. Its origin point is on the left of the circle. The BC (with id "red circle") from Figure 4 (a) is repeated four times, with each instance rotated by 90 degrees, resulting in a red flower, the RC ✿ in Figure 4 (b). The CC in Figure 4 (c) is employed to integrate multiple distinct containers and may define their spatial relationships. The CC with ID

TABLE I: The five atomic GDSL operations: three create containers, one modifies parameters, and one maps data to visual attributes.

Operation Type	Parameters	Example	Explanation
BC Creation	id, type, params, coordinate	`CreateBasic("rect1", "rect", {width, height, fill}, {origin, transform})`	Creates a container for a basic graphic element (polygon, line, rect, circle, text) with specified visual attributes.
RC Creation	id, target_id, coordinate, repeat_count, repeat_type, repeat_interval	`CreateRepeater("flower", "petal", "polar", 12, "uniform")`	Create a container by repeating a child container in polar or Cartesian coordinates.
CC Creation	id, components, spatial_relations	`CreateCompositor("chart", ["title", "bars"], [{source, target, type, distance}])`	Combines multiple child containers into a composite structure with spatial relationships.
Modify Parameters	target_id, params	`ModifyParams("rect1", {fill: "red", width: 150})`	Modifies static parameters of a container without data mapping, including visual attributes or structural properties.
Encode Data	target_id, attribute, data_list — data_expression	`EncodeData("bars", "height", [10, 45, 30])`	Maps data values to parameters or visual attributes, enabling different visual representations based on data values.

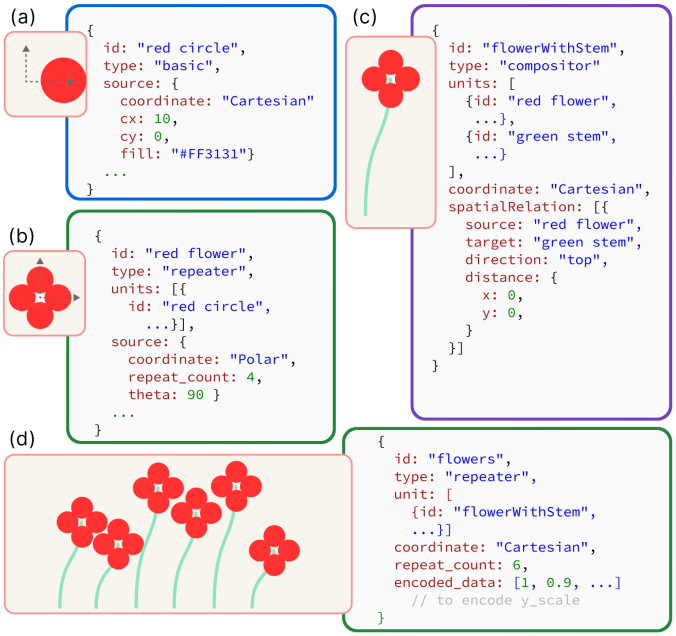


Fig. 4: A GDSL instance with basic container (a), repeater containers (b and d), and compositor container (c).

"flowerWithStem" integrates two elements, the red flower and the green stem. The CC establishes their spatial relationship: the red flower is attached to the top anchor of the green stem with a displacement of $\{x:0, y:0\}$. The RC in Figure 4 (d) demonstrates a more complex application where the repeating unit itself is a compositor container (with id "flowerWithStem"). This unit is repeated six times in a Cartesian coordinate system, with the `encoded_data` array providing different scalar values for the y-directional scaling of each "green stem" instance, thereby creating a floral arrangement with varying heights.

V. GDSL Operation Management Mechanism

To enable controllable editing of GDSL instances, we propose a GDSL operation management mechanism. It employs concise and semantically clear operations to ensure structural validity and enhance the editing accuracy and consistency. This mechanism consists of: (1) a set of well-defined atomic operations, which serve as the building blocks for GDSL editing; and (2) an MLLM-powered operation translator, which translates user intentions into valid atomic operations.

A. Atomic Operations

GDSL constructs visual structures through five atomic GDSL operations (Table I), grouped into container creation and modification. These composable semantic units build a structural skeleton and modify its parameters or bind data.

- **BC creation.** BCs encapsulate elements, including polygons, lines, rectangles, circles, and text. The operation `CreateBasic` defines the visual attributes for the basic element (e.g., position, size, color), with a coordinate system specification that includes an origin point and transformations (rotation, translation, and scaling).
- **RC creation.** The `CreateRepeater` operation replicates existing content to generate a new RC. `CreateRepeater` requires specifying the repeating unit's ID (an existing container), the repetition count, the coordinate type, and the repetition category (uniform, stacked, or flexible). Figure 4 illustrates a sequence of operations on GDSL instances. The BC containing one red circle in panel (a) serves as the repeating unit. Applying `CreateRepeater` with this target, a repetition count of four, and a uniform polar arrangement produces the four-circle RC in panel (b). Panel (c) then uses `CreateCompositor` to attach the resulting flower to a stem, and panel (d) repeats this composite unit six times and binds data to the stems' vertical scales.
- **CC creation.** `CreateCompositor` operation combines multiple existing units into a CC. Parameters include the information to be combined and potential spatial relationships between components. For each spatial relationship, the source and target IDs, the type of spatial dependency (e.g., top, bottom, left, right), and the spatial distance must be specified. For example, positioning container A above container B with a distance of 5 units would be expressed as: {`source ID = "container`

A", target ID = "container B", spatial type = "top", distance = {$x:0, y:5$}}.

- **Modifying parameters.** The operation ModifyParams alters the parameters of a target container. This operation requires providing the target container, visual attribute, and new value. For example, changing the repetition count of a RC (with id "repeat"), the parameters are: {target_ID = "repeat", attribute = "repeat count", value = 5}.
- **Encoding data.** The operation EncodeData enables data attributes to be mapped to visual attributes. This operation adjusts the parameters of target elements in batches. Unlike the ModifyParams operation, EncodeData allows different instances to reflect distinct data values. For example, as shown in Figure 5, binding the scale of petals to a data list can transform a flower with uniformly sized petals into one with petals of varying sizes; similarly, binding the fill color of rectangles to a data list can result in rectangles displaying different colors.



Fig. 5: The data encoding operation enables encoding various data items into visual attributes of different instances.

B. Data Pipeline

Data can come from user-provided values or a specified simulation function. GDSL organizes the resulting values according to the container hierarchy. Each repeat level introduces a corresponding array dimension, so the data dimensionality must match the target repeated structure. The data-binding component checks this correspondence; when the dimensions do not match, *GlyphWeaver* displays a mismatch warning and asks the user to revise the values or simulation function to the expected array shape. For example, if one petal is replicated into 12 petals, the resulting data structure is an array of size 1×12. If the flower is then further replicated into three flowers, the data structure becomes an array of size 3×12. To support such arrays, we allow users to specify data in two ways:

- **Math Expressions.** Users may provide a simulation function that generates data based on the variables index and random, allowing the construction of specific trends or random patterns. By composing these variables, the MLLM can generate a wide range of mathematical expressions, such as (index * 5 + random()) for a sequence with a linear trend and random variation. Synthetic data can also be specified through natural language descriptions, such as *"a dataset with an increasing trend"* or *"random values between 10 and 20"*. The translator then converts these descriptions into mathematical expressions. *GlyphWeaver* supports two variables, index, which represents the sequence number of each data point, and random, which generates a random value in $[0,1]$, together with basic arithmetic operations, such as $+$, $-$, $\times$, and $/$.
- **Direct Data Provision.** Beyond expression-based generation, *GlyphWeaver* supports binding user-provided or updated lists of values. For example, for text elements in a RC, a user can bind a list of U.S. state names to the text attribute. The user may request that the translator generate different state names or explicitly provide a list, such as California, Arizona, and Texas. Users may provide such data either inline or through file upload. For inline input, a user may paste an array into the chat box and specify its target mapping, such as using a one-dimensional array to control petal sizes. *GlyphWeaver* parses the input, identifies the corresponding target elements and attributes, such as scale, and then binds the data accordingly. Alternatively, users may upload data files in CSV or JSON format. The corresponding data processing workflow remains consistent with that of chat-based input.

C. MLLM-Powered Translator

To ensure diverse inputs are translated into GDSL operations in a controllable manner, *GlyphWeaver* uses a four-stage translation strategy. First, *GlyphWeaver* constrains the parsing scope. Within the controlled space, users can interactively adjust parameters to align results with their intent. *GlyphWeaver* then supports hierarchical parsing for complex designs, decomposing multi-level structures step by step. Finally, *GlyphWeaver* provides suggestive feedback for irrelevant inputs, ensuring even off-topic commands are redirected productively. In our implementation, natural-language translation is powered by GPT-4o or GPT-5.1, using a structured prompt template to enhance output consistency. However, we still cannot guarantee that the parsing will always perfectly match the user's intent. To address this, we provide an undo button as well as ambiguity widgets that allow users to adjust the parsing results (as shown in Figure 6).

- **Constrained Parsing Scope.** The translator first enforces a restricted output space: the MLLM is explicitly instructed to generate only five supported atomic operations (defined in subsection V-A). This scope prevents unsupported or overly broad outputs and guarantees predictability. When user instructions are underspecified but still valid, *GlyphWeaver* supplements them with default values. For instance, the input *"rotate and replicate the object six times"* is interpreted as six rotations of 60° each, forming a circular pattern.
- **Interactive Parameter Adjustment.** Even when an instruction maps to one of the five supported atomic operations, its inferred parameters may not align with user intent. The dialogue panel therefore displays parameters such as angle, scale, and repetition count in ambiguity widgets for direct adjustment (see subsection VI-B).
- **Hierarchical Parsing for Complex Designs.** For inputs describing multi-level or composite structures, *GlyphWeaver*

applies hierarchical parsing. The translator first identifies higher-level structural patterns and then recursively analyzes their components. Each step targets a container structure.

- **Suggestive Feedback for Irrelevant Inputs.** When the input cannot be mapped to one of the five supported GDSL operations, the dialogue panel identifies it as unsupported and prompts the user to reformulate it as a supported glyph-authoring operation. For example, for the user input *"play a song"* that is not relevant to glyph design or creation, it can return suggestions like *"try 'replicate the shape five times' instead."*

For an NL instruction within the supported operation space, the translator returns an atomic operation and its inferred parameters. When a parameter is missing or ambiguous, the dialogue panel exposes the inferred value in an ambiguity widget (subsection VI-B). If an instruction cannot be mapped to a valid GDSL operation, the dialogue panel reports that limitation and provides an example of a supported instruction.

VI. Multimodal Interaction Interface

We implemented *GlyphWeaver* as a Figma[1] plugin, given Figma's popularity in graphic design. *GlyphWeaver* has three views: the direct manipulation interface (Figma canvas), dialogue panel, and preview panel, as shown in Figure 6. Users can interact with *GlyphWeaver* in a multimodal manner, using direct manipulation or natural language. Users can directly manipulate graphics in the Figma canvas (Figure 6a). In the dialogue panel (Figure 6c), users can input natural language (NL) instructions and adjust parameters via ambiguity widgets (inline editable input boxes embedded in the dialogue text Figure 6c1). Users can view the rendered result and adjust parameters directly in the preview panel (Figure 6b).

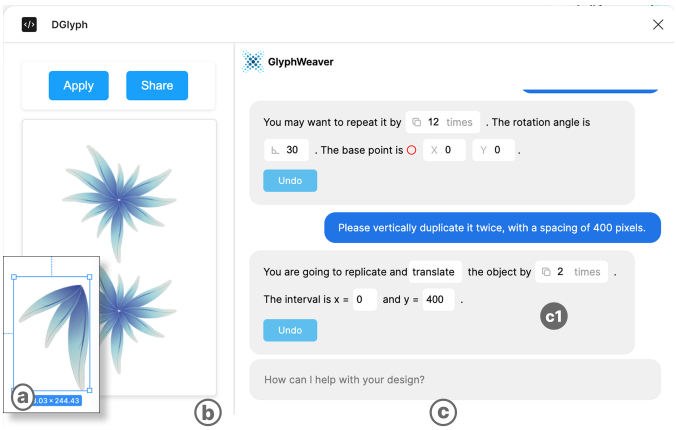


Fig. 6: The interface of *GlyphWeaver*. (a) the manipulation interface (Figma canvas); (b) the preview panel; (c) the dialogue panel. In the Figma canvas, users can create and edit graphics. In the dialogue panel, users can present textual instructions or modify parameters in the widgets. Users can view current rendered results in the preview panel.

[1]Figma: www.figma.com

A. Direct Manipulation Interface

We integrate *GlyphWeaver* with Figma via the Figma API to leverage Figma's direct manipulation interface. Users can select graphics directly on the Figma canvas, and *GlyphWeaver* monitors these selection changes in real time. When the selection is updated, *GlyphWeaver* prompts the user in the interface: *"You have selected [object]. Would you like to start the parsing process?"* The selected graphics are then converted to SVG and, upon confirmation, parsed into GDSL containers.

- BC **Creation**. For selected elements, *GlyphWeaver* creates BCs in the GDSL. Each basic container encapsulates a graphical element, including text, rectangles, circles, paths, or grouped elements treated as a single SVG unit.
- RC **Creation.** We examine whether any of these BCs form repeating patterns. Candidate repeated elements are identified using an MLLM, followed by a rule-based pipeline that infers the geometric transformations relating these elements. This pipeline analyzes the spatial coordinates of repeated elements and organizes them into a hierarchical framework of 2D affine transformations. The space of affine transformations comprises translation, rotation, scaling, and their combinations (shearing is rarely encountered in visual design tools and is therefore excluded). Accordingly, we focus on translations, rotations around a fixed center, and scalings, either overall (s) or along a single direction (s_x, s_y). To align with user intent, we adopt a simple-to-complex inference strategy: transformations are fitted in order of increasing complexity, translations first, then single-center rotations, and finally uniform or non-uniform scalings if necessary. Based on the inferred parameters, we construct RCs in either Cartesian or polar coordinates, associating scaling parameters with the corresponding data encoding. Elements for which no valid transformation can be inferred remain in their BCs.
- CC **Creation.** If the selection contains more than one container, we create a CC to group them. We position these containers based on their coordinates.

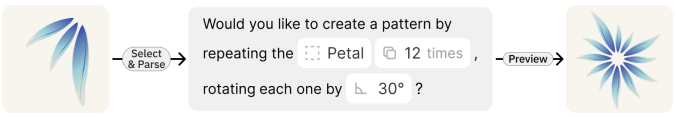


Fig. 7: Repeater container creation from Figma canvas. Users copy three repetitive petals in the Figma canvas. *GlyphWeaver* infers that the user intends to create a repeater container with a polar coordinate system. The user can further adjust the parameters of the repeater container in the dialogue panel.

As shown in Figure 7, *GlyphWeaver* captures user intent by recognizing qualitative and quantitative parameters from the MLLM-based container analysis. *GlyphWeaver* automatically identifies repeated or rotatable elements and infers minimal transformations. For repeated petals, *GlyphWeaver* automatically identifies a polar arrangement and infers a 30-degree default rotation interval. Based on the calculated transformations (repetition and rescaling), we represent user intent using operations for the GDSL. We also provide NL explanations

of the parsing results, as displayed in the dialogue panel of Figure 7, allowing users to understand the glyph-authoring actions.

B. Dialogue Panel

Users can input NL commands in the dialogue panel. Each command is processed by the MLLM translator (e.g., GPT-4o), which takes as input the current GDSL, other existing GDSLs on the canvas, and the user's NL input. The translator first classifies the instruction into one of four operations (except for creating a basic container) and then extracts the relevant parameters required for execution.

- **Parameter Modification.** Users can modify the parameters of a container by setting a single value on a visual attribute. For example, users may say, *"Change the circle's fill to blue."* The translator identifies the target element ID and specified attribute, after which the operation management mechanism applies the update.
- **Data Encoding.** Users can provide a list of values, or specify an expression. For instance, *"Randomize petal sizes between 1 and 1.5."* prompts the model to generate a data expression.
- **CC Creation.** Users can integrate existing DSL instances from the canvas into the current GDSL, specifying the spatial relationship. For example, *"Add a triangle above the square"* or *"Place a circle 50 units above the rectangle."* The translator determines the referenced element, placement rule (e.g., "top"), and horizontal or vertical offset before the operation management mechanism applies the composition.
- **RC Creation.** Users can construct **RCs** by giving NL instructions that describe repetition patterns. For instance, in Figure 6, a user may say, *"Vertically duplicate it twice"*. *GlyphWeaver* determines the target element, repetition count, and the parameters for translation or rotation.

When an instruction cannot be mapped to an operation, the dialogue panel suggests a supported form, such as *"Specify the element and color, e.g., 'Change the circle fill to blue'."* By leveraging an MLLM to translate natural language into precise GDSL operations, *GlyphWeaver* supports flexible interactions.

Ambiguity Widgets. Ambiguity widgets [14] are inline editable boxes (Figure 6c1) developed to expose and resolve ambiguities in NL commands by allowing users to edit parameters. *GlyphWeaver* embeds ambiguity widgets within the dialogue panel's textual feedback, enabling direct parameter adjustment by users. Users can modify:

- **Target element selection.** For each operation, *GlyphWeaver* infers a default target container ID from the current GDSL and displays it in the widget. The widget also provides a drop-down list of all valid IDs, allowing users to select a different target element.
- **Numerical values.** For parameters expressed as numeric values (e.g., repetition counts, angles, translation distances, or coordinates), users can edit these values via the widgets.
- **Color selection.** Users can refine color attributes by selecting or adjusting colors interactively.
- **Data expressions and arrays.** More generally, parameters represented as strings, including generated data expressions and data arrays, are editable as free-form string input.

For example, in Figure 7, the widgets displayed alongside the translator response allow users to modify the repeated unit and repetition count. Adjustments made through ambiguity widgets are immediately reflected in both the GDSL representation and the preview panel, providing instant feedback for refinement.

C. Preview Panel

GlyphWeaver provides a preview panel that is synchronized with the current GDSL instance. The Preview panel acts as a renderer for *GlyphWeaver*, where the DSL is instantiated layer by layer according to the rules of repetition, composition, spatial relationships, and data mappings. The basic elements are represented using SVG primitives such as `<rect>`, `<circle>`, and `<path>` (for polygons and lines), while images are rendered using `<rect>` with texture filters (`<filter>`). Repeating and compositional structures are expanded into corresponding `<g>` (group) elements, each with appropriate transformation parameters. The instantiation process unfolds the hierarchical GDSL structure in a top-down manner, recursively expanding **RCs** based on their parameters. As a result, the rendering time complexity is proportional to the total number of instantiated SVG groups, which is determined by the product of repetition counts across all nested levels in the GDSL. Each repeated structure maps directly to an SVG `<g>` element, so the overall complexity matches that of standard SVG rendering. While the actual rendering time may be slightly higher due to the overhead of recursively expanding the GDSL structure, the asymptotic complexity remains unchanged. These groups and basic elements are then rendered in SVG, as shown in Figure 6.

Once the user is satisfied with the rendered result, clicking the Apply button will send the SVG result onto the Figma canvas. This ensures synchronization among the GDSL instance, the SVG representations in *GlyphWeaver* interface, and the graphics in the Figma canvas. The rendered result synchronized to the Figma canvas is cached on the server via unique identifiers. In subsequent sessions, users can reuse previous GDSL instances. For example, if a user previously created a blue flower and later draws a set of lines, the composition algorithm can associate the blue flower with these lines, such as *"put a blue flower on top of each line."*

D. Interaction Pipeline and Latency

The interaction pipeline exports selected Figma graphics as SVG, passes the selected SVG representation from the Figma plugin to *GlyphWeaver*'s parsing pipeline, invokes the online MLLM when translation is required, and renders the resulting GDSL instance in the preview panel. Table III reports model latencies under the quantitative-evaluation setup in subsection VII-B; because response time varies with the model, network conditions, and input modality, we do not claim a general real-time bound.

VII. Evaluation

We evaluate *GlyphWeaver* through two case studies, a translation benchmark, and a user study to assess its performance and usability. We do not perform a direct quantitative

comparison with general-purpose design tools such as Figma, as these systems do not natively support structured glyph repetition, data binding, and natural-language-driven editing. Instead, our evaluation focuses on the usability and interaction effectiveness of the proposed workflow.

A. Case Studies

Figure 1 shows glyph-based visualizations reproduced with *GlyphWeaver*. All of them are reproductions of award-winning visualizations from the website *Information is Beautiful*. We use two examples to showcase the creation process.

TABLE II: Inputs and GDSL operations for constructing the snowflake glyph. **[Figma]** denotes canvas manipulation, **[NL]** a natural-language instruction, and **[Prototype]** an inferred operation.

User Input / Action	Parsed GDSL Operation
[Figma] Draw 2-3 polylines in canvas	`CreateBasic("polyline1", "polyline", {points, stroke, width}, {origin})`
[Prototype] Infer a repeated structure	`CreateRepeater( "MultiPolyline", "polyline1", "cartesian", 7, "uniform")`
[Figma] Draw a vertical line	`CreateBasic("line1", "line", {start, end, stroke}, {origin})`
[Figma] Combine vertical line and repeated polylines into a branch	`CreateCompositor( "branch1", ["line1", "MultiPolyline"], ...)`
[NL] *"rotate and copy the branch 6 times"*	`CreateRepeater( "snowflake", "branch1", "polar", 6, "uniform")`
[Figma] Draw a hexagon	`CreateBasic("hexagon1", "polygon", {sides: 6, stroke, width}, {origin})`
[Figma] Combine the hexagon and the 6 branches together	`CreateCompositor( "finalGlyph", ["snowflake", "hexagon1"], ...)`

Case 1: snowflake visualization. The visualization in Figure 1 (d) originates from Twitter discussion intensity data regarding different actors across six seasons of the television series *Game of Thrones* [51]. Each snowflake represents an actor. The six branches of a snowflake correspond to an actor's intensity over the six seasons. Each branch consists of several polylines, further detailing the mention frequency for each episode. The snowflake is authored in two stages. During design exploration, users first generate index-dependent placeholder values to inspect the repeated structure without entering the final dataset. These placeholder values can later be replaced by the character's actual Twitter mention frequencies, as in Figure 1(d). The polylines within each branch correspond to the episodes of that season in chronological order. Suppose the first user-drawn polyline has length L_1 and corresponds to the first nonzero reference value x_1, *GlyphWeaver* treats this polyline as the base visual unit with a scale factor of 1.0. A subsequent value x_i is assigned the relative scale factor x_i/x_1, producing $L_i = L_1(x_i/x_1)$. Thus, the absolute lengths depend on the initial geometry drawn by the user, whereas their relative lengths encode the mention frequencies. The central hexagon uses color as a categorical identifier for the character; its size does not encode a data value. The overall glyph size is not independently mapped to another variable, but results from the user-defined base geometry and the relative scaling of the constituent polylines. Traditionally, designing such snowflake glyphs requires manually editing each polyline, which is labor-intensive. The construction complexity increases significantly when polylines vary in size and users need to draw or edit dozens of polylines. As illustrated in Figure 8, with *GlyphWeaver* users can construct a complete snowflake visualization through a few steps (in Table II): *Step 1. Initial Pattern Creation.* Users only need to draw 2-3 polylines in the Figma canvas, after which *GlyphWeaver* infers the user's intention to create an RC with BC polylines through Cartesian repetition and scaling. Users can modify the number of repetitions, set to seven in this example, and inspect the rendered structure in the preview panel. *Step 2. Create a branch.* When users add a vertical line, *GlyphWeaver* creates a basic container for the line and composites the vertical line and the repeated polylines into a CC, a snowflake branch. *Step 3. Radial repetition.* Through the dialogue panel, users can specify *"rotate and copy the branch 6 times"*. *GlyphWeaver* takes the NL instruction and current GDSL instance as input and outputs the operation for creating an RC in a polar coordinate system with 60-degree intervals repeated 6 times. The results are also shown in the dialogue panel, with textual explanation and ambiguity widgets. Users can adjust parameters in the preview panel and the dialogue panel to optimize placement, for example, adjusting the rotation center. *Step 4. Add a hexagon.* Users add a hexagon to the snowflake structure, then replicate this snowflake and encode data by assigning different colors to the hexagons in each instance.

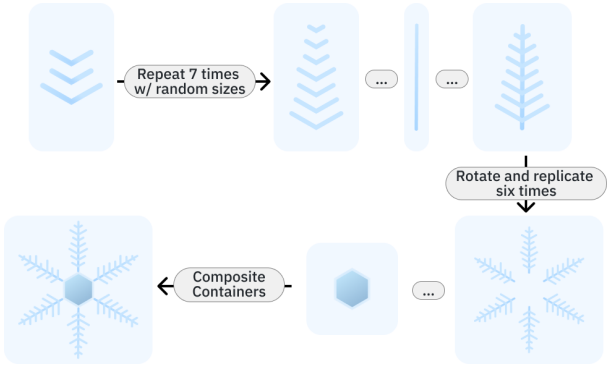


Fig. 8: Creating a snowflake glyph using *GlyphWeaver* includes four steps: creating a repeater container, compositing with the vertical line, creating a repeater container for branches, and compositing with the hexagon.

Case 2: protein visualization. The design [9] in Figure 1 (b) is an award-winning design from the *Information is Beautiful Award*. The visualization employs a plant-inspired structure where curved lines at the base connect to represent proteins of foods, with varying heights indicating different protein concentrations. Foods in the same category are clustered together as a "plant". As shown in Figure 9, using *GlyphWeaver*, users can create this visualization by creating a cluster of lines containing 2-3 curves. After analysis, *GlyphWeaver*

determines that these curves were generated using different x-direction scaling factors following an arithmetic sequence. *GlyphWeaver* reproduces the graphic by creating an RC with different x-scales. *GlyphWeaver* enables users to specify different heights to the curves by using an NL instruction *"give curves with different heights, randomly varying between 0.8 and 1.5"*. This step applies random y-direction scaling to the curves, effectively encoding protein concentration values. Subsequently, users can draw a diamond on the Figma canvas and enter *"place this diamond above each curve"* in the dialogue panel. *GlyphWeaver* uses a `CreateCompositor` operation to incorporate this diamond element (a BC) into the GDSL instance and positions it above each curve, establishing a spatial relationship. Users can then request vertical text placement above each diamond through an NL instruction. The text labels represent various food names, produced by a data encoding operation with a list of food names responded by the MLLM. This way, the corresponding textual information is encoded into the visualization. Figure 9 demonstrates this process through four stages: initial curve generation, height variation application, diamond placement above each curve, and finally, the addition of textual labels positioned vertically above each diamond. After completing these steps, users can further enhance the visualization by creating filled diamonds with different sizes and colors to create the complete structure shown in Figure 1 (b).

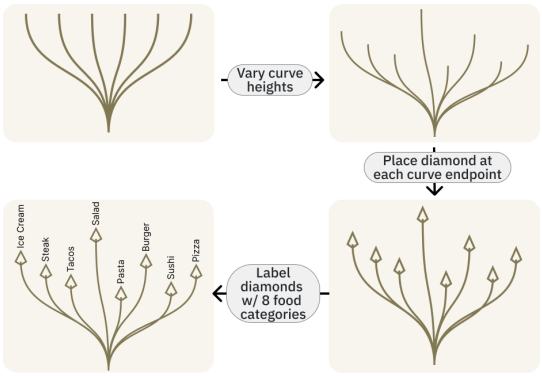


Fig. 9: The process of designing protein visualization.

B. Translation Benchmark

We evaluated how accurately four MLLM-based translators convert NL instructions into supported GDSL operations. We first extracted representative operation scenarios from the glyph-authoring cases and then designed 25 instructions for each of four visual-authoring operations: repeating content, combining an existing GDSL component with a target, modifying a fixed parameter, and encoding data. The resulting benchmark contains 100 unique instructions, each paired with an input GDSL instance and one or more accepted operation outputs. It tests translation within this balanced operation space rather than representing the full distribution of naturally occurring design requests. The supplementary material contains the parsing prompt (Appendix A) and the benchmark and release-file index (Appendix B).

TABLE III: NL-to-GDSL translation results on the 100-case benchmark. Latency is client-side wall-clock time from one recorded response per case and is reported descriptively.

Model	Category Acc (%)	Param Match (%)	Perfect Match (%)	Latency (s)
GPT-5.4	100.00	92.00	82.00	3.90
GPT-5.1	100.00	95.67	89.00	3.74
GPT-4o	100.00	92.33	84.00	2.27
GPT-4o-Mini	99.00	90.33	79.00	2.41

Benchmark Construction. Three annotators specified the expected operation type, target element, and operation-specific fields for each instruction. They resolved disagreements through discussion. When multiple GDSL outputs were judged semantically equivalent, the case-specific ground truth records the accepted alternatives rather than requiring textual identity across the complete serialization. We tested the snapshots `gpt-5.4-2026-03-05`, `gpt-5.1-2025-11-13`, `gpt-4o-2024-08-06`, and `gpt-4o-mini-2024-07-18` using the same prompt, a temperature of 0, and a maximum output length of 500 tokens. Each model produced one recorded response for each case.

Evaluation Metrics. Category Accuracy requires an exact match among the four operation types. Parameter Matching is the macro-average of the scored operation-specific fields for each case. Perfect Match requires the operation type and every scored field to match an accepted ground truth. For example, if a `ModifyParams` prediction selects the correct operation, attribute, and value but the wrong target ID, it is correct for Category Accuracy, matches two of the three scored parameters, and is incorrect for Perfect Match. Missing or unparseable outputs count as incorrect. Mean Recorded Latency is client-side wall-clock time; because network and provider load vary and each case was run once, it is descriptive rather than a controlled performance comparison.

Results and Analysis. Table III shows that the models almost always selected the intended operation category (99–100%). Parameter matching ranged from 90.33% to 95.67%, while perfect match ranged from 79% to 89%. GPT-5.1 obtained the highest parameter matching and perfect-match results; GPT-4o had the lowest recorded mean latency. The remaining mismatches primarily involved inferred repetition counts or rules, selected target IDs and composition relations, and data-expression fields. These results support the feasibility of constrained NL-to-GDSL translation for the tested instructions, but they do not establish performance for unrestricted design language or statistically compare model latency.

C. User Study

The user study examines the usability of *GlyphWeaver*, how participants perceive and use its integrated workflow combining natural-language input and direct manipulation, and their perceived efficiency when authoring glyph-based visualizations. Specifically, the study addresses the following two research questions (RQs): ***RQ1****: Can users easily learn and use GlyphWeaver to author glyph-based visualizations without experiencing a steep learning curve?* ***RQ2****: How do*

participants perceive and use the integrated workflow combining natural-language input and direct manipulation when expressing and refining glyph-design intentions? We additionally report participants' perceived efficiency as a descriptive study theme, rather than treating it as a research question about measured time or effort reduction.

We conducted a user study with 13 participants (P1 to P13; 4 males, 9 females) from the fields of visualization and infographic design. Specifically, six participants were from a school of design, majoring in infographic design. The remaining were Master's or Ph.D. students specializing in data visualization. Ten participants reported being relatively familiar with glyph design, with six of them (P2, P4, P9, P10, P11, P12) reporting high familiarity with glyph design. All participants were proficient in designing infographics using tools like *Figma*, and each had at least one year of experience in infographic design or visualization development. During the user study, *GlyphWeaver* used GPT-4o (`gpt-4o-2024-08-06`) to translate participants' natural-language instructions into GDSL operations. Our user study followed the structured process outlined below:

- **Prototype Introduction.** Participants were first provided with an overview of *GlyphWeaver*, including its integration with Figma. We demonstrated *GlyphWeaver*'s features, with emphasis on the two supported interaction modes: NL instructions and direct manipulation on the Figma canvas.
- **Task Execution.** Participants completed two activities: reproducing one glyph-based visualization from subsection VII-A to experience the supported authoring operations, and using the available features to create a visualization of their own interest. Each participant was allocated approximately one hour to complete all study activities.
- **Questionnaire.** Immediately after completing the design tasks, participants filled out a questionnaire using a 5-point Likert scale (1: Strongly Disagree, 5: Strongly Agree). As shown in Figure 10, the reported questionnaire items comprise two parts: (1) two study-specific items on efficiency and effectiveness and (2) the standard 10-item *System Usability Scale (SUS)*, evaluating usability, learnability, and perceived complexity.
- **Interview.** We conducted interviews with each participant to gain deeper insights into their user experience, challenges encountered, and suggestions for improvement.

The questionnaire results are shown in Figure 10. The questionnaire results are reported alongside corresponding observations from the post-study interviews.

Prototype Usability and Learnability. We calculated the SUS score from Q3–Q12; the mean score was 76.15, indicating good usability. Responses indicated that participants generally found the prototype easy to use (Q5: $\mu = 4.31$, $\sigma = 0.85$) and disagreed that it was unnecessarily complex (Q4: $\mu = 2.00$, $\sigma = 0.71$) or cumbersome (Q10: $\mu = 1.92$, $\sigma = 0.86$). Furthermore, participants agreed that they would learn to use it quickly (Q9: $\mu = 4.08$, $\sigma = 0.64$) and felt confident during its use (Q11: $\mu = 4.08$, $\sigma = 0.64$). In the interviews, P1, P4, and P8 associated their ease of adoption with the prototype's integration into their Figma workflow.

Experience of the Integrated Multimodal Workflow. The SUS integration and consistency items indicated that participants generally perceived the prototype's functions as well integrated (Q7: $\mu = 4.15$, $\sigma = 0.90$) and reported a low perceived level of inconsistency (Q8: $\mu = 1.77$, $\sigma = 0.83$). These items assess the consistency of the prototype as a whole; they do not compare the two interaction modalities. The study-specific item on fulfillment of visualization needs received a positive rating (Q2: $\mu = 4.54$, $\sigma = 0.52$). P3 described the dialogue panel as a natural complement to direct canvas manipulation. P1 reported that NL instructions were particularly useful for repetitive tasks because one instruction could replace multiple manual operations. P3 reported using ambiguity widgets to adjust inferred parameters, and P6 used the preview to inspect those adjustments. P9 and P11 referred to corrective feedback when an instruction could not be parsed. The study examined participants' experience of the integrated workflow and did not isolate the respective contributions of natural-language input and direct manipulation. A controlled comparison of the two modalities and their combination is left for future work.

Efficiency. The time-saving item received the highest mean among the questionnaire items (Q1: $\mu = 4.69$, $\sigma = 0.63$). In the interviews, most participants estimated that they completed the assigned glyph-authoring tasks with *GlyphWeaver* in approximately 10–15 minutes. P12 estimated from personal experience that producing a polished visualization through a traditional collaborative workflow involving design and development could take approximately two weeks. Because the latter estimate covers a broader process that may include ideation, refinement, and implementation, these self-reported durations are not directly comparable. We therefore interpret them as qualitative evidence of participants' perceived efficiency rather than as a measured speedup; no controlled baseline study was conducted.

Suggestions for Improvement. Participants also identified limitations and suggested improvements. One suggestion was to include enhanced data processing capabilities, such as data cleaning and transformation functions, which could reduce dependence on external tools before importing data (P5). The current prototype leaves data cleaning and transformation outside the glyph specification, so this suggestion would require an additional upstream data-processing component. Participants also suggested providing more educational resources, including example cases, reusable semi-finished templates, and a community-driven template repository to support learning and onboarding (P3, P9, P11). A participant (P12) expected higher-level interaction mechanisms, allowing them to describe design intentions at a conceptual level rather than constructing visualizations step by step through low-level operations. Several participants suggested generating initial visual structures from natural-language descriptions instead of first drawing them in Figma. P13 also requested clearer progress indicators for long-running operations.

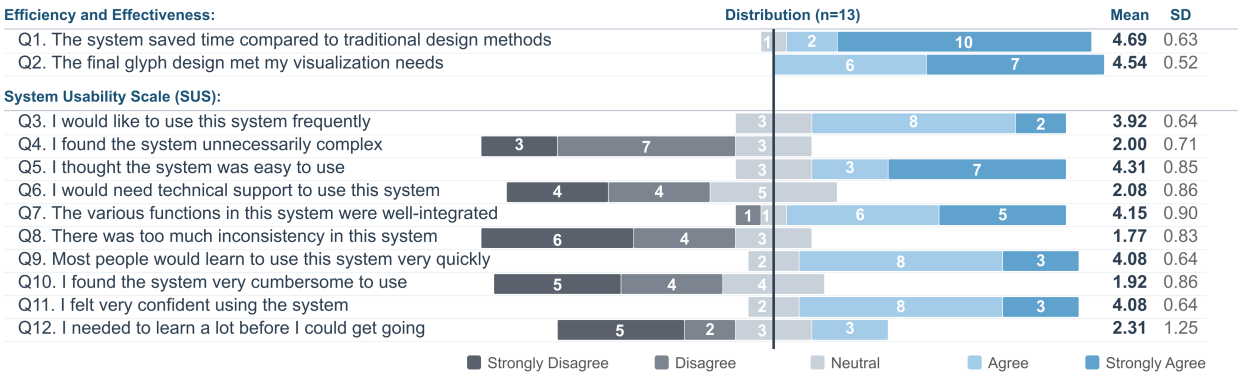


Fig. 10: Questionnaire responses from the user study. Q1–Q2 are study-specific items, and Q3–Q12 comprise the System Usability Scale. Bars show the distribution of 5-point Likert responses; labels report the mean and standard deviation.

VIII. Discussion

Mappings and reuse. *GlyphWeaver* helps bridge user intent and formal specifications of glyph-based visualizations through both natural language input and direct manipulation. Our case studies show that GDSL can represent a range of structures with repeated and hierarchical organizations, illustrating the flexibility of its composition model. Notably, the prototype supports nested data structures beyond simple tabular data, enabling more complex and diverse visualization scenarios. GDSL supports both one-to-one and many-to-one mappings. A basic container can map different variables to different attributes of one object, such as size and color (one-to-one). A repeater container can map multiple variables from one data record to the same attribute type on different repeated elements, such as the lengths of different petals in one glyph (many-to-one). Although *GlyphWeaver* is primarily demonstrated with standalone and infographic-style examples, it can also support interactive visual analytics workflows. Its D3.js-based rendering pipeline exports authored glyphs as reusable code that can be rebound to new data in application-specific interfaces, including small multiples, overview displays, dashboards, and map overlays.

System limits. However, the current GDSL does not natively support automatic layout computation, and some cases therefore require additional processing outside the specification itself. For example, for node link diagrams based on force directed layouts, *GlyphWeaver* allows users to bind data directly to node and link coordinates, but it does not support computing these positions within the specification. We treat such computation as an extra independent functionality that is decoupled from the visual design specification, since it depends on upstream data processing rather than on visual structure itself. This limitation reflects the broad variety of layout methods, which makes comprehensive built-in support difficult in the current prototype. As a result, the present implementation focuses on direct data mapping rather than automatic layout generation. When multiple data variables jointly determine a single visual attribute, users can also precompute a derived value outside GDSL and then bind that value to the attribute. Importantly, we note that data cleaning and wrangling are not the focus of glyph design itself. Such preprocessing tasks involve decisions about data semantics and transformation pipelines, which fall outside the scope of glyph specification.

Design flexibility. *GlyphWeaver* helps users turn design ideas into glyphs they can inspect and refine without writing code from scratch. Because glyphs often use metaphors and context-specific encodings, users can define data-to-visual bindings that suit their designs. The prototype leaves these choices to users without enforcing strict validation, which can lead to unsuitable encodings, especially for users with limited experience. We plan to add warnings that explain potential problems and suggest alternatives while letting users proceed. For example, a warning could explain that assigning nominal data to a continuous gradient implies an order and suggest a categorical palette.

Study limitations. All 13 participants had taken part in an evaluation of *GlyphWeaver* about one year earlier. In this study, they completed new glyph-authoring tasks and immediately filled out the questionnaire, including the ten-item SUS. Prior experience with *GlyphWeaver* may have influenced their learnability ratings (RQ1) and SUS scores.

IX. Conclusion and Future Work

We presented *GlyphWeaver*, an interactive prototype for authoring glyph-based visualizations. Its GDSL represents their structures as hierarchies of basic, repeater, and compositor containers, and its operation management mechanism restricts MLLM translations to five inspectable GDSL operations. The case studies show how this representation reconstructs the repeated and hierarchical structures of the examined glyph-based visualizations. The translation benchmark evaluates whether different models identify the intended operations and parameters, and the user study reports a mean SUS score of 76.15. The reported time estimates reflect perceived efficiency rather than a controlled comparison. These findings are limited to the examined corpus, benchmark instructions, participants, and tasks. The prototype does not compute derived layouts, perform data cleaning, or validate the semantic appropriateness of data mappings. Future work will investigate them and evaluate *GlyphWeaver* against instrumented baseline workflows.

Acknowledgment

This project is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (NTU

Tier 1: RG104/25) and Academic Research Fund Tier 2 (Proposal ID: T2EP20222-0049). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] D3 gallery. https://observablehq.com/@d3/gallery. Accessed: 2026-02-24.

[2] 15-minute in-state phone rates increments in U.S. state prisons. https://www.informationisbeautifulawards.com/showcase/5975-15-minute-in-state-phone-rates-increments-in-u-s-state-prisons, 2023. Accessed: 2026-02-24.

[3] R. Borgo, J. Kehrer, D. H. Chung, E. Maguire, R. S. Laramee, H. Hauser, M. Ward, and M. Chen. Glyph-based visualization: Foundations, design guidelines, techniques and applications. *Proceedings of Eurographics State of the Art Reports*, pp. 39–63, 2013. doi: 10.2312/conf/EG2013/stars/039-063

[4] M. Bostock, V. Ogievetsky, and J. Heer. D^3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 9 pages, 2011. doi: 10.1109/TVCG.2011.185

[5] X. Chen, L. Gong, A. Cheung, and D. Song. PlotCoder: Hierarchical decoding for synthesizing visualization code in programmatic context. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers*, pp. 2169–2181, 2021. doi: 10.18653/v1/2021.acl-long.169

[6] W. Cui, X. Zhang, Y. Wang, H. Huang, B. Chen, L. Fang, H. Zhang, J.-G. Lou, and D. Zhang. Text-to-viz: Automatic generation of infographics from proportion-related natural language statements. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):906–916, 2019. doi: 10.1109/TVCG.2019.2934785

[7] V. D'Efilippo. Poppy Field. https://www.informationisbeautifulawards.com/showcase/1005-poppy-field, 2015. Accessed: 2026-02-24.

[8] V. Dibia and Ç. Demiralp. Data2Vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks. *IEEE Computer Graphics and Applications*, 39(5):33–46, 2019. doi: 10.1109/MCG.2019.2924636

[9] I. Doliva. But what about protein. https://www.informationisbeautifulawards.com/showcase/6325-but-what-about-protein, 2023. Accessed: 2026-11-02.

[10] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein. Iris: A conversational agent for complex tasks. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, 12 pages, pp. 1–12, 2018. doi: 10.1145/3173574.3174047

[11] B. Flury and H. Riedwyl. Graphical representation of multivariate data by means of asymmetrical faces. *Journal of the American Statistical Association*, 76(376):757–765, 1981. doi: 10.1080/01621459.1981.10477718

[12] F. Fragapane. The Shape of Dreams for Google Trends. https://www.informationisbeautifulawards.com/showcase/5563-the-shape-of-dreams, 2022. Accessed: 2026-02-24.

[13] J. Fuchs, P. Isenberg, A. Bezerianos, and D. Keim. A systematic review of experimental studies on data glyphs. *IEEE Transactions on Visualization and Computer Graphics*, 23(7):1863–1879, 2016. doi: 10.1109/TVCG.2016.2549018

[14] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. G. Karahalios. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 489–500, 2015. doi: 10.1145/2807442.2807478

[15] D. Han, H. Zhu, W. Chen, J. Pan, R. Chen, X. Wang, Z. Wen, L. Weng, M. Zhu, Y. Wu, et al. Nuwa: An authoring tool for graph visualizations. In *Proceedings of IEEE Pacific Visualization Conference*, pp. 142–151. IEEE, 2024. doi: 10.1109/PacificVis60374.2024.00024

[16] J. Heer. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences*, 116(6):1844–1850, 2019. doi: 10.1073/pnas.1807184115

[17] K. Hu, M. A. Bakker, S. Li, T. Kraska, and C. Hidalgo. VizML: A machine learning approach to visualization recommendation. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2019. doi: 10.1145/3290605.3300358

[18] Information is Beautiful Awards. Information is beautiful awards. https://www.informationisbeautifulawards.com/, 3 2025. Accessed: 2026-11-02.

[19] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. ECharts: a declarative framework for rapid construction of web-based visualization. *Visual Informatics*, 2(2):136–146, 2018. doi: 10.1016/j.visinf.2018.04.011

[20] G. Li, M. Tian, Q. Xu, M. J. McGuffin, and X. Yuan. GoTree: A grammar of tree visualizations. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2020. doi: 10.1145/3313831.3376297

[21] H. Li, Y. Wang, S. Zhang, Y. Song, and H. Qu. Kg4vis: A knowledge graph-based approach for visualization recommendation. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):195–205, 2021. doi: 10.1109/TVCG.2021.3114863

[22] Y.-N. Li, D.-J. Li, and K. Zhang. Metaphoric transfer effect in information visualization using glyphs. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, pp. 121–130. ACM, 2015. doi: 10.1145/2801040.2801062

[23] C. Liu, Y. Han, R. Jiang, and X. Yuan. ADVISor: Automatic visualization answer for natural-language question on tabular data. In *Proceedings of the IEEE Pacific Visualization Symposium*, pp. 6–15, 2021. doi: 10.1109/PACIFICVIS52677.2021.00010

[24] Z. Liu, C. Chen, and J. Hooker. Manipulable semantic components: A computational representation of data visualization scenes. *IEEE Transactions on Visualization and Computer Graphics*, 31(1):732–742, 2025. doi: 10.1109/TVCG.2024.3456296

[25] Z. Liu, R. Metoyer, M. Dontcheva, M. Hoffman, A. Wilson, D. Delorey, and K. Wongsuphasawat. Data illustrator: Augmenting vector design tools with lazy data binding for expressive visualization authoring. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2018. doi: 10.1145/3173574.3173697

[26] Y. Luo, X. Qin, N. Tang, and G. Li. DeepEye: Towards automatic data visualization. In *Proceedings of the IEEE International Conference on Data Engineering*, pp. 101–112, 2018. doi: 10.1109/ICDE.2018.00019

[27] Y. Luo, N. Tang, G. Li, C. Chai, W. Li, and X. Qin. Synthesizing natural language to visualization (nl2vis) benchmarks from nl2sql benchmarks. In *Proceedings of the International Conference on Management of Data*, 13 pages, pp. 1235–1247, 2021. doi: 10.1145/3448016.3457261

[28] E. Maguire, P. Rocca-Serra, S.-A. Sansone, J. Davies, and M. Chen. Taxonomy-based glyph design—with a case study on visualizing workflows of biological experiments. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2603–2612, 2012. doi: 10.1109/TVCG.2012.271

[29] A. M. McNutt. No grammar to rule them all: A survey of JSON-style DSLs for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):160–170, 2022. doi: 10.1109/TVCG.2022.3209460

[30] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, 2019. doi: 10.1109/TVCG.2018.2865240

[31] A. Narechania, A. Srinivasan, and J. Stasko. NL4DV: A toolkit for generating analytic specifications for data visualization from natural language queries. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):369–379, 2021. doi: 10.1109/TVCG.2020.3030378

[32] M. Nylin, J. Lundberg, M. Bång, and K. Kucher. Glyph design for communication initiation in real-time human-automation collaboration. *Visual Informatics*, 8(4):23–35, 2024. doi: 10.1016/j.visinf.2024.09.006

[33] A. Offenwanger, T. Tsandilas, and F. Chevalier. DataGarden: Formalizing personal sketches into structured visualization templates. *IEEE Transactions on Visualization and Computer Graphics*, 31(1):1268–1278, 2025. doi: 10.1109/TVCG.2024.3456336

[34] J. Pollock, H. Johnson, C. Perin, and B. A. Myers. Bluefish: Composing diagrams with declarative relations. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2024. doi: 10.1145/3654777.3676465

[35] F. J. Post, T. van Walsum, F. H. Post, and D. Silver. Iconic techniques for feature visualization. In *Proceedings of Visualization*, pp. 288–295. IEEE, 1995. doi: 10.1109/VISUAL.1995.485141

[36] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. GlyphMaker: creating customized visualizations of complex data. *Computer*, 27(7):57–64, 1994. doi: 10.1109/2.299412

[37] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, vol. 33, pp. 351–360. Wiley Online Library, 2014. doi: 10.1111/cgf.12391

[38] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization*

and Computer Graphics, 23(1):341–350, 2017. doi: 10.1109/TVCG.2016.2599030

[39] J. Scott-Brown and B. Bach. Netpanorama: A declarative grammar for network construction, transformation, and visualization. *arXiv preprint arXiv:2310.18902*, 2023. doi: 10.48550/ARXIV.2310.18902

[40] C. D. Shaw, J. A. Hall, C. Blahut, D. S. Ebert, and D. A. Roberts. Using shape to visualize multivariate data. In *Proceedings of the Workshop on New Paradigms in Information Visualization and Manipulation*, pp. 17–20, 1999. doi: 10.1145/331770.331777

[41] H. Siirtola. The effect of data-relatedness in interactive glyphs. In *Proceedings of the Ninth International Conference on Information Visualisation*, pp. 869–876. IEEE, 2005. doi: 10.1109/IV.2005.123

[42] M. Stefaner. OECD Better Life Index. https://www.informationisbeautifulawards.com/showcase/457-better-life-index, 2011. Commissioned by the Organisation for Economic Co-operation and Development (OECD). Accessed: 2026-11-02.

[43] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002. doi: 10.1109/2945.981851

[44] Y. Sun, J. Leigh, A. Johnson, and S. Lee. Articulate: A semi-automated model for translating natural language queries into meaningful visualizations. In *Proceedings of the International Symposium on Smart Graphics*, pp. 184–195, 2010. doi: 10.1007/978-3-642-13544-6_18

[45] 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 & Computer Graphics*, 31(03):1731–1745, 2025. doi: 10.1109/TVCG.2024.3368621

[46] L. Wang, S. Zhang, Y. Wang, E. Lim, and Y. Wang. LLM4Vis: Explainable visualization recommendation using chatgpt. In M. Wang and I. Zitouni, eds., *Proceedings of the Conference on Empirical Methods in Natural Language Processing: (Industry Track)*, pp. 675–692, 2023. doi: 10.18653/V1/2023.EMNLP-INDUSTRY.64

[47] Q. Wang, Z. Chen, Y. Wang, and H. Qu. AI4VIS: Survey on artificial intelligence approaches for data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(9):3740–3758, 2021. doi: 10.1109/TVCG.2021.3067505

[48] Q. Wang, Z. Chen, Y. Wang, and H. Qu. A survey on ML4VIS: Applying machine learning advances to data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):808–828, 2021. doi: 10.1109/TVCG.2021.3099002

[49] M. O. Ward. Multivariate data glyphs: Principles and practice. In *Handbook of data visualization*, pp. 179–198. Springer, 2008. doi: 10.1007/978-3-540-33037-0_8

[50] C. Wittenbrink, A. Pang, and S. Lodha. Glyphs for visualizing uncertainty in vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):266–279, 1996. doi: 10.1109/2945.537309

[51] K. Wongsuphasawat. Winter is here. https://www.informationisbeautifulawards.com/showcase/3708-winter-is-here, 2018. Accessed: 2026-02-24.

[52] S. Xiao, S. Huang, Y. Lin, Y. Ye, and W. Zeng. Let the chart spark: Embedding semantic context into chart with text-to-image generative model. *IEEE Transactions on Visualization and Computer Graphics*, 30(01):284–294, 2024. doi: 10.1109/TVCG.2023.3326913

[53] L. Xie, Y. Lin, C. Liu, H. Qu, and X. Shu. DataWink: Reusing and Adapting SVG-Based Visualization Examples with Large Multimodal Models . *IEEE Transactions on Visualization & Computer Graphics*, 32(01):824–834, Jan. 2026. doi: 10.1109/TVCG.2025.3634635

[54] L. Xie, J. Zhou, A. Rao, H. Qu, and X. Shu. DataSway: Vivifying metaphoric visualization with animation clip generation and coordination. In *Proceedings of the ACM Designing Interactive Systems Conference*, 2026. doi: 10.48550/arXiv.2507.22051

[55] L. Ying, X. Shu, D. Deng, Y. Yang, T. Tang, L. Yu, and Y. Wu. MetaGlyph: Automatic generation of metaphoric glyph-based visualization. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):331–341, 2023. doi: 10.1109/TVCG.2022.3209447

[56] L. Ying, T. Tang, Y. Luo, L. Shen, X. Xie, L. Yu, and Y. Wu. GlyphCreator: Towards example-based automatic generation of circular glyphs. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):400–410, 2022. doi: 10.1109/TVCG.2021.3114877

[57] B. Yu and C. T. Silva. FlowSense: A natural language interface for visual data exploration within a dataflow system. *IEEE Transactions on Visualization and Computer Graphics*, 26(1):1–11, 2020. doi: 10.1109/TVCG.2019.2934668

[58] S. Zhang, H. Li, H. Qu, and Y. Wang. AdaVis: Adaptive and explainable visualization recommendation for tabular data. *IEEE Transactions on Visualization and Computer Graphics*, 30(9):5923–5938, 2023. doi: 10.1109/TVCG.2023.3316469



Can Liu is an assistant professor at the Central University of Finance and Economics. He was previously a research fellow in the College of Computing and Data Science at Nanyang Technological University. He received his Ph.D. in Intelligent Science (2023), B.Sc. in Computer Science (2018), and B.Ec in Economics (2018) degrees from Peking University. His research interests lie in the field of deep learning-driven visualization, especially intelligent interaction for visualization. Refer to https://vvoliucano.github.io for more details.



Shiwei Chen is a master student at the College of Computing and Data Science, Nanyang Technological University, Singapore. Her research interests include human–AI interaction and scientific visualization. She is currently exploring AI-assisted approaches to support creativity in visual representation.



Zhibang Jiang received a B.A. degree in journalism from Communication University of Zhejiang and an M.S. degree in data visualization from Parsons School of Design. His design interest is human-centered design for data-intensive services. His design footprints include tenures at leading organizations such as Providence, Tencent, Alibaba, Peking University, and GroupM, and collaborations with partners such as Nike, and the Smithsonian. He has shaped the data experience for millions of users.



Yong Wang is currently an assistant professor in the College of Computing and Data Science, Nanyang Technological University. Before that, he worked as an assistant professor at Singapore Management University from 2020 to 2024. His research interests include data visualization, HCI, and human-AI collaboration, with an emphasis on their application to FinTech, quantum computing and online learning. He obtained his Ph.D. in Computer Science from Hong Kong University of Science and Technology. He received his B.E. and M.E. from Harbin Institute of Technology and Huazhong University of Science and Technology, respectively. For more details, please refer to http://yong-wang.org.