arXiv:submit/7462035 [cs.HC] 11 Apr 2026

# Raiven: LLM-Based Visualization Authoring via Domain-Specific Language Mediation

Alexandra Irger*, Ella Hugie*, Minghao Guo, Simon Warchol,
Kenneth Moreland, David Pugmire, Wojciech Matusik, and Hanspeter Pfister

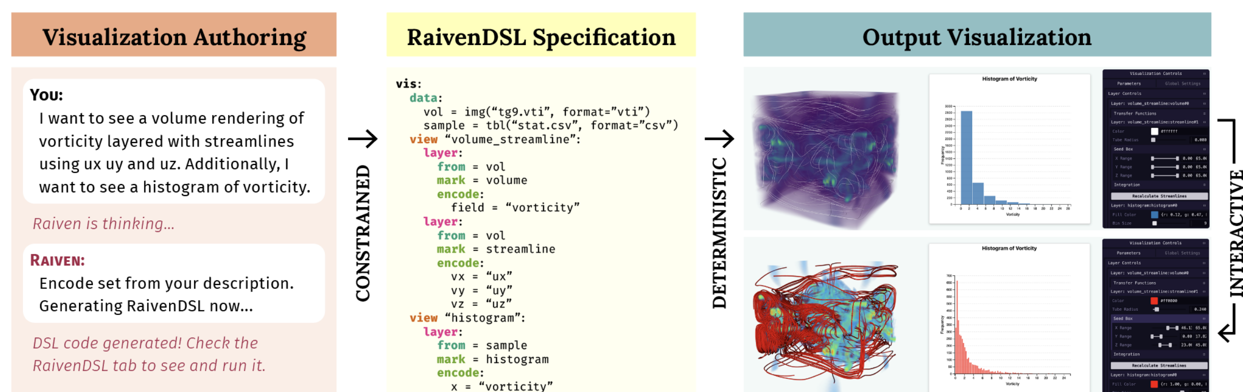



Fig. 1: **Raiven Overview.** The user describes a visualization task in natural language; the LLM translates the request into a schema-constrained RaivenDSL specification, which the compiler deterministically transforms into an interactive visualization and control panel.

**Abstract**—Visualization is central to scientific discovery, yet authoring tools remain split between information and scientific visualization, and expertise in one rarely transfers to the other. Large Language Model (LLM) based systems promise to bridge this gap through natural language, but current approaches generate code non-deterministically, with no guarantee of correctness and no protection against silent data fabrication. We present Raiven, a conversational system that mediates visualization authoring through a formally defined domain-specific language. RaivenDSL unifies scientific and information visualization in a single representation spanning 2D, 3D, and tabular data. The LLM produces a compact RaivenDSL specification under schema-guided constraints, and a deterministic compiler translates it to executable D3 or VTK.js code. Because the LLM operates only on dataset metadata, outputs are deterministic, specifications are verifiable before execution, and data fabrication is impossible by construction. In a 100-task benchmark, Raiven achieves 100% compilation, is up to six times faster and six times cheaper than state-of-the-art LLMs, while improving interaction quality, correctness, and data faithfulness. An expert user study shows that Raiven significantly reduces debugging effort and makes it easier to produce correct visualizations.

**Index Terms**—Visualization Authoring, Natural Language Interfaces, Domain-Specific Languages, Visualization Systems

## 1 Introduction

Despite decades of research and an ever-growing ecosystem of visualization tools, authoring visualizations remains surprisingly hard. This difficulty stems in part from fragmentation, both across domains and across tools. Scientific and information visualization have evolved as entirely separate disciplines, with distinct languages and conventions, meaning knowledge gained in one rarely carries over to the other. Even within a single domain, the landscape is crowded with libraries, frameworks, and applications each targeting a narrow use case. This challenge is compounded by the well-known tradeoff between expressiveness and ease of use, which forces users to choose between flexible but complex low-level libraries and simple but rigid turnkey applications [30]. Together, these challenges leave users with no single system that is both expressive enough to span domains and accessible enough for everyday use.

Natural language interfaces promise to bridge this gap, but current Large Language Model (LLM) based approaches are opaque, costly, non-deterministic, and fragile. The LLM generates visualization code directly, with no guarantee of correctness and no enforcement of visualization best practices. Control over the visualization process lies entirely in large technology companies and their training data. Recent evaluations confirm these limitations: LLM-based generation struggles beyond simple chart types [31, 42] and natural language interfaces remain incapable of supporting research-level visualization tasks [16]. These problems intensify for projects that span both scientific and information visualization, as no current LLM-based frameworks operate across both domains. Worse, because the model often operates directly on raw data values, LLMs can silently fabricate or substitute data, producing plausible-looking visualizations built on invented values.

We argue that domain-specific languages (DSLs) are a solution to these problems. DSLs occupy a middle ground between low-level libraries and simple turnkey applications [30]. A DSL defines a constrained vocabulary with formal semantics, enabling parsing, validation, and compilation before rendering. A DSL inserted between the

- *Alexandra Irger and Ella Hugie are co-first authors.*
- *Alexandra Irger, Ella Hugie and Hanspeter Pfister are with Harvard John A. Paulson School of Engineering and Applied Sciences. Email: airger|ehugie|pfister@g.harvard.edu*
- *Minghao Guo and Wojciech Matusik are with MIT Computer Science & Artificial Intelligence Laboratory.*
- *Simon Warchol is with the Laboratory of Systems Pharmacology, Harvard Medical School. Email: simonwarchol@g.harvard.edu*
- *Kenneth Moreland and David Pugmire are with ORNL. Email: morelandkd|pugmire@ornl.gov*

*Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx*

LLM and the visualization backend separates language interpretation from visualization execution. This separation provides *verifiability*, because a DSL specification can be parsed and validated before compilation, and *flexibility*, because a single specification can target multiple rendering backends, much as a shader language compiles to different GPU APIs. The visualization community itself has recognized the need for cross-domain tools: VAST, InfoVis, and SciVis merged into a single conference in 2021, and multifaceted scientific data increasingly demands both modalities, yet no existing visualization authoring system bridges this divide. Scientific visualization languages do not handle information visualization mark types, and information visualization grammars do not handle volumetric or flow data.

In this paper, we present *Raiven*, a system that mediates natural language visualization authoring through a pipeline that separates language interpretation from visualization execution. The user describes a visualization task in natural language, the LLM translates that request into a compact *RaivenDSL* specification under schema-guided constraints, and a deterministic compiler translates the specification into executable D3 or VTK.js code. Because the LLM operates only on dataset metadata and never accesses raw data values, the architecture preserves data privacy by design, remains independent of context-window limitations that constrain direct code-generation approaches, and makes silent data fabrication impossible. In our evaluation, Raiven achieves 100% compilation, runs up to six times faster, and costs up to six times less than direct code generation with state-of-the-art LLMs, while improving visualization correctness and data faithfulness.

Our contributions are as follows. We present *Raiven*, a conversational visualization authoring system with a type-checked LLM pipeline and a deterministic compiler, connected by *RaivenDSL*, a backend-agnostic DSL spanning scientific and information visualization that provides a compact, verifiable specification structure designed for safe LLM-based visualization generation. We introduce *VMPC* (Visualization Multi-view Prompt Compliance), an evaluation metric for multi-view visualization correctness that scores execution, data faithfulness, cross-view linking, and per-view compliance. We demonstrate Raiven's effectiveness through a novel *benchmark* of 100 visualization tasks spanning scientific, information, and combined workflows, on which Raiven outperforms state-of-the-art large language models across all categories. All code and benchmarks will be released upon acceptance.

## 2 Related Work

Prior work relevant to *Raiven* spans three areas: direct natural language (NL) visualization authoring, domain-specific languages (DSL) for visualization, and systems that mediate natural language through an intermediate structured representation. Table 1 summarizes these systems and their coverage.

### 2.1 Direct Natural Language Authoring

An emerging line of research treats visualization authoring as a direct natural language-to-code generation problem. In information visualization, Chat2VIS [23] generates visualization code directly from natural language, and LIDA [8] wraps code generation in a multi-stage pipeline that summarizes datasets before producing code. FlowSense [44] takes a different approach, mapping utterances to dataflow graph operations. In scientific visualization, ChatVis [25] generates ParaView Python scripts and iteratively repairs them, and Omega [32] generates Python code for bioimage analysis tasks within napari. ParaView-MCP [21] and napari-mcp[1], Omega's successor, replace free-form generation with structured tool invocation over their respective APIs. All of these systems remain tied to specific libraries or execution environments, and direct code generation inherits the fragility of code synthesis, including syntax errors, execution failures, and malformed outputs.

### 2.2 DSL Authoring

We distinguish *domain-specific languages* from *visualization libraries* such as D3 [4], VTK [35], Matplotlib [14], or ObservablePlot[2]. A library provides programmatic access to rendering primitives but imposes no structural constraints on how visualizations are specified. A DSL, by contrast, defines a fixed vocabulary with formal semantics that can be parsed and validated prior to rendering.

[1] https://github.com/royerlab/napari-mcp
[2] https://observablehq.com/plot/

Table 1: Existing NL and DSL based visualization authoring systems. *S* covers Scientific Visualization, *I* covers Information Visualization.

| System | Year | NL | DSL | *S* | *I* |
|---|---|---|---|---|---|
| FlowSense [44] | 2020 | ● | | | ● |
| Chat2Vis [23] | 2023 | ● | | | ● |
| LIDA [8] | 2023 | ● | | | ● |
| ChatVis [25] | 2024 | ● | | ● | |
| Omega [32] | 2024 | ● | | ● | |
| ParaView-MCP [21] | 2025 | ● | | ● | |
| Diderot [6] | 2012 | | Diderot | ● | |
| Vivaldi [7] | 2014 | | Vivaldi | ● | |
| ViSlang [30] | 2014 | | ● | ● | |
| Vega-Lite [33] | 2017 | | Vega-Lite | | ● |
| Shih et al. [36] | 2019 | | ● | ● | |
| DXR [37] | 2019 | | ● | | ● |
| Scholz [34] | 2021 | | ● | ● | ● |
| Harth et al. [10] | 2023 | | ● | | ● |
| GoFish [29] | 2026 | | GoFish | | ● |
| Gosling [17] | 2026 | | Gosling | | ● |
| NL4DV [28] | 2021 | ● | Vega-Lite | | ● |
| NMT2Vis [22] | 2022 | ● | VegaZero | | ● |
| NL2Viz [43] | 2022 | ● | ● | | ● |
| FlowNL [13] | 2023 | ● | ● | ● | |
| ChatModelling [15] | 2024 | ● | ● | ● | |
| YAC [19] | 2025 | ● | ● | | ● |
| ChartGPT [38] | 2025 | ● | ● | | ● |
| NLI4VolVis [1] | 2026 | ● | ● | ● | |
| VegaChat [12] | 2026 | ● | Vega-Lite | | ● |
| **Raiven** | 2026 | ● | **RaivenDSL** | ● | ● |

The landscape of DSLs for scientific visualization is sparse. ViSlang [30] composes procedural, declarative, and functional sublanguages for GPU-based scientific visualization. Shih et al. [36] introduce a declarative JSON grammar for volume visualization pipelines. Diderot [6] is a compiled language for continuous field visualization, and Vivaldi [7] provides a procedural DSL for distributed volume processing. None of these target information visualization mark types.

DSL development is much more mature on the information visualization side. Vega-Lite [33], building on the Grammar of Graphics [41], is the dominant declarative grammar. GoFish [29] formalizes Gestalt grouping as composable operators, Gosling [17] adapts the declarative model to genomic data, DXR [37] targets immersive environments, and Harth et al. [10] coordinate heterogeneous web views through an interaction grammar. None of these support volumetric, flow, or spatially continuous scientific data. Scholz [34] spans both domains with a JSON-based DSL, but renders 3D content through Three.js, a general-purpose graphics library rather than a visualization backend. Draco [27] encodes design knowledge as constraints over Vega-Lite and uses answer-set programming to rank valid designs; Dziban [20] extends this framework with anchored recommendations to support incremental and iterative visualization design. Both, however, operate as recommendation engines rather than generation targets.

Across both communities, visualization DSLs have been designed for direct human authoring or programmatic construction, not as generation targets for language models.

### 2.3 Natural Language Authoring via DSLs

The line of research most similar to our approach combines natural language authoring with a DSL or other intermediate structured representation. On the information visualization side, NL4DV [28] (NLP-based) and VegaChat [12] map natural language to Vega-Lite specifications. Vega-Zero [22] introduces a seq2seq-friendly linearization of Vega-Lite, and ChartGPT [38] uses a thin intermediate chart representation that resolves to an established grammar. NL2Viz [43] and YAC [19] intro-

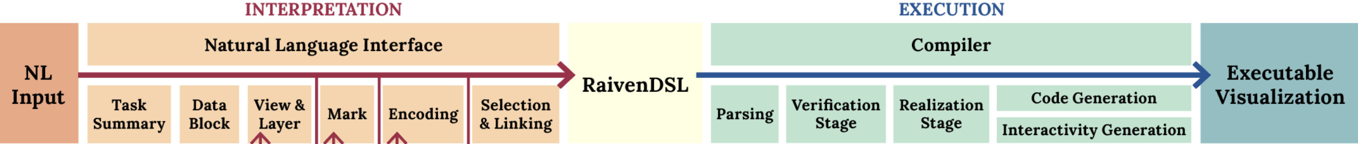



Fig. 2: **Raiven pipeline.** Left: in the *interpretation* phase, schema-mediated generation translates natural language into RaivenDSL through a sequence of validated stages. Right: in the *execution* phase, the compiler parses, validates, resolves backend-specific defaults, and generates executable code and interactive controls. The user can interact with the system at three points: natural language input (Describe), direct DSL editing (Edit), and interactive controls on the rendered visualization (Explore).

duce purpose-built intermediate languages, but both remain narrowly scoped to specific chart types or biomedical data exploration.

On the scientific visualization side, FlowNL [13] translates natural language into a declarative intermediate language for flow visualization. Chat Modeling [15] mediates natural language through a JSON-based intermediate format for procedural modeling of biological structures. NLI4VolVis [1] uses function-calling agents to invoke volume visualization commands iteratively. All three are tightly scoped to particular tasks and domains.

No existing system combines natural-language authoring, a purpose-built backend-agnostic intermediate language, and support for both scientific and information visualization within a single typed language.

## 3 System Overview

Raiven mediates natural language visualization authoring through a structured pipeline that separates language interpretation from visualization execution (Figure 2). The pipeline has two phases: *interpretation*, which translates a natural language request into RaivenDSL, and *execution*, which compiles that specification into an interactive browser-based visualization.

In the interpretation phase (Section 5), a user submits a visualization request through the chat interface. Rather than generating backend code directly, the LLM incrementally constructs a session schema, a structured representation that persists across conversational turns and records the evolving specification. The schema captures the task summary, typed data sources, and view structure, including layers, marks, encodings, styles, selections, and linked interactions. It is populated through a sequence of narrowly scoped prompts, each addressing a specific subproblem and followed by validation. When ambiguity remains, the system asks for clarification before proceeding. Once the schema is complete, a final prompt translates it into RaivenDSL.

The resulting RaivenDSL (Section 4) is returned directly to the user, who can inspect and edit the specification before compilation, render it immediately, or continue refining it through natural language. This transparency is deliberate: the DSL serves as a legible and verifiable mediator between user intent and compiler input.

In the execution phase (Section 6), the Raiven compiler parses the specification, resolves types, and assigns each view to an appropriate rendering backend. Raiven currently targets two rendering backends—D3 for information visualization and VTK.js for scientific visualization—both executed in the browser. This pairing was chosen deliberately: both are state-of-the-art libraries in their respective fields, together spanning both visualization domains. The compiler then assembles each view deterministically, resolving axes, color palettes, camera parameters, interaction defaults, and generated controls. Because this translation occurs per view within a single program, one RaivenDSL specification can produce coordinated multi-view visualizations that span both scientific and information visualization.

## 4 RaivenDSL

RaivenDSL specifies visualizations at the level of intent rather than execution. The user describes *what* they want to see without concern for which backend renders it. The language restricts itself to mark types with established semantics across visualization practice, trading fine-grained control for lower specification burden, stronger validation, and backend portability. Although designed as a generation target for the natural language interface, RaivenDSL is also a human-authored language: users may write or edit specifications directly and compile them without going through the natural language pipeline.

We introduce the language through a concrete example before describing its design principles.

### 4.1 A Running Example

Consider a neuroscientist exploring a CT scan of a human head. She wants a 3D volume rendering containing a 2D axial slice. In RaivenDSL, this visualization is a short program:

```
vis:
  data:
    vol = img("head.vti", format="vti")
  view "volume_slice":
    layer:
      from = vol
      mark = volume
    layer:
      from = vol
      mark = slice
      style:
        axes = ["XY"]
```

Even this minimal program illustrates several design decisions. The **data** source carries a typed constructor: `img()` tells the compiler this is volumetric image data, not a table or network. The program declares a single **view**, a named container that holds one complete chart or 3D scene, containing two **layers**, a volume rendering and a slice. Each layer adds a distinct visual representation to the view. Here, the first layer renders the volume and the second adds a slice plane, both drawn in the same 3D scene because they share the same view. Each layer specifies a **mark**, the primitive visual type to render (e.g., `volume`, `slice`, `scatter`, `bar`), and optionally an **encoding** that binds data variables to the mark's visual channels (e.g., mapping columns to axes and colors). The user specifies what to show; the compiler handles everything else, from axis construction to default color mappings.

Now suppose the neuroscientist wants a second linked view showing only the slice. The slice position gets synchronized with the first view and a shared color transfer function ensures that adjusting the mapping in one view updates the other (Figure 3).

```
  view "slice_xy":
    link(slice="xy_link", axes=["XY"],
         views=["volume_slice", "slice_xy"])
    link(tf="head.vti_shared",
         views=["volume_slice", "slice_xy"])
    layer:
      from = vol
      mark = slice
      style:
        axes = ["XY"]
```

The two **link** declarations introduce cross-view coordination. The first synchronizes slice position: dragging the slice plane in one view moves it in the other. The second shares a color mapping between the slice layers so that adjustments propagate across views. The DSL expresses the *intent* of the coordination, and the compiler generates the implementation, giving our neuroscientist a fully coordinated multi-view CT visualization in twenty-two lines of declarative code. The full RaivenDSL and corresponding visualization follow:

```
vis:
  data:
    vol = img("head.vti", format="vti")
  view "volume_slice":
    link(slice="xy_link", axes=["XY"],
         views=["volume_slice", "slice_xy"])
    link(tf="head.vti_shared",
         views=["volume_slice", "slice_xy"])
    layer: ...                      // layers as in example 1
  view "slice_xy": ...                  // view in example 2
```

Fig. 3: Output of the running example: a volume rendering and synchronized slice view of a CT head scan.

### 4.2 Design Principles

Four principles guided the design of RaivenDSL: typed data, separation of semantics from style, declarative coordination, and early rejection of invalid programs.

**Typed data for backend routing.** A central challenge in unifying scientific and information visualization is that the two domains rely on fundamentally different rendering architectures. Volume data compiles to VTK.js; tabular data compiles to D3. Rather than forcing the user to choose a backend, we encode the distinction in the data declaration itself. Each data source carries a typed constructor that identifies its modality (image, table, network, or geospatial). The compiler uses this data type to constrain which marks are valid using this data. Together with the mark type, it determines the appropriate rendering backend for each view and validates that the specified encodings are compatible with the data before generating code. The running example illustrates this: the `img()` constructor restricts layers to marks such as `volume` and `slice`, which the compiler routes to VTK.js, while a `tbl()` constructor restricts layers to marks that compile to D3.

**Separating semantics from style.** A key design decision is the separation of encoding from styling. Encodings define the semantic content of the visualization: which data variables map to which visual channels. Style properties (color schemes, opacity, transfer function parameters) are optional and resolved by the compiler when omitted. This separation matters for two reasons. First, it reduces specification burden: the user communicates what the visualization means, and the compiler resolves how it looks. Second, it makes the DSL a better generation target for the LLM, because the model only needs to produce the semantically meaningful part of the specification. Stylistic defaults that would be unreliable and unpredictable if inferred by an LLM become deterministic when encoded in the compiler.

**Declarative coordination.** Cross-view coordination is one of the most complex aspects of multi-view visualization, typically requiring manual event wiring, shared state management, and backend-specific callback code. We chose to make coordination a first-class language construct: the user declares *what* should be shared, and the compiler generates the synchronization logic. RaivenDSL currently supports five coordination types: brush-and-filter selections, point selection with cross-view highlighting, shared color mappings, synchronized color transfer functions, and linked slice indices. We deliberately made coordination explicit rather than inferred, because linked behaviors vary too widely across visualization tasks for any default to be reliable. A brush that filters in one context should highlight in another; a shared slice index is appropriate for medical imaging but not for a statistical dashboard. These decisions belong to the user, not the compiler. Currently, declared links only operate within a single backend (i.e., linking D3 views with D3 views, or VTK.js views with VTK.js views); cross-backend linking is a planned extension.

Table 2: Design-space coverage of RaivenDSL against the Tory & Möller [39] taxonomy. All types and techniques are from the original classification. ● = supported, ○ = not supported. Unsupported techniques in gray.

| Dim | Type | DSL | Techniques |
|---|---|---|---|
| **Continuous (given spatial layout)** | | | |
| 1D | Scalar | ● | Line graph |
| 2D | Scalar | ● | Colour map, isolines |
| 3D | Scalar | ● | Volume rendering, isosurfaces |
| 3D | Tensor | ○ | Tensor ellipsoids |
| 2D–3D | Vector | ● | LIC, Particle traces, glyphs |
| 1D–3D | Multivar. | ● | Combine scalar, vector & tensor |
| nD | — | ● | Multiple 1D, 2D, or 3D views |
| **Discrete (chosen / given layout)** | | | |
| 2D | Values | ● | Scatter plot, bar chart |
| 2D | Graphs | ● | Node-link diagrams (2D) |
| 3D | Values | ○ | 3D scatter plot, 3D bar chart |
| 3D | Graphs | ○ | Node-link (3D) |
| nD | Values | ● | Charts + colour, multiple views, glyphs, parallel coordinates |
| nD | Trees | ○ | Hierarchical graphs, Space-filling Mosaics |

**Structural validation at parse time.** Because RaivenDSL is designed as a generation target for an LLM, catching errors early is essential. The grammar encodes structural constraints directly: a layer must declare a data source and a mark, selections must be defined before they can be linked, and backend-incompatible layers cannot share a view. Invalid specifications are rejected at parse time rather than discovered at runtime. This is a deliberate tradeoff against flexibility: the language does not permit arbitrary code or unconstrained composition. The benefit is that every syntactically valid RaivenDSL program is guaranteed to compile, which eliminates an entire class of errors that plague direct code generation.

The language supports two types of syntax: a brace-based syntax aligned with existing visualization grammars [26] and an indentation-based syntax familiar to Python users. Both parse to the same abstract syntax tree. The full grammar is provided in Appendix A.1.

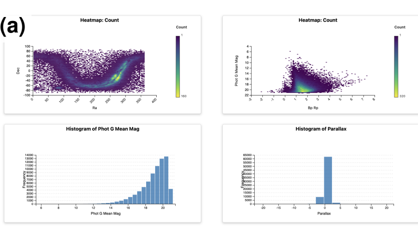
### 4.3 Expressiveness

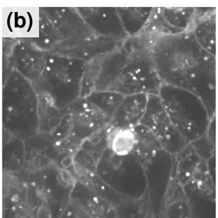
RaivenDSL targets mark types that are well-established and broadly applicable across visualization practice, spanning both scientific visualization (volume rendering, isosurfaces, slices, streamlines, line integral convolution) and information visualization (statistical, distributional, relational, network, compositional, and geospatial marks). A complete list of supported marks with their corresponding encodings, style parameters, and generated controls is provided in Appendix A.3.

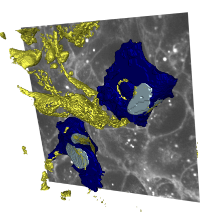
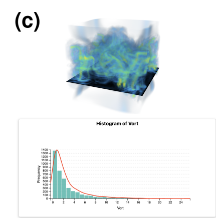
The key observation is that both domains share the same specification structure: a data source, a mark type, mark-dependent channel-to-field encodings, and optional styling. A volume rendering and a scatter plot differ in their backend realization, but at the specification level they are both a mark applied to data, typically through an encoding. RaivenDSL capitalizes on this structural similarity to provide a single language that spans both domains.

We evaluate RaivenDSL's coverage against the Tory and Möller taxonomy [39], which classifies visualization techniques by whether their spatial layout is continuous or discrete and whether it is given by the data or chosen by the designer. This taxonomy is uniquely suited to our evaluation because it is the only high-level classification spanning both scientific and information visualization. Table 2 summarizes the results. RaivenDSL covers 10 of 14 categories, including all scalar, vector, and multivariate continuous settings and all 2D discrete settings.

As a practical demonstration of this coverage, Figure 4 shows visualizations authored in RaivenDSL: recreations of key views from





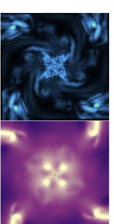


Fig. 4: Visualization types from published systems, recreated in RaivenDSL. (a) Mosaic [11]: linked heatmaps and histograms over a subset of the Gaia catalog [40]. (b) Neuroglancer [24] view recreated from [3]: 2D slice with 3D segmentations (Allen Cell Data [2]). (c) Polyphorm [9] view recreated from [18]: volume rendering with linked slices and charts.

Mosaic [11], Neuroglancer [24], and Polyphorm [9]. We recreate the visualization types these systems produce, not their full functionality. The four unsupported categories in Table 2 reflect deliberate scoping decisions. Tensors and hierarchical trees are left to future work. The discrete 3D categories (3D scatter plots, 3D node-link diagrams) are excluded because occlusion and depth ambiguity make them unreliable without careful view-dependent rendering, which conflicts with the language's goal of deterministic, intent-level specification.

## 5 Natural Language Interface

Rather than generating visualization code directly from a single prompt, Raiven first populates a structured intermediate schema that captures the user's intent, then compiles that schema into RaivenDSL only after validating it. The system is implemented as a FastAPI server that manages session state, orchestrates the multi-prompt pipeline, and communicates with the OpenAI API (ChatGPT 5.2) for all generation steps, exposing a browser-based chat interface to the user. We illustrate with a brief example before describing the mechanism.

### 5.1 Revisiting the Running Example

Returning to the running example from Sec 4.1: the neuroscientist uploads a VTI file of a CT head scan and types: "Show me a volume rendering of the head data with a linked axial slice and a shared color transfer function." Raiven does not generate code directly from this sentence. Instead, the system proceeds through a sequence of targeted nodes, each populating one component of the session schema and validating it before the next step begins:

1. **Task summary.** The LLM extracts a high-level goal: *volume rendering with linked axial slice of a 3D scalar dataset and shared transfer function.*
2. **Data block.** Raiven has already parsed the uploaded file. The schema records the source name, file path, format (`vti`), and variable descriptors (scalar field names, spatial dimensions), of which Raiven derives from the file header. The LLM never sees the raw voxel values.
3. **View and layer.** The LLM proposes two layers in a shared view: `volume` and `slice`. Raiven checks that at least one view with one layer exists, that every layer is assigned to a dataset, and that all layers in a view share a compatible backend.
4. **Mark type.** For each layer the LLM selects a primitive mark type: `volume` for the three-dimensional rendering and `slice` for the axial cut. Raiven normalizes the mark name and checks that it appears in the backend capability table.
5. **Encoding and Style.** The LLM updates the volumes field to the `vti`'s scalar variable and sets the slice axis to `XY`. Raiven checks that the mark and data types are compatible, that every referenced variable exists, and that all required channels are filled.
6. **Selection and Linking.** The LLM proposes a shared transfer function between the two views. Raiven records the link but enforces no additional validation.

At each step, Raiven validates the schema before the pipeline advances. If the user had omitted the slice orientation, the system would have asked for it at the "Encoding and Style" stage rather than guessing. Only after all fields are populated and checked does a final prompt translate the schema into the RaivenDSL program shown in Section 4.1. The result is the same specification the user could have written by hand, but arrived at through conversation.

### 5.2 Session Schema

The example above illustrates the role of the session schema: a structured intermediate object that persists across conversational turns and tracks the accumulated specification. The schema records a task summary, a data block keyed by source name (including each source's type, path, and variable descriptors), and a views block that enumerates views, layers, marks, encodings, and styles. It tracks selections and linking definitions for interactive multi-view specifications.

A core challenge in natural-language visualization authoring is that users routinely underspecify their requests. Traditional LLM-based approaches respond to ambiguity by inferring missing details, which can produce hallucinated variable names, fabricated data sources, or visualizations that do not reflect user intent. Raiven addresses this by requiring a complete and validated specification before generating RaivenDSL. Users describe their intent through an interactive chat interface, refining and extending their request across multiple turns until the system has gathered a full specification. Rather than generating RaivenDSL from a single prompt, the LLM populates the schema incrementally through a sequence of targeted prompts, each responsible for a narrowly defined subproblem. The schema serves as a flattened intermediate representation of the eventual DSL, allowing the system to track what has been specified and what remains outstanding before compilation. Appendix B describes the full schema structure and documents what triggers re-prompting at each pipeline stage, including the exact clarification messages presented to the user.

### 5.3 Validation and Completion Checking

The system advances through a series of sequential nodes that check each schema component before proceeding. When a session begins, Raiven immediately parses uploaded data files into the data block, grounding all subsequent generation in the actual dataset structure. Raiven exposes the LLM only to dataset metadata such as variable names and data types rather than raw values.

At each stage, Raiven validates proposed schema updates before committing them. Raiven checks mark types against its supported set, verifies referenced variables against the dataset, and checks encodings for compatibility with their data types. Crucially, the system will not proceed if required schema elements are missing or underspecified. If the user has not provided a mark type, the system asks for one. If encoding fields are absent or ambiguous, the system returns a message listing exactly what is still needed and waits for the user to respond. This continues until every required component is present and valid.

Only once the schema is fully populated and all checks pass does the system issue a final prompt that translates the schema into RaivenDSL. Because Raiven defers RaivenDSL generation until the schema is complete and validated, the translation step operates on well-formed input and avoids the hallucination and non-determinism that arise when LLMs generate visualization code directly from underspecified requests.

## 6 Compiler

The Raiven compiler translates RaivenDSL specifications into executable visualization code deterministically. Its design reflects a central architectural choice: all visualization logic, from default styling to interactive controls, is resolved by the compiler rather than the LLM. This division means that the language captures what the user specified, and the compiler fills in what the user did not, with deterministic, reproducible results.

### 6.1 Two-Stage Design

The compiler operates in two stages: a *verification stage* that validates the specification independently of any rendering backend, and a *realization stage* that assigns a rendering backend to each view, resolves defaults, and prepares the specification for code generation. We separate these stages so that the compiler catches structural errors (unknown data sources, invalid mark-encoding combinations, unresolved selection references) before making any backend-specific decisions. This ensures that validation logic and backend logic remain independent: adding a new backend requires only a new realization path, not changes to the validator.

Because a single RaivenDSL program can contain views that require different backends, for example volumetric views compiled to VTK.js alongside chart views compiled to D3, the realization stage assigns a backend independently to each view based on its mark types. This per-view routing is what makes cross-domain bridging work at the execution level: a single RaivenDSL program compiles to multiple backends, and the generated views are laid out together in the browser. We chose per-view rather than per-layer backend assignment because layers within a view must share a coordinate system and rendering context, a constraint that would be difficult to enforce across backends.

A validation pipeline runs at both stages. At the verification stage, the compiler checks for structural well-formedness. At the realization stage, it checks for backend compatibility. Because the specification has formal structure, the compiler can validate it before generating any code, which is not possible when an LLM emits arbitrary visualization-library programs.

### 6.2 Code Generation

Code generation translates the realized specification into executable D3 or VTK.js code. For each view, the compiler produces the complete rendering logic required by the target backend. The user specifies a mark type and, where applicable, an encoding; the compiler produces the full implementation deterministically. The volume mark from the running example (Section 4.1) illustrates the scale of this expansion: two lines of RaivenDSL (a data source and a mark type) expand into a complete VTK.js pipeline that loads the data, constructs a transfer function, configures a volume mapper, and registers the actor with the renderer.

The complexity of the generated code varies substantially across mark types, and this asymmetry is a key part of the compiler's value. A D3 scatter plot requires scales, axes, gridlines, margins, and SVG structure. A scientific-visualization mark such as streamlines requires a multi-step pipeline: resolving velocity components from the source data, generating seed points, running numerical integration, and registering the result in the 3D renderer. In both cases, the user writes the same kind of specification, a mark type and an encoding, and the compiler absorbs the complexity that would otherwise fall on the user or the LLM.

The compiler is also responsible for defaults. When a specification leaves style properties unset, the compiler supplies deterministic values derived from the mark type and data schema: color palettes, bin sizes, iso-values, transfer function parameters. We encode defaults in the compiler rather than rely on the LLM to infer them because LLM-inferred styling varies across runs and cannot guarantee consistency. Users can adjust most defaults through the generated controls described below, so they can override defaults without editing the specification.

### 6.3 Interactivity Generation

Rather than requiring users to specify all interaction in RaivenDSL, the compiler generates interactive controls automatically. The alternative, producing static output with no controls, would force users to re-edit and recompile the specification for every parameter change. For exploratory tasks, this would make the system impractical. We therefore chose to have the compiler generate controls directly from the specification, so that users can adjust parameters in the UI without returning to RaivenDSL or the natural language interface. Raiven generates interactive controls at three levels, each reflecting a different design rationale.

**Implicit controls.** The compiler provides standard navigation (rotation, zoom, pan) for all 3D views. We include these by default rather than require them in RaivenDSL, because free camera movement is necessary to explore any spatial visualization. Requiring users to declare navigation would add specification overhead without expressive benefit.

**Declared controls.** The compiler translates selections and links specified in the DSL into interactive behaviors. When a specification declares a brush selection linked across two views, the compiler generates the event listeners, data filtering logic, and visual highlighting needed to synchronize them. For example, a brush in a scatter plot that highlights corresponding points in a histogram requires the compiler to generate listener code in the source view, a filtering function that maps the brushed range to matching data points, and a highlight update in the target view. Shared transfer functions and synchronized slice indices work the same way: the compiler reads the declared link, determines which parameters must be shared, and generates the synchronization code. We chose to compile these from explicit declarations rather than infer them, consistent with the language design principle described in Section 4.2.

**Mark-specific controls.** For each mark, the compiler determines which parameters are user-adjustable and generates the corresponding interface elements (Figure 1). A volume mark produces a transfer function editor; a force-directed graph produces controls for simulation strength and link distance; a streamline mark produces controls for seed region bounds, tube radius, and integration step size. We chose to generate these automatically because the mark type and the resolved specification fully determine the set of meaningful parameters, requiring no additional input from the user or the LLM.

Not all controls behave the same way at runtime. Changing a color, opacity value, tube radius, isosurface threshold, or force-graph simulation parameter takes effect immediately. Changing a seed region or integration step size for streamlines triggers a full numerical re-integration. Raiven applies these expensive controls only on user confirmation because the underlying computation is too costly to run on every slider movement. Appendix A.3 provides a complete list of per-mark-type controls.

Raiven implements the compiler in TypeScript; it runs entirely in the browser. ANTLR4 parses the brace-based syntax; the indentation-based syntax uses a separate parser that produces the same abstract syntax tree. The compiler implements validation, realization, and code generation as sequential pipeline stages. Generated D3 and VTK.js code executes in-browser, arranging views in a responsive grid and rendering interactive controls through Tweakpane. Compilation and rendering require no server-side computation: once Raiven generates RaivenDSL, the entire execution pipeline is client-side. We chose this split so that the only server dependency is the LLM API call; a user who writes RaivenDSL by hand can compile and render without any server at all. Vite handles self-contained HTML export, bundling all generated code, library dependencies, and data references in a single file. RaivenDSL supports four data formats: VTI (2D and 3D image data), CSV, JSON, and GeoJSON.

## 7 Benchmark Evaluation

To evaluate Raiven's end-to-end performance, we designed a benchmark of 100 visualization prompts spanning the full RaivenDSL design space. Each prompt is run once through Raiven and 3 alternate general-purpose LLMs, each in a single-shot setting, producing a self-contained

HTML file scored for correctness and efficiency[3]. No manual repair, iterative prompting, or task-specific scaffolding is permitted. Outputs that fail to produce executable HTML are treated as execution failures.

**Prompts.** The 100 prompts are grouped into three categories: InfoVis ($I$, 40 prompts), SciVis ($S$, 30 prompts), and Combined ($C$, 30 prompts), where Combined prompts reference both spatial and tabular data. The higher InfoVis count reflects the larger number of mark types in that category. Prompts vary in complexity from single-view, single-mark specifications to multi-view dashboards with cross-view linking. Each prompt is fully specified with respect to its accompanying dataset: given the prompt text and the dataset, the intended visualization is unambiguous, with no missing parameters or underspecified intent. All prompts are listed in Appendix D.6.

**Datasets.** The benchmark uses real-world datasets across all three categories. SciVis prompts use 2D and 3D image data (VTI); InfoVis prompts use tabular (CSV), network (JSON), and geospatial (GeoJSON) data; Combined prompts reference both. CSV and JSON files are embedded in full in the model prompt, reflecting how a user would typically provide tabular data to an LLM; files exceeding 100K tokens are truncated, though this affects only a small subset of CSVs. VTI and GeoJSON files are too large to embed and are represented by their metadata headers only. Both are provided by URL, to avoid downsampling artifacts observed in early experiments. All files are also provided as URLs so that generated code can reference them directly at runtime. Full dataset descriptions, sources, and licenses are in Appendix D.1.

**Baselines.** We compare Raiven against three general-purpose LLMs: ChatGPT 5.4, Claude Opus 4.6, and Gemini 3.1 Pro. All models are called with temperature 0. Raiven itself uses ChatGPT 5.2 for all generation steps (Section 5), a less capable model than the ChatGPT 5.4 baseline. Each baseline receives the same prompt and data as described above. Raiven, by contrast, receives only dataset metadata (variable names, data types, and source paths) and never sees raw data values, as described in Section 5. All baselines share a system prompt (Appendix D.3) instructing the model to produce a self-contained HTML file and to load data at runtime via the provided URLs. No additional scaffolding, examples, or few-shot demonstrations are provided. This is a system-level comparison: it evaluates the end-to-end authoring pipeline each approach provides, not model capability in isolation. Per-model API configurations are documented in Appendix D.2.

### 7.1 Efficiency

We assess efficiency through generation time, token usage, and cost. Generation time is measured as wall-clock time from user request to produced visualization, reflecting the latency experienced during usage. Token usage is the total prompt and completion tokens consumed per task. Cost reflects the token-based API pricing applied to each task. Results are reported in Table 3.

### 7.2 Visualization Correctness: VMPC

Existing evaluation metrics for LLM-generated visualizations, such as VisEval [5] and VegaChat [12], are designed for single-chart tasks and do not address multi-view completeness, hallucinated data, or execution reliability. AstroVisBench [16] covers scientific visualization but uses coarse error categories insufficient for partially correct multi-view outputs. We introduce VMPC (Visualization Multi-View Prompt Compliance), a metric that extends these prior approaches to interactive multi-view visualization across both domains.

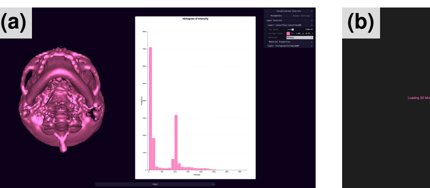
For each required view $v$, VMPC evaluates five binary criteria: view presence $V_v$ (whether the chart outline or container is rendered), mark correctness $M_v$ (whether the correct mark type is used), encoding correctness $E_v$ (whether the correct variables, color mappings, and channel assignments appear), data hallucination $H_v$ (whether the system used the original input data rather than fabricating or substituting values), and cross-view linking $L_v$ (whether requested coordination between views is present and functional). For single-view prompts or prompts that do not request linking, $L_v$ is set to 1. All criteria are binary: a criterion is either met or it is not. We chose binary scoring over partial-credit scales to eliminate subjectivity in grading. A global execution gate $X$ ensures that a system receives a score of zero if it fails to produce a runnable visualization. These per-view terms are averaged over all 5 criteria across the $N$ views specified in the prompt:

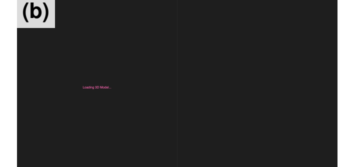
$$\mathrm{VMPC} = X \cdot \frac{1}{5N} \sum_{v=1}^{N} (V_v + M_v + E_v + H_v + L_v) \in [0,1] \quad (1)$$

Figure 5 illustrates VMPC scoring on a two-view prompt. Raiven produces both views correctly (VMPC=1.00); Gemini renders both view containers but with missing marks and encodings (VMPC=0.53). Because VMPC awards partial credit for incomplete outputs, the baseline scores reported in this paper represent generous lower bounds; the practical gap from a user's perspective is larger than the numbers suggest. Full VMPC grading example breakdowns are provided in Appendix D.5.

Table 3: Benchmark results across InfoVis ($I$), SciVis ($S$), and Combined ($C$) categories reporting mean ($\mu$). Cell shading interpolates linearly per row from best to worst.

| | | Gemini 3.1 Pro | ChatGPT 5.4 | Claude Opus 4.6 | Raiven |
|---|---|---|---|---|---|
| Compile (%) | $I$ | 1.0000 | 1.0000 | 0.9580 | 1.0000 |
| | $S$ | 0.6110 | 0.6670 | 0.5110 | 1.0000 |
| | $C$ | 0.9440 | 0.8780 | 0.8780 | 1.0000 |
| | $\Sigma$ | 0.8670 | 0.8630 | 0.8000 | 1.0000 |
| Time in minutes ($\mu$) | $I$ | 1.432 | 0.857 | 0.890 | 0.310 |
| | $S$ | 2.363 | 0.559 | 1.066 | 0.342 |
| | $C$ | 2.118 | 0.986 | 1.365 | 0.340 |
| | $\Sigma$ | 1.917 | 0.807 | 1.085 | 0.329 |
| Total Tokens ($\mu$) | $I$ | 81221 | 35211 | 42960 | 20880 |
| | $S$ | 19567 | 3979 | 6508 | 18240 |
| | $C$ | 96363 | 15906 | 51719 | 18824 |
| | $\Sigma$ | 67267 | 20050 | 34652 | 19471 |
| Estimated Cost in USD ($\mu$) | $I$ | 0.1761 | 0.1361 | 0.3018 | 0.0487 |
| | $S$ | 0.0356 | 0.0467 | 0.1371 | 0.0443 |
| | $C$ | 0.1957 | 0.0925 | 0.3928 | 0.0471 |
| | $\Sigma$ | 0.1398 | 0.0962 | 0.2797 | 0.0469 |
| VMPC ($\mu$) | $I$ | 0.9738 | 0.9722 | 0.8847 | 0.9900 |
| | $S$ | 0.3689 | 0.4100 | 0.3978 | 0.9894 |
| | $C$ | 0.7109 | 0.6966 | 0.6823 | 0.9851 |
| | $\Sigma$ | 0.7134 | 0.7209 | 0.6779 | 0.9884 |

(a) 

(b) 

| System | $X$ | $V_1$ | $M_1$ | $E_1$ | $H_1$ | $L_1$ | $V_2$ | $M_2$ | $E_2$ | $H_2$ | $L_2$ | VMPC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Raiven | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1.00 |
| Gemini | 1 | .67 | 0 | 0 | 1 | 1 | .67 | 0 | 0 | 1 | 1 | 0.53 |

Fig. 5: VMPC scoring for benchmark prompt #73: *"Render the CT isosurface in pink from `head.vti`, then summarize `head_sample.csv` using a histogram of intensity colored pink."* (a) Raiven output (VMPC = 1.00). (b) Gemini output (VMPC = 0.53): marks and encodings are missing. Fractional scores (e.g. $V = 0.67$) reflect disagreement among graders. The table shows per-view component scores averaged across three human graders; subscripts denote view 1 (isosurface) and view 2 (histogram). Both systems execute successfully ($X = 1$).

[3]All prompts, and outputs are available at https://alexandrairger.github.io/RaivenBenchmark/benchmark/processed/gallery.html.

### 7.3 Scoring Validation

Each visualization is scored independently by three human graders, and the final VMPC score is taken as the average across all three assessments. Inter-rater reliability across the three human graders was high (Krippendorff's $\alpha = 0.93$), indicating strong consistency in how graders applied the rubric. Because all graders are authors of this paper, we additionally validated human scores against an independent vision-language model (GPT-4.1), which received the benchmark prompt, rendered visualization, and VMPC rubric. We observe strong agreement between human and VLM evaluations (Pearson $r = 0.924$, Spearman $\rho = 0.881$, $n = 400$), confirming that the rubric is sufficiently well-specified to be applied objectively. All scores reported in this paper are human-assigned; VLM scores serve only as a validation instrument.

### 7.4 Results

The benchmark reveals a clear and consistent story. As shown in Table 3, Raiven scores near-perfect across all prompt groups and components (overall VMPC = 0.988, averaged across three human graders), while the three general-purpose baselines range between 0.678 and 0.721 on the same aggregate. Notably, Raiven achieves these results using ChatGPT 5.2, a less capable model than the ChatGPT 5.4 baseline, suggesting that the gains stem from the DSL-mediated architecture rather than from the underlying language model.

The performance gap is driven almost entirely by scientific visualization ($S$) prompts, which expose simultaneous failures across multiple VMPC components in all baselines. On $S$ prompts, general-purpose models fail at the most basic level, with execution rates suggesting a substantial portion of all prompts either crash or produce no visible output, and mark correctness and encoding correctness collapse among those that do. InfoVis ($I$) prompts tell a different story: execution is strong across all models and ChatGPT and Gemini nearly match Raiven on overall VMPC, with the remaining gap concentrated in mark and encoding correctness. Combined ($C$) prompts sit in between: execution recovers relative to $S$, but overall VMPC remains substantially below Raiven's, as the $S$ components within these prompts continue to drag down mark correctness, encoding correctness, and cross-view coordination for all baselines.

Across the full benchmark, data hallucination affects 4% of Claude outputs, 2% of ChatGPT and Gemini outputs, and 0% of Raiven outputs. A visualization that silently fabricates values is not an aesthetic failure but a correctness one, with real consequences ranging from flawed research findings to misdiagnoses in clinical settings. We note that VMPC does not score interaction, which understates Raiven's practical advantage: the compiler generates interactive controls for every visualization, including parameter sliders, transfer function editors, and cross-view coordination widgets, while the baselines produced little to no interactivity.

Raiven never fails to compile or hallucinates data across any prompt or category, not because it is a more capable language model but because its architecture makes these failure modes structurally impossible. However, this same property is also a limitation: because the LLM never directly inspects the data, it cannot apply reasoning to catch logical inconsistencies. For example, timestep labels such as `t1`, `t10`, ..., `t19`, `t2` were sorted alphabetically by the compiler rather than numerically. Similarly, Raiven plots all data as provided, with no automatic filtering of null or zero values, which can produce visually misleading results.

## 8 User Study

To validate whether the benchmark gains translate into measurable improvements in practice, we conducted an expert user study comparing Raiven against participants' own LLM-assisted visualization workflows.

### 8.1 Study Design

We recruited seven visualization experts (2 female, 5 male), consisting of three PhD students and four postdoctoral researchers whose research focuses on visualization. Participants reported an average visualization expertise of 4.43 on a 1–5 self-assessment scale. Each study session lasted approximately 60–90 minutes, and participants were compensated $20 for their participation. All participants provided informed consent. The study protocol was approved by the Harvard University Institutional Review Board (IRB25-1536) and the Department of Energy (ORAU001356).

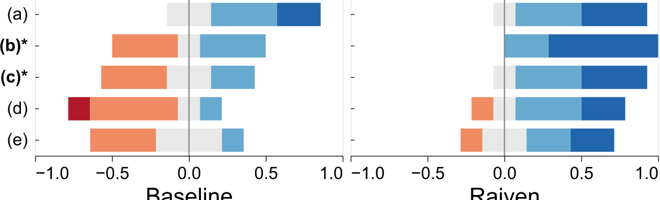



Fig. 6: Fraction of participants at each Likert level. Rows (a)–(e) index paired items in survey order: (a) ease of use; **(b)*** task efficiency; **(c)*** ease of creating correct visualizations; (d) trust in visualizations; (e) perceived control. Colors indicate Likert score 1 2 3 4 5 (1 = strongly disagree, 5 = strongly agree). $^*p < 0.05$.

Participants completed three visualization authoring tasks spanning information visualization, scientific visualization, and a combined workflow. Each task was presented as a printed image of a target visualization dashboard to replicate, simulating a common workflow in which a practitioner begins with a visualization concept and must recreate it using available tools. Typed instructions were intentionally avoided to prevent participants from directly copying task descriptions into either system.

Each participant completed all three tasks using both Raiven and a baseline workflow, with system order alternating across tasks and the starting assignment counterbalanced between participants. For their baseline, participants used their own preferred LLM-assisted tools: six of seven chose Claude in some form, spanning chat interfaces, notebook environments, and IDE-integrated agentic workflows (Claude web, Claude Code, Cursor, Antigravity, and Google Colab). All tools represented the highest-tier models available at the time of the study. To control for unequal access, we provided a standardized workstation with unlimited access to ChatGPT, Claude, Claude Code, Gemini, Codex, Cursor, Antigravity, VS Code, and Google Colab. The comparison is therefore between two LLM-mediated authoring strategies: Raiven's structured DSL pipeline versus free-form code generation through general-purpose models.

Each task began with a comprehension phase in which participants examined the printed target dashboard and wrote prompts or notes in a document. Participants then completed the task using the assigned system. We recorded comprehension time, task completion time, and the number of interaction iterations required to produce the visualization, and collected all generated outputs. After completing all tasks, participants filled out a post-study survey consisting of Likert-scale questions and open-ended feedback. Full task descriptions, target dashboards, and study materials are provided in Appendix E.2.

### 8.2 Raiven reduces debugging effort and iteration time

Raiven demonstrated clear efficiency advantages over baseline workflows across all three tasks. Participant ratings reflected this directly: Raiven scored significantly higher than baseline on task completion efficiency (Wilcoxon signed-rank, $p = 0.031$) and ease of creating correct visualizations (Wilcoxon signed-rank, $p = 0.031$), Figure 6. The strongest result in the study was for debugging effort, where Raiven was rated as significantly reducing debugging compared to baseline (Wilcoxon signed-rank vs. neutral, $p = 0.016$), Figure 7. Ease of use did not differ significantly between systems ($p = 0.75$), likely reflecting the high usability floor inherent to natural language interfaces generally. Raiven's gains were specific to output reliability and iteration speed rather than general usability.

Task completion patterns favored Raiven across all three tasks. Task 3 (combined) was the most consistent for Raiven, with nearly every participant completing it correctly on the first or second attempt. Task 2 (scientific visualization) proved the most challenging for base-

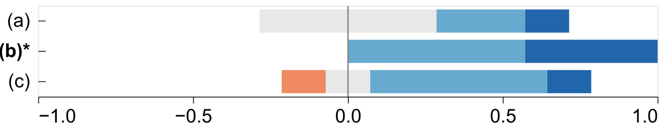



Fig. 7: Comparing Raiven with the Baseline: (a) required less mental effort; **(b)*** reduced debugging effort; (c) preferred for future tasks. Colors indicate Likert scores 1 2 3 4 5 (1 = strongly disagree, 5 = strongly agree). $^*p < 0.05$.

line workflows, with several participants spending upwards of 10–15 minutes with limited success. Baseline failures were varied: several participants received non-interactive static outputs when tasks required linked interactive views, one participant's map was generated for only a small geographic subset of the data, and another encountered repeated compile errors across two different tools and was unable to produce a working output at all.

Participants echoed these results in open-ended feedback, citing speed as Raiven's single biggest advantage: "by far the speed," "speed, better result in initial try," and "easy to iterate" were representative responses. One participant noted that with Raiven it was "extremely straightforward to get the correct result," attributing this to how the DSL clarified the mental model: "I need data, I need to know a visualization type, I need to know encodings — the system prompting for decisions makes this really clear, without implementation details."

Terminology and specification challenges were a recurring issue across both systems. Several participants used imprecise vocabulary, asking for a "bar chart" when the target was a histogram, or "volume" when they meant isosurface. Both systems took these descriptions literally, although baseline systems frequently missed parts of the specifications even when stated correctly. One of the most commonly noted limitations of Raiven was that its clarifying prompts could have been clearer and more intuitive, though all participants were ultimately able to answer them and allow Raiven to proceed.

### 8.3 Baseline systems silently fabricate data; Raiven's DSL makes outputs verifiable

The most consequential baseline failure was silent data fabrication, which surfaced repeatedly in Task 2. Despite VTI being a standard volumetric data format, several baseline systems failed to parse it correctly and silently substituted fabricated or simulated datasets, producing plausible-looking visualizations built on invented data. One participant's Cursor-based workflow produced a visualization with no real data at all; another spent over 20 minutes with Antigravity before realizing the same issue; a third participant received a synthetic skull volume generated with a Gaussian filter rather than the actual file. One participant caught the issue mid-task and added "don't mess with my data" explicitly to their prompt. In most cases, the output looked superficially correct, requiring active verification to detect.

Silent data fabrication of this kind is architecturally impossible in Raiven: data is handled entirely by deterministic code, and the LLM never directly accesses or processes the underlying data. Raiven handled Task 2 well, with most participants finishing in under 4 minutes using the actual data. Where Raiven did fall short, such as missing volume rendering components, it did so transparently rather than silently substituting data. This distinction between *honest failure* and *silent failure* has real consequences, particularly for users without the data familiarity and domain knowledge to recognize that the underlying data has been replaced entirely.

Trust emerged as a central theme across both qualitative feedback and behavioral observations. Baseline trust concerns centered on process opacity: "I had no idea what type of output it would produce, what software architecture or visualization libraries it would use. I was not confident it would read the data formats and understand their contents as expected." Raiven fostered trust primarily through the readability of its RaivenDSL output and confidence in data integrity. The RaivenDSL specification serves as a legible mediator between user intent and compiler input: participants can verify what the system understood before any code is generated. As one participant noted, "I could quickly read its RaivenDSL output," and another emphasized that "I could trust that the system was not manipulating my data." Participant ratings reflected this pattern, with trust in visualizations trending toward Raiven though falling short of significance (Wilcoxon signed-rank, $p = 0.063$), likely reflecting limited statistical power at $n = 7$ rather than a genuine null effect.

### 8.4 Experts prefer Raiven for prototyping and unfamiliar data, baselines for production workflows

Participants drew a clear distinction between the types of tasks each system was best suited for. Raiven was preferred for demos, prototyping, quick data exploration, standalone dashboards, and tasks involving niche or unfamiliar data formats, particularly those combining 3D volumetric data with information visualization. Baseline was preferred for production projects requiring many iterations, tasks with specific library or language requirements, extending existing codebases, and workflows demanding heavy verification and fine-grained control. One participant captured the baseline use case succinctly: "if I want to be a micromanager." When asked about overall system preference, five of seven participants reported preferring Raiven (one definitely, four probably), two reported no preference, and none preferred the baseline.

## 9 Conclusions and Future Work

A central result of this work is that an LLM does not need to generate backend code to support effective visualization authoring. In Raiven, the model operates only over RaivenDSL, while the compiler handles implementation logic deterministically. This separation reduces fragility, keeps visualization behavior under system control, and makes the pipeline easier to verify. By externalizing visualization knowledge into the language and compiler rather than relying on implicit model knowledge of libraries and conventions, Raiven makes data faithfulness and structural correctness easier to enforce than in direct code-generation pipelines. Our findings also highlight the value of a unified representation spanning both scientific and information visualization. Prior systems remain split between chart-focused information visualization and narrowly scoped scientific-visualization workflows, even though many real analysis tasks require both. A cross-domain DSL provides a practical way to bridge this divide. More broadly, the results suggest that visualization authoring systems should be evaluated not only by syntactic validity or visual plausibility, but also by data faithfulness, view completeness, and cross-view structure. VMPC is one step toward that broader evaluation framework.

The current limitations of Raiven fall along a natural boundary: some constrain only the implementation, while others reflect deliberate choices in the language design. On the implementation side, datasets are loaded entirely into the browser with a 100 MB cap, excluding large-scale data and streaming workflows. Animation, cross-backend linking, and unstructured mesh geometry are not yet supported but require only new compiler capabilities, not language changes. On the language side, RaivenDSL provides one rendering implementation per mark type, prioritizing determinism over algorithmic choice, and deliberately excludes data preprocessing to remain declarative. Tensor and hierarchical mark types would extend the design space. Adding new rendering backends such as Matplotlib or a WebGPU-based renderer requires only a new code-generation path, since backend routing already supports per-view assignment. The conversational flow assumes users with some visualization vocabulary; novice users would benefit from more guided clarification strategies. Fine-tuning a model specifically on RaivenDSL specifications is a tractable next step that would further improve generation reliability.

Raiven demonstrates that DSL-mediated generation is a practical foundation for trustworthy visualization authoring via natural language, and its architecture is designed to remain relevant as models continue to improve. The mediation layer, languages, compilers, and validation pipelines that sit between language models and rendering backends, is where the visualization community can most effectively assert its own standards of correctness, faithfulness, and design. Raiven is one step in building that foundation.

## 10 Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Computer Science Competitive Portfolios program under Contract No. DE-AC05-00OR22725.

## References

[1] K. Ai, K. Tang, and C. Wang. Nli4volvis: Natural language interaction for volume visualization via llm multi-agents and editable 3d gaussian splatting. *IEEE Transactions on Visualization and Computer Graphics*, 32(1):46–56, 2026. doi: 10.1109/TVCG.2025.3633888 2, 3

[2] Allen Institute for Cell Science. hiPSC single-cell image dataset [AICS-10_8], 2018. 5

[3] J. Beyer, J. Troidl, S. Boorboor, M. Hadwiger, A. Kaufman, and H. Pfister. A survey of visualization and analysis in high-resolution connectomics. *Computer Graphics Forum*, 41(3):573–607, 2022. doi: 10.1111/cgf.14574 5

[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] 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, 2025. doi: 10.1109/TVCG.2024.3456320 7

[6] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: a parallel dsl for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 10 pages, p. 111–120. Association for Computing Machinery, New York, NY, USA, 2012. doi: 10.1145/2254064.2254079 2

[7] H. Choi, W. Choi, T. M. Quan, D. G. C. Hildebrand, H. Pfister, and W.-K. Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2407–2416, 2014. doi: 10.1109/TVCG.2014.2346322 2

[8] V. Dibia. LIDA: A tool for automatic generation of grammar-agnostic visualizations and infographics using large language models. In D. Bollegala, R. Huang, and A. Ritter, eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 113–126. Association for Computational Linguistics, Toronto, Canada, July 2023. doi: 10.18653/v1/2023.acl-demo.11 2

[9] O. Elek, J. N. Burchett, J. X. Prochaska, and A. G. Forbes. Polyphorm: Structural analysis of cosmological datasets via interactive physarum polycephalum visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):806–816, 2021. doi: 10.1109/TVCG.2020.3030407 5

[10] P. Harth, A. Bast, J. Troidl, B. Meulemeester, H. Pfister, J. Beyer, M. Oberlaender, H.-C. Hege, and D. Baum. Rapid Prototyping for Coordinated Views of Multi-scale Spatial and Abstract Data: A Grammar-based Approach. In C. Hansen, J. Procter, R. G. Raidou, D. Jönsson, and T. Höllt, eds., *Eurographics Workshop on Visual Computing for Biology and Medicine*. The Eurographics Association, 2023. doi: 10.2312/vcbm.20231218 2

[11] J. Heer and D. Moritz. Mosaic: An architecture for scalable & interoperable data views. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):436–446, 2024. doi: 10.1109/TVCG.2023.3327189 5

[12] 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 2, 7

[13] J. Huang, Y. Xi, J. Hu, and J. Tao. Flownl: Asking the flow data in natural languages. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):1200–1210, 2023. doi: 10.1109/TVCG.2022.3209453 2, 3

[14] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55 2

[15] D. Jia, Y. Wang, and I. Viola. Chat modeling: Natural language-based procedural modeling of biological structures without training. 2024. doi: 10.48550/arXiv.2404.01063 2, 3

[16] S. A. Joseph, S. M. Husain, S. S. R. Offner, S. Juneau, P. Torrey, A. S. Bolton, J. P. Farias, N. Gaffney, G. Durrett, and J. J. Li. Astrovisbench: A code benchmark for scientific computing and visualization in astronomy. 2025. doi: 10.48550/arXiv.2505.20538 1, 7

[17] D. Kouřil, T. Manz, S. L'Yi, and N. Gehlenborg. Design space and declarative grammar for 3d genomic data visualization. *IEEE Transactions on Visualization and Computer Graphics*, 32(1):890–900, 2026. doi: 10.1109/TVCG.2025.3634654 2

[18] F. Lan, M. Young, L. Anderson, A. Ynnerman, A. Bock, M. A. Borkin, A. G. Forbes, J. A. Kollmeier, and B. Wang. Visualization in astrophysics: Developing new methods, discovering our universe, and educating the earth. *Computer Graphics Forum*, 40(3):635–663, 2021. doi: 10.1111/cgf.14332 5

[19] D. Lange, S. Gao, P. Sui, A. Money, P. Misner, M. Zitnik, and N. Gehlenborg. Yac: Bridging natural language and interactive visual exploration with generative ai for biomedical data discovery. 2025. doi: 10.48550/arXiv.2509.19182 2

[20] H. Lin, D. Moritz, and J. Heer. Dziban: Balancing agency & automation in visualization design via anchored recommendations. In *Proceedings of the 2020 CHI conference on human factors in computing systems*, pp. 1–12, 2020. 2

[21] S. Liu, H. Miao, and P.-T. Bremer. Paraview-mcp: An autonomous visualization agent with direct tool use. In *2025 IEEE Visualization and Visual Analytics (VIS)*, pp. 61–65, 2025. doi: 10.1109/VIS60296.2025.00018 2

[22] Y. Luo, N. Tang, G. Li, J. Tang, C. Chai, and X. Qin. Natural language to visualization by neural machine translation. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):217–226, 2022. doi: 10.1109/TVCG.2021.3114848 2

[23] 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 2

[24] J. Maitin-Shepard, A. Baden, W. Silversmith, E. Perlman, F. Collman, T. Blakely, J. Funke, C. Jordan, B. Falk, N. Kemnitz, tingzhao, C. Roat, M. Castro, S. Jagannathan, moenigin, J. Clements, A. Hoag, B. Katz, D. Parsons, J. Wu, L. Kamentsky, P. Chervakov, P. Hubbard, S. Berg, J. Hoffer, A. Halageri, C. Machacek, K. Mader, L. Roeder, and P. H. Li. google/neuroglancer:, Oct. 2021. doi: 10.5281/zenodo.5573294 5

[25] T. Mallick, O. Yildiz, D. Lenz, and T. Peterka. Chatvis: Automating scientific visualization with a large language model. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 49–55, 2024. doi: 10.1109/SCW63240.2024.00014 2

[26] 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, 2023. doi: 10.1109/TVCG.2022.3209460 4

[27] 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 2

[28] 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 2

[29] J. Pollock and A. Satyanarayan. Gofish: A grammar of more graphics! *IEEE Transactions on Visualization and Computer Graphics*, 32(1):549–559, 2026. doi: 10.1109/TVCG.2025.3634250 2

[30] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. Vislang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2388–2396, 2014. doi: 10.1109/TVCG.2014.2346318 1, 2

[31] M. Ribalta-Albado and P.-P. Vázquez. Evaluating llms' abilities to create charts, a systematic approach. *Computers & Graphics*, 135:104544, 2026. doi: 10.1016/j.cag.2026.104544 1

[32] L. A. Royer. Omega—harnessing the power of large language models for bioimage analysis. *nature methods*, 21(8):1371–1373, 2024. doi: 10.1038/s41592-024-02310-w 2

[33] 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 2

[34] D. Scholz. A modular domain-specific language for interactive 3D visualization. Diploma thesis, Technische Universität Wien, 2021. doi: 10.34726/hss.2021.80484 2

[35] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit (4th*

*ed.)*. Kitware, 2006. 2

[36] M. Shih, C. Rozhon, and K.-L. Ma. A declarative grammar of flexible volume visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):1050–1059, 2019. doi: 10.1109/TVCG.2018.2864841 2

[37] R. Sicat, J. Li, J. Choi, M. Cordeil, W.-K. Jeong, B. Bach, and H. Pfister. Dxr: A toolkit for building immersive data visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):715–725, 2019. doi: 10.1109/TVCG.2018.2865152 2

[38] Y. Tian, W. Cui, D. Deng, X. Yi, Y. Yang, H. Zhang, and Y. Wu. Chartgpt: Leveraging llms to generate charts from abstract natural language. *IEEE Transactions on Visualization and Computer Graphics*, 31(3):1731–1745, 2025. doi: 10.1109/TVCG.2024.3368621 2

[39] M. Tory and T. Moller. Rethinking visualization: A high-level taxonomy. In *IEEE Symposium on Information Visualization*, pp. 151–158, 2004. doi: 10.1109/INFVIS.2004.59 4

[40] A. Vallenari, A. G. Brown, T. Prusti, J. H. De Bruijne, F. Arenou, C. Babusiaux, M. Biermann, O. L. Creevey, C. Ducourant, D. W. Evans, et al. Gaia data release 3-summary of the content and survey properties. *Astronomy & Astrophysics*, 674:A1, 2023. doi: 10.1051/0004-6361/202243940 5

[41] L. Wilkinson. *The grammar of graphics*. Statistics and computing. Springer, New York, 1999. 2

[42] Y. Wu, Y. Wan, H. Zhang, Y. Sui, W. Wei, W. Zhao, G. Xu, and H. Jin. Automated data visualization from natural language via large language models: An exploratory study. *Proc. ACM Manag. Data*, 2(3), article no. 115, 28 pages, May 2024. doi: 10.1145/3654992 1

[43] Z. Wu, V. Le, A. Tiwari, S. Gulwani, A. Radhakrishna, I. Radiček, G. Soares, X. Wang, Z. Li, and T. Xie. Nl2viz: natural language to visualization via constrained syntax-guided synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, 12 pages, p. 972–983. Association for Computing Machinery, New York, NY, USA, 2022. doi: 10.1145/3540250.3549140 2

[44] 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 2

# A RAIVENDSL

This section provides the complete RaivenDSL language reference: the formal grammar (Section A.1), the language structure including data constructors, layer properties, link types, and selection types (Section A.2), and per-mark-type specifications (Section A.3).

## A.1 Grammar

As referenced by Section 4.2. RaivenDSL supports two surface syntaxes that parse to the same abstract syntax tree. Listings 1 and 2 show the same specification in both forms. The brace-based syntax is defined by the ANTLR4 grammar in Section A.1.1; the indentation-based variant is described in Section A.1.2.

Listing 1: Indentation-based syntax for the Teaser Figure 1 example.

```
vis:
  data:
    vol = img("taylorgreen_9.vti", format="vti")
    sample = tbl("tg9_sample.csv", format="csv")
  view "volume_streamline":
    layer:
      from = vol
      mark = volume
      encode:
        field = "vorticity"
    layer:
      from = vol
      mark = streamline
      encode:
        vx = "ux"
        vy = "uy"
        vz = "uz"
  view "histogram":
    layer:
      from = sample
      mark = histogram
      encode:
        x = "vorticity"
```

Listing 2: Brace-based syntax for the same specification. Both listings produce an identical AST.

```
vis {
  data {
    vol: img("taylorgreen_9.vti", format: "vti");
    sample: tbl("tg9_sample.csv", format: "csv");
  }
  view "volume_streamline" {
    layer {
      from: vol;
      mark: volume;
      encode: { field: "vorticity" };
    }
    layer {
      from: vol;
      mark: streamline;
      encode: { vx: "ux", vy: "uy", vz: "uz" };
    }
  }
  view "histogram" {
    layer {
      from: sample;
      mark: histogram;
      encode: { x: "vorticity" };
    }
  }
}
```

### A.1.1 Classic Brace-Based Syntax—ANTLR4 Grammar

```
grammar VisDSL;

program    : 'vis' '{' topStmt* '}' ;

topStmt    : dataBlock
           | viewDecl
           | selectionsBlock
           ;

dataBlock  : 'data' '{' (dataDecl ';')* '}' ;
dataDecl   : (IDENT | 'geo') ':' dataCtor ;

dataCtor   : 'img' '(' STRING (',' argList)? ')'
           | 'tbl' '(' STRING (',' argList)? ')'
           | 'net' '(' STRING (',' argList)? ')'
           | 'geo' '(' STRING (',' argList)? ')'
           | 'func' '(' argList? ')'
           ;

argList    : namedArg (',' namedArg)* ;
namedArg   : IDENT ':' value ;

value      : STRING
           | NUMBER
           | 'true' | 'false' | 'null'
           | obj | arr
           ;

obj        : '{' (namedArg (',' namedArg)*)? '}' ;
arr        : '[' (value (',' value)*)? ']' ;

viewDecl   : 'view' STRING '{' (layerDecl | linkDecl |
     interactionsBlock)* '}' ;

layerDecl  : 'layer' '{' layerStmt* '}' ;

layerStmt  : 'from'     ':' layerIdent  ';'
           | 'geo'      ':' layerIdent  ';'
           | 'mark'     ':' IDENT   ';'
           | 'encode'   ':' obj     ';'
           | 'style'    ':' obj     ';'
           | 'where'    ':' expr    ';'
           ;

selectionsBlock : 'selections' '{' selectionDecl* '}' ;
selectionDecl   : 'select' '(' argList ')' ';' ;

linkDecl   : 'link' '(' argList ')' ';' ;

interactionsBlock : 'interactions' '{' interactionDecl*
     '}' ;
interactionDecl   : 'on' '(' STRING ')' '{' action* '}' ;
action            : 'bind' '(' STRING (',' argList)? ')'
     ';' ;

expr       : ;

IDENT      : [A-Za-z_][A-Za-z0-9_]* ;
NUMBER     : [+-]?([0-9]+('.'[0-9]+)?)([eE][+-]?[0-9]+)? ;

STRING     : '"' ( '\\' . | ~["\\] )* '"' | '\'' ( '\\' .
      | ~['\\] )* '\'' ;
WS         : [ \t\r\n]+ -> skip ;
COMMENT    : '//' ~[\r\n]* -> skip ;
BLOCKCOMM  : '/*' .*? '*/' -> skip ;
```

### A.1.2 Pythonic Indentation-Based Syntax

The indentation-based syntax replaces braces with colon-plus-indentation blocks, semicolons with newlines, and uses `=` for property assignment within layers (e.g., `from = ct`) rather than `:`. Data constructors and function-call forms (`link(...)`, `select(...)`, `bind(...)`) retain their parenthesized syntax unchanged. The `encode` and `style` properties additionally accept a block form with indented key-value pairs as an alternative to inline objects. A hand-written to-

kenizer emits INDENT/DEDENT tokens from leading whitespace; the parser is recursive descent and produces an identical AST to the ANTLR-based parser.

## A.2 Language Structure

This section provides the complete language reference for RaivenDSL's compositional rules: top-level constructs, data constructors, layer properties, link types, and selection types. Per-mark-type specifications follow in Section A.3.

### A.2.1 Top-Level Constructs

A RaivenDSL program is a single vis block containing data declarations, view definitions, and an optional selections block.

Table 4: Top-level constructs in a RaivenDSL program.

| Construct | Contains | Cardinality |
|---|---|---|
| vis | data, view, selections | 1 per program |
| data | named data declarations | 1+ sources |
| view | layer, link, interactions | 1–9 views |
| selections | select(...) declarations | 0–1 blocks |

### A.2.2 Data Constructors

Each data source carries a typed constructor that determines which marks and backends are valid. The constructor name identifies the data modality; the compiler uses it together with the mark type to route each view to the appropriate rendering backend.

Table 5: Data constructors. All constructors except func() take a path string as their first positional argument.

| Constructor | Format | Backend | Optional Args |
|---|---|---|---|
| img() | VTI | VTK.js | format |
| tbl() | CSV, JSON | D3 | format |
| net() | JSON | D3 | format |
| geo() | GeoJSON | D3 | format, crs |
| func() | procedural | VTK.js | equations, dims, bounds, range, params |

### A.2.3 Layer Properties

Each layer declares a data source and a mark type; encodings, styles, and geographic references are optional at the language level but may be required by specific marks.

Table 6: Layer properties. ● = required, ○ = optional. Encoding keys and style keys are mark-dependent (Section A.3).

| Property | Type | Req. | Description |
|---|---|---|---|
| from | identifier | ● | Data source reference |
| mark | identifier | ● | Visual mark type |
| encode | object | ○ | Channel-to-field mappings |
| style | object | ○ | Appearance overrides |
| geo | identifier | ○ | Geographic data reference |

### A.2.4 Link Types

Links declare cross-view coordination. Each link specifies exactly one coordination type (tf, slice, or selection) and the participating views.

Table 7: Link parameters. A link must specify exactly one of selection, tf, or slice. selection links operate within D3; tf and slice links are within VTK.js.

| Parameter | Type | Required With |
|---|---|---|
| selection | brush-and-filter | target or views |
| tf | shared transfer fn. | views |
| slice | synced slice index | views, axes |
| axes | slice orientation | slice |
| views | view ID array | all link types |
| target | single view ID | selection only |

### A.2.5 Selections and Interactions

Selections declare named handles for cross-view coordination; the interactions block binds events to those handles. Brush-based selection is driven by the DSL; point selection with cross-view highlighting is generated automatically by the compiler for force_graph and heatmap marks. All selection-based coordination currently operates within D3 views only.

Table 8: Selection and interaction constructs.

| Construct | Required | Effect |
|---|---|---|
| select(*name*) | name | Registers a named selection |
| on("brush") | event = "brush" | Installs a rectangular D3 brush |
| bind(*sel.*) | selection name | Publishes brush to named selection |

## A.3 Marks

As referenced by Sections 4.3 and 6.3. RaivenDSL supports 22 mark types: 5 for scientific visualization (rendered via VTK.js) and 17 for information visualization (rendered via D3). Tables 9 and 12 list each mark with its encoding channels, style parameters, and compiler-generated controls. All style fields are optional; when omitted, the compiler resolves defaults deterministically. Encoding channels define the semantic content of the visualization (which data variables map to which visual channels), while style parameters control appearance and can be adjusted at runtime through the generated controls without re-editing the specification. All SciVis views additionally receive implicit navigation controls (rotation, zoom, pan).

### A.3.1 Scientific Visualization Marks

All scientific visualization marks accept img or func data sources. All encode and style fields are optional; when omitted, the compiler selects the first available scalar or vector field and resolves data-derived ranges.

Table 9 lists each mark's encoding channels, style parameters, and compiler-generated controls. Table 10 summarizes data type constraints, layering compatibility, implicit controls, and linking options.

**Slice rendering modes.** The slice mark's rendering mode and camera are determined by the view configuration. A single slice layer with a single axis renders as a 2D image with pan, zoom, and scroll-to-index navigation. Any other configuration—multiple axes (e.g. ["XY","XZ","oblique"]), or any additional layer in the same view—renders in 3D with a trackball camera. Oblique slices follow the same rule, with one exception: a single oblique slice defaults to 2D but can be forced to 3D by setting is3DPlane: true.

Table 10: Scientific visualization mark metadata. All SciVis marks except for LIC can be layered together within a single view. Implicit controls are provided automatically based on rendering mode.

| Mark | Layerable With | Implicit Controls | Linking |
|---|---|---|---|
| volume | iso., slice, stream. | 3D trackball camera | — |
| isosurface | vol., slice, stream. | 3D trackball camera | — |
| slice | vol., iso., stream. | 2D or 3D (see text) | slice, tf |
| streamline | vol., iso., slice | 3D trackball camera | — |
| lic | — | 2D image camera | — |

Table 9: Scientific visualization mark specifications. Gray encode fields are semantically important but syntactically optional (the compiler infers them from the data when omitted). All style fields are optional. Controls marked with † are runtime-only (generated by the compiler, not settable in the specification).

| Mark | Encode | Style | Generated Controls |
|---|---|---|---|
| `volume` | `field`: scalar array name defaults to first array | `sample_distance`: ray step size (default 0.7)<br>`palette`: named colormap (e.g. `"viridis"`)<br>`ctf`: color TF stops (overrides `palette`)<br>`otf`: opacity TF stops | Color transfer function editor<br>Opacity transfer function editor<br>Palette dropdown |
| `isosurface` | `field`: scalar array name defaults to first array | `iso_value`: threshold (default 1/3 of range)<br>`color`: hex color<br>`opacity`: 0–1 | Iso-value slider<br>RGBA color picker with opacity<br>Material preset† (matte/glossy/metallic/custom)<br>Specular, diffuse, ambient sliders† |
| `slice` | `field`: scalar array name defaults to first array | `axes`: `"XY"`, `"XZ"`, `"YZ"`, `"oblique"`, or array (e.g. `["XY","XZ"]`); default `"XY"`<br>`palette`: named colormap<br>`ctf`: color TF stops (overrides `palette`)<br>`quaternion`: orientation (oblique only)<br>`offset`: position (oblique only)<br>`is3DPlane`: oblique rendering mode (auto-inferred; see text) | *Axis-aligned* (per axis in `axes`):<br>Slice index slider<br>Visibility toggle†<br>Color range interval slider†<br>Color transfer function editor<br>Palette dropdown<br>*Oblique* (when `"oblique"` in `axes`):<br>3D rotation gizmo†<br>Offset slider† |
| `streamline` | `vx`, `vy`, `vz`: velocity component array names | `seed_bounds`: $[x_0, x_1, y_0, y_1, z_0, z_1]$<br>`seed_count`: number of seeds (default 100)<br>`integration_step`: step size (default 0.5)<br>`max_steps`: max steps (default 1000)<br>`color`: hex color<br>`tube_radius`: streamtube radius | Seed region†: dual-handle slider per axis<br>Seed count slider<br>Integration step slider<br>Max steps slider<br>Color picker<br>Tube radius slider<br>Recalculate button† |
| `lic` | `vx`, `vy`: velocity component array names | `number_of_steps`: integration steps (default 50)<br>`step_size`: step length (default 1.0)<br>`enhanced_lic`: contrast enhancement (default `true`)<br>`lic_intensity`: blending weight (default 0.8) | Step count slider<br>Step size slider<br>Enhanced LIC toggle<br>Intensity slider |

### A.3.2 Information Visualization Marks

Most information visualization marks consume tabular data (`tbl` with CSV or an inline table). Force-directed graph layers use a network source (`net`, JSON). Chord, Sankey, and most chart marks expect tables; choropleth layers typically pair a geographic source (`geo`, GeoJSON or similar) with table columns that supply region keys and values. Required encode channels are per mark (for example `x` and `y` for scatter plots, `dimensions` for parallel coordinates, `source` and `target` for flows and networks); optional channels such as `color` are listed in the schema when allowed. Style parameters are optional throughout. When optional encodings or styles are omitted, the compiler and renderer apply defaults (for example named color schemes, bin counts, or layout parameters). Table 12 lists each mark's encoding channels, style parameters, and compiler- or UI-generated controls. Table 11 summarizes data-type constraints, layering compatibility, implicit controls, and linking options.

**Multi-view interaction and layout.** Views that share a linked selection use `selections`, `link(selection: ...)`, and `interactions` (for example brush-to-filter) so brushing in one chart updates linked views; parallel coordinates additionally expose per-axis brushes in the runtime UI. Network and flow marks use dedicated layouts (force simulation, chord diagram, Sankey) rather than shared Cartesian axes; Cartesian marks (points, lines, bars, distributions, heatmaps, hexbins, etc.) are drawn on scales derived from the data unless overridden in the view or layer specification.

Table 11: Information visualization mark metadata. Single-view layering is supported only for the combinations listed (see text). Brush and linking are enabled when the program declares `selections`, `link(selection: ...)`, and `interactions` `on("brush")`.

| Mark | Layerable With | Implicit | Linking |
|---|---|---|---|
| `points` / `bubble` | hexbin, choropleth | — | brush (emit/follow) |
| `hexbin` | points, choropleth | — | — |
| `line` | line, band | — | — |
| `band` | line | — | — |
| `histogram` | KDE, histogram | — | brush (follow) |
| `kde` | histogram | — | brush (follow) |
| `heatmap` | — | — | brush (emit/follow), point (follow) |
| `bar` | — | — | — |
| `boxplot` | — | — | — |
| `violin` | — | — | — |
| `ridgeline` | — | — | — |
| `parallel coordinates` | — | — | brush (emit/follow) |
| `pie` | — | — | — |
| `chord` | — | — | — |
| `sankey` | — | — | — |
| `force_graph` | — | Node dragging | point (emit) |
| `choropleth` | points, hexbin | — | — |

Table 12: Information visualization mark specifications. Gray encode fields are semantically important but syntactically optional. All style fields are optional. Controls marked with † are runtime-only (generated by the compiler, not settable in the specification). `bubble` is treated as `points` with a `size` encoding in the generated IR.

| Mark | Encode | Style | Generated Controls |
|---|---|---|---|
| `points` | `x`, `y`; `color`: category or value | `radius`, `fill_color` | Fill color; Categories if `color`-encoded; brush interval† |
| `hexbin` | `x`, `y`; `color`: value field | `radius`, `color_scheme` | Color palette dropdown |
| `heatmap` | `x`, `y`; `color`: cell value | `color_scheme` | Color palette; flip x/y toggles; brush† (numeric density); point selection† (adjacency) |
| `histogram` | `x` (count implicit) | `bins`, `fill_color`, `stroke_color` | Fill color; bin size slider (5–100); brush follow† |
| `kde` | `x` | `bandwidth`, `stroke_width`, `stroke_color` | Stroke color; brush follow† |
| `boxplot` | `x`, `y`; `color`: group field | `width`, `fill_color`, `stroke_color`, `stroke_width` | Fill color / Categories |
| `violin` | `x`, `y`; `color`: group field | `bandwidth`, `fill_color`, `stroke_color`, `stroke_width`, `show_median` | Fill color / Categories |
| `ridgeline` | `x`, `y` (categorical group); `color`: group field | `bandwidth`, `fill_color`, `stroke_color`, `stroke_width`, `overlap`, `height` | Fill color / Categories |
| `line` | `x`, `y` (`y` may be array); `color`: series field | `stroke_width`, `stroke_color` | Stroke color; Categories if `color`-encoded; linked selection dims opacity† |
| `band` | `x`, `y0`, `y1`; `color`, `opacity`: optional | `fill_color`, `fill_opacity`, `stroke_color`, `stroke_width` | Fill color |
| `bar` | `x`, `y`; `color`: stacking or grouping | `fill_color`, `stroke_color` | Fill color / Categories |
| `pie` | `label`, `value`; `color`: label field | `inner_radius`, `outer_radius` | Categories |
| `chord` | `source`, `target`, `value`; `group`: optional | `pad_angle`, `inner_radius`, `outer_radius` | Categories |
| `sankey` | `source`, `target`, `value`; `node`: optional | `node_width`, `node_padding`, `link_opacity`, `align`, `link_color` | Link color mode (static / source / target / interpolate) |
| `force_graph` | `source`, `target`; `value`, `color`: optional | `node_radius`, `link_distance`, `link_strength`, `charge_strength`, `stroke_width`, `stroke_opacity`, `fill_color`, `stroke_color`, `color_scheme` | Node/link sliders; path source/target fields†; Categories if `color`-encoded; click-to-select† |
| `choropleth` | `region`, `value`; `color`: value field | `color_scheme`, `stroke_color`, `stroke_width`, `projection` | Color palette per layer |
| `parallel coordinates` | `dimensions`: list of columns; `color`: category field | `stroke_width`, `stroke_opacity`, `color_scheme` | Fill color / Categories; per-axis brush† |

## B NATURAL LANGUAGE INTERFACE

As referenced by Section 5.2.

### B.1 Schema Structure

The schema (Fig. B.1) is a structured representation used for Natural Language to RaivenDSL generation. It consists of the following components (fields shown in gray in Fig. B.1 denote conditionally included components, which appear only when required by the data type or visualization):

- **`task_summary`** (string): Natural-language description of the visualization goal.
- **`data`** (object): Mapping from source name to data source definition.
  - `type`: One of `tbl`, `img`, `net`, `geo`, `func`
  - `path`: File path or URL
  - `args`: Optional parameters
  - `variables`: List of variable definitions
    - `name`: Field/column name used in encodings
    - `data_type`: Primitive type (e.g., number, string, date)
    - `semantic_type`: `quantitative` or `qualitative` (optional)
    - `role_hint`: Optional role (e.g., category, value, id)
  - `dimensions`: Dimensions of image/volume data (e.g., VTI); required for volumetric sources and omitted otherwise
- **`views`** (array): List of view specifications.
  - `view_id`: Unique view identifier
  - `layers`: List of layers
    - `from`: Data source reference
    - `geo`: Geographic reference (if applicable)
    - `mark`: Visual mark type
    - `encode`: Channel-to-field mapping
    - `style`: Optional styling parameters
  - `links_out`: Target view IDs for linking (optional)
  - `interactions`: View-specific interaction configuration (optional)
- **`selections`** (array): Interaction definitions.
  - `name`: Selection identifier (optional)
  - `type`: `interval` or `point` (optional)
  - `bind_view`: View where interaction occurs (optional)
  - `bind_channels`: Channels or fields used in selection (optional)
- **`linking`** (object): Multi-view coordination.
  - `shared_data_source`: Data source used by linked views (optional)
  - `linked_view_ids`: Views participating in linking (optional)
  - `selection_name`: Selection used for coordination (optional)
  - `link_style`: Linking mode (optional)
- **`slice_linking`** (array): Slice-based interaction groups for volume visualizations, included only when slice interactions are present.
  - `linked_view_ids`: Views participating in slice linking
  - `axes`: Slice orientation (e.g., `XY`, `XZ`, `YZ`, or oblique)
  - `slice_link_id`: Identifier for synchronizing slice position across views
  - `tf_link_id`: Identifier linking slice interaction to transfer-function interaction
- **`tf_linking`** (array): Transfer-function linking groups, included only when transfer-function interactions are present.
  - `linked_view_ids`: Views participating in transfer-function linking
  - `tf_link_id`: Identifier for shared transfer-function interaction

```
{
  "task_summary": "",
  "data": {
    "<source_name>": {
      "type": "tbl|img|net|geo|func",
      "path": "",
      "args": {},
      "variables": [
        {
          "name": "",
          "data_type": "",
          "semantic_type": "quantitative|qualitative",
          "role_hint": ""
        }
      ],
      "dimensions": [x, y, z]
    }
  },
  "views": [
    {
      "view_id": "",
      "layers": [
        {
          "from": "",
          "geo": "",
          "mark": "",
          "encode": {},
          "style": {}
        }
      ],
      "links_out": [],
      "interactions": {}
    }
  ],
  "selections": [
    {
      "name": "",
      "type": "interval|point",
      "bind_view": "",
      "bind_channels": []
    }
  ],
  "linking": {
    "shared_data_source": "",
    "linked_view_ids": [],
    "selection_name": "",
    "link_style": "views"
  },
  "slice_linking": [
    {
      "linked_view_ids": [],
      "axes": "XY|list of XY/XZ/YZ/oblique",
      "slice_link_id": "",
      "tf_link_id": ""
    }
  ],
  "tf_linking": [
    {
      "linked_view_ids": [],
```

```
      "tf_link_id": ""
    }
  ]
}
```

**Validation and Progression:** The schema is constructed incrementally through a sequence of nodes. Each node must satisfy structural and semantic constraints before the workflow advances. Validation enforces correct data references, valid mark types, compatibility between variables and encoding channels, and completeness of required fields. If any constraint fails, the schema is not updated and the workflow remains on the current node, prompting for clarification until the requirements are satisfied.

1. **Task Definition**
   - Requirement: Non-empty textual summary derived from the user's initial description.
   - Behavior: Generates `task_summary` by summarizing the user's first message; always advances with no validation constraints.
2. **Data**
   - Requirement: Valid data sources with parsed variables.
   - Behavior: Enforced at session start through file upload. The workflow cannot begin unless a data file is successfully uploaded and parsed; unsupported or invalid files prevent initialization.
3. **View & Layer**
   - Requirements:
     - Each view must define a unique `view_id`.
     - Each layer must reference a valid data source (`from` $\in$ `data`).
     - When only one data source is present, all layers must resolve to that source.
   - Behavior:
     - Uniqueness of `view_id` and single-source assignment are enforced programmatically.
     - Missing or invalid layer-to-data assignments trigger re-prompting, requiring the user to explicitly specify the data source for each layer.
4. **Mark**
   - Requirement: Each layer must specify a valid mark from the RaivenDSL mark set.
   - Behavior: If no visualization type is specified or inferable for all layers, the user is re-prompted before proceeding; invalid marks block progression and require correction.
5. **Encode**
   - Requirements:
     - Encoded variables must exist in the associated data source.
     - Variable types must be compatible with the selected mark and channel.
   - Behavior: Non-existent or incompatible variables trigger re-prompting, and the workflow remains at this step until all required channels are assigned valid, compatible variables.
6. **Selections & Linking**
   - Requirement: None explicitly enforced.
   - Behavior: Inferred programmatically from the user description; always completes.

**LLM Output Requirements:** For LLM-driven nodes, progression additionally requires:

- Valid JSON output conforming to the expected schema structure
- A signal that the provided information is sufficient (e.g., "Enough")

If these conditions are not met, the schema is not updated and the workflow remains at the current node until valid output is produced.

## B.2 Prompt Templates

The following prompts define the LLM interactions at each stage of the schema construction pipeline. Each node uses a task-specific prompt to elicit structured outputs that progressively populate the schema.

### B.2.1 Task Summary Node Prompt

{user} denotes the user's natural language input describing the visualization task.

---

You are given a user's description of a visualization task. Summarize in a short paragraph what this visualization is mainly about.

**Output format:**

- Return a markdown block containing the summary:

```
'''markdown
<your answer>
'''
```

**Input:**

- User description: user

---

### B.2.2 Data Node Prompt

{user} denotes the user's natural language input describing the visualization task.

---

You are an assistant that helps structure user descriptions into a predefined schema. The current task is about the data block (data sources for the visualization).

**Schema:**

```
{
  "data": {
    "source_name": {
      "type": "",
      "path": "",
      "args": {},
      "variables": []
    },
    ...
  }
}
```

**Data source types:**

- tbl (table/CSV)
- img (image/volume)
- net (network)
- geo (geographic)
- func (computed)

**Variable extraction (tbl):**

- When the user mentions variables, add each to `variables`.
- Use the exact names provided by the user.
- Do not introduce new variable names.
- Leave `variables = []` if none are specified.

**Inference rule (volume / slice / isosurface):**

- If the user requests volume, slice, isosurface, triplanar, or oblique slice:

- set `type = img`
- set `path` to the provided file/URL
- assign a short `source_name`

- Return `Enough`; do not ask for missing type/path.

**Instructions:**

1. Read the user's description: {user}
2. Fill the schema:
   - `source_name`: short identifier
   - `type`: one of tbl, img, net, geo, func
   - `path`: exact file path or URL
   - `args`: optional arguments (used for func)
3. Insert any provided file path or URL into `path`, then decide Enough vs Not Enough.
4. Provide two outputs:
   - Schema output (JSON)
   - Feedback (Markdown)
     - If complete: first line `Enough`
     - If incomplete: first line `Not Enough`, followed by a specific request

Ensure both outputs are always produced.

### B.2.3 View & Layer Node Prompt

{user}, {data_ref}, and {data_ref_with_types} denote runtime-injected values from the current schema and user input.

You are an assistant that helps structure user descriptions into a predefined schema. The current task is about **view(s) and layer** — which view(s) to create and which data source each uses.

**Available data sources (name and type):** data_ref_with_types

**You may return either:**

- **A) Single view** (when the user asks for one chart):

```
{
"view_name": "",
"layer_from": "",
"geo": ""
}
```

- **B) Multiple views** (when the user asks for more than one chart, or one view with multiple layers):

```
{
"views": [
{ "view_id": "", "layer_from": "", "geo": ""
    },
{ "view_id": "", "layers": [ { "layer_from":
    "", "geo": "" }, { "layer_from": "", "
    geo": "" } ] },
...
]
}
```

**Field rules:**

- `view_id` / `view_name`: short identifier for the view. Only use `volume_triplanar` when the user explicitly asked for triplanar slices, three slices, or XY/XZ/YZ slices. If the user said only *a slice*, *volume with a slice*, or *volume and a slice*, use `volume_slice` or `combined`, not `volume_triplanar`.
- `layer_from`: the name of a data source — use the exact name from the list above (e.g., data_ref). Do not use generic names like `data` or `sample`. When there are multiple sources of the same type (e.g., two `img`), set `layer_from` only for the view(s) the user explicitly specified. If the user said *volume with a slice from vtk and a streamline*, set `vtk` only for the volume/slice view and leave the streamline source empty.
- `layers`: use when one view contains multiple layers (e.g., volume and slice in the same view). Each layer may use the same data source. Omit `layer_from` at the view level when using `layers`.
- `geo`: optional; for geographic layers only. Leave empty if not applicable.

**Inference rules:**

- **Single data source:** when there is only one data source (data_ref), use it for `layer_from` for every view and return `Enough`.
- **One img and one tbl:** when there is exactly one `img` and one `tbl`, and the user asked for both a volume/slice/isosurface view and a chart view, infer `layer_from`: assign the `img` source to volume-like view(s) and the `tbl` source to chart view(s). Return `Enough`.
- **Multiple data sources:** only output `Not Enough` when there are multiple sources of the same type and it is unclear which source each view/layer should use.

**Volume + slice(s): combined vs. separate views**

- **"With" means one view:** phrases such as *volume with streamline*, *volume with slice*, or more generally *X with Y* mean one view with two layers from the same data source.
- **Layered means one view:** if the user says *layered*, return one view with multiple layers. Examples include layered volumes, layered streamlines, or mixed layered volumes and streamlines.
- **Multichannel volume:** if the user says *multichannel volume* or *volume using layers A, B, C*, return one view with one layer per named field, all from the same source.
- **Volume + triplanar + streamline:** if the user explicitly asks for triplanar or three slices and a separate streamline, return two views:
  - `volume_triplanar` with volume + one slice layer
  - `streamline`
- **Volume + slice(s) + streamline (separate):** if volume+slice(s) and streamline are listed separately, return two views:
  - one combined view with volume + one slice layer
  - one streamline view
- **Combined volume + slice(s):** if the user says volume with slice(s) only, return one view with two layers:
  - first layer = volume (or isosurface)
  - second layer = slice

  Use `volume_triplanar` only if triplanar is explicitly requested; otherwise use `volume_slice` or `combined`. Even if the user says *two slices* or *N slices*, still use one slice layer; slice planes are chosen later in the encode step.
- **Combined with triplanar + oblique:** if the user says volume with xy, yz, xz, and oblique slice, return one combined view with three layers:
  - volume
  - slice for triplanar
  - slice for oblique

  If the user also asks for separate slice views, add four additional slice views: `slice_xy`, `slice_oblique`, `slice_xz`, `slice_yz`.
- **Separate views:** if the user lists volume and slice(s) as separate items (without *with* or *layered*), return separate views, one per item.

**General rules:**

- If the user describes two or more distinct charts/views, return multiple views.
- Treat *interactive scatterplot* and *interactive chart* as meaning a brushable scatterplot: one view only.
- Only when the user explicitly asks for linked views should multiple linked views be returned.
- If the user wants only multiple slice planes and no volume, use one view with one slice layer; slice axes are set later.
- A *linked* slice is a separate slice view using the same data source as the linked source view.

**Map / choropleth rules:**

- If the user wants a map and both a GeoJSON and CSV source are available:

- set `layer_from` to the table (CSV)
- set `geo` to the geographic (GeoJSON) source

- **Map always includes a choropleth layer:**
  - *map* only → one choropleth layer
  - *map with hexbin*, *map with bubble*, or *map with points* → choropleth + overlay layer
  - *map with hexbin and bubble* → choropleth + hexbin + bubble

**Histogram with KDE:**

- If the user says *histogram with kde*, *histogram and kde*, *kde over histogram*, or similar, return one view with two layers:
  - histogram
  - kde

  Both layers use the same table source.

**Instructions:**

1. Read the user's description: user. If it says *interactive scatterplot* or *interactive chart*, treat that as a brushable scatterplot: one view only, no second view.
2. If the user asked for multiple views/charts, fill `views` with one object per view. If the user asked for one chart, fill `view_name`, `layer_from`, and `geo`.
3. When there are multiple data sources:
   - if there is one `img` and one `tbl` and the user asked for both volume/slice and a chart type, infer `layer_from` and return `Enough`
   - only when there is real ambiguity (e.g., two `tbl` or two `img` sources) output `Not Enough` and ask which dataset each view/layer uses
4. Provide two outputs:
   - **Schema output** (inside `json ...`): set `layer_from` only when (a) there is one data source, or (b) the user explicitly said which dataset a view uses. If there are two or more sources of the same type and the user specified only one dataset, leave the others empty.
   - **Feedback** (inside `markdown ...`): use exact dataset names (e.g., data_ref). First line `Enough` when at least one view is defined and each `layer_from` matches a data source; otherwise `Not Enough` with a specific clarification request.

Make sure to always output both parts.

---

### B.2.4 Mark Node Prompt

{user}, {schema_desc}, and {MARK_TYPES_STR} denote runtime-injected values from the current schema and user input.

---

You are an assistant that helps structure user descriptions into a predefined schema. The current task is about the **mark type(s)** (chart/graph type) for the visualization.

**Valid mark types (you MUST use only these):** MARK_TYPES_STR

**Aliases:**

- If the user says **map** (e.g. *I want a map*, *create a map*), use mark type **choropleth**.
- **Map with overlay:** when the user said **map with hexbin** (or *map with bubble*, *map with points*) and the schema has **one view with two layers**, return `view_marks` **[ ["choropleth", "hexbin"] ]** (or **["choropleth", "bubble"]** / **["choropleth", "points"]**) — the first layer must be **choropleth**; do not return only **["hexbin"]** or put geo on hexbin.
- When the user said **map with hexbin and bubble** and the schema has one view with **three** layers, return **["choropleth", "hexbin", "bubble"]**.
- If the user says **bubble** or **bubble chart**, use mark type **bubble** (x, y, and size required; color optional).
- If the user says **time series**, **timeseries**, **time series plot**, or similar (e.g. *show over time*, *trend over time*), use mark type **line** (line plot).
- For an **oblique slice** (single plane at an angle), use mark **slice**; axes will be set to `"oblique"`.
- For **triplanar**, **two slices**, **three slices**, or **multiple slice planes** in one view, use a **single** mark **slice** for that view — one layer, not multiple. The axes (e.g. **["XY", "YZ"]** or **["XY", "XZ", "YZ"]**) are set in the encode step; do not create multiple slice layers for *two slices* or *three slices*.
- **Volume with triplanar + oblique in one view:** when the schema has one view with **three** layers (volume, slice, slice), use `view_marks` so that view gets **["volume", "slice", "slice"]** — first slice layer = triplanar (XY, XZ, YZ), second slice layer = oblique.
- **Histogram with KDE:** when the user said **histogram with kde**, **histogram and kde**, **kde over histogram**, or **histogram with density** and the schema has **one view with two layers**, return `view_marks` **[ ["histogram", "kde"] ]** — first layer **histogram**, second layer **kde** (KDE curve overlaid on the same variable as the histogram).

**Do not assume a chart type.** If the user did not specify what kind of visualization they want (e.g. they only said *show variable b* or *visualization of x* without saying histogram, scatterplot, heatmap, bar chart, etc.), output `"mark": ""` (or empty `"marks"` array) and in Feedback say `Not Enough` and ask them to specify the type of visualization, listing the valid mark types above.

**The schema is defined as follows:** schema_desc

**Instructions:**

1. Read the user's description: {user}
2. Only if the user clearly specified a chart/visualization type (e.g. histogram, scatter, heatmap, bar, line, pie, volume, slice), choose the matching mark type(s) from the list above.
   - **Layered (one view, multiple layers):** When the schema has one view with multiple layers from *layered* (e.g. two volume layers + one streamline layer), return one mark per layer in order (e.g. `view_marks: [ ["volume", "volume", "streamline"] ]`).
   - **Multichannel volume:** When the user specifies multiple fields (e.g. *volume using layers X, Y, Z*), return one volume mark per layer (e.g. `["volume", "volume", ...]`).
   - **Volume + slice(s):** When one view contains volume and slice, return two marks: first `volume` (or `isosurface`), second `slice`.
   - **"Two slices" or "three slices":** Still use one slice layer (e.g. `["volume", "slice"]`), not multiple slice layers.
   - **Three-layer case:** If the schema has volume + triplanar slice + oblique slice, return `["volume", "slice", "slice"]`.
   - **Separate views:** If volume and slices are in separate views, return one mark per view (e.g. `["volume"], ["slice"], ["slice"]`).
   - **Triplanar in one view:** Use a single `slice` mark; do not return multiple slice marks.
   - **Linked slice:** A linked slice is a separate view with mark `"slice"`.
   - **Single-layer view:** If a view has one layer, return `"mark"` as a string.
3. Provide **two outputs**:
   - **Schema output** (inside `json ...`)
   - **Feedback** (inside `markdown ...`)
     - First line `Enough` when valid marks are provided
     - Otherwise `Not Enough` with a request to specify the visualization type

Make sure to always output both parts.

---

### B.2.5 Encode Node Prompt

{user_intent_block}, {data_info}, and {refinement_context_block} denote runtime-injected values from the current schema and user input. {no_encode_marks} specifies mark types that do not require encoding.

---

You are an assistant that helps structure user descriptions into a predefined schema. The current task is about the **encode** block — which variables (fields) the user wants on which visual channels.

**Barrier rules:**

- You may ONLY use variable names from the allowed list above. If the list is empty, output `"encode":` and say `Not Enough`.
- Do not infer or guess encode from the data. Only fill encode when the user has explicitly said which variables to use (e.g. *use c and h*, *x is sales, y is revenue*, *plot column A vs B*).
- If the user only described the chart type (e.g. *I want a scatterplot*) without naming which columns/variables to use, output `"encode":` and say `Not Enough`.
- In your JSON output, include ONLY encode keys for channels the user explicitly assigned.
- Example: if the user said *x is sales, y is revenue* and *sales*, *revenue* are allowed, output `"encode": "x": "sales", "y": "revenue"`. Do not add `color` or any other key they did not specify.

**User intent block (when provided):** user_intent_block

**Context:** data_info

**Refinement context (when provided):** refinement_context_block

**Exception — encode step can be bypassed for these marks:** no_encode_marks
These display the whole dataset and do not map table columns to x/y/color. For these mark types, output `Enough` with an empty encode object.

**Output format:**

- Single view with one layer needing encode: `"encode": <only channels user assigned>`
- Single view with multiple layers needing encode, or multiple views: `"encodes": [ [ { ... }, { ... } ], ... ]`
  - one element per view
  - each element is an array of encode objects
  - one encode object per layer that needs encode, in layer order
- Example: one view with 2 histogram layers → → `"encodes": [ [ { "x": "s" }, { "x": "t" } ] ]`
- For a view whose layers are all no_encode_marks, use `[ {} ]`.

**Instructions:**

1. If single view and mark type is one of no_encode_marks, output empty encode and first line `Enough`.
2. If multiple views: for each view with mark in no_encode_marks, use for that view's encode. Only request or fill encode for views that need variable mapping (points, bar, line, etc.).
3. If allowed variable names is empty and at least one view needs encode, output empty encodes for those views and first line `Not Enough`; ask the user to specify which columns/variables to use (only for the view(s) that need encode, not for volume/slice/isosurface views).
4. If the user did not specify which variable goes on which channel for a view that needs encode, do not guess and do not pick from the allowed list — output empty encode for that view and say `Not Enough`, asking the user to choose which variables to use (only for that view).
5. **Underspecification:** If a view needs multiple variables (e.g. scatterplot/points needs x and y) but the user only specified one variable for that view, output empty encode for that view and say `Not Enough`, naming which view is underspecified, what is required (e.g. `x`, `y`), and what is optional for this mark if any (e.g. color). Example: *Scatterplot view: required encodings are x and y; optional encodings are color. You specified one variable. Please specify the other required variable(s).*
6. Use only the required and optional channels listed in Context for this mark. Include optional channels (e.g. color, size, opacity for scatterplot) only if the user explicitly assigned them.
7. Provide **two outputs**: Schema (inside `json ...`) and Feedback (inside `markdown ...`). First line of feedback: `Enough` when every view that needs encode has all required channels filled with names from the allowed list (and volume/slice/isosurface views have empty encode); otherwise `Not Enough`.

Make sure to always output both parts.

---

### B.2.6 Selections & Linking Node Prompt

{user}, {view_ids}, and {data_sources} denote runtime-injected values from the current schema and user input.

---

You are an assistant that helps structure user descriptions into a predefined schema. The current task is about **selections and linking** — whether the user wants to link views so that selecting in one view updates another.

**View IDs from previous step:** ", ".join(view_ids) or "(none)" **Data sources:** ", ".join(data_sources) or "(none)"

**Interpretation:** Treat **interactive scatterplot** and **interactive chart** as meaning **brushable scatterplot** (one brushable view).

If the user asked to **link** views, **brush/select**, or an **interactive** (brushable) chart so that selecting in one view updates another, fill in the schema below. Use the exact `view_id` names and the exact field names the user said for `bind_channels`.

**Linking works by data rows, not by variables:** Linked views can use different axes (e.g. one scatter a vs b, another scatter c vs d) as long as they use the same data source. The selection identifies rows; the other view highlights/filters those same rows even if it plots different columns. So `shared_data_source` is the key; `bind_channels` are only for the view where the user brushes (`bind_view`).

If the user did NOT ask for linking, brushing, or an interactive/brushable chart, return:

```
{ "selections": [], "linking": {} }
```

**Schema when linking is requested:**

```
{
"selections": [
{
"name": "<selection_name>",
"type": "interval",
"bind_view": "<view_id where user brushes>",
"bind_channels": ["<x_field>", "<y_field>"]
}
],
"linking": {
"shared_data_source": "<data source name
    used by all linked views>",
"linked_view_ids": ["<view_id1>", "<view_id2
    >"],
"selection_name": "<same as selections[0].
    name>"
}
}
```

**Instructions:**

1. Read the user's description: user
2. If they want linked views with brush/selection: fill `selections` (name, type `"interval"`, `bind_view`, `bind_channels` for that view only) and `linking` (`shared_data_source`, `linked_view_ids`, `selection_name`). Linked views may use different encode channels (e.g. scatter a vs b and scatter c vs d from the same table) — same data source is sufficient. Use exact `view_id` and field names from the task.
3. If they did not ask for linking: return `{ "selections": [], "linking": {} }` and first line `Enough`.
4. Provide **two outputs**:
   - **Schema output** (inside `json ...`)

- **Feedback** (inside `markdown ...`). First line Enough in both cases.

Make sure to always output both parts.

---

## B.3 Agent Clarification Messages

When Raiven cannot proceed with a workflow step due to missing or ambiguous information, it pauses and returns a clarification message to the user rather than making an uninformed assumption. Below we document what a user will see at each stage of the pipeline when clarification is needed.

**Task Definition.** This stage always completes successfully and never requests clarification from the user.

**Data Block.** If the dataset to use is unclear, the user sees:

```
Please specify the dataset (file path or URL) you want to
use.
```

**View & Layer.** If the user has multiple datasets and has not specified which dataset each view should use:

```
You have multiple datasets:  [names].  Each view uses
exactly one dataset; multiple views can share the same
dataset.  Please specify which dataset each view uses.
```

With a single dataset:

```
Please specify which dataset this view should use (or
multiple views can share the same dataset).
```

If ambiguity persists after the LLM has inferred views, Raiven provides a structured per-view breakdown:

```
You have multiple image data sources:  [names].
Which dataset should each view use?
- View 'spatial' (volume + triplanar slices):  ct_1.vti
```

**Mark.** If the user has not specified a visualization type:

```
You must specify the type of visualization (chart type)
you want.  Supported types:  [supported types].  For
example:  “I want a histogram of b”, “make a scatterplot
of x and y”, “show a heatmap of a, b, and c”.
```

If a chart type was specified but is not supported:

```
[LLM explanation].  The mark type must be one of:
[supported types].
```

**Encode.** The encoding stage produces the most detailed clarification messages, as it must resolve all variable-to-channel assignments. If the mark and data combination is invalid:

```
The variable encoding underspecified check is not
satisfied.
[per-view error details]
```

or:

```
This combination isn't allowed:
[per-view error details]
```

When encoding information is incomplete, Raiven returns a structured per-view breakdown of what has been specified and what is still needed:

```
I need a few more details to complete.
View 'main' (line):  Already specified:  y (all numerical
variables).  Still needed:  x (numerical or categorical).
Optionally:  color (categorical).  Please specify the x
variable (e.g. 'along d', 'plot them along x', or 'x is
d').
```

Available variable names from the loaded dataset are listed directly below each view block. For choropleth views, additional guidance on providing a matching GeoJSON boundary file is included.

**Selections & Linking.** This stage always completes successfully and never requests clarification from the user, as linking is inferred programmatically from the visualization structure.

## B.4 NL to Schema to RaivenDSL

We illustrate the end-to-end pipeline from natural language input to schema construction and final RaivenDSL generation. Given the datasets `stats.csv` and `tg9.vti`, the user provides the following request:

> "I want to see a volume rendering of vorticity layered with streamlines using ux, uy, and uz. Additionally, I want to see a histogram of vorticity."

This input corresponds to the visualization shown in Fig. 1. It is first translated into the intermediate schema representation (Fig. B.4), which captures the data sources, view structure, and encodings. The corresponding RaivenDSL program generated from this schema is shown in Listing 2.

```
{
  "task_summary": "The visualization focuses on exploring
     fluid dynamics by displaying a volume rendering of
     vorticity. It also includes a histogram of vorticity
     values to show the distribution and magnitude of
     rotational intensity within the dataset.",

  "data": {
    "stats.csv": {
      "type": "tbl",
      "path": "/var/folders/n4/
     vzm78dy505g2qkxwbtf2bh780000gp/T/vis_upload_ef_a9qzq/
     stats.csv",
      "args": { "format": "csv" },
      "variables": [
        { "name": "ux", "data_type": "number" },
        { "name": "uy", "data_type": "number" },
        { "name": "uz", "data_type": "number" },
        { "name": "vorticity", "data_type": "number" },
        { "name": "pp", "data_type": "number" },
        { "name": "critq", "data_type": "number" }
      ]
    },
    "tg9.vti": {
      "type": "img",
      "path": "/var/folders/n4/
     vzm78dy505g2qkxwbtf2bh780000gp/T/vis_upload_ef_a9qzq/
     tg9.vti",
      "args": { "format": "vti" },
      "variables": [
        { "name": "critq", "data_type": "number" },
        { "name": "pp", "data_type": "number" },
        { "name": "ux", "data_type": "number" },
        { "name": "uy", "data_type": "number" },
        { "name": "uz", "data_type": "number" },
        { "name": "vorticity", "data_type": "number" }
      ],
      "dimensions": [65, 65, 65]
    }
  },

  "views": [
    {
      "view_id": "volume",
      "layers": [
        {
          "from": "tg9.vti",
          "geo": "",
          "mark": "volume",
          "encode": {
            "field": "vorticity"
          },
          "style": {}
        }
      ],
      "links_out": [],
      "interactions": {}
    },
    {
      "view_id": "histogram",
```

```
      "layers": [
        {
          "from": "stats.csv",
          "geo": "",
          "mark": "histogram",
          "encode": {
            "x": "vorticity"
          },
          "style": {}
        }
      ],
      "links_out": [],
      "interactions": {}
    }
  ],

  ""selections"": [],
  ""linking"": {}
}
```

## C Compiler

This section details the two-stage compiler pipeline through the running example from Figure 1. The compiler translates a parsed RaivenDSL AST into executable visualization code through a verification stage (Section C.1) and a realization stage (Section C.2).

### C.1 Verification Stage

The verification stage transforms the raw AST into a typed `ProgramSpec`, validating structural constraints independently of any rendering backend. The key transformations are:

- Data constructors are resolved to typed kinds: `img` → `ImageData`, `tbl` → `Table`, `net` → `Network`, `geo` → `GeoJSON`.
- Each layer receives a unique identifier of the form `viewId:markType#index`.
- Data source references (`from`) are validated against the data block.
- Mark types are checked against the supported set.
- Selection and link declarations are validated for referential integrity.

For the running example, the AST:

```
{
  "kind": "Program",
  "data": {
    "vol":    { "ctor": "img", "args": { "path": "
     taylorgreen_9.vti", "format": "vti" }},
    "sample": { "ctor": "tbl", "args": { "path": "
     tg9_sample.csv", "format": "csv" }}
  },
  "views": [
    { "kind": "View", "id": "volume_streamline",
      "layers": [
        { "kind": "Layer", "from": "vol", "mark": "volume",
          "encode": { "field": "vorticity" }},
        { "kind": "Layer", "from": "vol", "mark": "
     streamline",
          "encode": { "vx": "ux", "vy": "uy", "vz": "uz" }}
      ]},
    { "kind": "View", "id": "histogram",
      "layers": [
        { "kind": "Layer", "from": "sample", "mark": "
     histogram",
          "encode": { "x": "vorticity" }}
      ]}
  ]
}
```

is lowered to the following `ProgramSpec`:

```
{
  "data": {
    "vol":    { "ctor": "DataLoader", "kind": "ImageData",
                "args": { "path": "taylorgreen_9.vti", "
     format": "vti" }},
    "sample": { "ctor": "DataLoader", "kind": "Table",
                "args": { "path": "tg9_sample.csv", "format":
      "csv" }}
  },
  "views": [
    { "id": "volume_streamline",
      "layers": [
        { "id": "volume_streamline:volume#0",
          "kind": "volume", "from": "vol",
          "encode": { "field": "vorticity" }},
        { "id": "volume_streamline:streamline#1",
          "kind": "streamline", "from": "vol",
          "encode": { "vx": "ux", "vy": "uy", "vz": "uz" }}
      ]},
    { "id": "histogram",
      "layers": [
        { "id": "histogram:histogram#0",
          "kind": "histogram", "from": "sample",
          "encode": { "x": "vorticity" }}
      ]}
  ]
}
```

Gray fields highlight the verification stage's contributions: data kinds are resolved from constructors, and each layer receives a unique identifier. The user's specification is otherwise preserved unchanged. At this stage, the compiler has confirmed that all data references resolve, all marks are valid, and all encodings are structurally well-formed—without knowing which backend will render each view.

### C.2 Realization Stage

The realization stage assigns a rendering backend to each view, resolves all defaults, and produces the `RenderIR`: the final, fully specified representation consumed by code generation. The key transformations are:

- Backend assignment: views are routed to `vtkjs` or `d3` based on their layer mark types. When a program contains views targeting both backends, the top-level backend is set to `multi`.
- Default resolution: color transfer functions, opacity profiles, sample distances, palettes, bin counts, fill colors, and all other style properties are resolved to concrete values.
- Control generation: for each mark, the compiler produces the set of runtime-adjustable parameters with their ranges and defaults.
- Data URLs are resolved to their runtime paths.

For the running example, the `ProgramSpec` is realized to:

```
{
  "backend": "multi",
  "views": [
    {
      "viewId": "volume_streamline",
      "backend": "vtkjs",
      "layers": [
        { "type": "volume",
          "id": "volume_streamline:volume#0",
          "field": "vorticity",
          "url": "/data/teaser/taylorgreen_9.vti",
          "sampleDistance": 0.7,
          "range": [0, 28.82],
          "ctf": [{"r":0.27,"g":0.00,"b":0.33,"s":0}, ...],
          "otf": [{"a":0,"s":0}, {"a":0.3,"s":10.09},
                  {"a":0.9,"s":28.82}],
          "_palette": "viridis"
        },
        { "type": "streamline",
          "id": "volume_streamline:streamline#1",
          "encode": { "vx": "ux", "vy": "uy", "vz": "uz" },
          "url": "/data/teaser/taylorgreen_9.vti",
          "integrator": { "step": 0.5, "max_steps": 1000 },
          "seedSpec": { "n": 100,
                        "region": { "type": "box" }},
          "style": { "color": null, "tubeRadius": null }
        }
      ],
      "controls": {
        "sampleDistance": { "min":0.1, "max":2,
                            "default":0.7, "step":0.01 },
        "palette": "viridis",
        "ctf_stops": [ ... ],
        "otf_stops": [ ... ],
        "layerControls": {
          "volume_streamline:streamline#1": {
            "streamline": {
              "color": "#ffffff",
              "count": 100,
              "integrationStep": 0.5,
              "maxSteps": 1000,
              "tubeRadius": 0,
              "seedBoxX": { "min":0, "max":65 },
              "seedBoxY": { "min":0, "max":65 },
              "seedBoxZ": { "min":0, "max":65 }
            }
          }
```

```
        }
      }
    },
    {
      "viewId": "histogram",
      "backend": "d3",
      "layers": [
        { "type": "histogram",
          "id": "histogram:histogram#0",
          "data": "/data/teaser/tg9_sample.csv",
          "encoding": { "x": "vorticity" }
        }
      ],
      "controls": {
        "colors": {
          "histogram:histogram#0": {
            "bins": 30,
            "fill_color": "#1f77b4"
          }
        },
        "palette": "viridis"
      }
    }
  ]
}
```

Gray fields highlight the realization stage's contributions: backend assignment, resolved defaults, and generated control specifications. The scalar range is extracted from the dataset (`range: [0, 28.82]` for vorticity) and used to position CTF and OTF stops within the data domain. Seed region control ranges are derived from the image dimensions (`seedBox: {min:0, max:65}` per axis), though the actual seed bounds are resolved at mount time after data loading. The volume layer's two-line DSL specification—a data source and a mark type—has expanded to a full rendering configuration; the streamline layer has acquired integrator and seed parameters; the histogram has received a default bin count and fill color. Bold fields denote user-specified values that pass through unchanged. This is the final representation consumed by the D3 and VTK.js code generators.

## D BENCHMARK

### D.1 Datasets

As referenced by Section 7. The benchmark uses real-world datasets across all three prompt categories. Tables 13, 14, and 15 summarize each dataset's format, dimensions, and source. CSV and JSON files are embedded in full in the model prompt; VTI and GeoJSON files are represented by metadata headers only and provided by URL for runtime loading.

#### D.1.1 Scientific Visualization Datasets

Four volumetric datasets provide the spatial data for SciVis and Combined prompts.

Table 13: Scientific visualization datasets (VTI format).

| Dataset | Dimensions | Source |
|---|---|---|
| `head.vti` | 64×64×93, 1 var. | Kitware[4] |
| `cells.vti` | 256×256×71, 6 vars. | Allen Inst.[5] |
| `divcurl_{t}.vti` | 2048×1024×1, 4 vars., 20 t | ORNL |
| `taylorgreen_{t}.vti` | 65×65×65, 6 vars., 10 t | ORNL |

#### D.1.2 Combined Datasets

Combined prompts pair volumetric data with tabular CSV extracts derived from the same sources. Each CSV provides a statistical summary or random sample of the parent volume, enabling InfoVis marks alongside SciVis marks in coordinated dashboards.

Table 14: Combined tabular datasets (CSV), derived from the volumetric datasets in Table 13.

| File | Rows | Description |
|---|---|---|
| *From* `head.vti` | | |
| `head_sample.csv` | 15k | Random CT intensities |
| `head_region_stats.csv` | 27 | 3×3×3 block stats |
| *From* `divcurl_{t}.vti` | | |
| `divcurl_sample.csv` | 4k | Raw field values (t = 10) |
| `divcurl_stats.csv` | 20 | Per-timestep summaries |
| `divcurl_binned_2d.csv` | ~2.5k | 50×50 spatial bins |
| *From* `taylorgreen_{t}.vti` | | |
| `tg_sample.csv` | 4k | Raw field values (t = 9) |
| `tg_stats.csv` | 10 | Per-timestep summaries |
| `tg_t9_binned.csv` | 512 | $8^3$ block averages |
| *From* `cells.vti` | | |
| `cells_channel_sample.csv` | 15k | Co-sampled channels |
| `cells_seg_stats.csv` | 6 | Segmentation stats |
| `cells_per_seg_sample.csv` | ~6k | Per-compartment samples |

#### D.1.3 Information Visualization Datasets

InfoVis prompts use tabular, network, and geospatial datasets drawn from public sources.

**Country statistics.** The primary tabular data source is a merged dataset combining World Bank Development Indicators[6] with UN Population Division median age estimates.[7] Three tabular views are provided: `countries_timeseries.csv` (13,760 rows; one row per country per year, 1960–2023), `countries_latest.csv` (215 rows; most recent snapshot per country), and aggregated files at continent, UN sub-region, and World Bank trade-region levels. An energy breakdown file (`countries_energy.csv`, 645 rows) provides long-format renewable/nuclear/fossil shares. A world life-expectancy aggregate (`world_life_exp.csv`, 64 rows) provides population-weighted global averages from 1960–2023.

**Trade flows.** Regional trade data (`trade_edges.csv`, 56 rows) reports flows between seven World Bank regions, sourced from the World Integrated Trade Solution (WITS).[8]

**Border network.** Land border data (`borders.csv`, 203 rows; `border_edges.csv`, 313 edges) is derived from Wikipedia's list of land borders.[9] Network JSON files (`border_network.json`, `trade_network.json`) provide node-link representations for force-directed graph and Sankey visualizations.

**Capital cities.** Coordinates and populations for primary capital cities are from SimpleMaps World Cities.[10]

**Geospatial.** GeoJSON boundary files are derived from Natural Earth Admin-0 boundaries,[11] split into per-continent, per-region, and per-trade-region subsets. A full world boundary file (`world_iso3.geojson`) is also provided.

**Gaia star catalog.** A random sample of ∼74,300 sources from ESA Gaia DR3[12] provides right ascension, declination, G-band magnitude, parallax, and BP–RP colour for scatter plots and sky maps.

Table 15: Information visualization datasets.

| File | Fmt | Rows | Source |
|---|---|---|---|
| `countries_timeseries` | CSV | 13.8k | WB, UN |
| `countries_latest` | CSV | 215 | WB, UN |
| `world_life_exp` | CSV | 64 | Derived |
| `countries_energy` | CSV | 645 | WB |
| `*_latest/ts/energy` | CSV | var. | Aggreg. |
| `borders` | CSV | 203 | Wikipedia |
| `border_edges` | CSV | 313 | Wikipedia |
| `trade_edges` | CSV | 56 | WITS |
| `border_network` | JSON | 164n | Derived |
| `trade_network` | JSON | 8n | Derived |
| `world_iso3` | GeoJ | 249f | Nat. Earth |
| `gaia` | CSV | 74.3k | ESA Gaia |

### D.2 Baseline Configuration

As referenced by Section 7. The four baselines use the model versions listed in Table 16. The three LLM baselines use **temperature** = 0, enforcing deterministic (greedy) decoding across every trial.

Table 16: Model version strings used in the benchmark.

| **Baseline** | **Model ID** |
|---|---|
| ChatGPT | `gpt-5.4` |
| Claude | `claude-opus-4-6` |
| Gemini | `gemini-3.1-pro-preview` |
| Raiven | `gpt-5.2-chat-latest` |

**Data Delivery.** Data delivery differs across baselines, as each API exposes a distinct interface for attaching non-text content, as summarised in Table 17. Data files are attached via each API's native file-upload mechanism alongside the benchmark prompt text.

**Context Size Limits and Truncation.** The output parameters applied to ChatGPT, Claude, and Gemini are given in Table 18. Per-file-type truncation behaviour is as follows:

[4] Kitware, Inc. (2016). VTK.js sample data. https://github.com/Kitware/vtk-js. BSD-3-Clause.

[5] Allen Institute for Cell Science (2018). hiPSC Single-cell Image Dataset [AICS-10_8]. https://allencell.org/3d-cell-viewer.

[6] The World Bank: World Development Indicators. https://data.worldbank.org. CC BY 4.0.

[7] United Nations, Dept. of Economic and Social Affairs, Population Division (2024). World Population Prospects 2024. https://population.un.org/dataportal. CC BY 3.0 IGO.

[8] World Bank WITS. https://wits.worldbank.org. World Bank license.

[9] Wikipedia: List of countries and territories by number of land borders. CC BY-SA 4.0.

[10] SimpleMaps. https://simplemaps.com/data/world-cities. Basic license.

[11] Natural Earth. https://naturalearthdata.com. Public domain.

[12] ESA Gaia mission. https://www.cosmos.esa.int/web/gaia. CC BY-NC 3.0 IGO. This work has made use of data from the European Space Agency (ESA) mission Gaia, processed by the Gaia Data Processing and Analysis Consortium (DPAC).

Table 17: Data delivery mechanism per baseline.

| Baseline | API Surface | Mechanism |
|---|---|---|
| ChatGPT | OpenAI Responses API | User message composed of `input_text` parts and `input_file` parts encoded as base64 data URLs. System prompt passed via `instructions`. |
| Claude | Anthropic Messages API (streaming) | User message composed of `text` blocks and `document` blocks (`text/plain` source). System prompt passed via `system`. |
| Gemini | Google Gen AI | Files $\leq$20 MB sent inline via `Part.from_bytes`; files $>$20 MB uploaded via the Files API and referenced by URI via `Part.from_uri`. System prompt passed via `system_instruction`. |
| Raiven | OpenAI Responses API | Files downloaded to local temporary paths and delivered as absolute local filesystem paths to the Raiven session. |

Table 18: Output token parameters.

| Maximum output tokens | Value | Purpose |
|---|---|---|
| Claude | 32,768 | Set explicitly via `max_tokens`. |
| Gemini | 1,000,000 | Set via `max_output_tokens`. |
| ChatGPT | Not set explicitly | Governed by Responses API default. |

- **VTI:** XML header only, truncated at the `<AppendedData` element. The full binary data is injected into the generated HTML at post-processing time.
- **GeoJSON:** Full content sent as a plaintext attachment.
- **CSV / JSON:** Full content embedded as a file attachment.

### D.3 LLM Prompt

As referenced by Section 7. All LLM baselines receive the system prompt below, instructing them to return a single self-contained HTML file. It is passed via the native system-prompt parameter of each API (`instructions` for ChatGPT, `system` for Claude, `system_instruction` for Gemini; Raiven handles it internally).

```
You are an expert data visualization developer.  Given
a user's visualization request and any attached data
files, generate a complete, self-contained HTML file that
renders the visualization.

Requirements:
- Return only the raw HTML.
- For CSV and JSON data:  embed all data directly in the
HTML. Do not fetch external data files.
- For .vti data:  load files using fetch('./FILENAME')
from the same directory.
- Ensure the page renders correctly with basic structure
(HTML, head, body).

Output a single complete HTML file and nothing else.
```

### D.4 VLM Scoring Prompt

To evaluate VMPC automatically, we prompted a VLM to act as a visualization evaluator. For each submission, the VLM was provided with four inputs: (1) the original prompt given to the model under evaluation, (2) the HTML source code of the generated output, (3) a rendered screenshot of that output, and (4) $N$, the number of views the prompt requested. The VLM was then asked to score each binary criterion and return a structured score table with per-criterion justifications. The full prompt given to the VLM is reproduced below.

---

**VMPC Scoring Prompt — VLM Evaluation of Generated HTML Visualizations**

You are an expert visualization evaluator. You will be given:

1. A **prompt** that was given to an LLM-based visualization system.
2. The **HTML source code** of the output that system produced.
3. A **rendered view** of that HTML, which you can interact with (click, hover, brush, select).
4. **N**, the number of views the prompt requested.

Your job is to evaluate the output by scoring each component described below.

**Core Principles**

- **Everything is grounded in the prompt.** Score based on what the prompt asked for, not what you think a good visualization should look like.
- **Overcompletion is fine.** Extra views, annotations, or interactions beyond what the prompt asked for are neither penalized nor credited.
- **N is provided to you.** It is the number of views the prompt requested. If the output has more views than $N$, identify which views correspond to the $N$ prompted ones and disregard the extras. If fewer, the missing views get all zeros.
- **When in doubt, score 0.** VMPC is a minimum compliance metric. Do not give the benefit of the doubt.

**Compilation Gate**

$X$ — **Compilation (binary: 0 or 1).** Does the HTML compile and render at least one view?

- **1**: The page loads and at least one visualization view is visible on screen — any chart, plot, or map with allocated space.
- **0**: The page shows only a permanent loading spinner, an error/stack trace, a completely blank/white page, or zero views render.

If $X = 0$, the entire score is 0 regardless of all other components.

**Per-View Scores**

For each of the $N$ views, evaluate the following five criteria.

*$V_v$ — View Existence (binary: 0 or 1).* Does view $v$ have allocated screen space for its intended visualization purpose?

- **1**: There is a region on the page dedicated to this view — a visible outline, frame, axes, container, panel, or box that indicates a visualization is intended there. An empty box or placeholder with allocated space counts as $V = 1$.
- **0**: No region exists on the page for this view. It is completely absent from the layout.

If $V = 0$, then $M$, $E$, and $H$ for that view are all automatically 0.

*$M_v$ — Mark Type (binary: 0 or 1).* Does view $v$ actually render visible marks of the correct type?

- **1**: The view contains visible, rendered marks that match what the prompt specified — bars for "bar chart", points for "scatter plot", lines for "line chart", a 3D surface for "isosurface", a 3D volume for "volume rendering", etc. The marks must be actually drawn on screen, not just implied by code or axes.
- **0**: No marks are rendered (the view is blank, shows only axes with no data, shows a loading spinner, or displays an error message), OR the wrong mark type is used.

**Critical:** Do not give $M = 1$ based on what the code *intends* to render — only what is *actually visible* in the rendered output. Domain-specific checks include:

- **LIC**: A correct LIC shows coherent, directional flow-structure textures. Uniform random noise/static with no directional structure means the LIC computation failed: $M = 0$.
- **Volume rendering**: Should show a 3D volumetric object with depth, transparency, and shading.
- **Streamlines**: Should show curved lines or tubes following flow directions.

*$E_v$ — Encoding (binary: 0 or 1).* Are the visual encodings actually visible and correct for view $v$ as specified in the prompt?

- **1**: The encodings the prompt specified are correct AND visible — the right variables on the right axes with actual data shown, correct color mapping applied to visible marks, etc.
- **0**: Encodings are wrong, OR the view has no visible data to evaluate encodings on (blank, loading, error), OR axes show wrong variables, OR a requested encoding is missing.

**Critical:** If a view shows no data, $E = 0$ regardless of what the code says. Only evaluate encodings the prompt explicitly specifies.

*$H_v$ — Data Hallucination (binary: 0 or 1).* Is the data in view $v$ real and sourced correctly, or is it hallucinated/fabricated? Evaluate by inspecting both the code and the rendered output:

- Check all `fetch()`, `d3.csv()`, `d3.json()` calls, inline data arrays, and `<script>` blocks. Does the code load from the source the prompt specified?
- **Critical — check for try/catch fallback patterns.** Code that tries to fetch real data but falls back to `Math.random()`, `Math.sin()`, hardcoded arrays, or synthetic generation in the `catch` block is $H = 0$. The correct fetch URL being present does *not* mean real data was used.
- Cross-check with the rendered output. Look for text saying "fallback", "sample data", or "generated", and for data patterns that look procedurally generated (perfect sine waves, uniformly random distributions).
- **1 (Real data)**: The code loads from the correct source, there is no synthetic fallback, and the rendered content is consistent with that dataset.
- **0 (Hallucinated)**: Data is fetched from a different source, fabricated, generated via a try/catch fallback, or inconsistent with the specified dataset.

If $V = 0$, then $H = 0$ automatically. If $V = 1$ but $M = 0$ (view exists but renders no marks), then $H = 1$ — a view that shows no data cannot be hallucinating data.

*$L_v$ — Linking (binary: 0 or 1).* Does view $v$ participate in cross-view linking as requested by the prompt?

- If the prompt does **not** request linking for this view, $L = 1$ automatically (not applicable).
- If linking is requested, inspect the code for event listeners, shared state, and dispatch/callback patterns connecting views, and test the interaction by clicking, hovering, or brushing.
- **1**: Linking works as specified, or no linking was requested.
- **0**: Linking was requested but is broken, missing, or produces no visible cross-view response.

**Output Format**

Provide your evaluation as a raw score table. $N$ is provided to you — score each of the $N$ views in order.

```
PROMPT SUMMARY: [1-2 sentence summary of what the
prompt asked for]

N = [provided value]

X (Compilation): [0 or 1] - [brief justification]

| view | V | M | E | H | L |
|------|---|---|---|---|---|
| v1   | _ | _ | _ | _ | _ |
| v2   | _ | _ | _ | _ | _ |
| ...  | _ | _ | _ | _ | _ |

JUSTIFICATIONS:
v1: [1-2 sentences explaining the scores for this view]
v2: [1-2 sentences explaining the scores for this view]
...
```

All scores are binary: 0 or 1. If $X = 0$, fill the entire table with zeros. If $V = 0$ for a view, $M$, $E$, and $H$ are all 0. If $V = 1$ but the view is blank, loading, or errored, then $M = 0$, $E = 0$, but $H = 1$. $L = 1$ if no linking was requested for this view.

---

## D.5 VMPC Scoring Examples

As referenced by Section 7.2. We illustrate VMPC scoring through two worked examples. First, we walk through how a grader would evaluate four different models' outputs for the same prompt (case 73), demonstrating how the metric behaves across a range of partial and full compliance. Second, we show a case where the execution gate $X = 0$ short-circuits scoring entirely. In all cases, scores reflect the majority judgment of three independent graders, with each binary criterion requiring agreement from at least two of the three.

### D.5.1 Case 73: Multi-View Compliance Across Four Models

The prompt for case 73 reads:

> *"Render the CT isosurface in pink from* `head.vti`*, then summarize* `head_sample.csv` *using a histogram of 'intensity' colored pink."*

This prompt specifies $N = 2$ views. No cross-view linking is requested, so $L_v = 1$ by default for both views across all four model outputs. A grader evaluating any response to this prompt would first check whether the code runs and produces visible output (execution gate $X$), then assess each view in turn against the five binary criteria: view presence $V_v$, mark correctness $M_v$, encoding correctness $E_v$, data hallucination $H_v$, and cross-view linking $L_v$.

**Model 1: Raiven (VMPC = 1.0).** The code executes without error and produces a visible output, so $X = 1$.

**View 1 (CT Isosurface).** A clearly rendered 3D surface is present in the expected panel, so $V_1 = 1$. The output uses an isosurface rendering as requested rather than a volume rendering or point cloud, so $M_1 = 1$. The surface displays a consistent pink hue with no unexpected color mapping, so $E_1 = 1$. The geometry is consistent with a human head CT scan loaded from `head.vti` rather than a substitute or fabricated mesh, so $H_1 = 1$. As no linking was requested, $L_1 = 1$.

**View 2 (Intensity Histogram).** A histogram panel is clearly visible alongside the 3D view, so $V_2 = 1$. The mark type is a binned bar chart as expected, so $M_2 = 1$. The x-axis encodes the `intensity` variable from `head_sample.csv`,

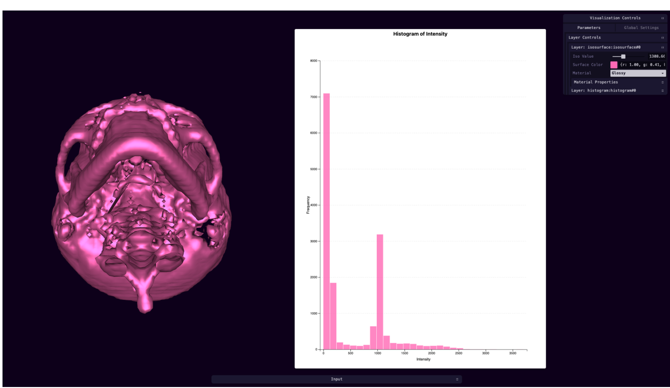



Fig. 8: Case 73 output from Raiven: a pink CT isosurface alongside a pink histogram of CT intensity values.

Fig. 10: Case 73 output from Claude: the isosurface panel renders only an empty container, while the pink intensity histogram is fully correct.

the y-axis encodes frequency counts, and the bars are colored pink as specified, so $E_2 = 1$. The distribution shape and value range are consistent with CT intensity data from the provided file rather than synthetic values, so $H_2 = 1$. No linking was requested, so $L_2 = 1$.

**Calculation.** All ten binary criteria are satisfied:

$$\text{VMPC} = 1 \cdot \frac{1}{5 \times 2} \left( \underbrace{(1+1+1+1+1)}_{\text{View 1}} + \underbrace{(1+1+1+1+1)}_{\text{View 2}} \right) = \frac{10}{10} = 1.0$$

All three graders assigned full credit to every criterion, reflecting clear and unambiguous compliance with the prompt across both views.

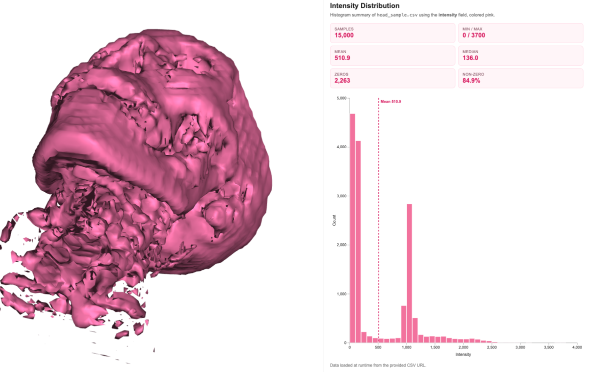



Fig. 9: Case 73 output from ChatGPT: a pink CT isosurface alongside a pink histogram of CT intensity values, with additional unrequested summary statistics.

**Model 2: ChatGPT (VMPC = 1.0).** The code executes without error, so $X = 1$. Both views satisfy all five criteria by the same reasoning as Raiven: the isosurface is present, correctly typed, and pink ($V_1 = M_1 = E_1 = H_1 = L_1 = 1$), and the histogram correctly encodes `intensity` with pink bars ($V_2 = M_2 = E_2 = H_2 = L_2 = 1$).

The grader also notices that the output includes additional summary statistics — such as mean and standard deviation annotations — that were not requested by the prompt. Importantly, VMPC does not penalize unrequested additions; the metric evaluates only whether what the prompt asked for is present and correct. These extra elements are ignored during scoring.

**Calculation.** All ten criteria are satisfied:

$$\text{VMPC} = 1 \cdot \frac{1}{5 \times 2} \left( \underbrace{(1+1+1+1+1)}_{\text{View 1}} + \underbrace{(1+1+1+1+1)}_{\text{View 2}} \right) = \frac{10}{10} = 1.0$$

**Model 3: Claude (VMPC = 0.8).** The code executes without error, so $X = 1$.

**View 1 (CT Isosurface).** A panel container for the isosurface is present and visible, so $V_1 = 1$. However, no isosurface geometry appears inside it — the panel renders but is empty. Because no mark is visible, the grader cannot confirm the correct mark type was used, so $M_1 = 0$. Similarly, with no rendered geometry, color, channel assignments, and variable mappings are all unverifiable, so $E_1 = 0$. Crucially, the absence of any rendered geometry also means no data could have been fabricated or substituted, so $H_1 = 1$. No linking was requested, so $L_1 = 1$.

**View 2 (Intensity Histogram).** The histogram is fully correct: the panel is present ($V_2 = 1$), the mark type is a binned bar chart ($M_2 = 1$), the x-axis encodes `intensity` with pink bars and the y-axis encodes frequency counts ($E_2 = 1$), the values are consistent with `head_sample.csv` ($H_2 = 1$), and no linking was requested ($L_2 = 1$).

**Calculation.** Eight of ten criteria are satisfied. The two deductions stem directly from the empty isosurface panel: without a visible mark, neither $M_1$ nor $E_1$ can be awarded.

$$\text{VMPC} = 1 \cdot \frac{1}{5 \times 2} \left( \underbrace{(1+0+0+1+1)}_{\text{View 1}} + \underbrace{(1+1+1+1+1)}_{\text{View 2}} \right) = \frac{8}{10} = 0.8$$

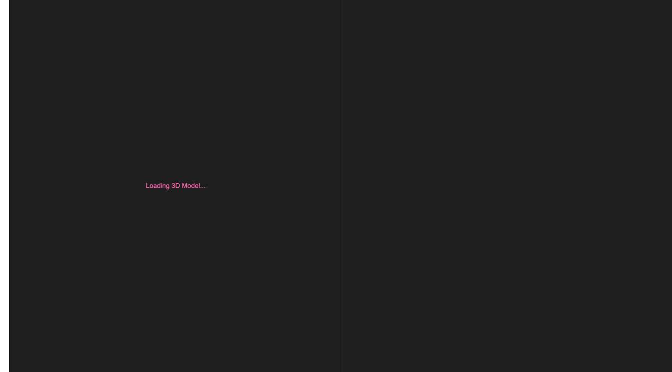



Fig. 11: Case 73 output from Gemini: two faintly outlined view containers are visible, but neither renders any marks or data. The isosurface panel shows only a "loading 3D model" indicator; the histogram panel is a blank space.

**Model 4: Gemini (VMPC = 0.6).** The code executes without producing a hard error, so $X = 1$, but neither view renders any visible marks or data.

**View 1 (CT Isosurface).** A faint panel outline is visible and a "loading 3D model" indicator appears within it, confirming that a view container was instantiated. Graders were instructed to apply a generous interpretation of view presence: $V_v = 1$ whenever there is a discernible indication that a view was intended, even if its content failed to load. The grader therefore awards $V_1 = 1$. However, since no isosurface geometry ever renders, there is no mark to evaluate ($M_1 = 0$) and no encoding to verify ($E_1 = 0$). The absence of any rendered data also means no hallucination is possible, so $H_1 = 1$. No linking was requested, so $L_1 = 1$.

**View 2 (Intensity Histogram).** The second half of the screen is occupied by a blank space with a faint outline. No histogram, bars, or axis labels are visible. The graders debated $V_2$ carefully: awarding $V_2 = 0$ would conflate a *missing* view with a *failed render*, conflating two distinct failure modes. Since the blank space is clearly allocated as a view container, at least two of three graders agreed to award $V_2 = 1$ in the interest of generous but consistent grading. With no marks rendered, $M_2 = 0$ and $E_2 = 0$. As no data is displayed, $H_2 = 1$, and with no linking requested, $L_2 = 1$.

**Calculation.** Six of ten criteria are satisfied. Both views lose $M_v$ and $E_v$ due to the absence of any rendered content:

$$\text{VMPC} = 1 \cdot \frac{1}{5 \times 2} \left( \underbrace{(1+0+0+1+1)}_{\text{View 1}} + \underbrace{(1+0+0+1+1)}_{\text{View 2}} \right) = \frac{6}{10} = 0.6$$

**Summary.** Table 19 summarizes the per-criterion scores for all four models on case 73. The progression from Raiven and ChatGPT (full compliance) through Claude (one empty view) to Gemini (both views empty) illustrates how VMPC captures graduated levels of partial compliance within the same prompt.

Table 19: Per-criterion VMPC scores for case 73 across four models. $L_v = 1$ for all entries as no linking was requested.

| | | View 1 (Isosurface) | | | | | View 2 (Histogram) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Model** | $X$ | $V_1$ | $M_1$ | $E_1$ | $H_1$ | $L_1$ | $V_2$ | $M_2$ | $E_2$ | $H_2$ | $L_2$ | **VMPC** |
| Raiven | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1.0 |
| ChatGPT | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1.0 |
| Claude | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.8 |
| Gemini | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0.6 |

### D.5.2 Execution Failure: Case s1 (ChatGPT, VMPC = 0.0)

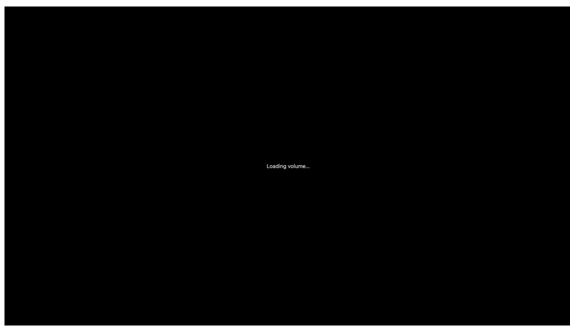


Fig. 12: Case s1 output from ChatGPT: a blank screen with only a “loading volume” indicator. No view ever renders.

The prompt for case s1 reads:

> *“Create a volume rendering of the skull CT data from* `head.vti`*.”*

This specifies $N = 1$ view with no cross-view linking ($L_1 = 1$ by default).

A grader approaching this output would first attempt to run the submitted code. The code either crashes outright or produces only a blank screen with a persistent “loading volume” indicator and no evidence of a rendered view. Throughout our evaluation, outputs that displayed only loading indicators with no rendered content were treated as execution failures on par with hard crashes: a loading symbol with no subsequent render provides no visualizable output for a grader to assess. The execution gate is therefore set to $X = 0$.

Once $X = 0$, VMPC collapses to zero regardless of any per-criterion judgments:

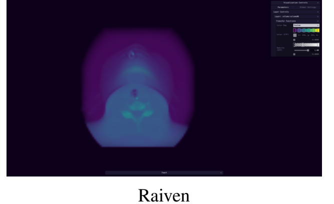
$$\text{VMPC} = 0 \cdot \frac{1}{5 \times 1} \underbrace{(V_1 + M_1 + E_1 + H_1 + L_1)}_{\text{View 1}} = 0.0$$

The grader need not evaluate any individual criterion — the execution gate short-circuits the entire computation. This reflects the core requirement that a visualization must be visible to provide value: a system that fails to render anything, whether due to a runtime error or an indefinitely stalled load, has not fulfilled the prompt regardless of what the underlying code attempted.

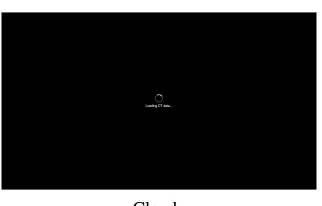
## D.6 Full Results

As referenced by Section 7. Results are organized by category: SciVis (S, prompts S1–S30), InfoVis (I, prompts I31–I70), and Combined (C, prompts C71–C100). For each prompt we show the rendered output from all four systems, the average VMPC score across three human graders ($G_1$–$G_3$, shown individually), the independent VLM score, and generation time in minutes.

### D.6.1 SciVis (S)


Raiven

ChatGPT


Claude

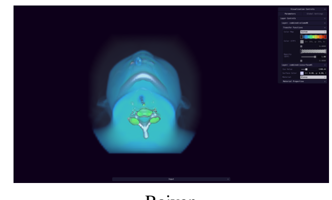
Gemini

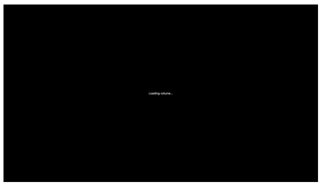
**S1** Views: 1

*Create a volume rendering of the skull CT data from head.vti.*

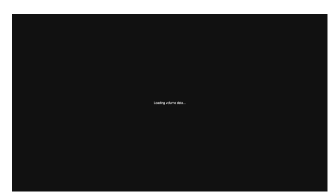
Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.93 | 1.0 | 1.0 | 0.8 | 1.00 | 0.24 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.35 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.67 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.89 |


Raiven


ChatGPT

Claude

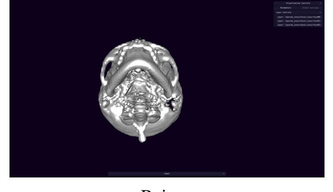

Gemini

**S2** Views: 1

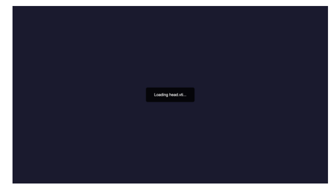
*Display a semi-transparent volume from head.vti with an overlayed isosurface highlighting the bone boundary within the tissue, use color palette turbo.*

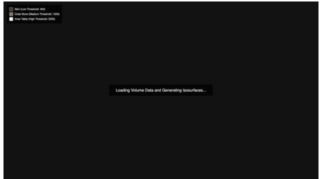
Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.26 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.45 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.54 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.71 |


Raiven

ChatGPT


Claude

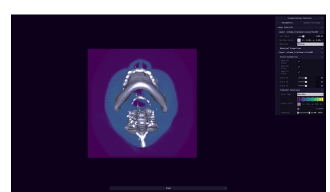

Gemini

**S3** Views: 1

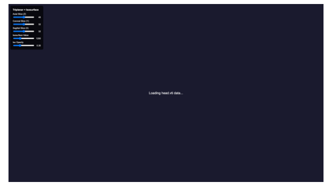
*Generate three layered, differently colored isosurfaces from head.vti at low, medium, and high intensity thresholds to represent skin, outer bone, and inner table.*

Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.93 | 1.0 | 1.0 | 0.8 | 0.80 | 0.56 |
| ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.38 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.41 |
| Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.80 |


Raiven

ChatGPT


Claude

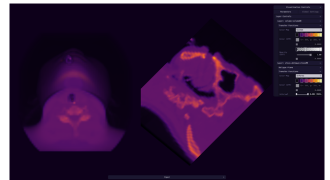
Gemini

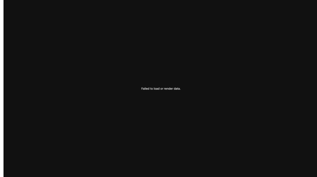
**S4** Views: 1

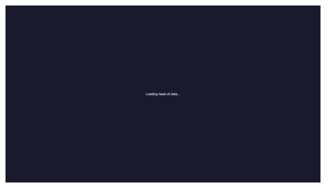
*Display axial, sagittal, and coronal triplanar slices of head.vti in one view with an overlayed isosurface to provide 3D surface context.*

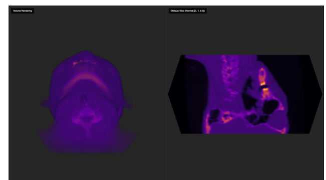
Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.35 |
| ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.62 |
| Claude | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.59 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.30 |


Raiven


ChatGPT


Claude

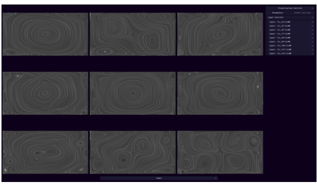

Gemini

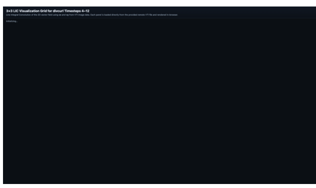
**S5** Views: 2

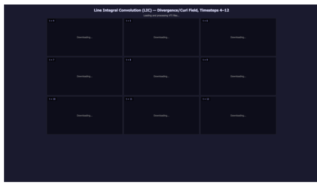
*Show a volume rendering from head.vti in one view and an oblique slice rotated at an arbitrary non-axis-aligned angle in a second view, both using color palette inferno.*

Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.43 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.19 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.36 |


Raiven


ChatGPT


Claude

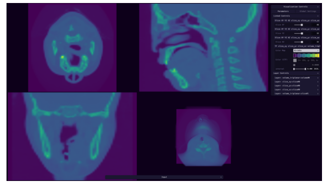
Gemini

**S6** Views: 9

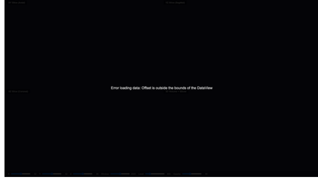
*Use divcurl_{t}.vti to create a 3×3 grid of LIC visualizations across timesteps t = 4 to 12, using the 2D vector field defined by vx and vy.*

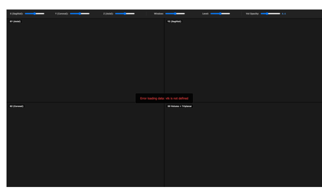
Datasets: `divcurl_4.vti, divcurl_5.vti, divcurl_6.vti, divcurl_7.vti, divcurl_8.vti, divcurl_9.vti, divcurl_10.vti, divcurl_11.vti, divcurl_12.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.40 |
| ChatGPT | 0.07 | 0.0 | 0.0 | 0.2 | 0.40 | 0.79 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.95 |
| Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 3.12 |


Raiven

ChatGPT


Claude

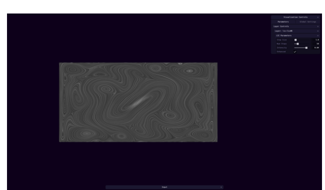

Gemini

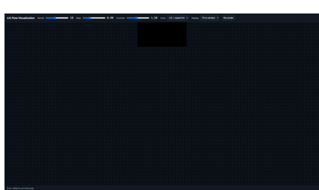
**S7** Views: 4

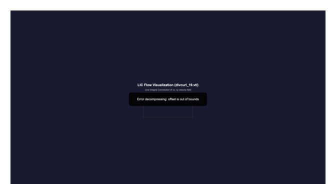
*Set up three slice views (XY, YZ, XZ) and a fourth 3D view combining volume rendering and triplanar slices from head.vti, synchronizing slice positions and the transfer function.*

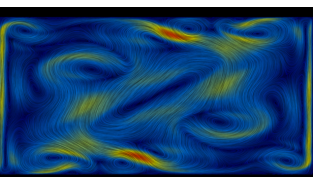
Datasets: `head.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.37 |
| ChatGPT | 0.47 | 0.4 | 0.4 | 0.6 | 0.40 | 0.78 |
| Claude | 0.13 | 0.0 | 0.4 | 0.0 | 0.40 | 1.81 |
| Gemini | 0.47 | 0.4 | 0.4 | 0.6 | 0.40 | 2.18 |


Raiven


ChatGPT


Claude

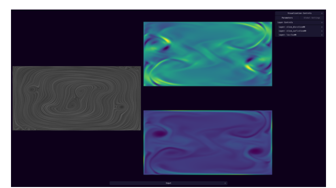

Gemini

**S8** Views: 1

*Produce an LIC visualization from divcurl_19.vti using vx and vy to reveal flow patterns.*

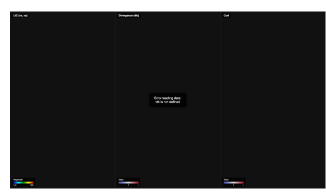
Datasets: `divcurl_19.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.23 |
| ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.79 |
| Claude | 0.40 | 0.0 | 0.6 | 0.6 | 0.00 | 1.28 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.94 |


Raiven

ChatGPT

Claude

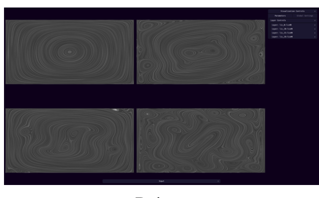

Gemini

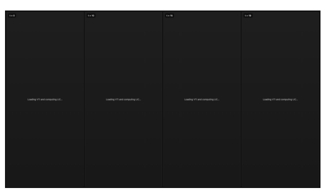
**S9** Views: 3

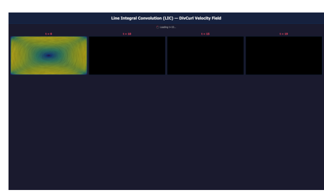
*Arrange three views from divcurl_15.vti: an LIC of vx and vy, a slice of div, and a slice of curl.*

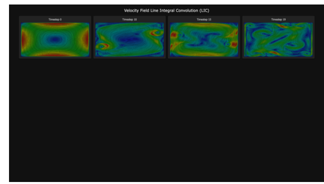
Datasets: `divcurl_15.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.36 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.77 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 1.11 |
| Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 2.59 |


Raiven


ChatGPT


Claude

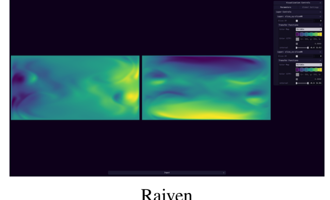

Gemini

**S10** Views: 4

*Use divcurl_0.vti, divcurl_10.vti, divcurl_15,vti, and divcurl_19.vti to create four side-by-side LIC views, one per timestep, each computed from the velocity vector with vx = "vx" and vy = "vy".*

Datasets: `divcurl_0.vti, divcurl_10.vti, divcurl_15.vti, divcurl_19.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| ChatGPT | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 0.35 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 0.70 | 0.80 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 3.31 |


Raiven

ChatGPT

Claude

Gemini

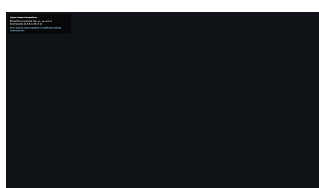
**S11** Views: 2

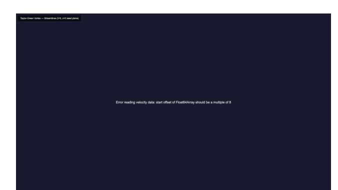
*Provide two slice views from divcurl_15.vti comparing velocity components by setting the scalar field to vy in one and vx in the other.*

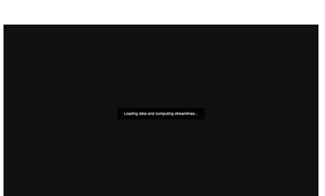
Datasets: `divcurl_15.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.45 |
| ChatGPT | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 0.34 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.04 |
| Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.18 |

Raiven


ChatGPT


Claude

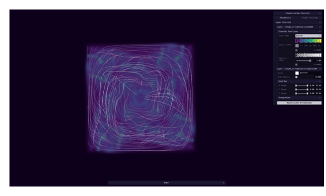
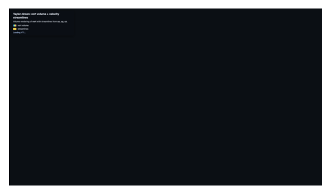
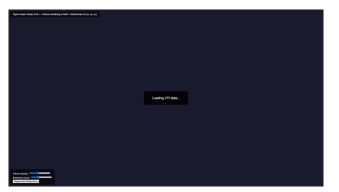
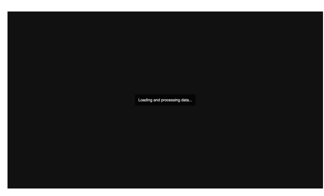

Gemini

**S12** Views: 1

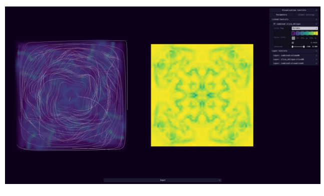
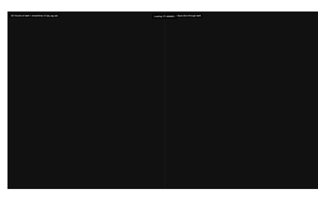
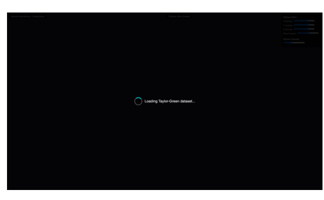
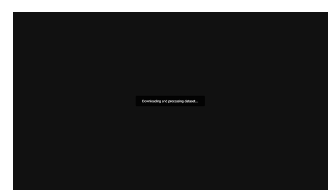
*Compute streamlines from taylorgreen_9.vti using ux, uy, and uz, with seeds in the bounds [0, 64, 0, 64, 0, 0].*

Datasets: `taylorgreen_9.vti`

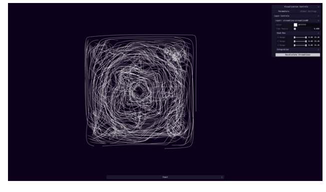
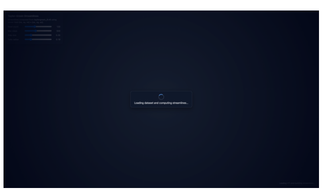
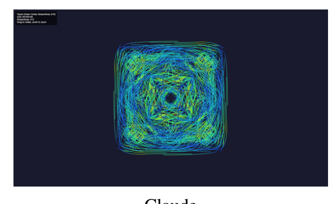
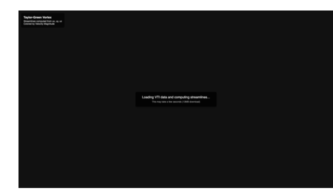
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.24 |
| ChatGPT | 0.20 | 0.6 | 0.0 | 0.0 | 0.40 | 0.32 |
| Claude | 0.20 | 0.6 | 0.0 | 0.0 | 0.40 | 1.23 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 2.58 |


Raiven

ChatGPT

Claude

Gemini

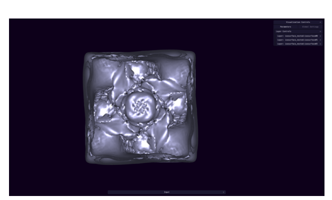
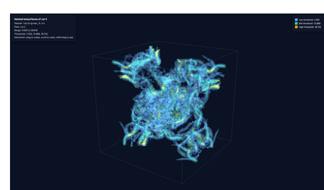
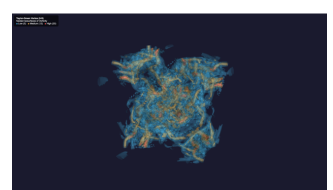
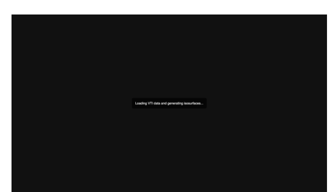
| **S13** Views: 1 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Make a volume of "vort" and also streamlines of ux, uy, and uz layered over volume from taylorgreen_9.vti.* Datasets: `taylorgreen_9.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| | ChatGPT | 0.73 | 0.6 | 1.0 | 0.6 | 0.40 | 0.57 |
| | Claude | 0.60 | 0.6 | 0.6 | 0.6 | 1.00 | 1.13 |
| | Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.62 |


Raiven

ChatGPT

Claude

Gemini

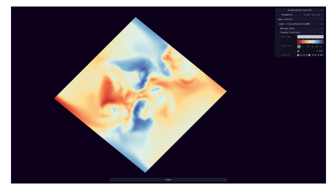
| **S14** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Use dataset taylorgreen_9.vti to make a volume of "vort" with streamlines of ux, uy, and uz layered over volume as well as a linked oblique slice.* Datasets: `taylorgreen_9.vti` | Raiven | 0.87 | 0.9 | 0.9 | 0.8 | 1.00 | 0.42 |
| | ChatGPT | 0.47 | 0.6 | 0.4 | 0.4 | 0.40 | 0.98 |
| | Claude | 0.13 | 0.0 | 0.4 | 0.0 | 0.40 | 1.66 |
| | Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 2.77 |

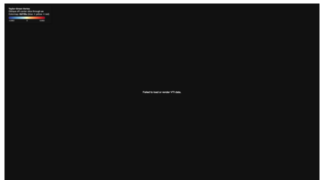

Raiven

ChatGPT

Claude

Gemini

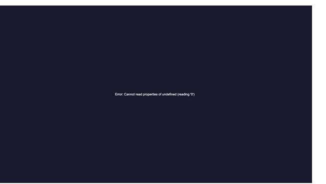
| **S15** Views: 1 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Make streamline from taylorgreen_9.vti using variables vx = ux, vy = uy, and vz = uz.* Datasets: `taylorgreen_9.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.23 |
| | ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.90 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.80 |
| | Gemini | 0.20 | 0.6 | 0.0 | 0.0 | 0.40 | 2.73 |

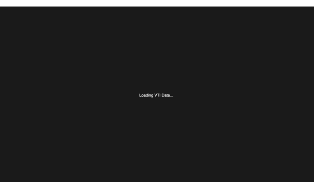

Raiven

ChatGPT

Claude

Gemini

| **S16** Views: 1 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Generate three nested isosurfaces of vort from taylorgreen_9.vti at progressively higher thresholds.* Datasets: `taylorgreen_9.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.38 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.38 |
| | Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 7.65 |


Raiven


ChatGPT


Claude

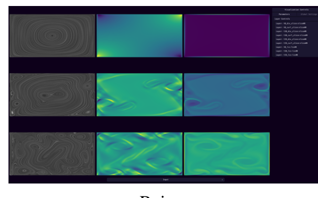

Gemini

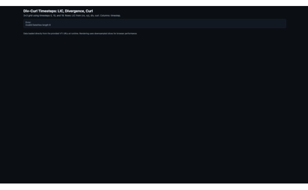
**S17** Views: 1

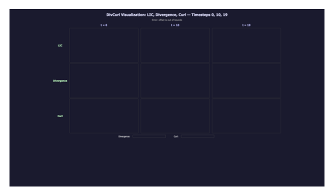
*Extract an oblique slice through the ux field in taylorgreen_9.vti that is rotated off-center and "Rdylbu" color scheme.*

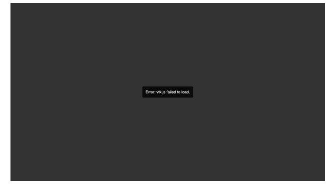
Datasets: `taylorgreen_9.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.26 |
| ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.34 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.11 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.36 |


Raiven


ChatGPT


Claude

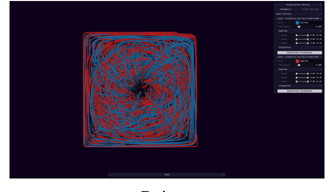

Gemini

**S18** Views: 9

*Use divcurl_0.vti, divcurl_10.vti, and divcurl_19.vti to create a 3×3 grid across timesteps, with LIC generated from the 2D vector field using vx and vy, and slices of div and curl.*

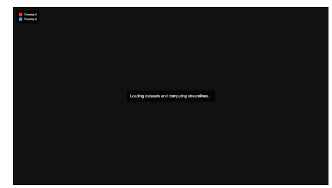
Datasets: `divcurl_0.vti, divcurl_10.vti, divcurl_19.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.62 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 0.75 |
| Claude | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.36 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 3.24 |


Raiven

ChatGPT

Claude

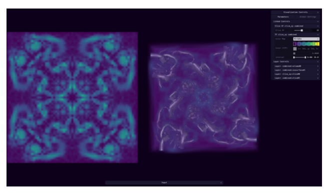

Gemini

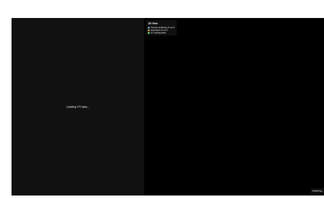
**S19** Views: 1

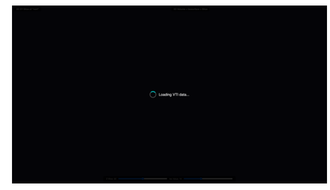
*Overlay streamlines from taylorgreen_6.vti and taylorgreen_8.vti in a single scene using vx = "ux", vy = "uy", and vz = "uz", distinguishing timesteps with different colors.*

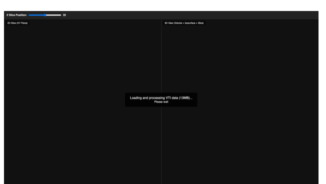
Datasets: `taylorgreen_6.vti, taylorgreen_8.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.25 |
| ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.39 |
| Claude | 0.60 | 0.6 | 0.6 | 0.6 | 0.00 | 1.04 |
| Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 1.47 |


Raiven


ChatGPT


Claude

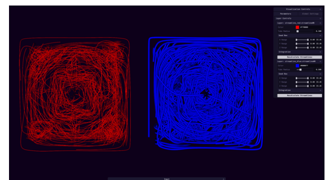

Gemini

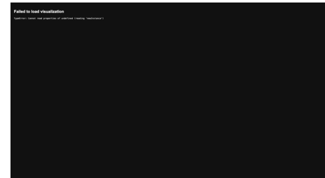
**S20** Views: 2

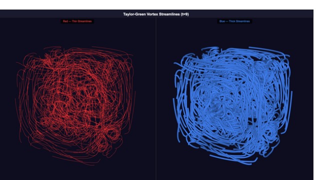
*Using taylorGreen_t_9.vti, create two coordinated views of "vort": a slice in the XY plane at mid-depth and a 3D view combining a volume rendering of "vort" with an isosurface of "vort" and the XY slice. Link the slice position along Z so moving the 2D slice updates the cutting plane in the 3D view.*

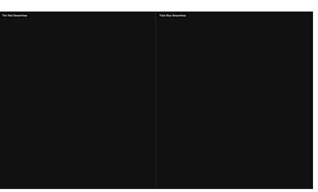
Datasets: `taylorgreen_9.vti`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.43 |
| ChatGPT | 0.43 | 0.4 | 0.4 | 0.5 | 0.40 | 0.56 |
| Claude | 0.13 | 0.0 | 0.4 | 0.0 | 0.40 | 0.87 |
| Gemini | 0.40 | 0.4 | 0.4 | 0.4 | 0.40 | 1.93 |

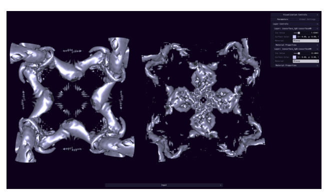

Raiven

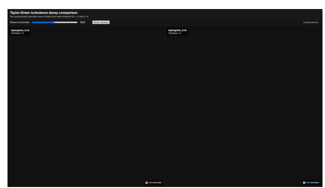

ChatGPT

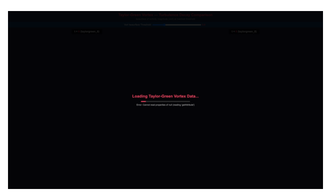

Claude

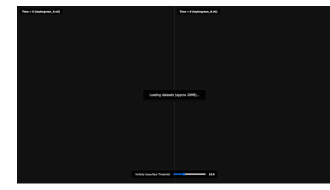

Gemini

| **S21** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Display two side-by-side streamline visualizations from taylorgreen_9.vti using vy = "ux", vy = "uy", and vz = "uz", one in red with thin streamlines and one in blue with thick streamlines.*<br>Datasets: `taylorgreen_9.vti` | Raiven<br>ChatGPT<br>Claude<br>Gemini | 1.00<br>0.00<br>1.00<br>0.53 | 1.0<br>0.0<br>1.0<br>0.6 | 1.0<br>0.0<br>1.0<br>0.6 | 1.0<br>0.0<br>1.0<br>0.4 | 1.00<br>0.00<br>1.00<br>0.40 | 0.29<br>0.34<br>1.12<br>8.38 |

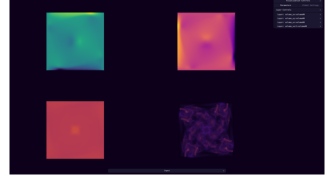

Raiven

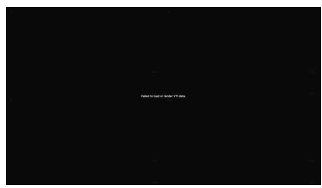

ChatGPT

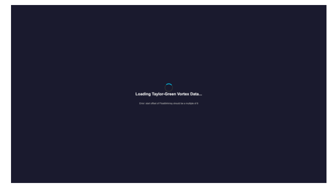

Claude

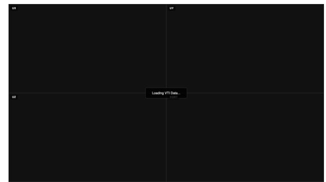

Gemini

| **S22** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Show two separate isosurface views of vort at the same threshold from taylorgreen_6.vti and taylorgreen_8.vti to compare turbulence decay.*<br>Datasets: `taylorgreen_6.vti, taylorgreen_8.vti` | Raiven<br>ChatGPT<br>Claude<br>Gemini | 1.00<br>0.60<br>0.20<br>0.53 | 1.0<br>0.6<br>0.0<br>0.6 | 1.0<br>0.6<br>0.6<br>0.6 | 1.0<br>0.6<br>0.0<br>0.4 | 1.00<br>0.40<br>0.40<br>0.40 | 0.25<br>0.45<br>3.75<br>1.21 |

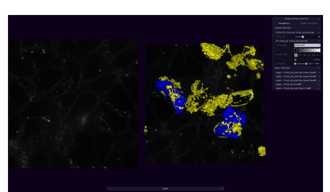

Raiven

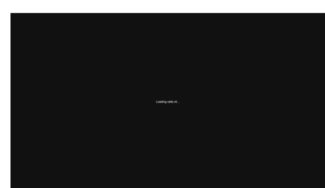

ChatGPT

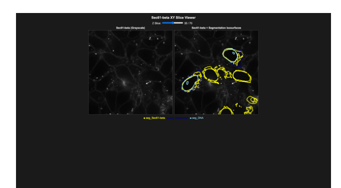

Claude

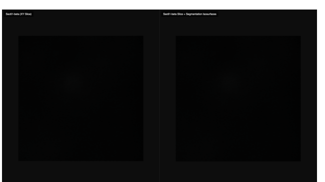

Gemini

| **S23** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Arrange a four-view grid from taylorgreen_9.vti with volume renderings of ux, uy, uz, and vort, each using different color palette.*<br>Datasets: `taylorgreen_9.vti` | Raiven<br>ChatGPT<br>Claude<br>Gemini | 0.95<br>0.20<br>0.00<br>0.53 | 1.0<br>0.0<br>0.0<br>0.6 | 1.0<br>0.6<br>0.0<br>0.6 | 0.8<br>0.0<br>0.0<br>0.4 | 1.00<br>0.00<br>0.00<br>0.40 | 0.29<br>0.78<br>1.18<br>1.82 |

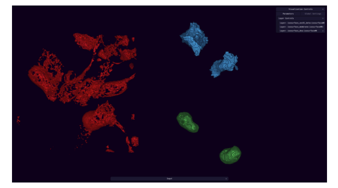

Raiven

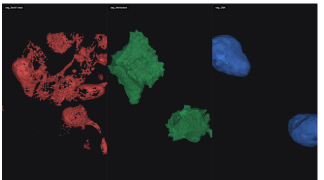

ChatGPT

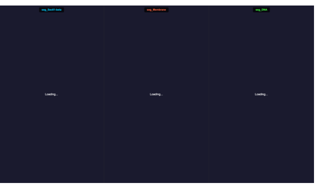

Claude

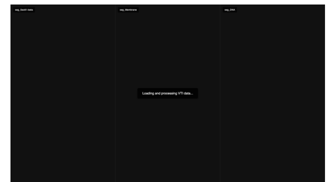

Gemini

| **S24** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create a view with an XY slice of the fluorescence field "Sec61-beta" from cells.vti in grayscale. Add a second linked view that shows the same "Sec61-beta" slice in grayscale overlaid with isosurfaces of "seg_Sec61-beta" in yellow, "seg_Membrane" in dark blue, and "seg_DNA" in light blue.*<br>Datasets: `cells.vti` | Raiven<br>ChatGPT<br>Claude<br>Gemini | 1.00<br>0.00<br>0.87<br>0.60 | 1.0<br>0.0<br>0.7<br>0.6 | 1.0<br>0.0<br>0.9<br>0.8 | 1.0<br>0.0<br>1.0<br>0.4 | 1.00<br>0.00<br>1.00<br>0.40 | 0.44<br>0.90<br>0.95<br>1.63 |

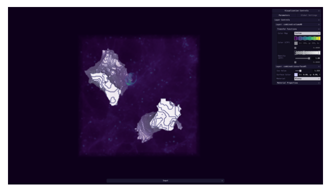

Raiven


ChatGPT

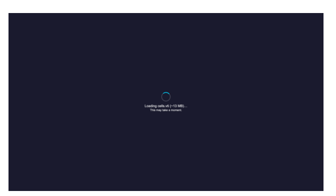

Claude

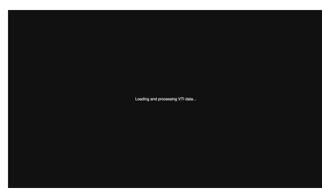

Gemini

| **S25** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create three views each showing an isosurface from cells.vti, one using seg_Sec61-beta, one using seg_Membrane, and one using seg_DNA, each should be semi-transparent and a unique color.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.31 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.35 |
| | Claude | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 0.56 |
| | Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.28 |

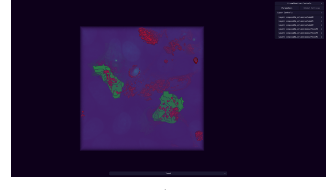

Raiven

ChatGPT

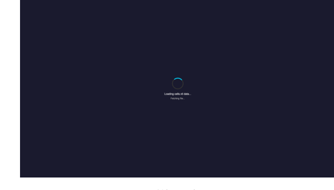

Claude

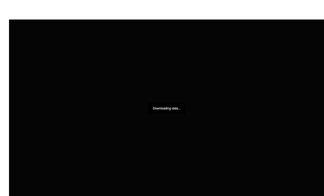

Gemini

| **S26** Views: 1 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Combine a volume rendering of Sec61-beta with an isosurface from seg_Membrane in a single view from cells.vti.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| | ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.54 |
| | Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 0.65 |
| | Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.91 |

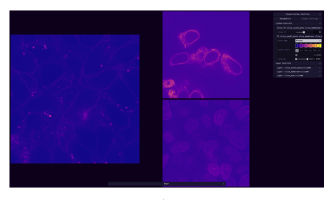

Raiven

ChatGPT

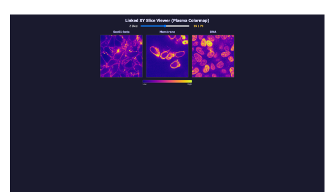

Claude

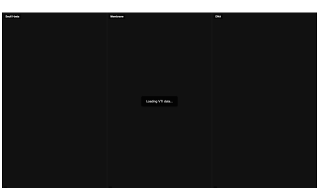

Gemini

| **S27** Views: 1 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Layer volume renderings of Sec61-beta, Membrane, and DNA with isosurfaces from seg_Sec61-beta, seg_Membrane, and seg_DNA in one composite view from cells.vti. All volumes should use same color palette, while all isosurfaces should be distinct colors.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.44 |
| | ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 0.56 |
| | Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.84 |
| | Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 2.02 |

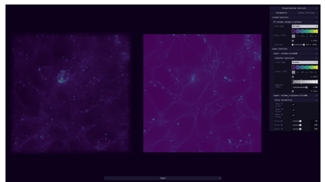

Raiven

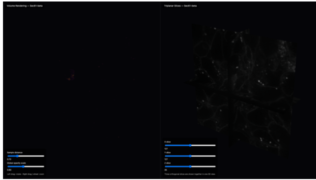
ChatGPT

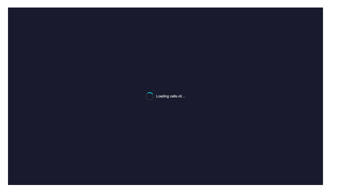

Claude

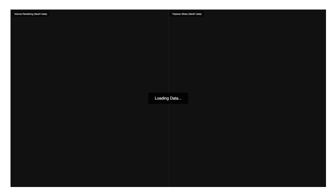

Gemini

| **S28** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Using cells.vti, create three slice (XY) views using color palette plasma: one for "Sec61-beta", one for "Membrane", and one for "DNA", and link their slice positions so scrolling in any view updates the other two.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.42 |
| | ChatGPT | 0.40 | 0.4 | 0.4 | 0.4 | 0.40 | 0.40 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.22 |
| | Gemini | 0.40 | 0.4 | 0.4 | 0.4 | 0.40 | 2.34 |

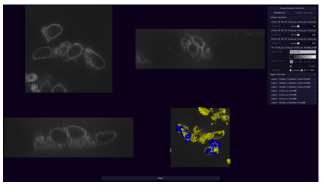

Raiven

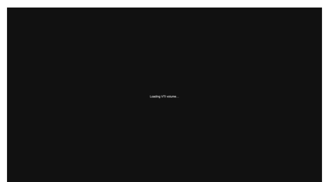

ChatGPT

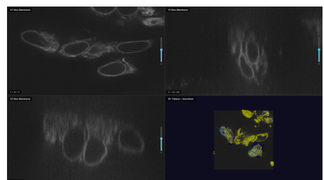

Claude

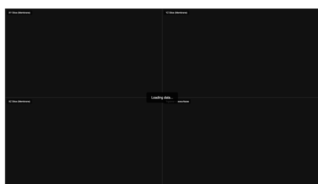

Gemini

| **S29** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Display a volume rendering of Sec61-beta in one view and triplanar slices of the same field in another from cells.vti.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| | ChatGPT | 0.87 | 0.6 | 1.0 | 1.0 | 1.00 | 0.60 |
| | Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 0.69 |
| | Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.83 |


Raiven

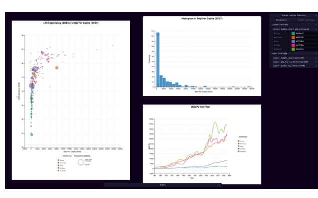

ChatGPT

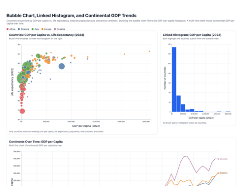

Claude

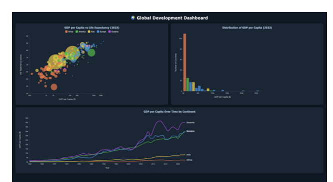

Gemini

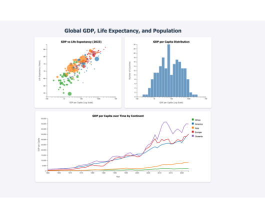
| **S30** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *cells.vti contains fluorescence channels ("Sec61-beta", "Membrane", "DNA") and segmentation channels ("seg_Sec61-beta", "seg_Membrane", "seg_DNA"). Create four views: (1) an XY slice of the fluorescence channel "Membrane" in grayscale, (2) a YZ slice of "Membrane" in grayscale, (3) an XZ slice of "Membrane" in grayscale, and (4) a combined view showing triplanar slices of "Membrane" in grayscale overlaid with isosurfaces of "seg_Sec61-beta" in yellow, "seg_Membrane" in dark blue, and "seg_DNA" in light blue. Link the slice positions across all four views so scrolling in any slice view updates the others and the triplanar.* Datasets: `cells.vti` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.47 |
| | ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.62 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.06 |
| | Gemini | 0.40 | 0.4 | 0.4 | 0.4 | 0.40 | 2.72 |

### D.6.2 InfoVis (I)

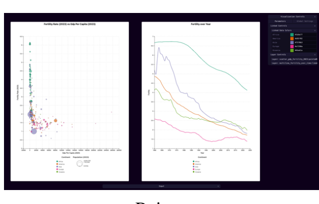

Raiven

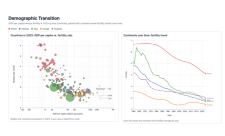

ChatGPT

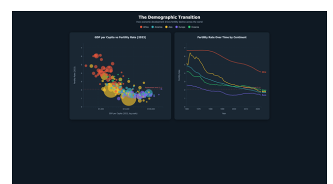

Claude


Gemini

| **I31** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Build a bubble chart from countries_latest.csv plotting "gdp_per_capita (2023)" versus "life_expectancy (2023)", sized by "population (2023)" and colored by "continent", with a linked histogram of "gdp_per_capita (2023)" that filters when brushing the bubbles. Add a multiline chart from continents_timeseries.csv tracking "gdp_pc" over "year" with lines colored by "continent".* Datasets: `countries_latest.csv, continents_timeseries.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.00 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.41 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.46 |

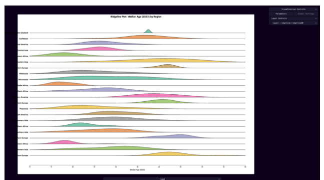

Raiven

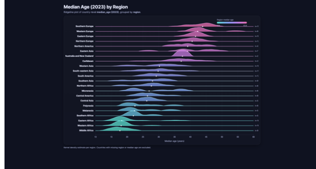

ChatGPT

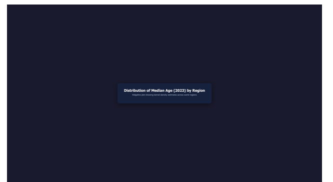

Claude

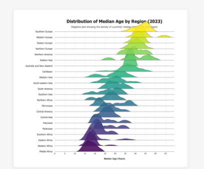
Gemini

| **I32** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Show the demographic transition: scatter "gdp_per_capita (2023)" versus "fertility_rate (2023)" from countries_latest.csv, sized by "population (2023)" and colored by "continent", paired with a multiline chart from continents_timeseries.csv plotting "fertility" over "year" with lines colored by "continent".* Datasets: `countries_latest.csv, continents_timeseries.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.20 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.68 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.99 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.38 |


Raiven


ChatGPT


Claude


Gemini

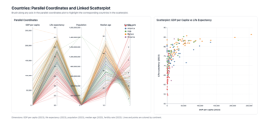
**I33** Views: 1

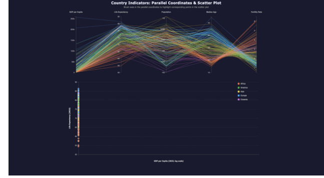
*Produce a ridgeline plot of "median_age (2023)" grouped by "region" from countries_latest.csv.*

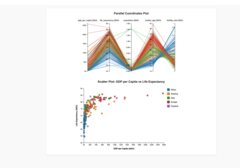
Datasets: `countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.22 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.70 |
| Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 0.68 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.37 |

Raiven


ChatGPT


Claude

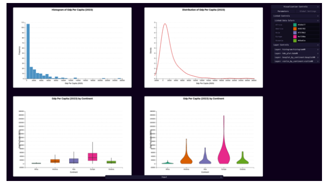

Gemini

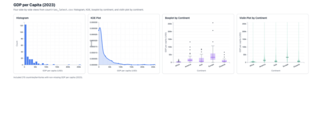
**I34** Views: 2

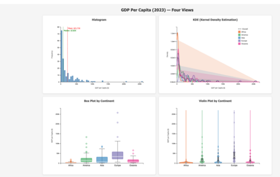
*Lay out a parallel-coordinates plot from countries_latest.csv spanning "gdp_per_capita (2023)", "life_expectancy (2023)", "population (2023)", "median_age (2023)", and "fertility_rate (2023)" with lines colored by "continent", linked to a scatter of "gdp_per_capita (2023)" versus "life_expectancy (2023)" colored by "continent". Brushing lines in the parallel coordinates should highlight the corresponding points in the scatter.*

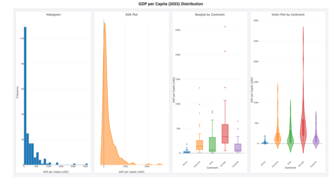
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| ChatGPT | 0.93 | 0.8 | 1.0 | 1.0 | 1.00 | 0.77 |
| Claude | 0.83 | 0.7 | 0.9 | 0.9 | 1.00 | 0.79 |
| Gemini | 0.93 | 0.8 | 1.0 | 1.0 | 1.00 | 1.68 |


Raiven


ChatGPT


Claude


Gemini

**I35** Views: 4

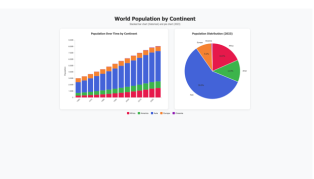
*Display "gdp_per_capita (2023)" from countries_latest.csv four ways side by side: a histogram, a KDE plot, a boxplot grouped by "continent", and a violin plot grouped by "continent".*

Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.06 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.13 |
| Gemini | 0.98 | 1.0 | 1.0 | 0.9 | 1.00 | 1.05 |

Raiven

ChatGPT


Claude

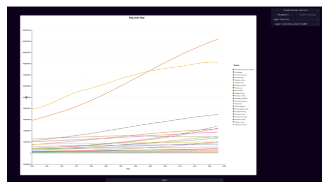
Gemini

**I36** Views: 2

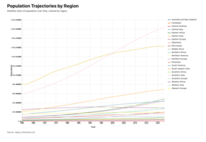
*Create a stacked bar chart from countries_timeseries.csv with "year" on the x-axis and "pop" aggregated by "continent", next to a pie chart of "population (2023)" from countries_latest.csv grouped by "continent". Use consistent continent colors across both views.*

Datasets: `countries_timeseries.csv, countries_latest.csv`

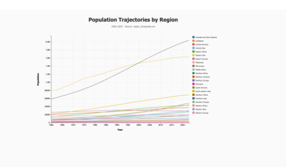
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.65 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.72 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.90 |



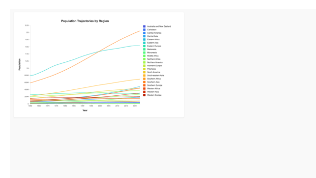
Raiven



ChatGPT



Claude



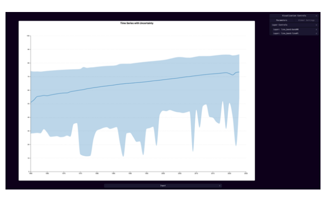
Gemini

**I37** Views: 1

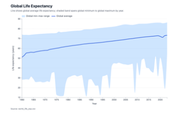
*Plot population trajectories from regions_timeseries.csv as a multiline chart with "year" on the x-axis, "pop" on the y-axis, and lines colored by "region".*

Datasets: `region_timeseries.csv`

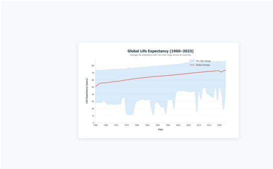
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.17 |
| ChatGPT | 0.93 | 1.0 | 1.0 | 0.8 | 1.00 | 0.50 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.51 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.71 |



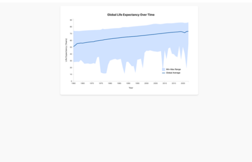
Raiven



ChatGPT



Claude



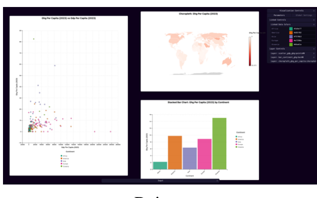
Gemini

**I38** Views: 1

*Layer a line of "global_average" life expectancy over a shaded band spanning "global_min" to "global_max" from world_life_exp.csv, with "year" on the x-axis.*

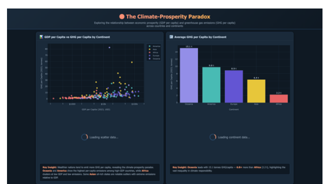
Datasets: `world_life_exp.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.93 | 1.0 | 1.0 | 0.8 | 1.00 | 0.31 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.44 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.00 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.51 |

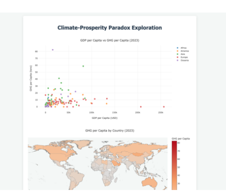


Raiven

ChatGPT



Claude



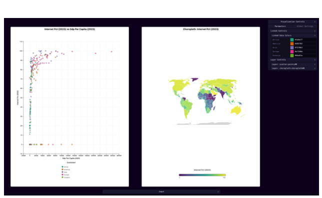
Gemini

**I39** Views: 3

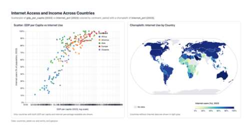
*Explore the climate-prosperity paradox: scatter "gdp_per_capita (2023)" versus "ghg_per_capita (2023)" colored by "continent" from countries_latest.csv, add a choropleth colored by "ghg_per_capita (2023)" using country geometries from world_iso3.geojson and a red colorscale, and include a bar chart from continents_latest.csv with "continent" on the x-axis and "ghg_per_capita (2023)" on the y-axis, colored by "continent".*

Datasets: `countries_latest.csv, world_iso3.geojson, continents_latest.csv`

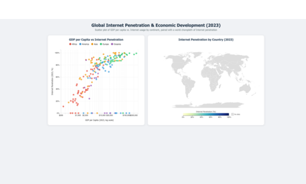
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.34 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.55 |
| Claude | 0.89 | 0.9 | 1.0 | 0.8 | 0.40 | 0.95 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.19 |



Raiven



ChatGPT



Claude

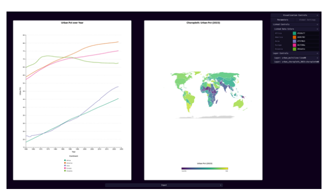
Gemini

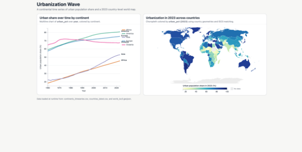
**I40** Views: 2

*Pair a scatter of "gdp_per_capita (2023)" versus "internet_pct (2023)" colored by "continent" from countries_latest.csv with a choropleth colored by "internet_pct (2023)" using country geometries from world_iso3.geojson.*

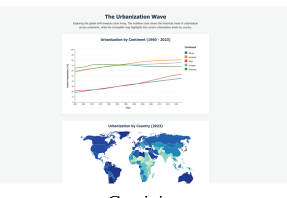
Datasets: `countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.31 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.81 |
| Claude | 0.90 | 0.9 | 0.9 | 0.9 | 1.00 | 0.82 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.10 |


Raiven


ChatGPT

Claude

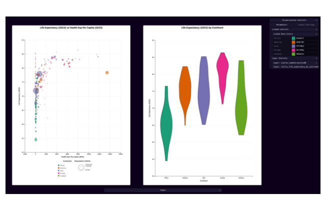

Gemini

**I41** Views: 2

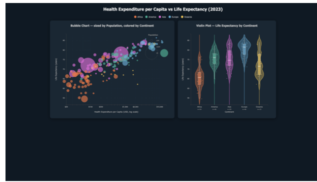
*Chart the urbanization wave: a multiline of "urban_pct" over "year" colored by "continent" from continents_timeseries.csv, plus a choropleth colored by "urban_pct (2023)" using country geometries from world_iso3.geojson and countries_latest.csv.*

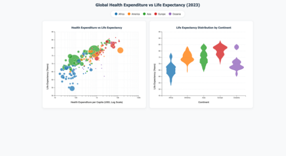
Datasets: `continents_timeseries.csv, countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.21 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.95 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.89 |


Raiven

ChatGPT


Claude


Gemini

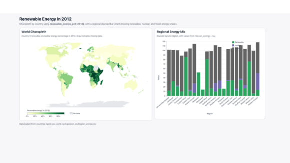
**I42** Views: 2

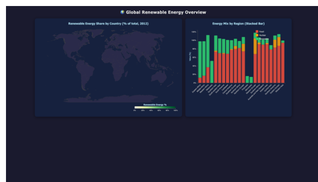
*Scatter "health_exp_per_capita (2023)" versus "life_expectancy (2023)" from countries_latest.csv as bubbles sized by "population (2023)" and colored by "continent", with a companion violin plot of "life_expectancy (2023)" grouped by "continent".*

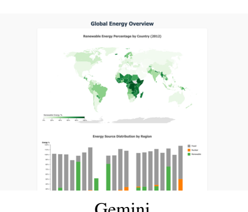
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.33 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.97 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.87 |

Raiven


ChatGPT


Claude

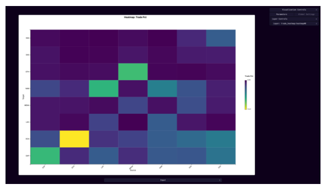

Gemini

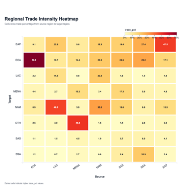
**I43** Views: 2

*Map renewable energy: a choropleth colored by "renewable_energy_pct (2012)" using country geometries from world_iso3.geojson and countries_latest.csv, alongside a stacked bar chart from region_energy.csv with "region" on the x-axis, "value" on the y-axis, and stacked by "category" (renewable, nuclear, fossil).*

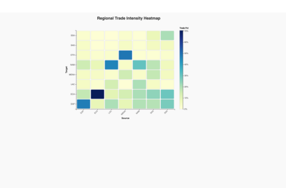
Datasets: `countries_latest.csv, world_iso3.geojson, region_energy.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.92 |
| Claude | 0.90 | 0.9 | 0.9 | 0.9 | 1.00 | 0.80 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.48 |


Raiven


ChatGPT

Claude

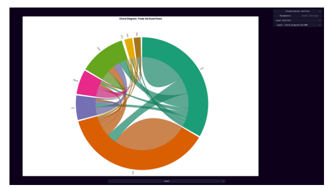

Gemini

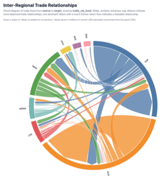
**I44** Views: 1

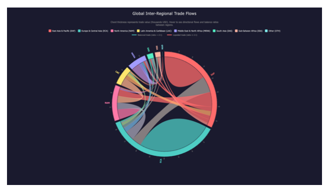
*Visualize regional trade intensity from trade_edges.csv as a heatmap with "source" on the x-axis, "target" on the y-axis, and cells colored by "trade_pct".*

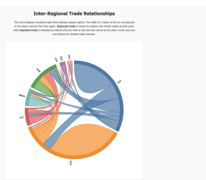
Datasets: `trade_edges.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.21 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.46 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.51 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.59 |


Raiven


ChatGPT


Claude


Gemini

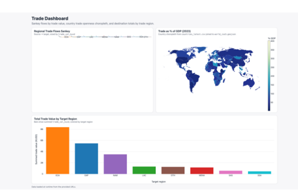
**I45** Views: 1

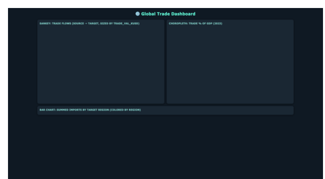
*Construct a chord diagram from trade_edges.csv connecting "source" to "target", sized by "trade_val_kusd", to reveal which inter-regional trade relationships are balanced and which are lopsided.*

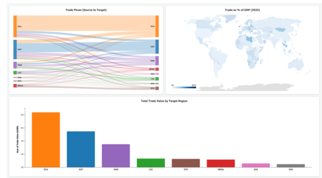
Datasets: `trade_edges.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.24 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.64 |
| Claude | 0.93 | 1.0 | 1.0 | 0.8 | 1.00 | 0.90 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.07 |

Raiven


ChatGPT


Claude

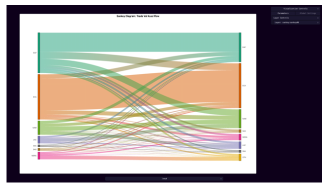

Gemini

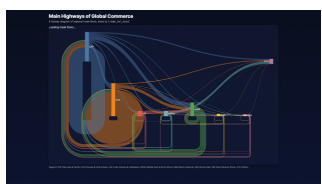
**I46** Views: 3

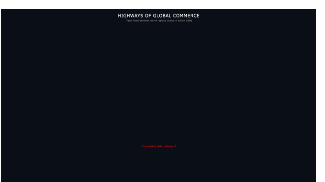
*Assemble a three-view trade dashboard: a sankey from trade_edges.csv with "source" flowing to "target" sized by "trade_val_kusd", a choropleth colored by "trade_pct_gdp (2023)" using countries_latest.csv and world_iso3.geojson, and a bar chart with "target" on the x-axis and summed "trade_val_kusd" on the y-axis, colored by "target" region.*

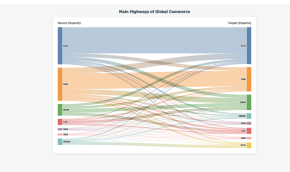
Datasets: `trade_edges.csv, countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.98 | 1.0 | 1.0 | 0.9 | 1.00 | 0.31 |
| ChatGPT | 0.89 | 0.9 | 0.9 | 0.8 | 1.00 | 0.83 |
| Claude | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.02 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.60 |


Raiven


ChatGPT


Claude

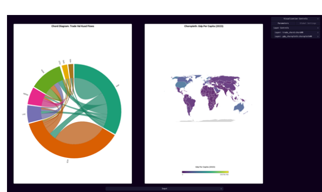

Gemini

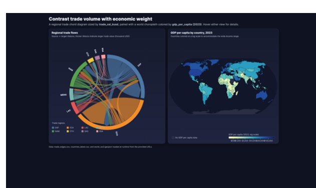
**I47** Views: 1

*Show the main highways of global commerce as a sankey diagram from trade_edges.csv, with "source" flowing to "target" sized by "trade_val_kusd".*

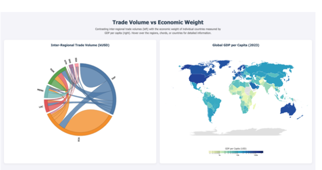
Datasets: `trade_edges.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.52 |
| Claude | 0.20 | 0.6 | 0.0 | 0.0 | 0.00 | 0.65 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.27 |


Raiven


ChatGPT

Claude

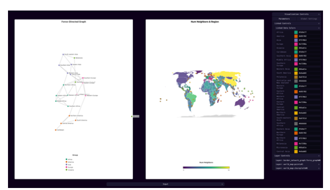

Gemini

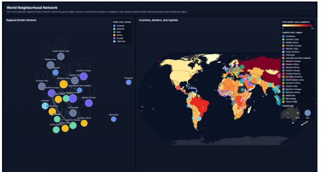
**I48** Views: 2

*Contrast trade volume with economic weight: a chord diagram from trade_edges.csv connecting "source" to "target" sized by "trade_val_kusd", paired with a choropleth colored by "gdp_per_capita (2023)" using countries_latest.csv and world_iso3.geojson.*

Datasets: `trade_edges.csv, countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.31 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.22 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.88 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.06 |


Raiven


ChatGPT

Claude

Gemini

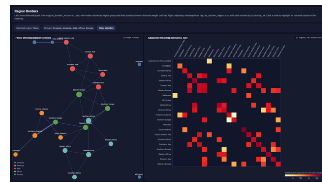
**I49** Views: 2

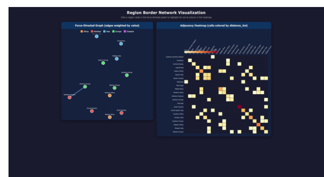
*Map the world's neighborhood network: a force-directed graph from region_border_network.json with nodes colored by "group" and edges connecting bordering pairs, alongside a choropleth colored by "num_neighbors" using world_iso3.geojson with a point layer at "capital_lat", "capital_lon" where the points are sized by "capital_population (2025)" from countries_latest.csv and colored by "region".*

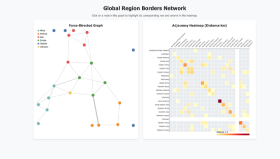
Datasets: `region_border_network.json, countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.49 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.34 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.05 |
| Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.28 |

Raiven


ChatGPT


Claude

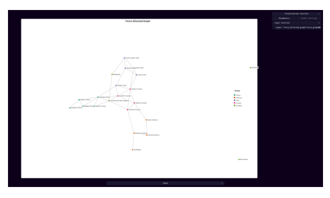

Gemini

**I50** Views: 2

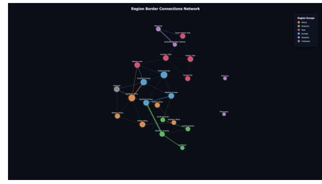
*Present borders two ways: (1) A force-directed graph from region_border_network.json with region nodes colored by "group" and edges weighted by "value". (2) An adjacency heatmap from region_border_edges.csv with "region_a" on one axis, "region_b" on the other, and cells colored by "distance_km". Selecting a region node in the graph should highlight its row and column in the heatmap.*

Datasets: `region_border_network.json, region_border_edges.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.36 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.28 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.99 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.21 |


Raiven

ChatGPT


Claude

Gemini

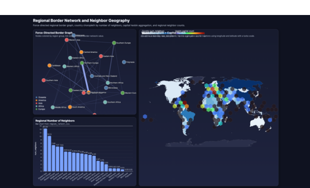
**I51** Views: 1

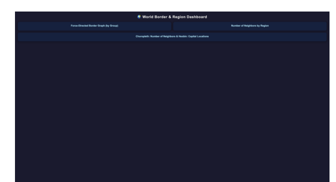
*Build a force-directed graph of border connections using region_border_network.json with region nodes colored by "group".*

Datasets: `region_border_network.json`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.26 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.81 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.67 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.78 |

Raiven


ChatGPT


Claude

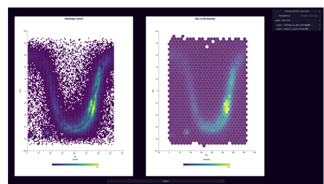
Gemini

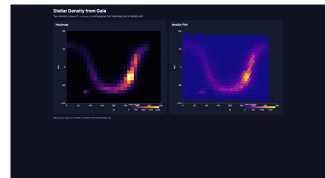
**I52** Views: 3

*Make a force-directed border graph from region_border_network.json with nodes colored by "group", a choropleth colored by "num_neighbors" from countries_latest.csv and world_iso3.geojson with a hexbin layer aggregating "capital_lat" using the "turbo" color scheme and "capital_lon", and a bar chart of "num_neighbors" from region_latest.csv.*

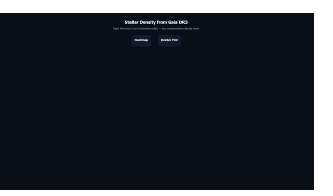
Datasets: `region_border_network.json, region_latest.csv, world_iso3.geojson, countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.45 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.09 |
| Claude | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 0.94 |
| Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.28 |

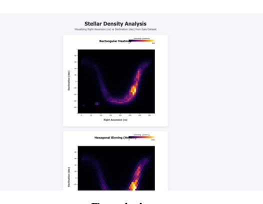

Raiven



ChatGPT



Claude



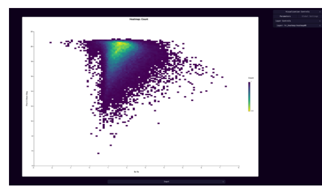
Gemini

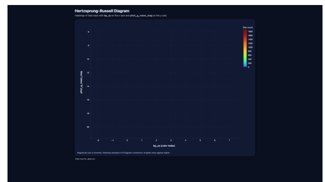
**I53** Views: 2

*Render two views of stellar density from gaia.csv: a heatmap with "ra" on the x-axis and "dec" on the y-axis, and a hexbin plot of the same "ra" versus "dec" coordinates.*

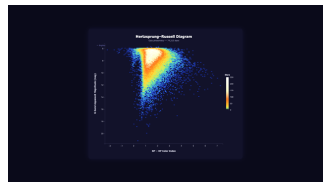
Datasets: `gaia.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.24 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.70 |
| Claude | 0.53 | 0.6 | 0.6 | 0.4 | 1.00 | 0.81 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.83 |

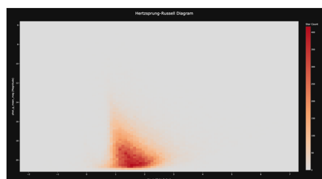

Raiven



ChatGPT



Claude



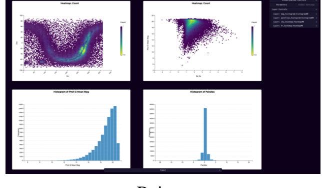
Gemini

**I54** Views: 1

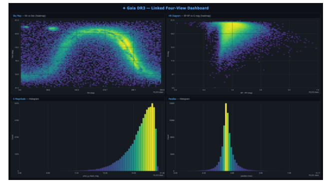
*Generate a Hertzsprung-Russell diagram from gaia.csv as a heatmap with "bp_rp" on the x-axis and "phot_g_mean_mag" on the y-axis.*

Datasets: `gaia.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| ChatGPT | 0.67 | 0.6 | 0.6 | 0.8 | 0.40 | 0.62 |
| Claude | 0.93 | 1.0 | 1.0 | 0.8 | 1.00 | 0.67 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.54 |


Raiven

ChatGPT


Claude

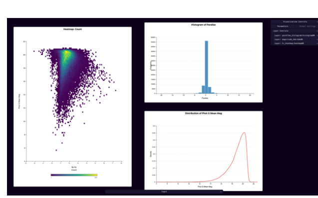
Gemini

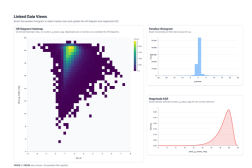
**I55** Views: 4

*Wire up a four-view linked dashboard from gaia.csv: a heatmap of "ra" versus "dec" (sky map), a heatmap of "bp_rp" versus "phot_g_mean_mag" (HR diagram), a histogram of "phot_g_mean_mag", and a histogram of "parallax". Brushing in any view should filter the others.*

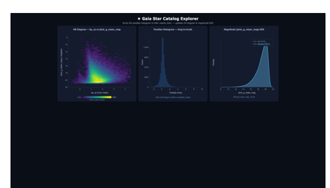
Datasets: `gaia.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.93 | 1.0 | 0.9 | 0.9 | 1.00 | 0.30 |
| ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 1.00 | 1.09 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.27 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.32 |

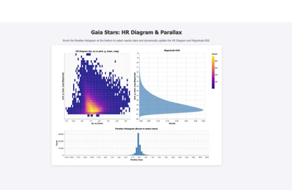

Raiven



ChatGPT



Claude



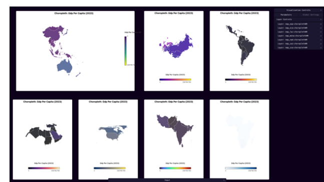
Gemini

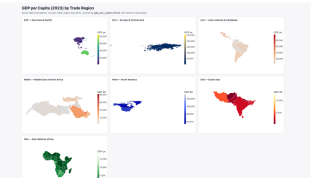
**I56** Views: 3

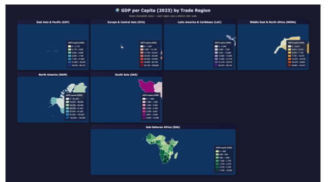
*Link three views from gaia.csv: a heatmap of "bp_rp" versus "phot_g_mean_mag" (HR diagram), a histogram of "parallax", and a KDE of "phot_g_mean_mag". Brushing the parallax histogram to select nearby stars should update the HR diagram and the magnitude KDE.*

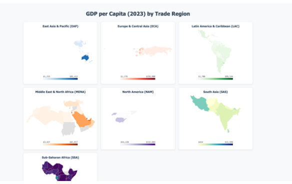
Datasets: `gaia.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.96 | 1.0 | 0.9 | 0.9 | 1.00 | 0.30 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.92 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.38 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.28 |


Raiven


ChatGPT


Claude

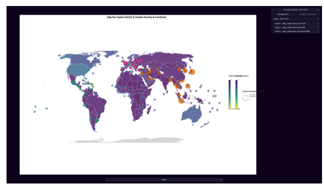

Gemini

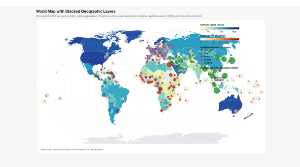
**I57** Views: 7

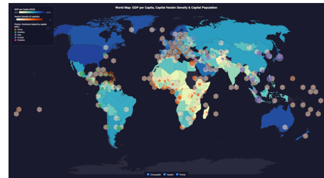
*Tile seven choropleth views from countries_latest.csv, one per trade-region GeoJSON file (EAP.geojson, ECA.geojson, LAC.geojson, MENA.geojson, NAM.geojson, SAS.geojson, SSA.geojson), each colored by "gdp_per_capita (2023)". Use different colorscales for each choropleth.*

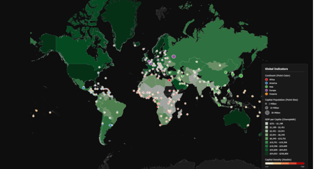
Datasets: `countries_latest.csv, EAP.geojson, ECA.geojson, LAC.geojson, MENA.geojson, NAM.geojson, SAS.geojson, SSA.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.72 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.48 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.80 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.39 |


Raiven


ChatGPT


Claude

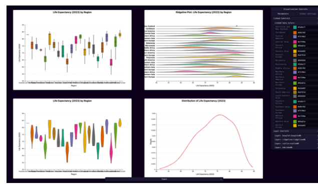

Gemini

**I58** Views: 1

*Stack three geographic layers in a single view from countries_latest.csv and world_iso3.geojson: a choropleth colored by "gdp_per_capita (2023)", a hexbin layer aggregating "capital_lat" and "capital_lon", and a point layer at "capital_lat", "capital_lon" sized by "capital_population (2025)" and colored by "continent".*

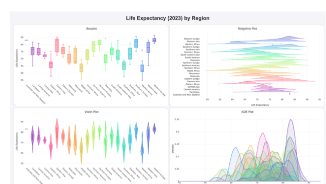
Datasets: `countries_latest.csv, world_iso3.geojson`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.42 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.83 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.99 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.65 |


Raiven

ChatGPT

Claude

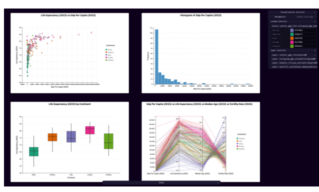

Gemini

**I59** Views: 4

*Display "life_expectancy (2023)" from countries_latest.csv four ways grouped by "region": a boxplot, a ridgeline plot (region on y axis, life expectancy on x axis), a violin plot, and a KDE plot, all sharing a consistent region color mapping.*

Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.97 | 1.0 | 0.9 | 0.9 | 1.00 | 0.38 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.04 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.95 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.35 |


Raiven

ChatGPT

Claude

Gemini

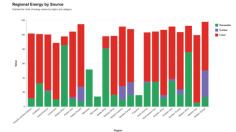
**I60** Views: 4

*Set up four linked views from countries_latest.csv: a scatter of "gdp_per_capita (2023)" versus "life_expectancy (2023)" colored by "continent", a stacked histogram of "gdp_per_capita (2023)" colored by "continent", a boxplot of "life_expectancy (2023)" grouped by "continent", and a parallel-coordinates plot of "gdp_per_capita (2023)", "life_expectancy (2023)", "median_age (2023)", and "fertility_rate (2023)" colored by "continent". Brushing any view should filter the others.*

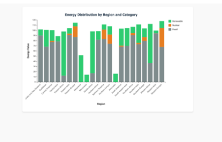
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.97 | 1.0 | 0.9 | 1.0 | 1.00 | 0.33 |
| ChatGPT | 0.73 | 0.8 | 0.7 | 0.8 | 1.00 | 1.32 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.30 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 3.03 |

Raiven


ChatGPT

Claude

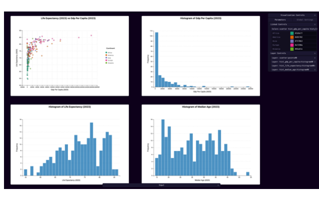

Gemini

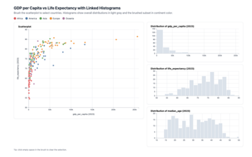
**I61** Views: 1

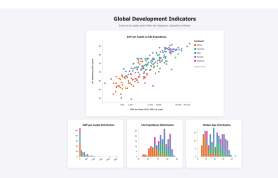
*Create a stacked bar chart from region_energy.csv with "region" on the x-axis, "value" on the y-axis, and bars stacked by "category" (renewable, nuclear, fossil), using distinct colors per energy type.*

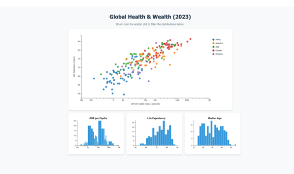
Datasets: `region_energy.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.26 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.40 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.50 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.90 |


Raiven


ChatGPT


Claude

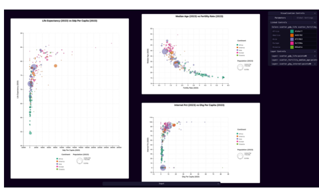

Gemini

**I62** Views: 4

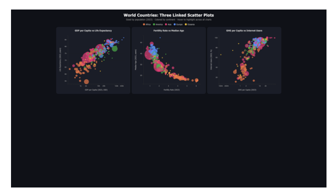
*Make a brushable scatter of "gdp_per_capita (2023)" versus "life_expectancy (2023)" colored by "continent" from countries_latest.csv and connect it to three separate histograms, showing distributions of "gdp_per_capita (2023)", "life_expectancy (2023)", and "median_age (2023)".*

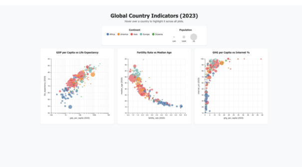
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.33 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.77 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.84 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.20 |


Raiven

ChatGPT


Claude

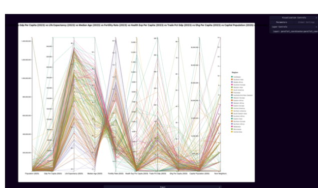

Gemini

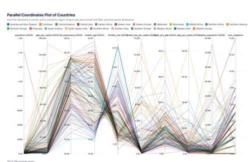
**I63** Views: 3

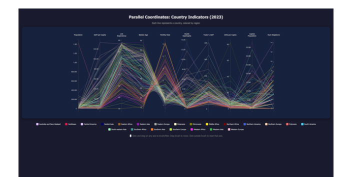
*Show three linked scatter plots from countries_latest.csv, all sized by "population (2023)" and colored by "continent": (1) "gdp_per_capita (2023)" versus "life_expectancy (2023)", (2) "fertility_rate (2023)" versus "median_age (2023)", (3) "ghg_per_capita (2023)" versus "internet_pct (2023)".*

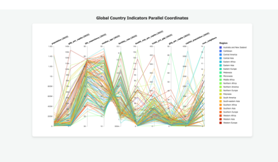
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.32 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.66 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.72 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.04 |


Raiven


ChatGPT


Claude

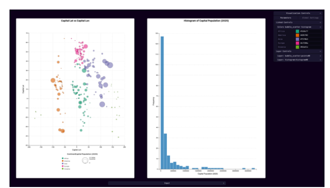

Gemini

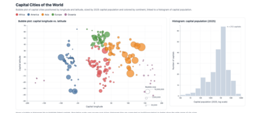
**I64** Views: 1

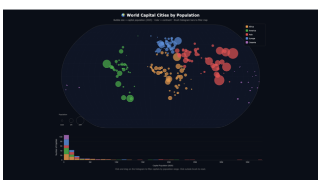
*Create a parallel-coordinates plot from countries_latest.csv spanning "population (2023)", "gdp_per_capita (2023)", "life_expectancy (2023)", "median_age (2023)", "fertility_rate (2023)", "health_exp_per_capita (2023)", "trade_pct_gdp (2023)", "ghg_per_capita (2023)", "capital_population (2025)" and "num_neighbors" with each line representing a country and lines colored by "region". Make the plot brushable.*

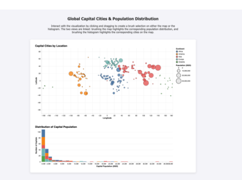
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| ChatGPT | 0.93 | 1.0 | 0.8 | 1.0 | 1.00 | 0.64 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.81 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.51 |


Raiven


ChatGPT


Claude


Gemini

**I65** Views: 2

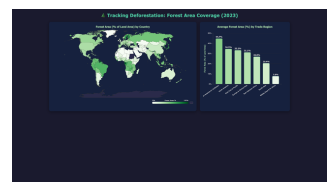
*Plot capital cities from countries_latest.csv as bubbles at y= "capital_lat", x= "capital_lon" sized by "capital_population (2025)" and colored by "continent", linked to a histogram of "capital_population (2025)".*

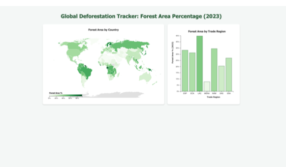
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.97 | 1.0 | 1.0 | 0.9 | 1.00 | 0.25 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.89 |
| Claude | 0.97 | 1.0 | 1.0 | 0.9 | 1.00 | 1.00 |
| Gemini | 0.97 | 1.0 | 1.0 | 0.9 | 1.00 | 1.24 |

Raiven

ChatGPT


Claude

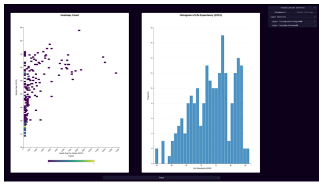

Gemini

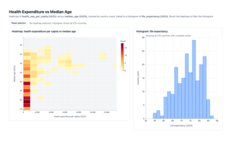
**I66** Views: 2

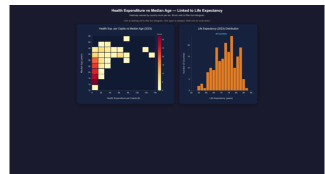
*Track deforestation: a choropleth colored by "forest_area_pct (2023)" using the "Greens" color scheme with country geometries from world_iso3.geojson and countries_latest.csv, alongside a bar chart from trade_region_latest.csv with "trade_region" on the x-axis and "forest_area_pct (2023)" on the y-axis.*

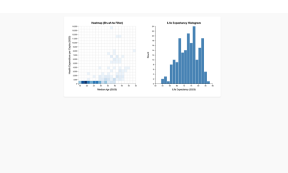
Datasets: `countries_latest.csv, world_iso3.geojson, trade_region_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.90 | 0.9 | 0.9 | 0.9 | 0.90 | 0.37 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.66 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.78 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.82 |


Raiven


ChatGPT


Claude

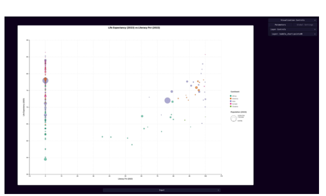

Gemini

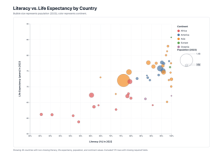
**I67** Views: 2

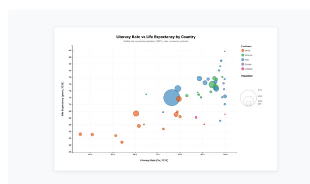
*Create a heatmap of "health_exp_per_capita (2023)" versus "median_age (2023)" from countries_latest.csv with cells colored by count, linked to a histogram of "life_expectancy (2023)". Brushing bins in the heatmap should filter the histogram.*

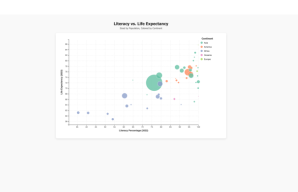
Datasets: `countries_latest.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.32 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.08 |
| Claude | 0.90 | 1.0 | 0.8 | 0.9 | 1.00 | 1.03 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.07 |


Raiven


ChatGPT


Claude

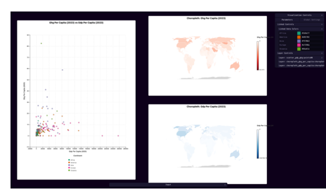
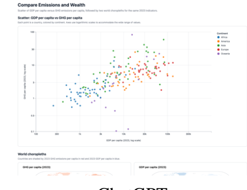

Gemini

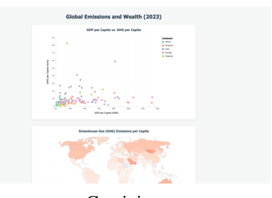
**I68** Views: 1

*Plot a bubble chart from countries_latest.csv with "literacy_pct (2022)" versus "life_expectancy (2023)", sized by "population (2023)" and colored by "continent".*

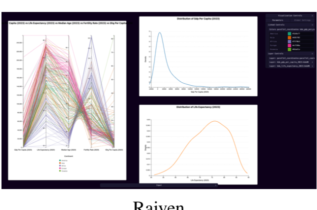
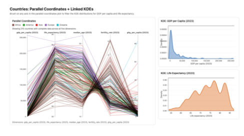
Datasets: `countries_latest.csv`

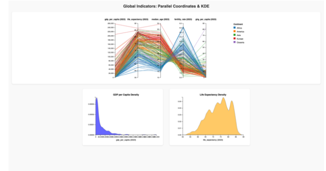
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.19 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.63 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.62 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.76 |


Raiven

ChatGPT

Claude

Gemini

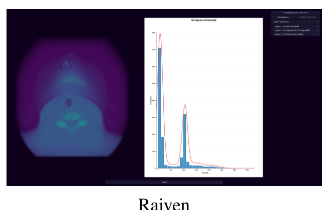
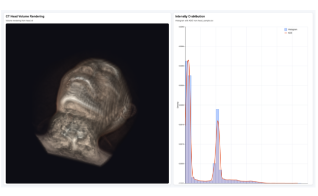
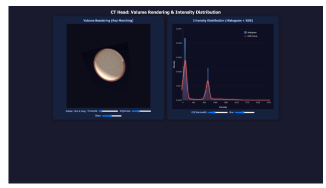
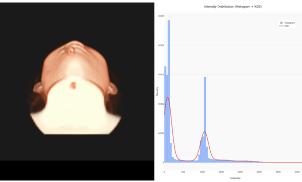
| **I69** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Compare emissions and wealth: a scatter of "gdp_per_capita (2023)" versus "ghg_per_capita (2023)" colored by "continent" from countries_latest.csv, followed by two choropleths using world_iso3.geojson — one colored by "ghg_per_capita (2023)" using a red sequential color scale, one colored by "gdp_per_capita (2023)" using a blue sequential color scale.* Datasets: `countries_latest.csv, world_iso3.geojson` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.36 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.81 |
| | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.93 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.07 |


Raiven

ChatGPT

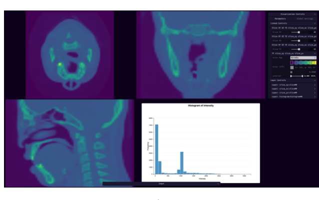
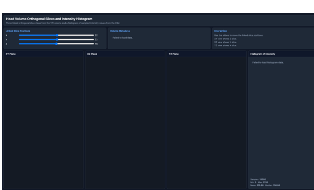
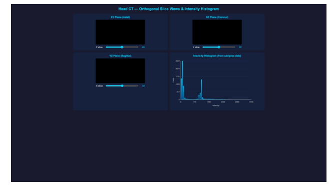
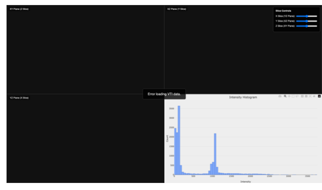
Claude

Gemini

| **I70** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create a parallel-coordinates plot from countries_latest.csv spanning "gdp_per_capita (2023)", "life_expectancy (2023)", "median_age (2023)", "fertility_rate (2023)", and "ghg_per_capita (2023)" with lines colored by "continent", linked to two KDE charts: one of "gdp_per_capita (2023)" in blue and one of "life_expectancy (2023)" in orange. Brushing the parallel coordinates should filter both KDE charts.* Datasets: `countries_latest.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.93 |
| | Claude | 0.33 | 0.3 | 0.3 | 0.3 | 1.00 | 0.89 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.53 |

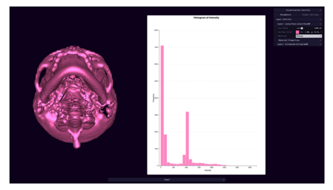
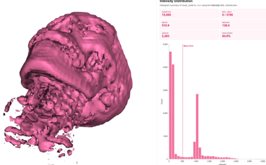
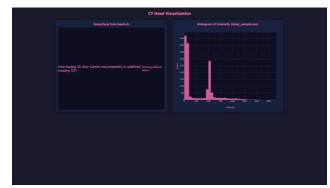
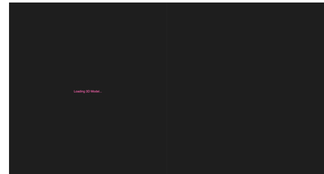
### D.6.3 Combined (C)


Raiven

ChatGPT

Claude

Gemini

| **C71** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *From head.vti, generate a volume rendering of the CT data, and then use head_sample.csv to produce a layered distribution plot showing a histogram of intensity with a KDE curve.* Datasets: `head.vti, head_sample.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.58 |
| | Claude | 0.83 | 0.9 | 0.9 | 0.7 | 0.90 | 2.02 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.87 |

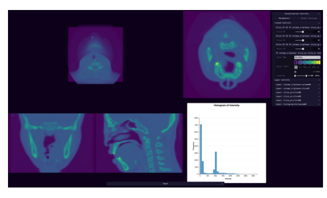
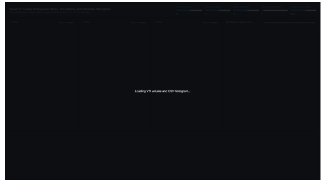

Raiven

ChatGPT

Claude

Gemini

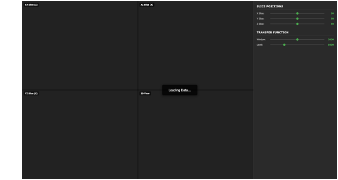
| **C72** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create seperate three orthogonal slice views in the XY, XZ, and YZ planes from head.vti, link their slice positions together, and also generate a histogram of "intensity" using head_sample.csv.* Datasets: `head.vti, head_sample.csv` | Raiven | 0.95 | 1.0 | 1.0 | 0.8 | 1.00 | 0.32 |
| | ChatGPT | 0.52 | 0.7 | 0.6 | 0.2 | 0.55 | 3.23 |
| | Claude | 0.62 | 0.6 | 0.7 | 0.6 | 1.00 | 1.52 |
| | Gemini | 0.65 | 0.7 | 0.7 | 0.6 | 0.55 | 2.41 |

Raiven | ChatGPT | Claude | Gemini

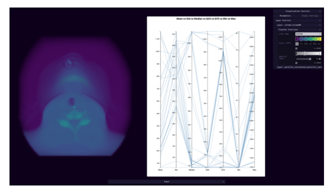
| **C73** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Render the CT isosurface in pink from head.vti, then summarize head_sample.csv using a histogram of "intensity" colored pink.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.42 |
| Datasets: `head.vti, head_sample.csv` | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.62 |
| | Claude | 0.77 | 0.8 | 0.8 | 0.7 | 0.70 | 0.53 |
| | Gemini | 0.53 | 0.6 | 0.6 | 0.4 | 0.40 | 1.09 |

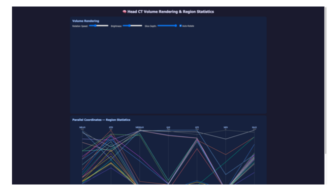
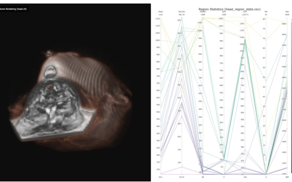





Raiven | ChatGPT | Claude | Gemini

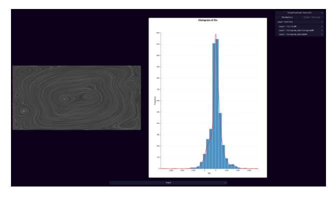
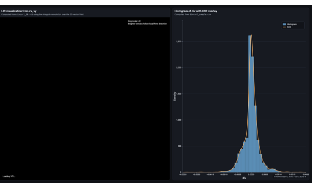
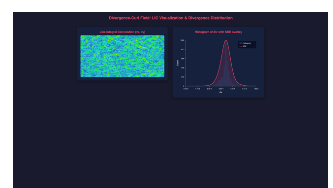
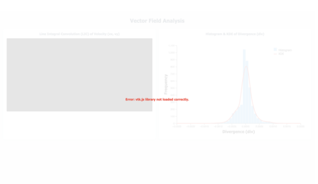
| **C74** Views: 5 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Using head.vti, assemble three linked slice views (XY, XZ, YZ) along with a 3D view that combines volume rendering and triplanar slices, making sure the slice positions and transfer function remain synchronized across all four views, and also include a histogram of intensity from head_sample.csv.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.40 |
| Datasets: `head.vti, head_sample.csv` | ChatGPT | 0.15 | 0.0 | 0.4 | 0.0 | 0.40 | 1.14 |
| | Claude | 0.57 | 0.5 | 0.5 | 0.7 | 0.40 | 2.52 |
| | Gemini | 0.44 | 0.4 | 0.4 | 0.6 | 0.40 | 3.14 |



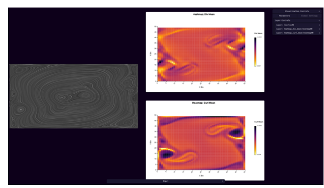
Raiven | ChatGPT | Claude | Gemini

| **C75** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Produce a volume rendering from head.vti, and then visualize the statistics in head_region_stats.csv as a parallel coordinates chart including mean, std, median, q25, q75, min, and max.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.27 |
| Datasets: `head.vti, head_region_stats.csv` | ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 0.65 |
| | Claude | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 1.57 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.11 |

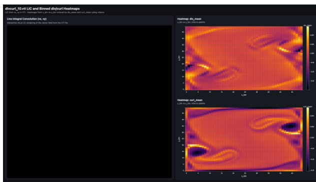





Raiven | ChatGPT | Claude | Gemini

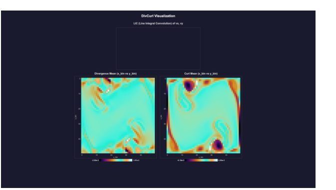
| **C76** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *With divcurl_10.vti, compute an LIC visualization based on vx and vy, and alongside it create a histogram of div with an overlaid KDE using divcurl_sample.csv.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| Datasets: `divcurl_10.vti, divcurl_sample.csv` | ChatGPT | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 1.17 |
| | Claude | 0.77 | 0.7 | 0.7 | 0.9 | 0.80 | 1.40 |
| | Gemini | 0.53 | 0.8 | 0.8 | 0.0 | 0.70 | 2.66 |



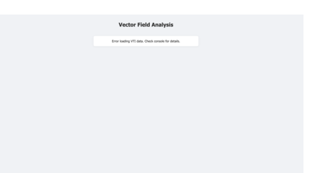
Raiven



ChatGPT



Claude



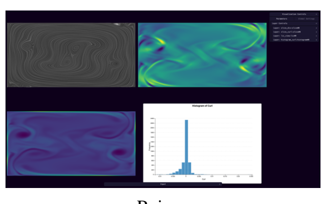
Gemini

**C77** Views: 3

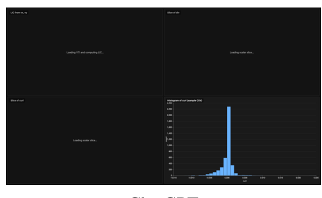
*Generate an LIC view from divcurl_10.vti using vx and vy. Also represent divcurl_binned_2d.csv as two heatmaps of x_bin vs. y_bin, one colored by div_mean and one colored by curl_mean, both using color palette inferno.*

Datasets: `divcurl_10.vti, divcurl_binned_2d.csv`

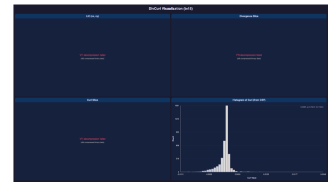
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.25 |
| ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.72 |
| Claude | 0.91 | 1.0 | 0.9 | 0.9 | 1.00 | 2.02 |
| Gemini | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 2.63 |



Raiven



ChatGPT



Claude

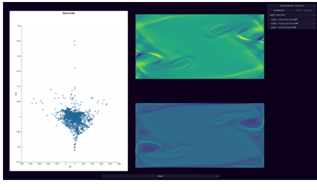
Gemini

**C78** Views: 4

*Using divcurl_15.vti, display four views consisting of an LIC using vx and vy, a slice of div, and a slice of curl, and a histogram of curl from divcurl_sample.csv.*

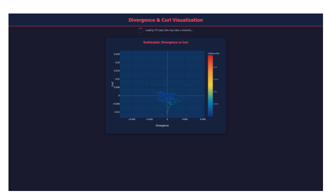
Datasets: `divcurl_15.vti, divcurl_sample.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.42 |
| ChatGPT | 0.65 | 0.6 | 0.7 | 0.7 | 0.85 | 0.89 |
| Claude | 0.70 | 0.7 | 0.7 | 0.7 | 0.55 | 1.81 |
| Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 3.23 |

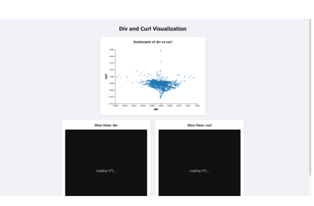


Raiven

ChatGPT



Claude



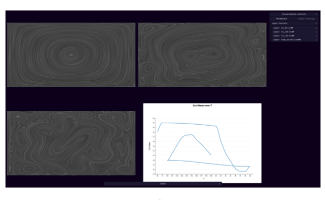
Gemini

**C79** Views: 3

*Make a scatterplot of div versus curl generated from divcurl_sample.csv. Also, show two slice views from divcurl_10.vti, one visualizing div and the other curl.*

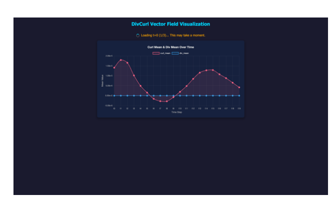
Datasets: `divcurl_10.vti, divcurl_sample.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| ChatGPT | 0.73 | 0.7 | 0.7 | 0.7 | 0.60 | 1.03 |
| Claude | 1.00 | 1.0 | 1.0 | 1.0 | 0.87 | 1.64 |
| Gemini | 0.73 | 0.7 | 0.7 | 0.7 | 0.60 | 2.16 |



Raiven

ChatGPT



Claude

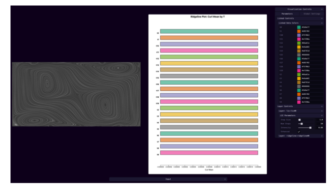
Gemini

**C80** Views: 4

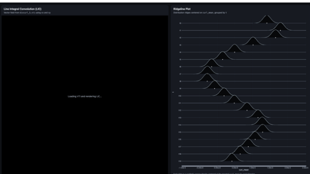
*Load divcurl_0.vti, divcurl_10.vti, and divcurl_19.vti to create three LIC visualizations based on vx and vy, and use divcurl_stats.csv to build a time-series chart plotting curl_mean and div_mean against t.*

Datasets: `divcurl_0.vti, divcurl_10.vti, divcurl_19.vti, divcurl_stats.csv`

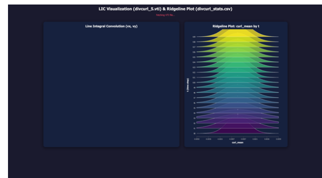
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.93 | 0.9 | 0.9 | 0.9 | 0.50 | 0.32 |
| ChatGPT | 0.70 | 0.7 | 0.7 | 0.7 | 0.55 | 0.80 |
| Claude | 0.70 | 0.7 | 0.7 | 0.7 | 0.55 | 1.26 |
| Gemini | 0.70 | 0.7 | 0.7 | 0.7 | 0.55 | 2.16 |



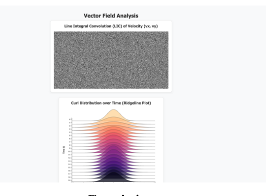
Raiven



ChatGPT



Claude



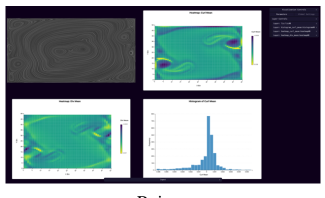
Gemini

**C81** Views: 2

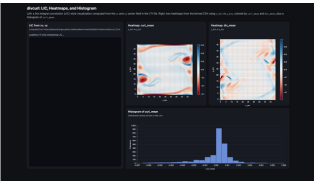
*Create a LIC from divcurl_5.vti using vx and vy, then generate a ridgeline plot from divcurl_stats.csv with x = curl_mean and y = t.*

Datasets: `divcurl_5.vti, divcurl_stats.csv`

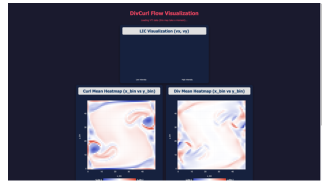
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 0.90 | 0.9 | 1.0 | 0.8 | 1.00 | 0.29 |
| ChatGPT | 0.73 | 0.7 | 0.7 | 0.8 | 0.70 | 0.68 |
| Claude | 0.93 | 0.9 | 0.9 | 1.0 | 0.80 | 1.80 |
| Gemini | 0.83 | 0.7 | 0.9 | 0.9 | 0.70 | 2.45 |



Raiven



ChatGPT



Claude

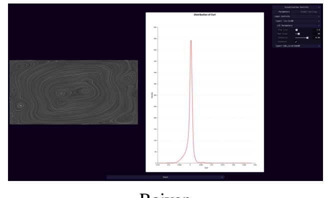
Gemini

**C82** Views: 4

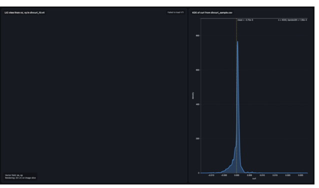
*Begin with an LIC visualization from divcurl_10.vti computed from vx and vy, then construct two heatmaps from divcurl_binned_2d.csv using x_bin versus y_bin , one colored by curl_mean and the other by div_mean. Also, add a histogram of curl_mean.*

Datasets: `divcurl_10.vti, divcurl_binned_2d.csv`

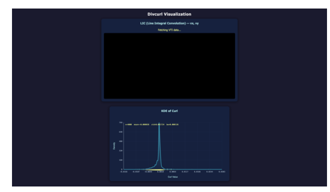
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.37 |
| ChatGPT | 0.93 | 0.9 | 0.9 | 0.9 | 1.00 | 0.97 |
| Claude | 0.67 | 1.0 | 1.0 | 0.0 | 1.00 | 1.72 |
| Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 2.51 |



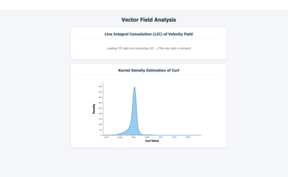
Raiven



ChatGPT



Claude



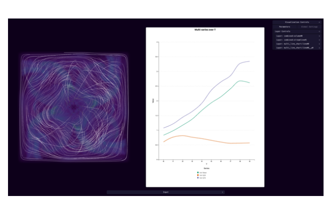
Gemini

**C83** Views: 2

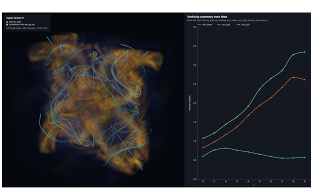
*Build an LIC view using vx and vy from divcurl_10.vti, and also make a KDE curve for curl based on data from divcurl_sample.csv.*

Datasets: `divcurl_10.vti, divcurl_sample.csv`

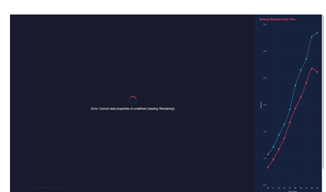
| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.28 |
| ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 0.58 |
| Claude | 0.87 | 0.9 | 0.9 | 0.8 | 0.90 | 1.79 |
| Gemini | 0.87 | 0.9 | 0.9 | 0.8 | 1.00 | 2.03 |



Raiven



ChatGPT



Claude

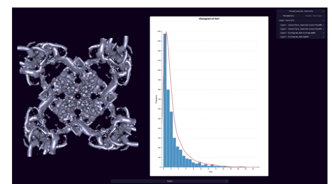
Gemini

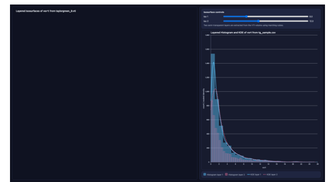
**C84** Views: 2

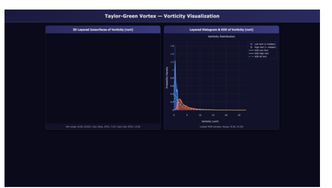
*From taylorgreen_7.vti, make a volume of "vort" and also layer streamlines using vx = "ux", vy = "uy", and vz = "uz". Then use tg_stats.csv to create a multi-line chart of vort_mean, vort_q25 and vort_q75 over t.*

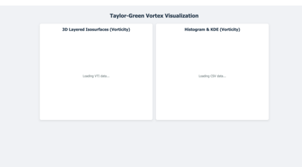
Datasets: `taylorgreen_7.vti, tg_stats.csv`

| System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|
| Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.31 |
| ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.99 |
| Claude | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 1.24 |
| Gemini | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 1.99 |

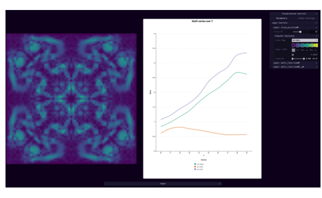

Raiven

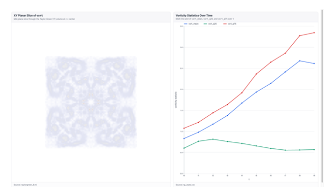

ChatGPT


Claude

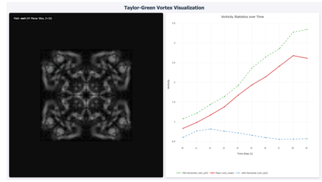

Gemini

| **C85** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Construct two layered isosurfaces of vort from taylorgreen_9.vti. Then make a layered histogram and KDE of vort generated from tg_sample.csv.* | Raiven | 0.97 | 1.0 | 1.0 | 0.9 | 1.00 | 0.30 |
| | ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 1.00 | 0.82 |
| | Claude | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 1.11 |
| Datasets: `taylorgreen_9.vti, tg_sample.csv` | Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 1.65 |

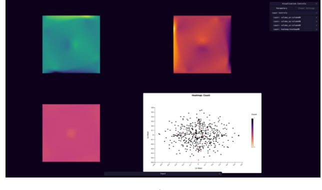

Raiven

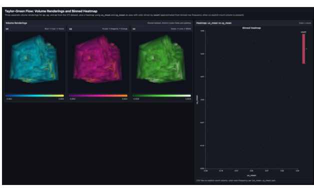

ChatGPT

Claude

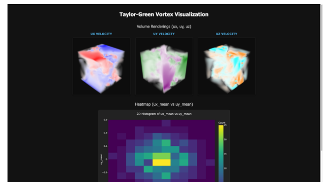

Gemini

| **C86** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Make an XY planar slice of vort from taylorgreen_9.vti and use tg_stats.csv to create a multi-line plot of vort_mean, vort_q25, and vort_q75 over t.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.37 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.69 |
| | Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 1.11 |
| Datasets: `taylorgreen_9.vti, tg_stats.csv` | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.36 |

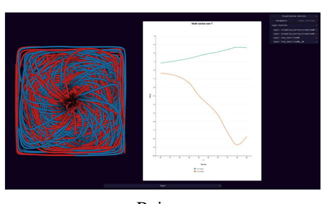

Raiven

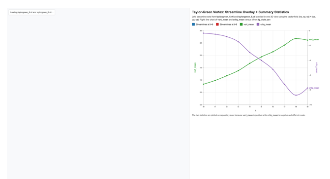

ChatGPT

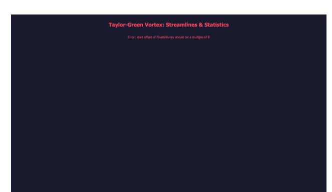
Claude

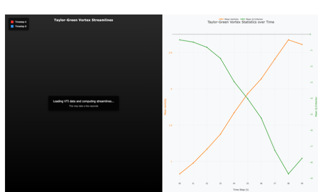

Gemini

| **C87** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Visualize ux, uy, and uz from taylorgreen_9.vti as three separate volume renderings, each with a unique color palette, and use tg_t9_binned.csv to create a heatmap where ux_mean and uy_mean define the axes and count controls the color scale.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.37 |
| | ChatGPT | 0.88 | 0.9 | 0.9 | 0.8 | 1.00 | 1.01 |
| | Claude | 0.70 | 0.7 | 0.7 | 0.7 | 0.55 | 1.86 |
| Datasets: `taylorgreen_9.vti, tg_t9_binned.csv` | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.95 |

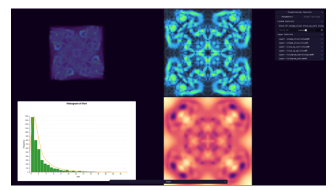

Raiven

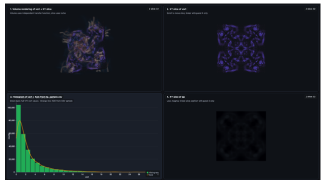

ChatGPT


Claude

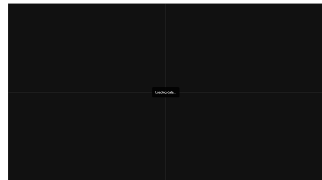

Gemini

| **C88** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Overlay streamline sets computed using vx = ux, vy = uy, and vz = uz in taylorgreen_6.vti and taylorgreen_8.vti within a single view using a different color per timestep, and then construct a line chart from tg_stats.csv plotting vort_mean and critq_mean against t.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.43 |
| | ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 0.80 |
| | Claude | 0.00 | 0.0 | 0.0 | 0.0 | 0.40 | 1.41 |
| Datasets: `taylorgreen_6.vti, taylorgreen_8.vti, tg_stats.csv` | Gemini | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 2.64 |

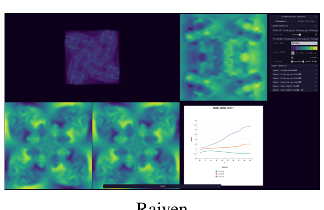

Raiven

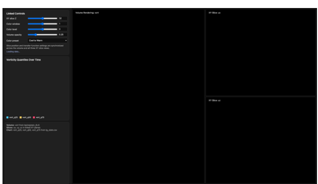

ChatGPT

Claude


Gemini

| **C89** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Display four views of the Taylor-Green vortex from taylorgreen_9.vti and tg_sample.csv: (1) a volume rendering of "vort" with an XY slice using "turbo", (2) the same XY slice of "vort" using "turbo", (3) a histogram of "vort" in green with an overlaid orange KDE chart from tg_sample.csv, and (4) an XY slice of "pp" using "magma". Link only the slice positions between the slice views so scrolling one updates the other. Do not link transfer functions.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.53 |
| | ChatGPT | 0.90 | 1.0 | 0.8 | 0.8 | 1.00 | 0.90 |
| | Claude | 0.60 | 0.6 | 0.6 | 0.7 | 0.55 | 1.66 |
| Datasets: `tg_sample.csv, taylorgreen_9.vti` | Gemini | 0.30 | 0.5 | 0.5 | 0.0 | 0.00 | 6.00 |

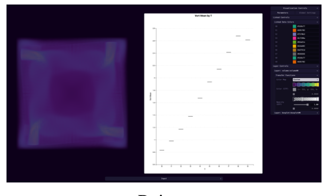

Raiven

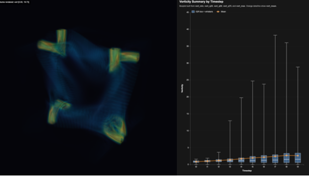

ChatGPT

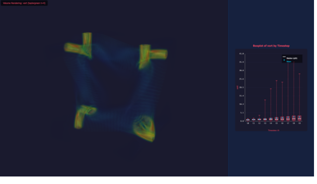
Claude

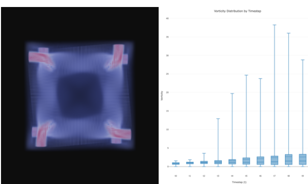
Gemini

| **C90** Views: 5 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Display a volume rendering of vort from taylorgreen_9.vti together with three linked XY planar slice views of ux, uy, and uz, ensure slice positions and transfer functions are synchronized across the views, and include a multi-line chart of vort_q25, vort_q50, and vort_q75 over t from tg_stats.csv.* | Raiven | 0.99 | 1.0 | 1.0 | 1.0 | 1.00 | 0.41 |
| | ChatGPT | 0.44 | 0.5 | 0.4 | 0.4 | 0.40 | 1.15 |
| | Claude | 0.17 | 0.0 | 0.5 | 0.0 | 0.52 | 1.63 |
| Datasets: `taylorgreen_9.vti, tg_stats.csv` | Gemini | 0.43 | 0.4 | 0.4 | 0.4 | 0.40 | 2.18 |

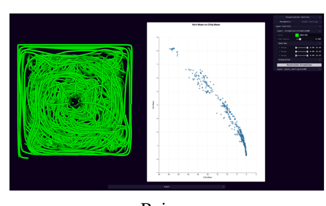

Raiven

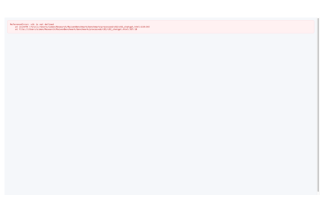

ChatGPT

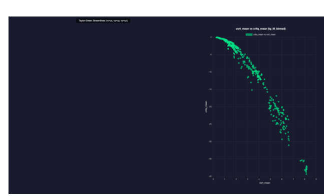

Claude

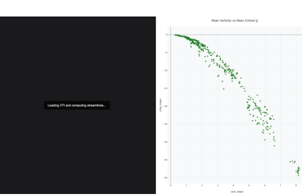

Gemini

| **C91** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Render vort as a volume from taylorgreen_4.vti and use tg_stats.csv to create a boxplot of vort_mean by timestep t using the fields vort_min, vort_q25, vort_q50, vort_q75, and vort_max.* | Raiven | 0.87 | 0.9 | 0.8 | 0.9 | 1.00 | 0.32 |
| | ChatGPT | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.12 |
| Datasets: `taylorgreen_4.vti, tg_stats.csv` | Claude | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.84 |
| | Gemini | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 1.48 |

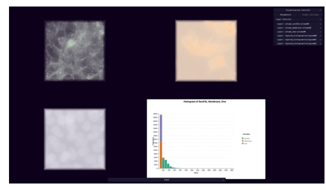

Raiven

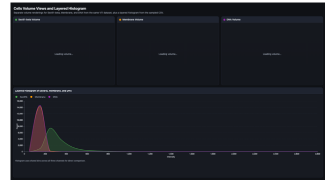

ChatGPT

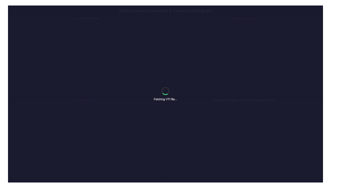

Claude


Gemini

| **C92** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Compute thick, green streamlines using vx = ux, vy = uy, and vz = uz in taylorgreen_9.vti and produce a point chart from tg_t9_binned.csv plotting vort_mean against critq_mean.* | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.29 |
| | ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.99 |
| Datasets: `taylorgreen_9.vti, tg_t9_binned.csv` | Claude | 0.80 | 0.8 | 0.8 | 0.8 | 1.00 | 0.86 |
| | Gemini | 0.73 | 0.6 | 0.8 | 0.8 | 0.70 | 1.85 |

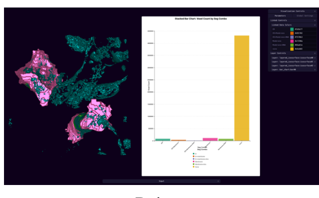

Raiven

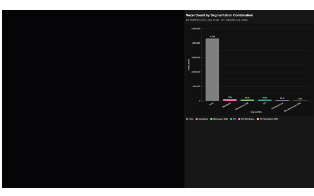

ChatGPT

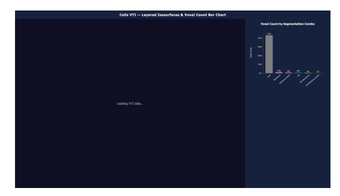

Claude

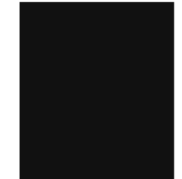
Gemini

| **C93** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create a volume for Sec61-beta using green color palette, a volume for Membrane using orange color palette, and a volume for DNA using purple color palette, all of which are seperate views but use cells.vti. Also add a layered histogram of Sec61b, Membrane, and DNA in a single chart using cells_channel_sample.csv.* Datasets: `cells.vti, cells_channel_sample.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| | ChatGPT | 0.83 | 0.6 | 0.9 | 1.0 | 1.00 | 0.89 |
| | Claude | 0.20 | 0.0 | 0.6 | 0.0 | 0.40 | 0.96 |
| | Gemini | 0.82 | 1.0 | 0.8 | 0.7 | 1.00 | 1.34 |

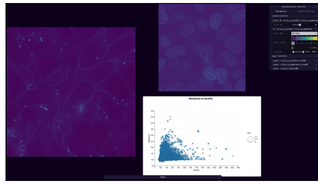

Raiven

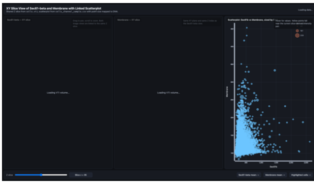

ChatGPT

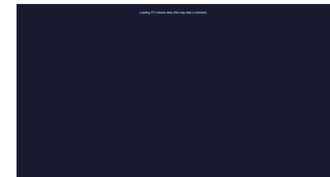

Claude

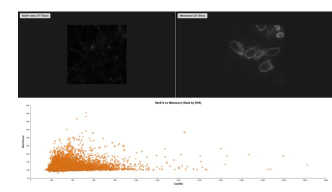

Gemini

| **C94** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Make layered isosurfaces using seg_Sec61-beta with color = teal, seg_Membrane with color = pink, and seg_DNA with color = green from cells.vti and construct a bar chart from cells_seg_stats.csv showing voxel_count that is colored by seg_combo.* Datasets: `cells.vti, cells_seg_stats.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.34 |
| | ChatGPT | 0.00 | 0.0 | 0.0 | 0.0 | 0.00 | 0.70 |
| | Claude | 0.90 | 0.9 | 0.9 | 0.9 | 1.00 | 0.77 |
| | Gemini | 0.50 | 0.5 | 0.4 | 0.6 | 0.40 | 1.68 |

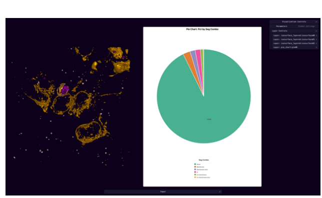

Raiven

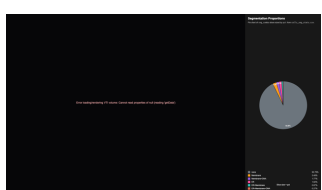

ChatGPT


Claude

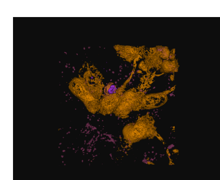
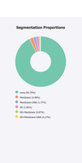

Gemini

| **C95** Views: 3 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Generate an XY slice view of Sec61-beta and a linked XY slice of Membrane from cells.vti and accompany it with a scatterplot of Sec61b versus Membrane, sized by DNA, created from cells_channel_sample.csv.* Datasets: `cells.vti, cells_channel_sample.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.37 |
| | ChatGPT | 0.91 | 0.9 | 0.9 | 0.9 | 1.00 | 1.35 |
| | Claude | 0.51 | 0.5 | 0.5 | 0.6 | 0.33 | 1.46 |
| | Gemini | 0.91 | 0.9 | 1.0 | 0.9 | 1.00 | 1.87 |

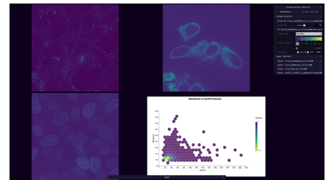

Raiven

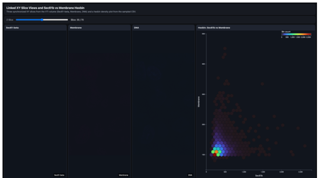

ChatGPT

Claude



Gemini

| **C96** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Make three layered isosurfaces from cells.vti, one using Sec61-beta and colored pink, another using Membrane and colored orange, and one using DNA colored purple, also summarize segmentation proportions with a pie chart built from cells_seg_stats.csv using seg_combo for slices sized by pct.* Datasets: `cells.vti, cells_seg_stats.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.35 |
| | ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 0.68 |
| | Claude | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 0.61 |
| | Gemini | 0.93 | 1.0 | 0.9 | 0.9 | 1.00 | 1.57 |

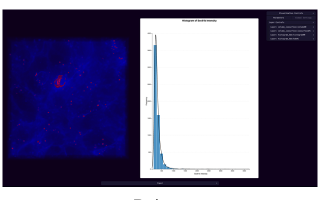

Raiven

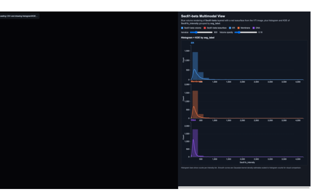

ChatGPT

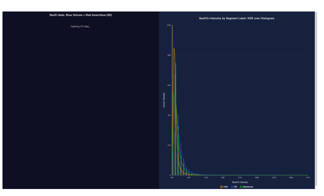
Claude

Gemini

| **C97** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Create three linked XY slice views for Sec61-beta, Membrane, and DNA from cells.vti with synchronized slice positions, and add a hexbin plot of Sec61b versus Membrane using cells_channel_sample.csv.* Datasets: `cells.vti, cells_channel_sample.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.30 |
| | ChatGPT | 0.67 | 0.6 | 0.6 | 0.8 | 1.00 | 0.96 |
| | Claude | 0.50 | 0.5 | 0.5 | 0.6 | 0.40 | 1.24 |
| | Gemini | 0.50 | 0.5 | 0.5 | 0.6 | 0.40 | 1.35 |

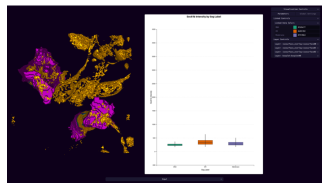

Raiven

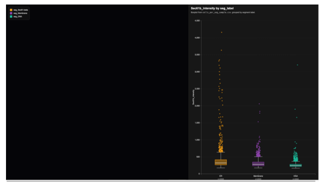

ChatGPT

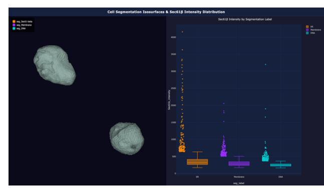

Claude

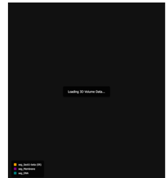
Gemini

| **C98** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Visualize Sec61-beta as a blue colored volume layered over red colored isosurface of Sec61-beta from cells.vti and overlay KDE over histogram of Sec61b_intensity grouped by seg_label using cells_per_seg_sample.csv.* Datasets: `cells.vti, cells_per_seg_sample.csv` | Raiven | 0.97 | 1.0 | 1.0 | 0.9 | 1.00 | 0.34 |
| | ChatGPT | 0.80 | 0.8 | 0.8 | 0.8 | 1.00 | 1.09 |
| | Claude | 0.80 | 0.8 | 0.8 | 0.8 | 0.70 | 1.04 |
| | Gemini | 0.60 | 0.6 | 0.6 | 0.6 | 0.40 | 2.22 |

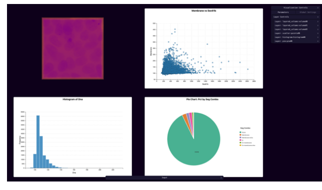

Raiven

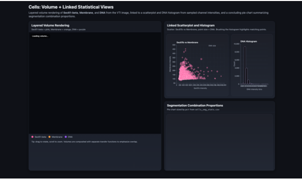

ChatGPT

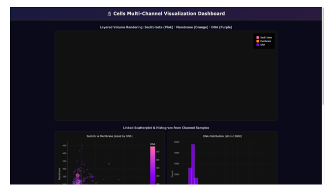

Claude

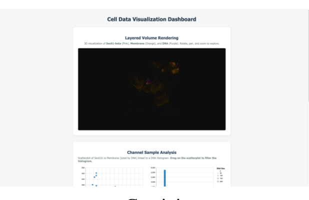

Gemini

| **C99** Views: 2 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Overlay isosurface representations of seg_Sec61-beta color = orange, seg_Membrane color = purple, and seg_DNA color = teal from cells.vti and present a boxplot of Sec61b_intensity grouped by seg_label based on cells_per_seg_sample.csv.* Datasets: `cells.vti, cells_per_seg_sample.csv` | Raiven | 1.00 | 1.0 | 1.0 | 1.0 | 1.00 | 0.31 |
| | ChatGPT | 0.73 | 0.6 | 0.8 | 0.8 | 1.00 | 0.94 |
| | Claude | 0.90 | 0.9 | 0.9 | 0.9 | 1.00 | 0.60 |
| | Gemini | 0.57 | 0.6 | 0.5 | 0.6 | 0.40 | 1.70 |


Raiven


ChatGPT


Claude


Gemini

| **C100** Views: 4 | System | VMPC | $G_1$ | $G_2$ | $G_3$ | VLM | Time |
|---|---|---|---|---|---|---|---|
| *Display layered volumes of Sec61-beta with color = pink, Membrane with color= orange, and DNA with color = purple from cells.vti, generate a scatterplot of Sec61b versus Membrane sixed by DNA which is linked to a histogram of DNA using cells_channel_sample.csv, and conclude with a pie chart of seg_combo proportions sized by pct from cells_seg_stats.csv.* Datasets: `cells.vti, cells_channel_sample.csv, cells_seg_stats.csv` | Raiven | 0.98 | 1.0 | 1.0 | 0.9 | 1.00 | 0.35 |
| | ChatGPT | 0.72 | 0.6 | 0.8 | 0.8 | 0.85 | 1.46 |
| | Claude | 0.85 | 0.9 | 0.8 | 0.8 | 0.85 | 0.96 |
| | Gemini | 0.95 | 1.0 | 0.9 | 0.9 | 1.00 | 1.25 |

# E USER STUDY

## E.1 Setup

All sessions were conducted between March 10–16, 2026 on a standardized workstation (MacBook Pro 14-inch, November 2024, Apple M4, 24 GB RAM). The workstation provided unlimited access to the following tools and services: ChatGPT (Pro subscription, $200/mo), Claude and Claude Code (Max Plan 5x, $100/mo), Gemini (Google AI Ultra, $125/mo), Cursor (Pro+, $60/mo), Codex, Antigravity, VS Code, and Google Colab. All AI services were on paid plans ensuring access to the highest-tier models available on each platform. Raiven was served locally on the same machine.

## E.2 Task Descriptions

As referenced by Section 8.1. Participants were given printed task sheets describing the target dashboard they were asked to recreate. Each sheet specified the required views, the data files available in the task folder on their computer, the visual encodings expected for each chart, and any interactions or cross-view linking required. Participants could refer to these sheets throughout the study.

### E.2.1 Task 1: InfoVis Dashboard

Task 1 required participants to build a four-view interactive dashboard using world development data. The dashboard comprised: (1) a choropleth world map encoding population by country, (2) a bubble plot of GDP per capita versus life expectancy, with bubbles grouped by world region and sized by population, (3) a histogram of life expectancy, linked to the bubble plot, and (4) a multi-line chart with year on the x-axis and average life expectancy on the y-axis, with lines colored by world region. The printed task sheet, including the target dashboard layout, data sources, and interaction requirements, is reproduced in Figure 13.

### E.2.2 Task 2: SciVis Dashboard

Task 2 required participants to build a four-view scientific visualization dashboard using volumetric CT skull data. The dashboard comprised: (1) a primary 3D view showing a volume rendering with a layered isosurface of the skull and triplanar slice overlays, (2) an XY slice view, (3) an XZ slice view, and (4) a YZ slice view. The three planar slice views were required to be linked to the primary 3D view such that scrolling through slices in any planar view updates the corresponding slice position in the 3D volume. The printed task sheet, including the target dashboard layout, data sources, and interaction requirements, is reproduced in Figure 14.

### E.2.3 Task 3: Combined InfoVis/SciVis Dashboard

Task 3 required participants to build a three-view dashboard combining scientific and information visualization using computational fluid dynamics data. The dashboard comprised: (1) a primary 3D view showing a volume rendering of vorticity with layered streamlines computed from the velocity components `vx`, `vy`, and `vz`, (2) a second 3D view showing layered streamlines from two different timesteps overlaid in the same scene, again using `vx`, `vy`, and `vz` for both, and (3) a violin plot with time on the x-axis and vorticity on the y-axis, summarizing the distribution of vorticity values across timesteps. The printed task sheet, including the target dashboard layout and data sources, is reproduced in Figure 15.

## E.3 Baseline Tool Selection

All seven participants reported writing code as their primary method of creating visualizations; only two regularly incorporated AI tools into their visualization workflow, and one reported no AI use at all. Among AI systems used generally, Claude and ChatGPT were most prevalent (5 and 4 participants respectively), followed by Gemini (3) and Cursor (2). This prior familiarity with Claude largely predicted baseline tool choices during the study: six of seven participants used Claude in some form, with approaches ranging from the web interface alone, to Claude paired with a Colab notebook, to Claude Code, to combinations of Claude and Cursor. The sole exception used Cursor exclusively with auto model selection. One participant switched tools between every task (Gemini in Colab, then Antigravity with Opus, then Claude web), driven by dissatisfaction with the results of each. All tools used represented the highest-tier models available at the time of the study.

Despite access to ChatGPT, Codex, VS Code, and other tools on the provided workstation, participants almost universally defaulted to familiar tools, reflecting a strong familiarity bias even among experts.

## E.4 Pre-Study Survey

Seven participants completed a pre-study survey prior to the study covering demographics, visualization expertise, library familiarity, and workflow habits. All seven reported regularly using data visualization in their work. Tables 20, 21, and 22 summarize the responses.

Table 20: Participant demographics and self-reported visualization expertise (1–5 scale).

| P | Age | Gender | Position | Overall | InfoVis | SciVis |
|---|---|---|---|---|---|---|
| P1 | 27 | M | PhD | 4 | 4 | 3 |
| P2 | 34 | M | Post-Doc | 5 | 4 | 5 |
| P3 | 22 | M | PhD | 4 | 4 | 4 |
| P4 | 28 | M | Post-Doc | 4 | 4 | 3 |
| P5 | 32 | M | Post-Doc | 5 | 5 | 5 |
| P6 | 30 | F | Post-Doc | 4 | 5 | 4 |
| P7 | 29 | F | PhD | 5 | 5 | 4 |

Table 21: Self-reported familiarity with visualization libraries (1 = Unfamiliar, 2 = Vaguely Familiar, 3 = Familiar, 4 = Proficient, 5 = Very Proficient). Top group: InfoVis-oriented libraries. Bottom group: SciVis/WebGL-oriented libraries.

| P | D3 | Vega | Mpl | Sns | ggplot2 | Altair | MATLAB |
|---|---|---|---|---|---|---|---|
| P1 | 4 | 5 | 4 | 4 | 4 | 5 | 2 |
| P2 | 3 | 4 | 3 | 2 | 2 | 3 | 2 |
| P3 | 3 | 2 | 5 | 4 | 2 | 2 | 2 |
| P4 | 4 | 4 | 4 | 3 | 3 | 4 | 2 |
| P5 | 5 | 4 | 4 | 4 | 4 | 4 | 3 |
| P6 | 4 | 4 | 4 | 4 | 4 | 4 | 2 |
| P7 | 4 | 4 | 4 | 4 | 5 | 4 | 2 |
| **P** | **VTK** | **ParaView** | **Neuro.** | **Three.js** | **Deck.gl** | **PyVista** | |
| P1 | 1 | 1 | 1 | 3 | 1 | 1 | |
| P2 | 2 | 1 | 1 | 5 | 2 | 1 | |
| P3 | 1 | 1 | 1 | 2 | 1 | 1 | |
| P4 | 2 | 2 | 3 | 2 | 5 | 1 | |
| P5 | 5 | 5 | 5 | 5 | 4 | 4 | |
| P6 | 1 | 1 | 1 | 1 | 1 | 1 | |
| P7 | 1 | 1 | 1 | 2 | 1 | 1 | |

Table 22: Participant workflow habits. All participants primarily create visualizations by writing code; P1 and P5 also reported regularly using AI tools as part of their workflow.

| P | AI Systems Used | Other Libraries |
|---|---|---|
| P1 | ChatGPT, Claude, Gemini, Cursor | — |
| P2 | ChatGPT, Claude | Observable Plot |
| P3 | Claude | — |
| P4 | Claude, Gemini, Cursor | — |
| P5 | ChatGPT, Claude, Gemini | — |
| P6 | ChatGPT, Claude | plot.js, gosling.js |
| P7 | None | p5.js, Illustrator, ggplot2 ext. |

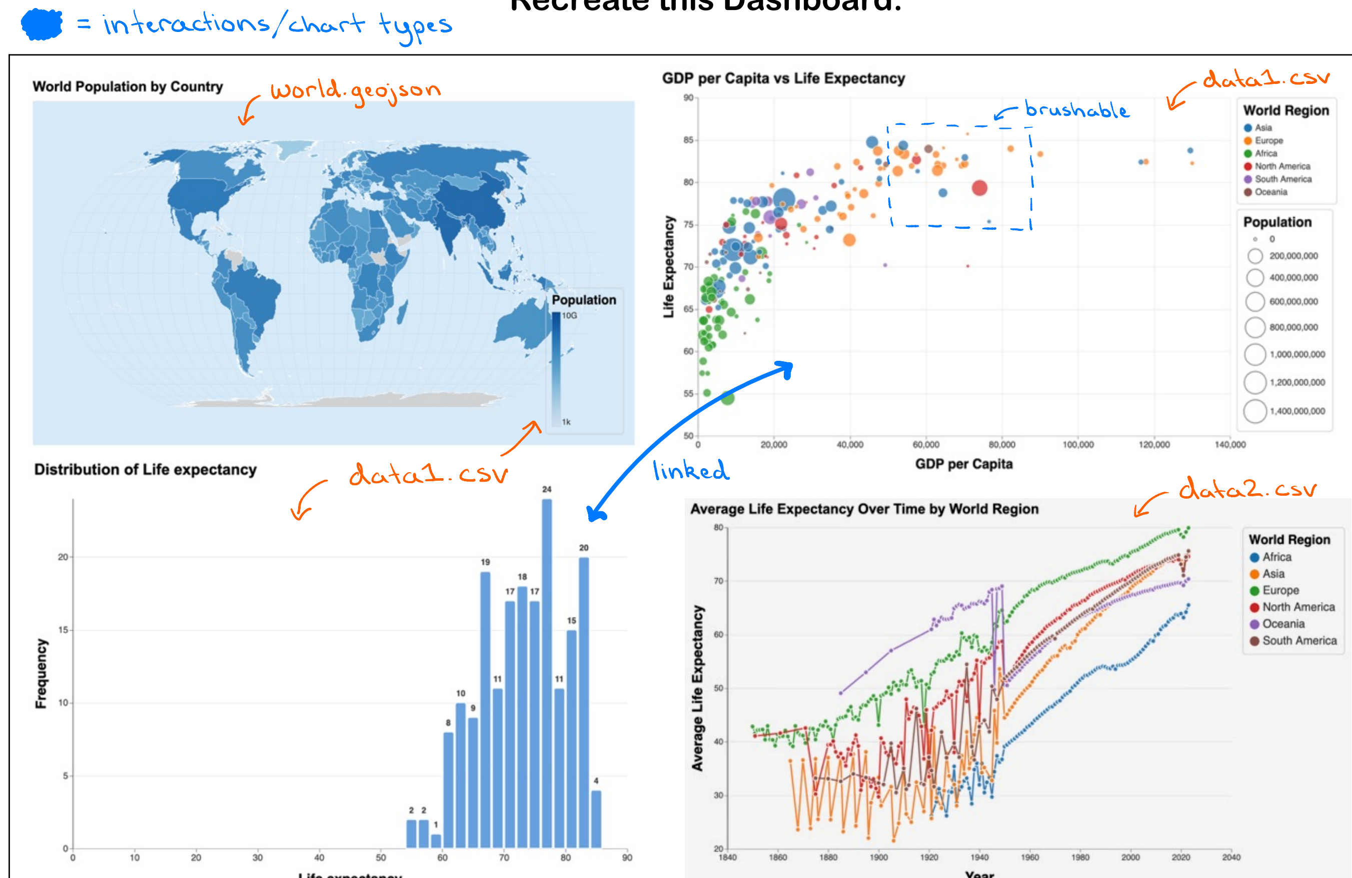



Fig. 13: Printed task sheet given to participants for Task 1 (InfoVis). The sheet specifies the target dashboard layout, required views, data files, encodings, and interactions.

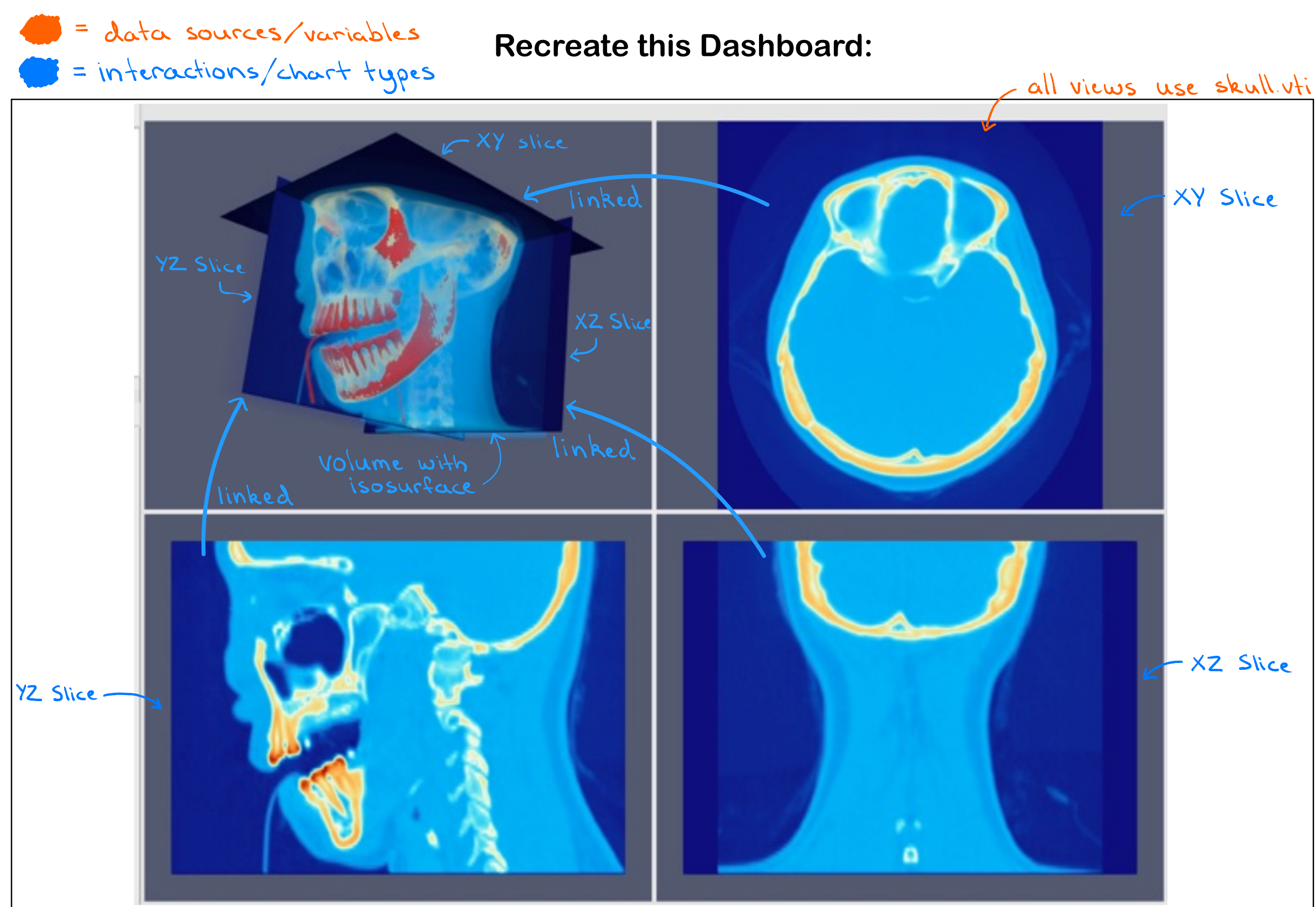



Fig. 14: Printed task sheet given to participants for Task 2 (SciVis). The sheet specifies the target dashboard layout, required views, data files, encodings, and slice-linking interactions.

= data sources/variables
= interactions/chart types

**Recreate this Dashboard:**

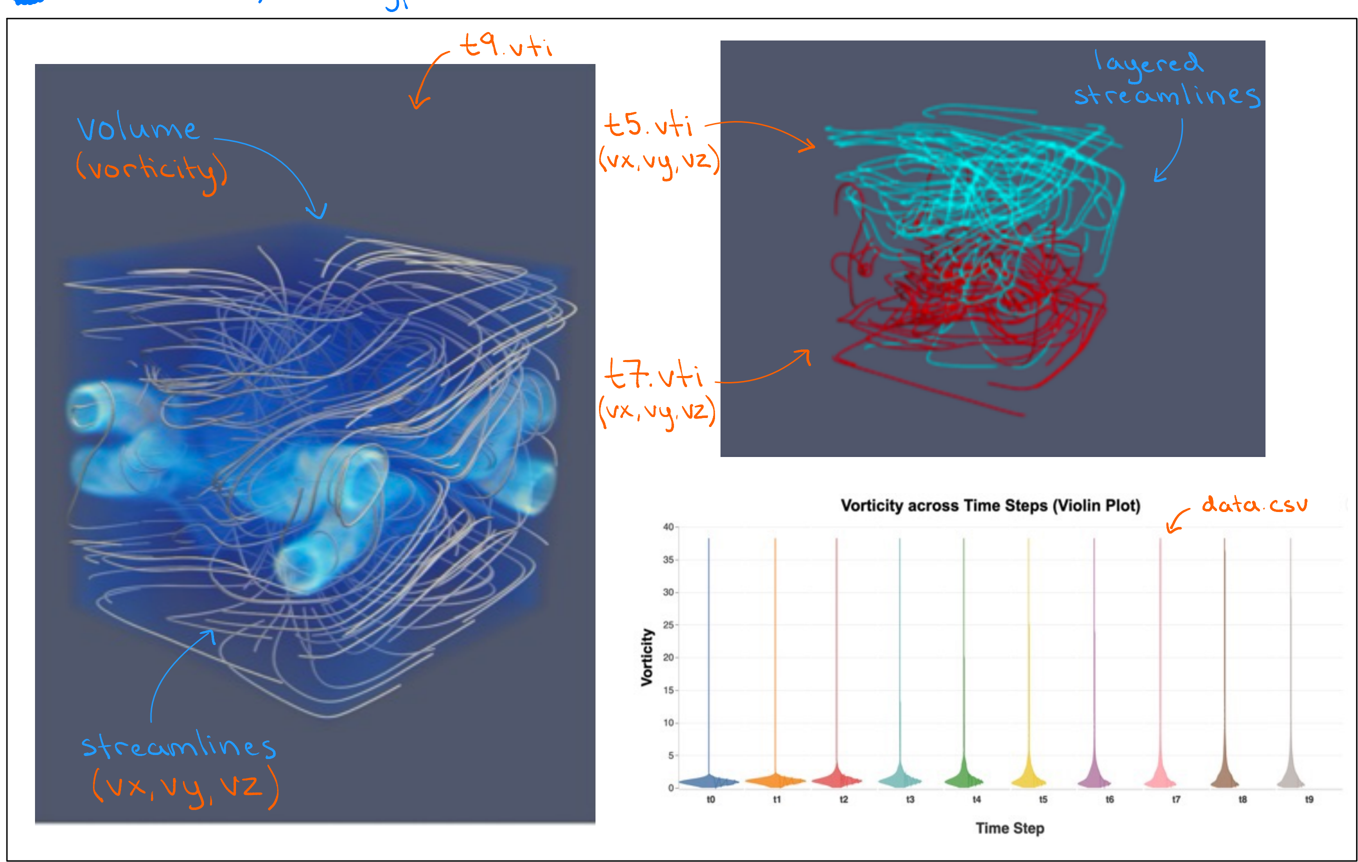



Fig. 15: Printed task sheet given to participants for Task 3 (Combined). The sheet specifies the target dashboard layout, required views, data files, and velocity component encodings.