Flint: A Semantics-Driven Data Visualization Intermediate Language

microsoft.github.io/flint-chart/

Yunhai Wang, Kecheng Lu, Junhao Chen, Alper Sarikaya, Chenglong Wang

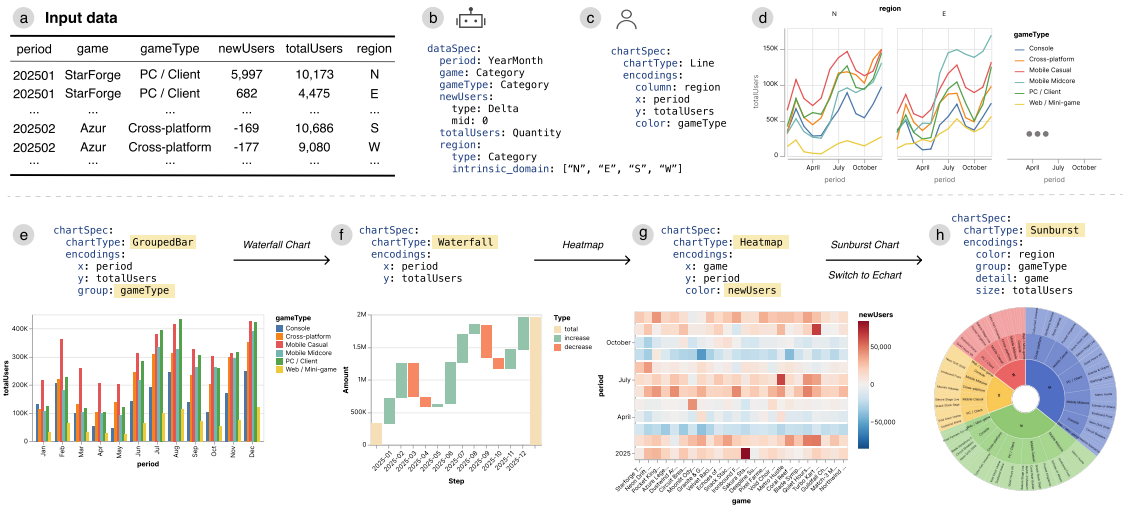

Figure 1: Flint is an intermediate language for data visualization. A Flint specification has two parts: a data spec, which can be inferred by an LLM agent from the data, and a simple chart spec. Given the input data, the compiler uses semantic information to choose low-level parameters and translate the Flint spec into a well-configured specification for a target chart type, such as a Vega-Lite faceted line chart. As the user edits the Flint spec to create new visualizations, the compiler automatically updates these configurations to produce *high-quality charts* without requiring the user to manually adjust detailed parameters such as axes, layers, or color schemes. This allows users to rapidly explore data across diverse chart types and rendering backends, for example by switching to ECharts to support a sunburst chart.

Abstract—We present Flint, an intermediate language that enables authors to create high-quality visualizations from concise, semantics-driven specifications without explicitly configuring low-level parameters such as scales, axes, and formatting. Unlike prior systems that infer default configurations from surface-level data representations, often producing brittle choices, Flint introduces a hierarchical data semantic model that allows users to specify the meanings of data fields structurally and helps the compiler derive appropriate visualization configurations. From a concise specification, the system generates and optimizes library-agnostic visualization configurations and translates them into complete, executable specifications for multiple target grammars, including Vega-Lite, Apache ECharts, and Chart.js. We demonstrate that Flint simplifies the authoring process without compromising on visual quality, and it is an effective intermediate language for both humans and AI agents to create visualizations.

Index Terms—LLM, semantic type, intermediate language

1 Introduction

Effective data exploration requires analysts to rapidly create and adapt visualizations, frequently changing chart types, encodings, and layouts as they refine hypotheses. Consequently, visualization specifications must remain simple while maintaining the capacity to produce high-quality designs. Declarative languages like Vega-Lite [34] and ECharts [22] have advanced this goal, with recent work combining these grammars with large language models (LLMs) to enable natural language to visualization (NL2Vis) systems [5] and autonomous data agents [41, 44]. These approaches lower the barrier to iterative chart creation by generating complete specifications from user descriptions.

Despite these advances, authors still face a persistent trade-off. Simple high-level specifications that fully rely on system "smart default" low-level configurations [47] often yield poor designs, whereas detailed specifications can improve quality but become verbose, brittle, and difficult to modify because of complex parameter dependencies. The fundamental issue is that

- *Yunhai Wang, Kecheng Lu, and Junhao Chen are with Renmin University of China and Bigdata and Responsible Artificial Intelligence for National Governance, Beijing, China. E-mail: {wang.yh, lukecheng0407, chenjunhao11}@ruc.edu.cn.*
- *Alper Sarikaya and Chenglong Wang are with Microsoft, Redmond, WA, USA. E-mail: {alper.sarikaya, chenglong.wang}@microsoft.com.*

existing libraries must infer data semantics from *raw representations* to configure smart defaults. Incorrect inferences produce misleading visualizations: stacking non-additive measures like correlation, applying a sequential color scale to diverging-class fields, or parsing integer dates (e.g., 20080101) as UNIX seconds, resulting in erroneous 1970 dates. Recommendation systems like Draco [30] alleviate generic configuration challenges, but modeling the exponential spaces of all parameters, especially those for layered and composite charts (e.g., waterfall, pyramid), across different libraries remains impractical.

As LLM-based agents are increasingly used to generate specifications from natural language or demonstrations, overcoming this configuration barrier has become urgent. The challenge is twofold. *From the user perspective*, even when agents produce visually effective charts, the resulting specifications are often verbose and fragile, with hard-coded interdependent low-level parameters. Small changes such as altering encodings or swapping chart types often break the code, forcing users to request full regeneration and incurring high token costs and latency that hinder interactive exploration. *From the developer perspective*, building agents that consistently follow best practices across libraries remains difficult. Visualization grammars have long-tailed parameter spaces and use different vocabularies. Training data is also uneven across libraries. These factors make agents unreliable for less common chart types and libraries. Developers therefore face substantial challenges in training or instructing LLM agents to robustly generate correct and well-designed visualizations.

In this paper, we argue that the trade-off between fragile, high-quality charts and flexible, poorly designed defaults comes from a language limitation: existing visualization grammars lack an explicit way to specify the semantics of data. Many low-level parameters implicitly encode semantic decisions, such as parsing rules, meaningful baselines (e.g., zero), valid aggregations, and formatting. Without semantics as first-class objects, human authors and LLM agents must encode their intent indirectly through monolithic, library-specific parameters. Conversely, when semantics are omitted entirely, systems are forced to "guess" data meanings, often producing suboptimal or misleading designs.

To address these challenges, we introduce Flint, a semantics-driven intermediate language for visualization. Inspired by database research that leverages semantic types for data integration and cleaning [18, 20], our key insight is to formalize a hierarchical semantic type system as first-class objects in visualization specifications. Rather than requiring users or LLMs to manipulate low-level parameters, Flint explicitly represents data semantics as types to guide visualization design. Flint's data-driven compiler integrates data semantics, raw data representations, and chart specifications to automatically infer critical configurations, including scales, axes, aggregations, formatting, and layout. As a result, users only need to specify a chart type and field-to-channel mappings, while the system derives the remaining low-level details. This design aligns with users' preference for using simple chart-type-based approaches to describe visualization goals during data exploration [1, 2, 29].

Designed as a library-agnostic intermediate language, Flint bridges high-level chart intent and executable rendering grammars. Like intermediate representations in general-purpose programming (e.g., LLVM [19]), it does more than resolve semantics: the compiler also optimizes chart specifications and generates output for the target library selected by the user, such as Vega-Lite, Apache ECharts, or Chart.js. For example, the same heatmap specification can be adapted to data density: a dense matrix yields compact cells, fewer ticks, and abbreviated labels, whereas a sparse matrix produces larger cells and more detailed annotations. By embedding such design knowledge into compiler passes, Flint reduces reliance on carefully designed prompting and ad hoc defaults, making specifications more robust and easier to revise during iterative exploration. Its flexible backend architecture also lets users switch rendering libraries without rewriting chart specifications. In our implementation, we realize backends for three libraries with substantially different specification models: Vega-Lite, Apache ECharts, and Chart.js.

Through a case analysis (Section 6), we show that Flint can compile concise semantic specifications into detailed, backend-specific code. We also show that Flint is well suited to LLM-based generation: semantic types are often easy for AI agents to infer from data field names, value patterns, and common sense, enabling the model to produce high-quality charts from compact semantic inputs. Compared with directly generating library-level specifications, Flint also enables LLMs to produce much shorter programs while maintaining strong visual quality, as verified by LLM judges [7]. Our contributions are:

- We introduce Flint, an intermediate language for specifying charts using data semantics and simple visual encodings.
- We develop a compiler that resolves and optimizes low-level chart configurations holistically from semantic intent, transpiling them to different visualization backends.
- We release Flint with compilers for Vega-Lite, Apache ECharts, and Chart.js, and evaluate its effectiveness through comprehensive chart galleries and LLM-generation experiments.

2 Related Work

Declarative Visualization Specification. Modern chart authoring is fundamentally shaped by declarative visualization grammars such as Wilkinson's *Grammar of Graphics* [47], alongside implementations like ggplot2 [42], Vega [35], and Vega-Lite [34]. These grammars provide a structured space of encodings, composition operators, and interaction mechanisms. This foundation enables authors to systematically reason about design decisions and iteratively refine charts, vastly improving upon ad hoc imperative code. Simultaneously, widely adopted libraries like D3 [4], Mascot.js [23], Matplotlib [40], and Apache ECharts [22] provide highly expressive rendering capabilities and extensive chart-type coverage. However, these systems (even high-level grammars like Vega-Lite) often demand low-level parameter configurations for marks, axes, scales, layouts, and styling to better reflect data semantics, because system defaults inferred from surface data representations can be fragile and hard-coded design choices can be difficult to edit [38]. Our work addresses this gap; Flint leverages data semantics to manage low-level library-specific chart configuration details, and it leverages existing libraries for final rendering.

Template-Based Visualization Authoring. Code reuse through templating is a common practice in both general programming and visualization design [39]. Many visualization tools (e.g., Excel) and grammars utilize templates or predefined "chart types" to accelerate the generation process, such as for waterfall, lollipop, and sunburst charts. However, traditional template-based approaches [1, 29] are often very restrictive. Static templates tightly couple a chart's structural scaffold with its underlying data configuration, making them restrictive and difficult to adapt to changes in data type or cardinality. More configurable templates offer greater flexibility, but they often expose a large number of low-level parameters, which can be difficult for both users and LLM agents to use effectively. Flint can be viewed as a form of "dynamic template" that decouples visual structure from

data semantics: semantic configurations are determined dynamically from the data, while structural composition is governed by reusable templates. In this way, Flint complements prior work on visualization reuse by enabling flexible templates for bespoke charts that can automatically adapt to diverse datasets.

LLM-based NL2VIS. Recent advancements in LLM-based NL2Vis techniques [12, 13] have enabled the translation of natural language (NL) queries into visualizations by generating grammar-specific code (e.g., Vega-Lite) or utilizing multi-stage pipelines. Systems such as Chat2VIS [28] and LIDA [11] demonstrate that general-purpose LLMs can autonomously select chart types and produce viable specifications. Subsequent research has focused on improving robustness: NL4DV-LLM [33] uses LLMs to infer analytic tasks and attribute roles, while VegaChat [17] introduces automated heuristic corrections and schema validation to guide error recovery. Recent state-of-the-art systems use LLMs to directly generate low-level visualization code [5, 44]. Although these specifications can yield high-quality charts, they are often complex and fragile, with interdependent parameters—even small edits require full regeneration—which increases both latency and cost for exploration. LLM performance also remains uneven across libraries because training data is imbalanced [7]. In response, we introduce Flint, a semantic intermediate language that lets LLMs translate natural-language intent into a concise representation, while the compiler automatically handles low-level configuration and adapts the result to different visualization backends based on data semantics.

Visualization Design Knowledge. Research in visualization has established foundational principles of visual encoding. Early work by Bertin [3] and Cleveland and McGill [10] demonstrated that the accuracy of information recovery depends on the alignment between data types and perceptual tasks. Subsequent research has extended these insights to higher-level analytical tasks, such as interpreting aggregates, distributions, and correlations, while articulating specific perceptual guidelines. Examples include "banking to 45°" for optimal aspect ratios [8, 45, 46], selecting perceptually distinct hues for multi-class categorical data [14, 25], and addressing the sine illusion in streamgraph ordering [6, 43]. These studies underscore that effectiveness is not merely a matter of encoding choice, but a complex interaction between data semantics, layout, and human perception.

Embedding such design knowledge into practical systems remains difficult, especially in AI-based ones where LLM agents do not reliably follow best practices because of uneven training data, even with complex prompting. Flint addresses this issue by encoding design knowledge directly as deterministic optimization passes that systematically adjust visualization parameters. This separation lets LLMs focus on *what* to visualize, while the compiler determines *how* to visualize it well. Because Flint is library-agnostic, it also lowers the cost of transferring these design principles across practical rendering systems.

Recommendation systems. Automatically recommending visualizations from data or partial inputs can help users better explore data. Early systems like APT [26] and ShowMe [27] relied on physical data types to select encodings, while modern systems like Draco [30] formalize design knowledge into strict constraints and preference rules to rank Vega-Lite charts. However, because low-level configuration spaces are large and difficult to cover exhaustively, existing recommendation engines are often restricted to a small set of common chart types with simplified parameterizations, limiting the exploration space. Flint complements these systems by abstracting away low-level configuration details and exposing semantic information directly. This allows recommendation engines to focus on higher-level design decisions, such as chart type, encoding composition, and analytical task fit, while relying on the compiler to handle downstream parameterization.

Semantic Types. Semantic type modeling has been widely studied in the database community for data cleaning and integration. Systems such as Sherlock [18] and Sato [50] predict fine-grained concepts from tabular data using machine learning, while more recent approaches use LLMs [21] to further improve detection. Modern analytics platforms [20, 24] also increasingly deploy semantic layers to define entities and governed metrics, ensuring consistent aggregation and interpretation. These efforts highlight the value of explicit data semantics, but existing models are designed primarily for data preparation and are not fine-grained enough for visualization needs. To address this gap, we introduce a visualization-oriented semantic type hierarchy. By treating semantic types as first-class objects, our compiler can constrain parameter choices and systematically generate higher-quality charts while reducing user configuration burden.

3 Overview

In this section, we first use an example to explain challenges for systematically handling low-level chart configurations in existing libraries. Then, we provide an overview of our language design and the compilation approach to address this problem.

3.1 Motivating Examples

We use a user-engagement dataset from a consulting scenario as a running example. It contains four fields: Period, Week, AgeGroup, and Users. The dataset records active-user counts over 72 monthly periods (202001 to 202512), with each period further broken down by week (wk1–wk4) and age group (18–24, 25–34, . . . , 60+). Sample rows are shown in Figure 2.

Period	Week	AgeGroup	Users
202001	wk1	18–24	15412
202001	wk1	25–34	24440
202001	wk1	35–44	20241
		...	
202001	wk2	18–24	16255
		...	
202512	wk4	60+	7449

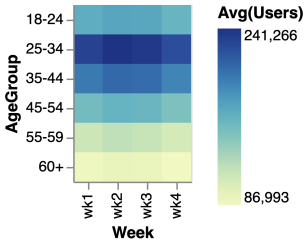

Figure 2: Example user-engagement dataset and a simple heatmap.

From this dataset, the user can easily create a heatmap to compare week and user group trends: Week on the x-axis, AgeGroup on the y-axis, and avg(Users) as color. This chart is easy to create in Vega-Lite using the rect mark.

```
{ "mark": "rect", "encoding": {
    "x": {"field": "Week"},
    "y": {"field": "AgeGroup"},
    "color": {"field": "Users", "aggregate": "mean"} } }
```

Because the specification is short and the default primitive data types align perfectly with visual mappings, it is easy to adapt: with minor edits, the user can swap the x and y encodings or try a different chart type, such as a grouped bar chart.

The analyst now wants a small conceptual edit: replace Week with Period on the x-axis and change the aggregation from avg(Users) to sum(Users). Ideally, the user should simply be able to swap the field and update the aggregate, an edit of only two tokens, and obtain a properly rendered chart. However, this is where the gap between primitive data types and **semantic types** becomes a critical hurdle. While Period values (e.g., 202001) are stored as raw integers, their semantic type is a structured temporal unit (Year-Month). Because standard visualization grammars

do not natively understand these rich semantic types, the user faces a dilemma: accept a simple specification that produces poor results, or write a verbose specification burdened with tightly coupled, low-level parameters.

Option 1: Simple but problematic specification. A short specification no longer produces a satisfactory chart because the system misinterprets the underlying semantics of the data. If Period is naively cast as a temporal type directly from the integer 202001, the system misparses the dates, resulting in an invalid timeline and wrong geometry (Figure 3, top). Conversely, if it is treated as a nominal string, the chart blindly expands into 72 discrete categories (Figure 3, bottom). Without understanding that Period here can simultaneously behave as discrete values to control tile configuration and a continuous temporal axis for parsing and axis formatting, the compiler cannot automatically decide a set of default parameters to produce a good visualization.

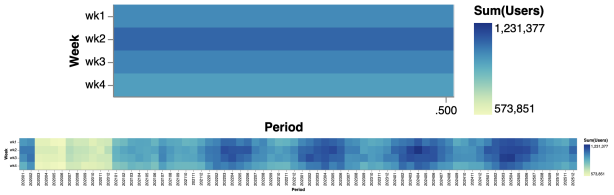

Figure 3: A simple specification fails in two different ways: temporal interpretation misparses the period field, while nominal interpretation produces an overcrowded layout.

Option 2: Detailed but fragile specification. An expert can create a clean heatmap for Period × Week × sum(Users). However, doing so requires coordinating a fragile set of low-level decisions that compensate for the system's lack of semantic awareness:

- *Manually enforcing temporal semantics:* Because the system sees 202001 merely as a raw integer, the expert must explicitly parse it using a %Y%m format and force the axis to be temporal. This explicit intervention prevents the chart from rendering arbitrary numbers and forces the use of time-based tick generation algorithms for readable labels.
- *Adapting physical geometry to accommodate high data cardinality:* To reconcile the broken positional behavior, the expert must coordinate a fragile dual-treatment: treating Period as continuous for chronological parsing and axis tick generation, while managing it as discrete for heatmap tile alignment. This requires hard-coding interdependent layout parameters, specifically a 7px mark width and a 504px canvas span, and manually padding the axis domain bounds to ["2019-12-17T00:00:00", "2026-01-16T00:00:00"]. This padding ensures that the first and last time-based ticks precisely align with the outer edges of the dense tiles without text overlap.
- *Refining color domain:* Because sum(Users) encodes absolute quantitative magnitude, its visual encoding must remain mathematically proportional. The expert therefore fixes the color scale to a zero baseline, preventing auto-scaling from distorting the perceived ratios of user counts.

Figure 4 shows the resulting Vega-Lite specification and rendered heatmap. While effective, the solution is fragile because it depends on hard-coded, interdependent parameters that a semantics-aware system should ideally infer automatically. This limitation becomes apparent when the user makes a seemingly simple change, such as rotating the chart to AgeGroup × Date × sum(Users) and faceting by Week. The chart representation changes substantially: Week becomes a facet, Date moves to a vertical temporal axis, and AgeGroup becomes a horizontal categorical axis, so the previous configuration no longer transfers.

```
{ "transform": [{
    "calculate": "timeParse(toString(datum.Period), '%Y%m')",
    "as": "Date" }],
  "mark": {"type": "rect", "width": 7},
  "encoding": {
    "x": { "field": "Date", "type": "temporal",
      "scale": {
        "nice": false,
        "domain": ["2019-12-17T00:00:00", "2026-01-16T00:00:00"] } },
    "y": { "field": "Week" },
    "color": {
      "field": "Users", "aggregate": "sum",
      "scale": {"domainMin": 0} } },
  "config": { "view": {"continuousWidth": 504} }}
```

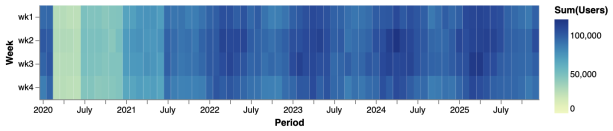

Figure 4: An expert can produce a clean heatmap for Period × Week × sum(Users), but only by coordinating tightly coupled low-level parameters, including parsing, scale, mark width, and layout.

The user must manually redefine layout, spacing, geometry, and cross-facet color consistency.

These challenges extend beyond this example and are especially pronounced in compositional visualizations such as pyramid or waterfall charts, where parameters must be coordinated within and across layers. The issue is further amplified across visualization libraries with different abstractions. For example, Vega-Lite controls layout through encodings, scales, and view properties, while ECharts organizes charts using series, grids, and component configurations. Even in the heatmap example, categorical spacing is handled through step size in Vega-Lite but through layout regions and paddings in ECharts. This diversity forces users and AI agents to reason about different parameterizations for the same visual intent. This scenario motivates the need for a unified, library-agnostic model that derives low-level specifications from high-level semantics.

3.2 System Architecture

Our solution is to design an intermediate language that lets users specify visualization intent through high-level declarative representations while the compiler handles low-level configuration automatically across libraries. The specification includes the semantic structure of the data and the intended chart. The full specifications for the experts' heatmaps are shown here.

```
dataSpec:
  Period: YearMonth
  Week: Week
  AgeGroup: Range
  Users: Count
```

```
chartSpec:
  chartType: Heatmap
  encodings:
    x: Week
    y: AgeGroup
    color: avg(Users)
```

```
chartSpec:
  chartType: Heatmap
  encodings:
    x: Period
    y: Week
    color: sum(Users)
```

Such simple specs make chart adaptation easy: the user only updates the high-level chart intent, while the data specification remains unchanged. When the chart specification changes, the compiler combines semantic information with raw data characteristics to (re)infer the parameters required by the target library, freeing users from low-level specification management while still producing high-quality charts.

Figure 5 shows our system architecture: a modular three-stage compilation pipeline inspired by the LLVM architecture [19]. The key idea is to maintain and optimize a set of abstract visualization properties in a library-agnostic fashion that reflects data semantics before translating them into backend-specific executable specifications.

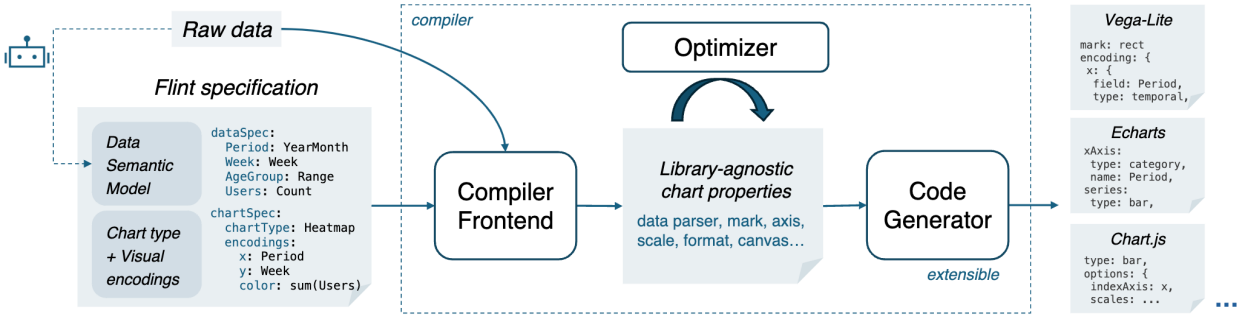

Figure 5: Overview of the Flint architecture. Flint employs a modular three-stage design: a *compiler frontend* for translating user intents into library-agnostic properties, an *optimization stage* that refines configurations, and *extensible code generators* that produce library-native specs for desired rendering engines. LLM agents can infer data semantic specs from the raw data to reduce authoring efforts.

Compiler Frontend. The pipeline begins with raw data and a user-defined Flint specification, including data semantic specs, the chart type, and visual encodings. The frontend resolves encoding properties from these inputs and represents them as a shared *compilation context*. Properties produced from this stage capture core encoding decisions reflecting the semantic meaning of data, including data parsing, encoding type, formatting behavior, aggregation, scale, domain, ordering, and midpoints.

Optimizer. The optimizer enriches this compilation context with layout decisions, coordinating the sizes of visual elements with the available canvas while accounting for data cardinality, encoding types, grouping, and faceting. It first resolves local layout properties for each axis, such as axis span, band step, padding, group width, radius, and facet panel size, and then refines them globally to resolve dependencies like aspect ratio and facet wraps. Thus, this stage augments the compilation context with *optimized layout properties* that ensure the chart will be readable and well-proportioned, avoiding overcrowded, distorted, or excessively sparse layouts.

Code Generator. Finally, the code generator decides final chart-specific properties and translates the optimized compilation context into an executable specification for a target visualization library (e.g., Vega-Lite, ECharts, or Chart.js). The code generator handles the diverse vocabulary and programming models used by different libraries. The backend is modular and extensible: each library provides its own templates and compilation parameters, allowing new libraries to be added without changing the frontend or optimization stages. The output is a library-native specification ready for rendering.

4 The Flint Language

In this section, we present the design of our intermediate language Flint. Flint represents visualization-relevant data semantics as first-class objects in the specification, so that low-level chart decisions can be inferred systematically from them. Our design follows these two principles:

1. We decouple *what* data means from *how* it is mapped to visual properties. Instead of relying solely on coarse encoding-oriented data types such as quantitative, nominal, temporal, and ordinal, we introduce fine-grained, encoding-agnostic semantic types that model the meaning of data fields directly. This builds a reusable semantic representation that can support multiple visualizations of the same data.
2. We adopt a chart-type-based approach to visualization specification, where the compiler manages layering, composition, and instance-specific parameters using configuration decisions inferred from data semantics. This keeps specifications concise while supporting expressive visualizations and simplifying adaptation across chart instances.

Table 1: Data semantics hierarchy examples.

L1: Domain	L2: Family	L3: Types
Temporal	DateTime	Date | Time | Timestamp | ...
	DateGranule	Year | Quarter | Month | YearMonth | ...
	Duration	Duration
Categorical	Entity	Category | Name
	Coded	Status | Boolean | Direction
	Binned	Range
Measure	Amount	Revenue | Cost | Price | ...
	Proportion	Percentage
	Rating	Score | Rating
	Signed	Profit | Correlation | Delta | ...
	Generic	Quantity | Number | Temperature | ...
Discrete	Rank	Rank
Identifier	ID	ID
Geographic	Coordinate	Latitude | Longitude
	GeoPlace	Country | State | City | ZipCode | ...

4.1 Hierarchical data semantic model

Our goal is to formalize a semantic type system that lets users formally express data meaning that can guide visualization design. Although semantic types have been used in database research for data mining tasks [18, 20], existing definitions are often too coarse for visualization. For example, two columns associated with month may require very different visual treatments if one stores month-of-year codes (1–12) and the other stores year–month timestamps (e.g., 2001-01 to 2010-12). The former is cyclically ordered discrete representation, whereas the latter forms a chronological sequence suits continuous temporal scales. Treating both as the same Temporal or Month type obscures visualization-relevant distinctions. At the same time, overly fine-grained labels such as MonthName, MonthNumber, or UnixTimestamp increase specification complexity without improving visualization intent, making schemas brittle.

Our key insight here is to formalize a semantic type system based on *how data meaning influences visualization behavior*, and to organize these types hierarchically so that each level captures a different class of compilation decisions. We formalize semantic types as a three-level hierarchy: Level 1 determines how raw values should be interpreted and parsed; Level 2 captures scale- and transformation-level semantics, such as aggregation

behavior, zero interpretation, and domain structure; and Level 3 refines type-specific presentation choices, such as formatting, ordering, and other display defaults. This hierarchy follows an inheritance structure: lower-level types inherit the compilation decisions made by their ancestors and then add more specific constraints when needed. For example, a Level 3 type such as `YearMonth` inherits the temporal parsing behavior from its Level 1 `Temporal` domain and temporal scale behavior from its Level 2 family, while refining presentation-level details such as date formatting and tick labeling. Thus, deeper levels do not alter the higher-level interpretation of the data; they specialize it for more reliable visualization generation.

- *Level 1: Semantic Domain.* We first group semantic types based on *how raw data should be parsed*. Our design leverages this level to determine the core parser class (e.g., temporal or numeric) to ensure correctness of data processing. For example, a column with values 202001, 202002, 202003 might represent year–month timestamps, or quantities such as revenue or account identifiers depending on their column names and contexts. If a system relies only on automatic parser tests, it may silently accept revenue as a datetime object and configure it into a temporal axis; they may also turn a numerical field representing sales with missing values into nominal just because the column has values represented as "N/A". With Level 1 semantic type guidance, the field can be parsed and interpreted with greater confidence.
- *Level 2: Semantic Family.* We next group semantic types based on *how data behaves with respect to key encoding properties* like aggregation, zero, and domain shape. For example, despite both revenue and percentage are numeric measures that should be processed by numerical parsers (per L1 decision), they imply different aggregation and domain properties. Revenue is an *Amount*: it is additive, zero denotes the absence of the measured quantity, and its domain is typically open-ended. Percentage, in contrast, is a *Proportion*: it is intensive and often has either a bounded or context-dependent domain. These distinctions affect chart configurations: when they are used as y-axis of a bar chart but color encoding is not provided, an *Amount* field such as revenue should typically default to summation if there are duplicate values, whereas a *Proportion* field such as percentage should default to average. Other example properties determined in L2 are whether including zero in the domain should be enforced, optional or disabled; whether the scale domain should be fixed or data-dependent.
- *Level 3: Semantic Type.* Finally, L3 captures fine-grained semantic types that define presentation decisions, such as diverging, formatting, ordering, and ticks. For example, a Rank field should be displayed on a reversed axis so that rank 1 at the top. Similarly, in a dot plot that compares the longitudinal positions of cities along an east-west axis, a Longitude field should respect its closed semantic domain $[-180, 180]$ rather than being displayed on an arbitrarily padded axis like $[-200, 200]$. Such fine-grained distinctions provide more consistent visual presentation.

In Flint, this hierarchy is implemented as a global semantic type registry containing 6 L1 domains, 15 L2 families, and 44 L3 types (Table 1). The registry operationalizes the inheritance structure described above: each L3 leaf type inherits parser decisions from its L1 domain and encoding properties from its L2 family, then contributes type-specific presentation refinements such as domain shape, divergence, formatting, and ordering. This structure also supports progressive disclosure. LLM agents can use the full L3 registry for precise visualization configuration, while human authors can specify only intuitive L1/L2 types and let the compiler fill in best-effort defaults. Table 2 illustrates how concrete leaf types populate this compilation registry.

```
dataSpec:
  <field>:
    semanticType: ...
    intrinsicDomain: null | [<min>, <max>] | [<v1>, <v2>, ...]
    mid: null | <value>
chartSpec:
  chartType: Scatter | Bar | Grouped Bar | Line | Area | Heatmap |
  ↪ Radar | Pyramid | ...
  encodings:
    <channel>:
      field: <field>
      aggregate: null | sum | average | min | max | unique
      overrides: ...
```

Figure 6: Schema of Flint. dataSpec defines field semantics; chartSpec defines the chart type and simple encodings.

Besides conceptual clarity, the hierarchical design provides direct engineering benefits, especially *reusability* and *extensibility*. For example, instead of building one set of parser for each semantic type, we only need to maintain a core set of parsers for each semantic domain. If a developer wants to introduce a new semantic type FiscalQuarter to support specialized labeling and formatting for financial use cases, they only need to implement the L3 semantics while reusing others based on parent types. In this way, the hierarchy reduces implementation redundancy, keeps the taxonomy maintainable, and allows new semantic types to be added without ad hoc parsing and visualization rules.

Note that semantic types do not replace the role of raw data representations (i.e., the actual input data values) in visualization. Rather, they are metadata of raw values that gives the compiler actionable guidance on how to appropriately interpret data for encodings, chart, and formatting defaults. In the next sections, we show how semantic types are complemented by data-specific semantic attributes, such as intrinsic_domain and mid, and how these specifications are transformed into detailed, semantically aligned chart configurations.

4.2 The language

Building on the data semantics model, Flint defines a visualization specification in two separate sections: (1) a dataSpec that describes the semantic representation of data fields and (2) a chartSpec section that describes the intended visual mapping. Figure 6 illustrates the grammar of this visualization language. Because the data specification captures the semantics of the underlying fields independently of charts, it only needs to be created once and can be reused for charts of the same dataset. In practice, users do not need to author all of this information manually, as much of the data specification can be inferred by an LLM from column names, observed values, and surrounding context [18]. Although data semantics and chart specifications are conceptually separated, they can still be authored together in a single Flint specification to reduce cognitive load during manual authoring. In LLM-powered interactive systems, the dataSpec can be inferred or cached when data is uploaded, making it easy for users to adapt the chartSpec when creating subsequent visualizations.

Data Specification. In dataSpec (Figure 6), in addition to a semantic type, users can provide optional properties such as an intrinsic domain and midpoint to complement a field's semantic description. Semantic type captures the general meaning of a field, but it does not fully determine its intended representation for visualization. For example, a field with semantic type *Rating* may use a 1–5 scale, a 0–100 scale, or an ordered discrete domain such as [strong reject, weak reject, weak accept, strong accept], even if the observed table values cover only part of that range.

Table 2: Illustrative semantic type assignments across the seven abstract properties.

Semantic Type	Parser	Encoding Candidates	Zero Class	Agg. Role	Domain Shape	Diverging	Formatter Ex.
YearMonth	temporal	temporal / ordinal	none	dimension	open	none	`%Y-%m`
Month	temporal	ordinal	none	dimension	cyclic	none	Jan / 1,2,3
Percentage	numeric	quantitative	contextual	intensive	bounded	none	`12.3%`
Revenue	numeric	quantitative	meaningful	additive	open	none	`$12.4K`
Profit	numeric	quantitative	meaningful	signed-additive	open	conditional	`+$2.1K`
Latitude	numeric	geographic	arbitrary	dimension	fixed	none	`37.77`
Status	string	nominal	none	dimension	open	none	Active
Direction	string	ordinal / nominal	none	dimension	cyclic	none	NE

Likewise, accuracy and recall are naturally bounded to $[0, 100]$ (or $[0, 1]$), whereas other percentage-valued fields may not imply the same closed range. Ordered fields such as age groups require an intrinsic domain to preserve the intended order rather than a lexicographic order [<18, >60, 18–29, 30–44, 45–60]. The midpoint property specifies a meaningful reference value, such as 0 for signed scores or a neutral baseline for sentiment, which may differ from the numerical center of the domain (e.g., 32°F for Fahrenheit temperatures). These properties are helpful for visual presentations: intrinsic domains support correct axis ranges and ordering, while midpoints enable centered color scales and diverging encodings. Together, these optional properties complement semantic types to constrain chart configurations.

Chart Specification. The chartSpec component describes the intended visualization. The user first chooses a chartType, which selects a visual template such as Scatter, Line, Bar, or Heatmap, and then specifies an encodings map that binds visual channels to fields or aggregated fields, such as x $\mapsto$ Month, y $\mapsto$ sum(Revenue), or color $\mapsto$ Region.

We adopt a chart type-based specification so that users can express a wide range of visualizations, including radar charts, pyramids, waterfall charts, and candlestick charts, without explicitly constructing them through composition algebra [34]. This design matches Flint's role as an intermediate language centered on semantic intent rather than bespoke visual construction. At the same time, Flint remains expressive through extensible chart templates, allowing it to support new visualization forms without expanding the core language. Unlike traditional static templates, which are often rigid with respect to data types, cardinalities, or formats, these compiler-backed templates adapt to the data and semantics in the specification, automatically resolving chart-specific parameter dependencies to produce high-quality visualizations.

We deliberately keep the chart language simple by omitting low-level view configuration such as axes, scales, and formatting, leaving these decisions to the compiler (Section 5). At the same time, Flint preserves flexibility through optional chart properties and encoding-level overrides, which allow users to steer common chart behaviors without editing low-level generated code directly. For example, users may specify a map center for geographic charts or override the sort direction of the x-axis in a bar chart from ascending to descending. Finally, because the Flint compiler produces library-native code, users who want additional customization can further edit the compiled specification.

Remarks. Flint currently supports static chart specifications and does not expose interactive visualization features, such as brushing, zooming or cross-filtering, as first-class constructs. When these interactions are supported by backend libraries, users who need them can add or modify interactive components in the downstream backend specification generated by Flint. In addition, Flint's current semantic type design assumes a one-to-one mapping between data fields and semantic types. This can be insufficient for fields that admit multiple possible interpretations. We consider first-class support for interaction and ambiguous semantic types as future extensions to Flint's core language.

5 Flint-Compiler

As described in Section 3.2, the compiler maintains a shared compilation context derived from the data and chart specification inputs, and progressively refines this context across three stages: semantics resolution, optimization, and code generation. Each stage resolves a different subset of visualization properties needed for high-quality chart design, namely *encoding properties*, *layout properties*, and *template-specific properties*. Flint treats data as part of the program: the compiler uses data characteristics, such as cardinality and distribution, alongside the specification during optimization. When the input data changes, Flint triggers a synchronous recompilation pass to adaptively recompute the visual layout. Figure 7 provides an overview of this pipeline's first two stages.

5.1 Stage 1: Semantics Resolution

In this stage, the compiler frontend resolves the chart *encoding properties* that capture the meaning of the data, including parsing, encoding type, formatting, scale type, tick constraints, axis reversal, stacking behavior, color semantics, and sort order. It does so by first resolving data ambiguity using both semantic types and raw data representations, and then grounds the decisions into chart-specific encoding properties.

The first step derives a set of *field properties* that characterize each field independently of chart context. This step resolves the many-to-many correspondence between semantic types and raw representations described in Section 4. Guided by the semantic type, the compiler first selects an appropriate parser and then resolves the detailed field properties. Semantic guidance is crucial here, as it allows the compiler to disambiguate these representations and determine reusable semantics, such as whether the field can support temporal or ordinal treatment in later stages. This step also helps the compiler to resolve mismatches between semantic types and physical data information.

Next, the compiler derives *encoding properties* for particular visual channels from field properties by adding chart contexts. For example, in the heatmap of Figure 4, placing YearMonth on the x-axis yields a temporal positional encoding with time formatting and an ordered temporal scale, but with domainPadding=0 to accommodate discrete tiles. The same field may lead to different channel-specific decisions in other contexts; for instance, when used on the color channel of a line chart, YearMonth may be treated as an ordered or categorical progression rather than as a temporal positional axis depending on cardinality. Guided by the semantic type throughout, this step ensures that channel encodings remain semantically aligned, avoiding errors such as treating YearMonth as quantitative values or applying inappropriate aggregation to fields such as accuracy.

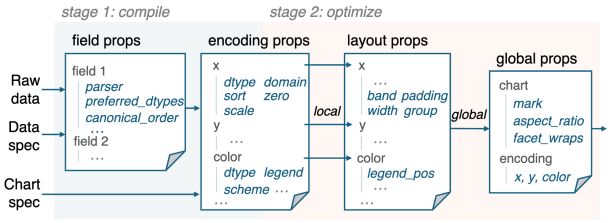

Figure 7: The compilation and optimization workflow. The system compiles raw inputs into intermediate field and encoding properties, then optimizes local and global properties to refine chart sizing.

5.2 Stage 2: Optimization

In the second stage, the compiler optimizes chart layout for the available display space. Users provide a preferred chart size and a maximum allowable area (e.g., preferred: 300×300, max: 800×600), and the compiler resolves *layout properties* so the chart fits the canvas gracefully. It does this by balancing compression of individual visual components with expansion of the canvas within the allowed range. The properties considered include axis span, discrete step size, padding, group width, inter-group spacing, facet panel size, facet row and column counts, and, for radial layouts, radius and angular allocation. Without this stage, default settings can easily produce overflow, excessive density, or distorted layouts (e.g., the heatmap in Figure 3). We formulate this stage as a physics-based optimization process: it first solves local parameters for each dimension, then coordinates these decisions to handle global properties such as aspect ratio.

Local optimizations. The optimizer first determines the size of each layout dimension that affects chart sizing, including the *x*- and *y*-axes, group (e.g., grouped bars), radius (for pie and radial charts), and row/column in faceted layouts. Each dimension acts as a 1D flexible container capable of expanding within a bounded budget. To ensure readability without relying on rigid static breakpoints, we adopt constraint-based approaches [16, 36] to dynamically control element sizes. Depending on the encoding type, the optimizer resolves spatial demand as follows:

- *Discrete layouts.* For items occupying discrete slots (e.g., bars or heatmap cells), each mark acts as a compressible unit constrained by a preferred and minimum readable size. If total spatial demand exceeds the available span, the optimizer resolves the conflict by expanding the layout or compressing items until all minimum size constraints are met.
- *Continuous layouts.* For continuous axes (e.g., scatterplots or line charts), constraints are defined by visual density and allowable mark overlap. When high spatial density causes marks to violate these overlap constraints, the optimizer resolves the conflict by stretching the canvas dimensions to reduce crowding and ensure visual separation.

For nested charts (e.g., grouped bars, strip plots, small multiples), constraints are applied hierarchically. Spatial demand is first resolved for items within each group, followed by a joint optimization of how the groups themselves pack into the broader canvas budget. This ensures individual marks remain readable without violating the overarching limits of the stretched canvas.

Global optimizations. Building on local decisions, the optimizer resolves overarching chart-level properties, primarily aspect ratio (AR). For continuous charts, it blends an AR derived from the "banking to 45°" principle [9, 15, 45, 46] with locally stretched width and height, balancing visual interpretability against density reduction. For banded charts, it shrinks the orthogonal dimension to preserve proportional mark sizes and prevent excessive elongation. Second, it optimizes facet wrapping by balancing overall compactness against subplot legibility. Finally, for non-axis-bound charts (e.g., treemaps, gauge charts, funnel charts), it calculates canvas size directly from visual component counts. These final decisions yield an optimized compilation context ready for code generation. For advanced users who wish to customize the compiler's layout behavior, Flint accepts compilation options such as stretch ratio, canvas size, base chart size, and minimum step size. These parameters are passed to the compiler at runtime and combined with data characteristics and chart design to constrain layout optimization, while preserving the same high-level visualization specification.

5.3 Stage 3: Code generation

Finally, backend code generators translate the optimized, library-agnostic chart properties into library-native specs. They first resolve *chart-specific properties* that depend on the selected chart template. Each backend provides a collection of *dynamic chart templates*, where each template provides a chart skeleton with an instantiation function that consumes the compilation context, finalizes instance-specific properties, and emits a specification in the target library's native vocabulary. For example, a bubble chart template may adjust the pixel size range of bubbles based on data density, while a radar chart template may normalize radial domains using intrinsic field domains. The same optimized layout can therefore be realized differently across libraries: a Vega-Lite template may emit `width: {step: ...}` together with scale padding, whereas an ECharts template may use series-level properties such as `barCategoryGap`, `barGap`, and `barWidth`. This dynamic-template design allows each chart instance to inspect the full compilation context and adapt accordingly, improving flexibility for compositional charts. Overall, the compiler maintains a clear separation between optimization and code generation: chart intent is determined once, and each backend implements it according to the conventions of its target library.

6 Evaluation

We evaluate Flint's authoring expressiveness via a case study, followed by an evaluation of its benefits for LLM agents in NL2VIS tasks. We implement Flint as a JavaScript library supporting three backends (Vega-Lite, ECharts, Chart.js) and 30 chart types, including variants of scatter, bar, line, pie, density, maps and compositional chart. We also integrate the library into the visualization pipeline of the open-source data analysis tool Data Formulator,[1] where AI agents generate flexible Flint specs (rather than Vega-Lite directly) to facilitate exploration.

6.1 Case Study

To demonstrate the effectiveness of Flint in supporting rapid, iterative data exploration, we conducted a case study using a synthetic dataset representing the 2025 global gaming market. The dataset tracks 24 games across 6 platform types (gameType) and 4 regions (region). The exploration process follows three phases, showing how Flint resolves the trade-off between specification simplicity and design quality. The example is shown in Figure 1.

Phase 1: Quick overview. The analyst begins by examining user trends across game types and regions using Vega-Lite. In traditional systems, this usually requires manual data and chart configuration to avoid misleading defaults: the period field could be parsed and formatted as integers, regions could be sorted alphabetically (E,N,S,W), and default subplot widths could become too large for four regions. With Flint, the compiler uses

[1] https://data-formulator.ai

dataSpec, inferred by an LLM agent from the raw data, to configure the chart. With a simple *chartSpec*, the analyst obtains a well-formatted multi-faceted visualization with optimized *x*- and *y*-axis labels, adapted chart widths, and semantically ordered facets, as shown in Figure 1-d.

Phase 2: Exploring different visualizations. The analyst now wants to explore different data field combinations to better understand regional and user growth trends. Traditionally, as the visualization changes, the user must reconfigure the chart to accommodate new encoding properties and data cardinality: (1) when creating a grouped bar chart to compare user numbers across game types by month, the analyst must remap period from a temporal to a discrete axis and reconcile bar widths and axis lengths to avoid skewed chart proportions (Figure 1-e); (2) when creating a waterfall chart to show user flux (Figure 1-f), they must explicitly configure bar starting points to align adjacent bars; (3) when using newUsers in a heatmap (Figure 1-g) to show each game's popularity trend over time, they must apply a diverging color scale with midpoint 0 to reflect "delta" rather than relying on default linear color (e.g., blues) and revise *x*-labels to accommodate longer names. These reconfiguration steps accumulate and slow down analysis. Flint eliminates this burden: the analyst simply changes the chart type and swaps encodings in the chartSpec, and the compiler reuses the dataSpec from the first step to automatically infer all downstream parameters to produce high-quality charts.

Phase 3: Switching backends. In the final phase, the analyst explores the user distribution hierarchy (Region / GameType / Game) using a sunburst chart. Since Vega-Lite does not support interactive sunburst charts, the analyst needs to use ECharts instead. However, the two libraries have very different programming models, so manually porting a chart can be costly. With Flint, the analyst only needs to provide a simple sunburst chart spec (Figure 1-h) and switch the backend, thanks to Flint's library-agnostic design. If the analyst later wants to render any of the earlier Vega-Lite charts in ECharts to take advantage of ECharts' animation features, they can simply switch the backend without modifying the specification itself.

In summary, Flint ensures: (1) *concise*—specifications were on average 85% shorter than native backend code; (2) *robust*—explicit semantic types prevent common design errors such as incorrectly formatted dates, invalid stacking of non-additive measures or diverging colors at the domain midpoint instead of zero; and (3) *portable*—the same intent was seamlessly compiled into Vega-Lite for initial sketches and ECharts for final presentation.

6.2 LLM Chart Generation

To evaluate whether Flint is LLM-friendly, we compare the *Flint* agent (Vega-Lite backend) with a baseline *DirectVL* agent that outputs Vega-Lite directly from datasets and natural language queries. We follow the VisEval [7] protocol, which uses a vision-language model to assess charts generated from natural language queries. We exclude the standard VisEval dataset due to its SQL-focused complexity and limited visual diversity: 84% of 1,150 cases are bar or pie charts (7 types total), 91% use only two visual encodings, and tables are very small (median 4 rows, 86% $\leq$10). Instead, we evaluate on the more complex *TidyTuesday* 2025 datasets.[2] To address the lack of annotated queries, we introduce an automated three-stage LLM pipeline: a *Question Generator* synthesizes diverse analytical queries [31], a *Chart* agent generates the visualizations using either the induced Flint-agent or the DirectVL-agent, and a vision-language *Grader* evaluates the rendered images using the VisEval evaluator. Following VisEval, outputs are scored 0–5 on Relevance, Chart Errors, Clarity, and Design Quality, for a maximum of 20 points. Additional details on prompts and agent design are available in the appendix.

[2] https://github.com/rfordatascience/tidytuesday

Table 3: Comparison of overall scores between Flint and DirectVL agents. Values show the number and percentage of wins/ties.

Model	#Flint wins	#Tie	#VL wins	p
GPT-5.1	129 (41%)	65 (21%)	118 (38%)	0.05
GPT-5-mini	140 (45%)	65 (21%)	106 (34%)	0.001
GPT-4.1	135 (43%)	70 (23%)	106 (34%)	0.004

Observations. The experiment yields 315 questions (5 per dataset) across 10 chart types: Bar (18.7%), Lollipop (15.6%), Grouped Bar (14.6%), Line (14.3%), Boxplot (13.7%), Scatter (11.4%), Heatmap (7.0%), Pie (2.5%), Map (1.3%), and Area (1.0%). The median number of output rows is 37 across agents. Table 3 shows pairwise comparisons: Flint wins more often than DirectVL, with statistical significance on GPT-5-mini (p=0.001) and GPT-4.1 (p=0.004), and borderline significance on GPT-5.1 (p=0.05). The advantage grows as model capability decreases, showing that embedding structured visualization knowledge into the language yields more reliable chart generation than unconstrained Vega-Lite, while remaining simple and flexible. Flint excels on charts requiring semantic coordination, such as mapping fields to grouped encodings, configuring sort orders, or creating multi-layer compositions. Ties or DirectVL wins occur on simpler charts (e.g., single-series bars, basic boxplots) that require minimal semantic reasoning. Results also show that an intermediate representation is especially valuable for models with limited capacity to manage free-form specifications. We show detailed experiment results and examples in the appendix.

7 Discussion and Conclusion

We introduced Flint, an intermediate language that uses a semantic type hierarchy to optimize chart configurations and generate library-native specifications. This abstraction opens new research opportunities:

- *Extending to new visualization languages.* Emerging languages such as Gofish [32] and PiCCL [37] introduce new visualization capabilities but often come with steep learning curves and limited training data. With Flint, researchers can implement a new backend that compiles Flint specifications to the target language. Downstream users can then plug the new language into existing tools without rewriting authoring interfaces, retraining agents, or learning a new low-level grammar from scratch.
- *Visualization recommendation.* Existing engines like Draco [30, 49] and CompassQL [48] rely on physical data representations, which introduce many rules over low-level parameters. With Flint's abstract semantic layer, recommenders can focus purely on high-level design choices (chart types and channel mappings) while delegating downstream adjustments to the compiler. This enables engines to learn generalizable visual idioms with significantly less training data.

Overall, Flint bridges visualization research and engineering practice through concise specifications and extensible backends.

Acknowledgments

We thank supports from the NSFC grants (No. 62502523, No. U2436209), the Beijing Natural Science Foundation (L247027), the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China.

REFERENCES

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

[2] H. K. Bako, X. Liu, L. Battle, and Z. Liu. Understanding how designers find and use data visualization examples. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):1048–1058, 2022. doi: 10.1109/TVCG.2022.3209490 2

[3] J. Bertin. *Semiology of graphics*. University of Wisconsin press, 1983. 3

[4] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185 2

[5] M. Brossier, T. Isenberg, K. Schönborn, J. Unger, M. Romero, J. Björklund et al. State of the art of LLM-enabled interaction with visualization, 2026. doi: 10.48550/arXiv.2601.14943 1, 3

[6] C. Bu, Q. Zhang, Q. Wang, J. Zhang, M. Sedlmair, O. Deussen et al. Sinestream: Improving the readability of streamgraphs by minimizing sine illusion effects. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1634–1643, 2020. doi: 10.1109/TVCG.2020.3030404 3

[7] N. Chen, Y. Zhang, J. Xu, K. Ren, and Y. Yang. Viseval: A benchmark for data visualization in the era of large language models. *IEEE Transactions on Visualization and Computer Graphics*, 31(1):1301–1311, 2024. doi: 10.1109/TVCG.2024.3456320 2, 3, 9

[8] W. S. Cleveland. A model for studying display methods of statistical graphics. *Journal of Computational and Graphical Statistics*, 2(4):323–343, 1993. doi: 10.1080/10618600.1993.10474616 3

[9] W. S. Cleveland. A model for studying display methods of statistical graphics. *Journal of Computational and Graphical Statistics*, 2(4):323–343, 1993. doi: 10.1080/10618600.1993.10474616 8

[10] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984. doi: 10.1080/01621459.1984.10478080 3

[11] V. Dibia. Lida: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 113–126, 2023. doi: 10.18653/v1/2023.acl-demo.11 3

[12] M. Elshehaly, R. Jianu, A. Slingsby, G. Andrienko, and N. Andrienko. Designing for collaboration: Visualization to enable human–LLM analytical partnership. *IEEE Computer Graphics and Applications*, 45(5):107–116, 2025. doi: 10.1109/MCG.2025.3583451 3

[13] 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 28th annual acm symposium on user interface software & technology*, pp. 489–500, 2015. doi: 10.1145/2807442.2807478 3

[14] C. C. Gramazio, K. B. Schloss, and D. H. Laidlaw. The relation between visualization size, grouping, and user performance. *IEEE transactions on visualization and computer graphics*, 20(12):1953–1962, 2014. doi: 10.1109/TVCG.2014.2346983 3

[15] J. Heer and M. Agrawala. Multi-scale banking to 45 degrees. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):701–708, 2006. doi: 10.1109/TVCG.2006.163 8

[16] J. Hoffswell, W. Li, and Z. Liu. Techniques for flexible responsive visualization design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–13, 2020. doi: 10.1145/3313831.3376777 8

[17] M. Hostnik, R. Kurbanov, Y. Sokolov, and A. Trofimov. Vegachat: A robust framework for LLM-based chart generation and assessment, 2026. doi: 10.48550/arXiv.2601.15385 3

[18] M. Hulsebos, K. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska et al. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on knowledge discovery & data mining*, pp. 1500–1508, 2019. doi: 10.1145/3292500.3330993 2, 3, 5, 6

[19] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004. doi: 10.1109/CGO.2004.1281665 2, 4

[20] D. J. L. Lee, D. Tang, K. Agarwal, T. Boonmark, C. Chen, J. Kang et al. Lux: Always-on visualization recommendations for exploratory data science. *CoRR*, abs/2105.00121, 2021. doi: 10.14778/3494124.349415 2, 3, 5

[21] C. Li, D. Zhang, and J. Wang. LLM-assisted labeling function generation for semantic type detection. Presented at the Data-driven AI Workshop, VLDB 2024, 2024. doi: 10.48550/arXiv.2408.16173 3

[22] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang et al. 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 1, 2

[23] 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, 2024. doi: 10.1109/TVCG.2024.3456296 2

[24] Looker. `https://developers.google.com/looker-studio/connector/semantics?hl=zh-cn`. Accessed: 2026-02-26. 3

[25] K. Lu, M. Feng, X. Chen, M. Sedlmair, O. Deussen, D. Lischinski et al. Palettailor: Discriminable colorization for categorical data. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):475–484, 2020. doi: 10.1109/TVCG.2020.3030406 3

[26] J. Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986. doi: 10.1145/22949.22950 3

[27] J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *IEEE transactions on visualization and computer graphics*, 13(6):1137–1144, 2007. doi: 10.1109/TVCG.2007.70594 3

[28] P. Maddigan and T. Susnjak. Chat2vis: Generating data visualizations via natural language using chatgpt, codex and gpt-3 large language models. *Ieee Access*, 11:45181–45193, 2023. doi: 10.1109/ACCESS.2023.3274199 3

[29] 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*, pp. 1–14, 2021. doi: 10.1145/3411764.3445356 2

[30] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe et al. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, 2018. doi: 10.1109/TVCG.2018.2865240 2, 3, 9

[31] J. Nam, J. Yoon, J. Chen, and T. Pfister. DS-STAR: data science agent via iterative planning and verification. *CoRR*, abs/2509.21825, 2025. doi: 10.48550/ARXIV.2509.21825 9

[32] J. Pollock and A. Satyanarayan. Gofish: a grammar of more graphics! *IEEE Transactions on Visualization and Computer Graphics*, 2025. doi: 10.1109/TVCG.2025.3634250 9

[33] S. Sah, R. Mitra, A. Narechania, A. Endert, J. Stasko, and W. Dou. Generating Analytic Specifications for Data Visualization from Natural Language Queries using Large Language Models. Presented at the NLVIZ Workshop, IEEE VIS 2024, 2024. doi: 10.48550/arXiv.2408.13391 3

[34] 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, 2016. doi: 10.1109/TVCG.2016.2599030 1, 2, 7

[35] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2015. doi: 10.1109/TVCG.2015.

2467091 2

[36] S. Schöttler, J. Dykes, J. Wood, U. Hinrichs, and B. Bach. Constraint-based breakpoints for responsive visualization design and development. *IEEE Transactions on Visualization and Computer Graphics*, 2024. doi: 10.1109/TVCG.2024.3410097 8

[37] H. Shi, Y. Wang, J. Chen, C. Wang, and B. Lee. PiCCL: Data-driven composition of bespoke pictorial charts. *IEEE Transactions on Visualization and Computer Graphics*, 2025. doi: 10.1109/TVCG.2025.3634264 9

[38] L. S. Snyder and J. Heer. Divi: Dynamically interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):403–413, 2023. doi: 10.1109/TVCG.2023.3327172 2

[39] L. S. Snyder, C. Wang, and S. M. Drucker. Challenges & opportunities with LLM-assisted visualization retargeting. *2025 IEEE Visualization and Visual Analytics (VIS)*, pp. 141–145, 2025. doi: 10.1109/VIS60296.2025.00034 2

[40] S. Tosi. *Matplotlib for Python developers*, vol. 307. Packt Publishing Birmingham, UK, 2009. 2

[41] P. Vaithilingam, E. L. Glassman, J. P. Inala, and C. Wang. Dynavis: Dynamically synthesized ui widgets for visualization editing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–17, 2024. doi: 10.1145/3613904.3642639 1

[42] P. M. Valero-Mora. ggplot2: elegant graphics for data analysis. *Journal of statistical software*, 35:1–3, 2010. doi: doi/10.5555/1795559 2

[43] S. VanderPlas and H. Hofmann. Signs of the sine illusion—why we need to care. *Journal of Computational and Graphical Statistics*, 24(4):1170–1190, 2015. doi: 10.1080/10618600.2014.951547 3

[44] C. Wang, B. Lee, S. M. Drucker, D. Marshall, and J. Gao. Data formulator 2: Iterative creation of data visualizations, with ai transforming data along the way. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pp. 1–17, 2025. doi: 10.1145/3706598.3713296 1, 3

[45] Y. Wang, Z. Wang, C.-W. Fu, H. Schmauder, O. Deussen, and D. Weiskopf. Image-based aspect ratio selection. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):840–849, 2018. doi: 10.1109/TVCG.2018.2865266 3, 8

[46] Y. Wang, Z. Wang, L. Zhu, J. Zhang, C.-W. Fu, Z. Cheng et al. Is there a robust technique for selecting aspect ratios in line charts? *IEEE Transactions on Visualization and Computer Graphics*, 24(12):3096–3110, 2017. doi: 10.1109/TVCG.2017.2787113 3, 8

[47] L. Wilkinson. *The Grammar of Graphics*. Springer Science & Business Media, 2006. 1, 2

[48] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the workshop on human-in-the-loop data analytics*, pp. 1–6, 2016. doi: 10.1145/2939502.293950 9

[49] J. Yang, P. F. Gyarmati, Z. Zeng, and D. Moritz. Draco 2: An extensible platform to model visualization design. In *2023 IEEE Visualization and Visual Analytics (VIS)*, pp. 166–170. IEEE, 2023. doi: 10.1109/VIS54172.2023.00042 9

[50] D. Zhang, Y. Suhara, J. Li, M. Hulsebos, C. Demiralp, and W.-C. Tan. Sato: Contextual semantic type detection in tables. *Proceedings of the VLDB Endowment*, 13(11). doi: 10.14778/3407790.3407793 3