

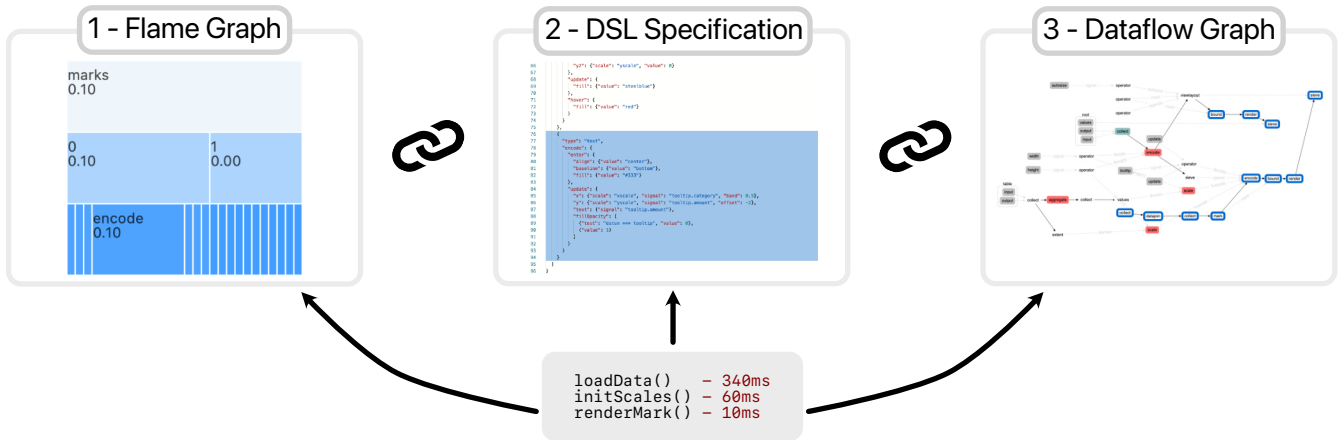
# VegaProf: Profiling Vega Visualizations

J. Yang<sup>1</sup> , A. Bäuerle<sup>2</sup> , D. Moritz<sup>3</sup> , and Ç. Demiralp<sup>2</sup>

<sup>1</sup>University of Washington, WA, USA, work done at Sigma Computing

<sup>2</sup>Sigma Computing, CA, USA

<sup>3</sup>Carnegie Mellon University, PA, USA



**Figure 1:** VegaProf records low-level execution times and encodes them in coordinated visualizations corresponding to different abstraction levels of Vega’s DSL execution. (1) A flame graph provides an overview of where the major amount of time is spent. (2) In contrast to previous debugging approaches, which relied on trial and error-based changes of the visualization specification, VegaProf maps performance measures directly to the Vega specification. (3) To analyze performance on different levels, VegaProf augments the dataflow graph with performance measures.

## Abstract

Vega is a popular domain-specific language (DSL) for visualization specification. At runtime, Vega’s DSL is first transformed into a dataflow graph and then functions to render visualization primitives. While the Vega abstraction of implementation details simplifies visualization creation, it also makes Vega visualizations challenging to debug and profile without adequate tools. Our formative interviews with three practitioners at Sigma Computing showed that existing developer tools are not suited for visualization profiling as they are disconnected from the semantics of the Vega DSL specification and its resulting dataflow graph. We introduce VegaProf, the first performance profiler for Vega visualizations. VegaProf effectively instruments the Vega library by associating the declarative specification with its compilation and execution. Using interactive visualizations, VegaProf enables visualization engineers to interactively profile visualization performance at three abstraction levels: function, dataflow graph, and visualization specification. Our evaluation through two use cases and feedback from five visualization engineers at Sigma Computing shows that VegaProf makes visualization profiling tractable and actionable.

## CCS Concepts

• *Human-centered computing* → *Visualization toolkits*; • *Software and its engineering* → *Domain specific languages*;

## 1. Introduction

Domain-specific languages (DSLs) such as Vega [Veg22a], ggplot2 [Wic16], and D3 [BOH11] simplify and accelerate visualization creation by abstracting implementation details from practitioners. Using intermediate representations (IRs), DSLs also open opportunities for automated optimizations, which otherwise require

deep domain knowledge and programming expertise. For example, Vega parses a visualization specification that is written in JSON—whose format has been informed by the grammar of graphics [Wil12]—into a dataflow graph. The resulting dataflow graph, in turn, triggers the execution of low-level visualization rendering functions. This successive creation of IRs to execute DSL code is also called the *lowering process*.

However, the abstraction that DSLs provide through the lowering process comes at a price. One limitation that Vega trades in for its simplicity is the limited access to code execution [HSH15]; there is no direct mapping between the DSL specification, the dataflow graph, and measured function performance. Since Vega specifications are parsed and then finally rendered by the underlying framework, there is currently no way of connecting the elements defined in the DSL and the functions executed to generate visualization primitives. Vega is often developed in a browser environment, which typically provide integrated profiling instruments. However, because of the missing connection between Vega’s IRs, these instruments cannot provide such a multi-level performance trace. In turn, one might know which rendering function slowed down the visualization drawing, but this knowledge cannot be mapped back to parts of the DSL or its resulting dataflow graph. As a result of this missing link between performance measures and elements of the DSL, we found that visualization engineers often have to use trial-and-error procedures to fix their performance problems. This calls for dedicated visualization profiling tools that operate under the premise of DSLs, visualization parsers, and their rendering engines.

To support visualization engineers discover and resolve performance problems of their Vega visualizations, we present VegaProf (video demo available [online](#)), the first profiler for the Vega visualization grammar. We enable such debugging capabilities by tracing execution time measured at a function execution level all the way back to the DSL, creating a bidirectional mapping between performance issues on the function level, the dataflow graph, and the user-defined or computer-generated DSL code. Based on this mapping, we provide timings for DSL segments, nodes of the resulting dataflow graph, as well as rendering functions generated from dataflow graph nodes. In addition to the mere recording of performance measures, we further present interactive profiling visualizations within a familiar developer tool, the Vega Editor. Bridging the gap of current profiling tools through a direct-manipulation visualization interface [HHN85], practitioners can trace performance bottlenecks from rendering functions all the way back to the DSL. Our evaluation with five visualization engineers shows that this form of performance tracing and visualization introduces means to reason about and resolve performance bottlenecks in a way that was not possible before. VegaProf is available as open-source software on [GitHub](#).

In summary, we contribute VegaProf, an interactive profiler enabling visualization practitioners to quickly explore the time performance of Vega visualizations at different IRs of the Vega DSL. We demonstrate VegaProf’s usage workflow with two use cases of visualization-based profiling and evaluate VegaProf by eliciting feedback from five visualization engineers through a user study. Our findings from the user study show the necessity for and usability of VegaProf. In addition, these findings indicate the effectiveness of encoding performance measurements in a flame graph visualization linked to the underlying visualization specification. While VegaProf is the first profiler dedicated to Vega, the presented concepts could be applied to any visualization DSL. We release VegaProf as open-source software to support future research and applications.

## 2. Related Work

Our work relates to prior research on both visualization debugging and dataflow system profiling. VegaProf adds a new element to both lines of earlier work as it is the first time-performance profiler for DSL-specified visualizations. Through interactive visualizations, VegaProf enables profiling across all underlying abstraction levels of the DSL.

### 2.1. Debugging Visualizations

While data visualization has a long history of tools [KWHH17, WQM\*17, MWN\*18] and grammars [Veg22a, SMWH16, BOH11] for visualization specification, research into linting and debugging visualizations is nascent. McNutt and Kindlmann [MK18] introduce one of the first visualization linters. Their linter checks a pre-defined set of rules on a given visualization and returns the list of failed rules with explanations. This postprocessing approach is disconnected from the development workflow and does not localize errors for rendered visualizations directly in their specifications. In contrast, VisuLint [HCS20] annotates visualizations in situ with red marks. These marks are akin to conventional linting-error visualizations in IDEs, but cannot be traced back to the visualization specification. To rectify defective visualization designs, VizLinter [CSX\*21] highlights flaws directly in the visualization specification. VizLinter maps flaws to the DSL code while suggesting potential fixes. Since interactions can be particularly hard to debug, Hoffswell et al. [HSH16] propose debugging techniques specifically designed for reactive visualizations. Prior to their work, users could only use the JavaScript console to traverse the system internals, which required existing knowledge and was hard to track changes. To provide the needed detail to the visualization engineer, they track state through interactions and map that to a visual debugging interface.

As these works focus on discovering errors in the visualization specification rather than providing performance insights, they target a different problem space than VegaProf. With VegaProf, we provide the first performance profiler for visualizations specified with a DSL.

### 2.2. Profiling Dataflow Systems

A bidirectional coupling of the dataflow graph with the underlying DSL code and visualization rendering functions is central to interactive profiling in VegaProf. Dataflow graphs are a common abstraction used by myriad tools and DSLs across domains beyond data visualization (e.g., PyTorch, TensorFlow, Spark, Flink, Naiad, SQL). Earlier work presents profiling tools to help discover performance issues in dataflow systems [GLB20]. For example, Perfopticon [MHHH15] shows the runtime distribution of individual query operators and per-worker execution traces. Similar to our approach, Perfopticon also maps the profiling result to user input. Battle et al. propose StreamTrace [BFD\*16], disentangling SQL queries as a series of intermediate queries to help developers debug the behavior of their queries. In a similar vein, Grust et al. [GKRS11] link intermediate query results directly to the SQL code that generated them instead of using representative visualizations.

Other approaches from the deep learning domain [GGGP21, WCLR22, CLC\*22] focus on program code performance optimization. Skyline [YGP20] embeds neural network training performance predictions in the code editor to help machine learning practitioners tune hyperparameters. Umlaut [SHH21], another profiling instrument for the deep learning domain, provides heuristics and error messages by analyzing the program structure and model behavior. Mapping performance directly to code has been a common paradigm in recent research [CLRG19]. However, none of the approaches discussed in this paragraph target visualization engineers or consider dataflow graphs as a profiling entity.

Beischl et al. [BKB\*21] propose a multi-level performance profiling technique specifically for dataflow-based systems. We build on this approach and adopt it for data visualization, where different abstraction levels and, more importantly, user-facing specification code need to be considered. Additionally, we provide an interactive, visualization-based interface for mitigating performance problems, whereas Beischl et al. focus on the technical aspects of performance profiling.

### 3. Formative Interviews

To assess the needs of visualization engineers, we interviewed 3 professional visualization engineers at Sigma Computing. For all participants, the programmatic generation of Vega visualizations is part of their daily work. We conducted and recorded our interviews in a semi-structured manner via online video conferencing software. A list of predefined topics and questions was covered, while at the same time leaving room for open discussion. We specifically asked our interviewees about their performance optimization needs, their current practice and tools, and potential improvements.

#### 3.1. Performance Optimization

We discovered that performance issues “*they usually have large impacts*” (P1). Our interviewees attributed this to the fact that performance issues are currently hard to localize, debug, and fix. This manifests in the fact that “*performance issues are often neglected to fix*” (P2) and “*once they find the causes, they always tell the customers not to perform such operations*” (P1). Furthermore, visualization engineers typically “*limit the data input to Vega spec to be less than 25k [data points] to prevent a lot of slow rendering issues*” (P3). Moving forward, these cannot be the go-to solutions, especially since interviewees “*have seen a lot of questions regarding visualization performance*” (P1).

#### 3.2. Current Tooling

To reason about poor performance, visualization engineers typically “*have zoom meetings with customers to talk about problematic visualizations*” (P1). Then, they often “*simulate the configurations*” (P1 and P3) and test them in a sandbox environment. As such, they do not have specialized tooling for performance debugging, but instead rely on “*the Vega Editor in combination with Chrome’s devtools*” (All participants). However, the problem with this is that “*devtools can only tell you that the issues are caused by Vega function calls, but it can’t help with locating them in the spec*” (P1).

### 3.3. Potential Improvements

When asked about what tooling could improve their situation, interviewees asked for a “*breakdown of the transforms, mark rendering, etc. that can immediately indicate which lines [in the DSL] caused the issue*” (P3). Thus, what professional visualization engineers are asking for is a mapping from performance issues back to the visualization specification and the IRs of the underlying DSL representation. Therefore, adequate developer tooling for performance profiling must surface this mapping effectively.

### 4. Design Goals

Based on our insights from the aforementioned formative interviews, we distilled the following requirements for a successful Vega performance profiler:

**Bidirectional timing mapping.** To help visualization engineers discover the root cause of performance bottlenecks, it is not sufficient to just measure function execution times. Instead, to be able to make informed decisions about performance optimizations, practitioners need a bidirectional mapping that connects these timings to the DSL, its associated dataflow graph, and individual rendering function calls.

**Multi-level profiling insights.** Once a bidirectional mapping of timings between function execution and the DSL exists, practitioners need to be able to investigate the results of this mapping. Visualization instruments that surface timing measurements can support this investigation. Using such visualizations, practitioners can trace performance measurements through the IRs used in the lowering pipeline of a DSL.

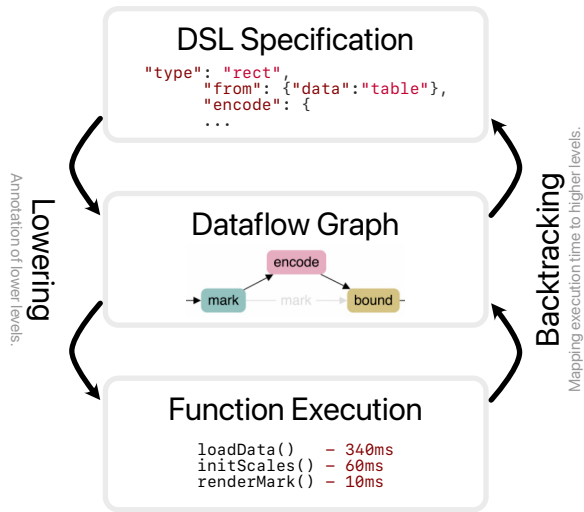
**Familiar development environment.** Finally, we want to support visualization engineers in a familiar environment. Visualization engineers adopt a tool only if the burden of entry does not outweigh its benefits. Therefore, we want to provide these visual insights for visualization performance debugging in a familiar environment, in our case, the Vega Editor [Veg22b].

### 5. Bidirectional Profiling Map

Like other profiling instruments, we measure function execution times. To do this, we hook into how Vega parses specifications and instantiates dataflow operators and record the runtime when operators’ evaluation functions are executed.

While recording function execution times is well-established for time-profiling, effective profiling instruments for a visualization DSL rely on a bidirectional mapping of execution time measurements and DSL segments with semantic meanings. Only with such a mapping can visualization engineers put performance bottlenecks in the context of the IRs that the DSL gets transformed into (cf. Figure 2).

To realize such a mapping, we annotate the nodes of the *dataflow graph description* as they are created when parsing the visualization specification. This way, we are also able to reverse this mapping, associating dataflow graph nodes with the corresponding lines of DSL code. Once the dataflow graph description is transformed into a *dataflow runtime* where the nodes represent functions to be



**Figure 2:** During the lowering process, the DSL specification is parsed into a computation graph and then functions to be evaluated eventually. We hook into this lowering process and add annotations that indicate which element on a higher-level IR corresponds to what part of the lowered representation, e.g., what part of the spec corresponds to which data flow graph nodes. Based on these annotations, we can track measured function evaluation times back to higher levels, i.e., assigning time-measurements to the nodes of the dataflow graph and visualization specification.

evaluated, we further annotate these functions with the respective dataflow graph nodes to realize such a mapping for this second lowering process. This way, our measurements of function execution time can be mapped back to the node of the dataflow graph that triggered the execution. By chaining the aforementioned annotations, we are able to assign the execution of individual functions not only to nodes in the dataflow graph but also to lines and segments of DSL code. Using this bidirectional mapping, execution times can be traced from the function level all the way to the highest level of operation, namely the DSL specification for the visualization.

Altogether, this directly addresses our first design goal of creating a bidirectional mapping between Vega’s IRs. While we only use this inter-IR indexing approach to provide better profiling instruments, it could be helpful for other introspection tools such as educating about Vega’s lowering process or dataflow debugging as well. In theory, this bidirectional profiling map can be visualized and analyzed in any environment. We integrated this bidirectional mapping into a well-established visualization development tool, the Vega Editor. In the following section, we explain how this technology is used for visually supporting visualization engineer’s performance analysis needs.

## 6. Visual Performance Inspection

On the basis of the information obtained through the aforementioned bidirectional mapping of profiling results, we provide a visual interface that enables visualization engineers to take action and

improve the performance of their visualization designs. We implemented this interactive performance profiling interface as an extension to the Vega Editor to place visualization engineers in a familiar environment. A new performance tab provides a performance flame graph (cf. Figure 3 (B)) and augments the dataflow graph (cf. Figure 3 (C)) as well as the DSL specification editor (cf. Figure 3 (A)). All three of those components are connected through brushing and linking techniques using our bidirectional profiling map (cf. Section 5) as the underlying data source. This way, we map selections, *mouseover* events, and zoom transitions that happen in one of the three views to the other two. Throughout our visualizations, such interactions are indicated through blue highlights. Hereby, hovered items are assigned a semi-transparent blue highlight, whereas selected items are highlighted in full blue consistently across all visualizations.

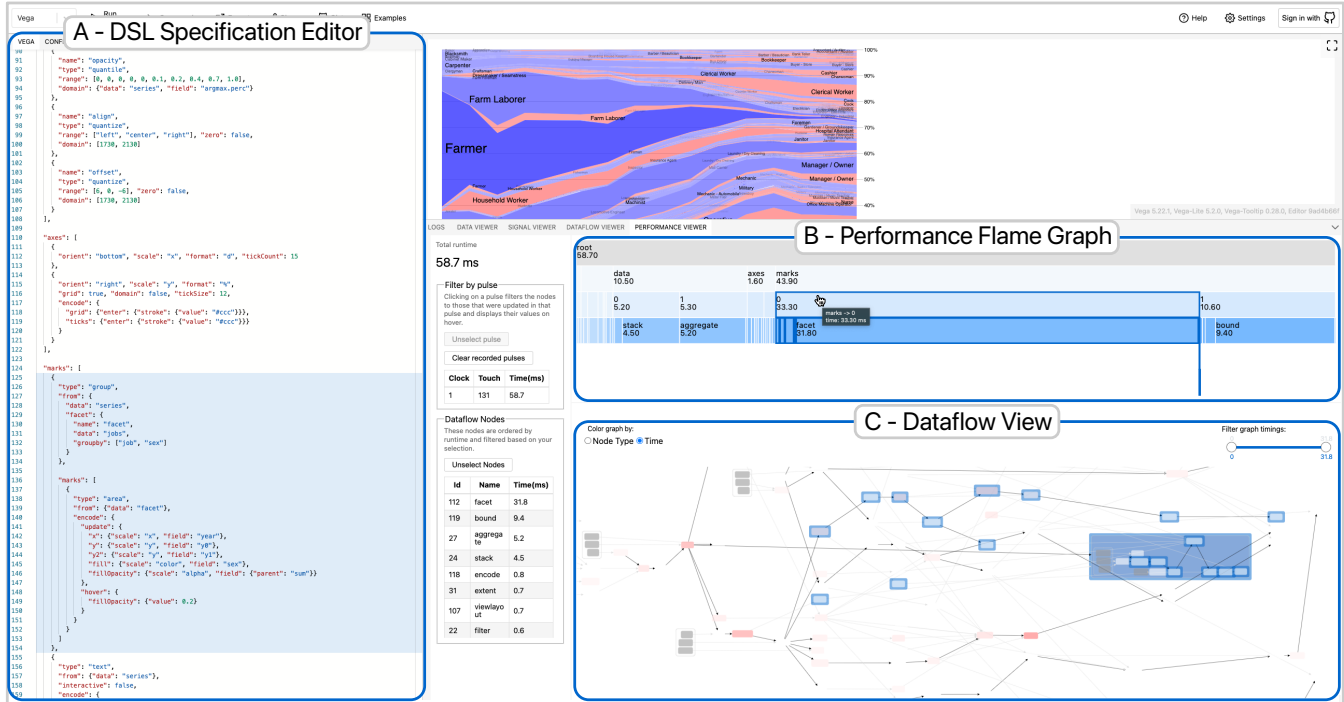
Aside from these main views, the Vega Editor further displays the visualization that results from the provided specification. If the visualization is interactive, the resulting profiling and operator states from interaction events are recorded as multiple *pulses*. The first pulse marks the initial rendering of the visualization. Subsequent pulses are added whenever the visualization needs to be updated based on user interaction. Pulses can be selected from the pulse table, updating the flame graph and dataflow graph to only show the operators being re-evaluated and their timings. In addition, pulses augment the dataflow graph via the node tooltips by providing insights of how data changes in the individual nodes along every pulses. By default, we show profiling results for the initial rendering pulse for both static and interactive charts. This way, visualization engineers can use our visualizations and profiling results not only to debug the initial rendering process but also to improve interaction performance. Directly above the pulse selection, we prominently show the total runtime of the selected pulse. This way, users have an anchor to put all the timings in the visual interface into context.

### 6.1. DSL Specification Editor

The DSL specification editor is prominently positioned at the left edge of the Vega Editor (cf. Figure 3 (A)), marking a natural entry point for visualization engineers. It represents the highest level of abstraction for VegaProf, directly connecting performance profiles to the DSL code that defines the visualization. Since this level can be directly influenced by visualization engineers, it is often where their time-performance analysis begins.

To map function execution times to blocks of the DSL specification, we consider different levels of ranges in the specification. These blocks directly map to JSON’s hierarchical object structure in the Vega specification. For example, a user would specify both the *x-axis* and *y-axis* blocks under the *axis* block. In Vega, these blocks are the units that visualization engineers would associate with visual components. Hence, this is the level at which they would make edits, e.g., changing the type of mark that is used for rendering or how data is mapped to these marks. As such, this way of clustering parts of the DSL naturally aligns with how Vega users understand and modify the specifications.

Hovering over one of these blocks of DSL code highlights the respective specification segment. When clicking on such a high-



**Figure 3:** Our visual inspection tools were implemented in a familiar development environment – the Vega Editor. (A) We highlight selected regions of DSL code when inspecting performance bottlenecks. (B) A flame graph depicts the measurement of rendering function execution time. (C) In Vega’s dataflow graph, we highlight nodes that contribute to selected timing measurements. Note how hovering over the flame graph highlights the corresponding elements in the dataflow graph and DSL editor.

lighted block of code, it is selected for further inspection. As mentioned at the beginning of this Section, highlights and selections are transferred to the according elements in the dataflow graph and the performance flame graph. Maybe even more importantly, we also implemented the reverse linking directions from the flame graph and the dataflow graph. As a result, one can easily interpret, the high-level responsibility of a certain element of the flame chart and the dataflow graph by inspecting the highlighted block in the specification. Meanwhile, visualization engineers get insight into how much time individual blocks of the visualization specification require during rendering. For example, investigating elements that require a large portion of the rendering time in the flame graph scrolls to the corresponding code segment of the DSL and highlights it. This makes such performance measures insightful and actionable, as visualization engineers can make direct adjustments to relevant parts of the DSL code.

## 6.2. Dataflow View

As a first IR of the Vega visualization grammar, the DSL specification provided by the visualization engineer is transformed into a dataflow graph. With the dataflow view, our visual performance inspection interface also allows for analysis at this more detailed level. The dataflow view contains a visualization of the parsed dataflow graph that Vega’s DSL gets transformed into (cf. Figure 3 (C)). It can be color-coded by node type or, more conveniently

for performance analysis, by node runtime. We use D3’s *interpolateReds* color scale to encode node runtime since red is often used as an alarm color, again drawing attention to the nodes that are most performance-intensive during rendering.

Whenever a node is selected from this graph visualization, the dataflow graph gets transformed to show only the subgraph with connections to the selected node based on dependency. A zoom-in animation further puts the focus on selected nodes. If a selection comes from any other visualization, such as the DSL editor or the performance flame graph, we analyze the nodes that are involved in the selected subset of performance analysis elements and employ a filtering and zooming similar to the one for direct node selection. This interaction concept further embraces the combined analysis of Vega’s different IRs, similar to how interaction with the DSL specification editor is mapped to all other visualizations.

To directly surface the most time-consuming nodes, VegaProf further includes a table of all nodes, positioned next to the dataflow graph. This table is based on the dataflow graph and ordered by node execution time, placing the most performance-intensive nodes at the top. With this ordering, this tabular visualization can be used as an entry-point of the analysis on the dataflow level as it guides the visualization engineer’s attention directly to nodes of interest.

### 6.3. Performance Flame Graph

Positioned directly above the dataflow graph, the performance flame graph (cf. Figure 3 (B)) functions as an intermediate representation between the dataflow graph and the DSL specification editor. The flame graph is defined by its different levels of aggregation, going from coarse performance elements to more fine-grained ones. To symbolize this aggregation structure, coarser levels are colored in grey and light blue, whereas a dark blue coloring is employed for the most detailed performance analysis levels. The most detailed level in this flame graph directly represents nodes of the dataflow graph. However, the flame graph also visualizes the hierarchical structure of the Vega DSL at its higher levels, connecting the two other views in one visualization.

Hovering over and selecting elements in the flame graph works just as it does in our other visualizations. The flame graph additionally zooms into selected elements to provide more detailed information about a selection. Similar to the dataflow graph, this zooming and highlighting might also be triggered by events from other visualizations. Altogether, the flame graph, with its different levels of performance aggregation and linked interaction concepts, serves as a bridge between both the specification editor and the dataflow view.

## 7. Use Cases

This section describes two example use cases for VegaProf. It demonstrates how VegaProf can help visualization engineers in discovering and resolving performance problems of their Vega visualizations. These use cases highlight how connecting different IRs help debug performance and, specifically, how a direct linking of performance bottlenecks to Vega's specification make such analyses actionable.

### 7.1. Visualization Design Decisions

Mary is a visualization engineer in the data analysis team of a large airline. She wants to analyze the effect of flight distance on the delay of flights based on a dataset that contains information on three million flights. She considers a scatter plot to visualize the data. When she specifies the scatter plot in Vega, she notices that the visualization she has created is too slow to be usable.

Having heard of VegaProf, Mary loads her data and visualization specification in the Vega Editor and analyzes the performance of her visualization. Through an investigation of the Flame Graph, she immediately notices that mark rendering takes most of the overall visualization generation time. Looking at the connected location in the visualization specification, she notices that rendering individual scatter marks for millions of flights is just too slow to sustain interactivity.

Since Mary is looking after general trends rather than individual flights anyway, she decides to get rid of these marks and instead render a heatmap for binned results.

Next, Mary notices that loading the data was rather slow. Using the dataflow graph, she locates an operation that copies part of the data during the transformation stage. As the relevant part of the

Vega specification is highlighted when she hovers the corresponding dataflow node, she identifies the problem and is able to modify the transformation code so that data processing becomes much more performant.

After these modifications, Mary notices that while much faster than before, data processing is still her main performance bottleneck. The final step she could take is to pre-aggregate data instead of binning it at present time. However, since the data frequently changes, she decides against it and accepts the initial loading time because of the data transformation.

### 7.2. Offloading Computational-Intensive Operations

Alice works as a visualization engineer for a software company. Her team is responsible for implementing product features around UI-based visualization authoring. Within their product, users can create various types of visualizations to explore data in a cloud data warehouse (CDW) without programming expertise. Alice's team uses Vega as the underlying technology to specify visualizations through the product's UI.

By default, Vega requires all data to be loaded and processed in the client's browser. However, it is computationally impossible to query the CDW for the raw data and transfer it to the browser's memory for processing in Vega. Therefore, Alice decides to pre-process the data with SQL queries so that the query result is ready to be directly mapped to visual channels without further Vega transforms. However, her testing visualization is not fast in its initial rendering and does not seamlessly respond to user interactions.

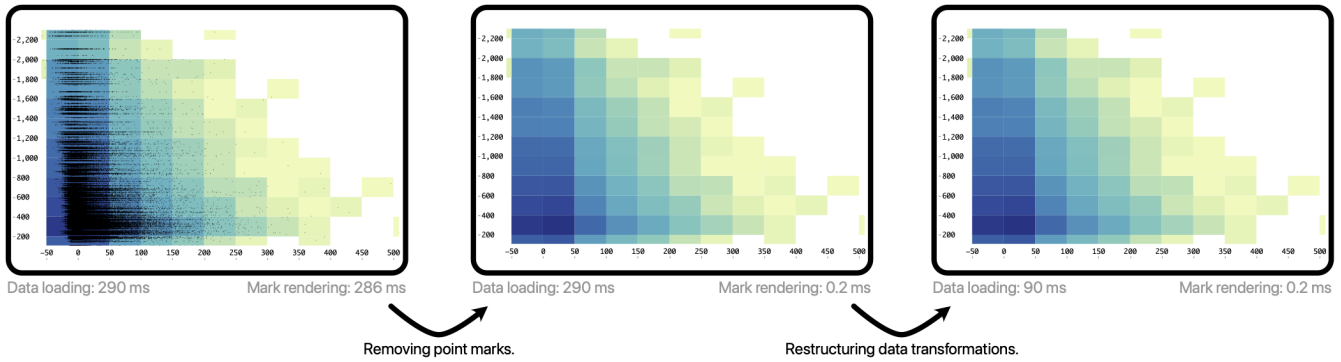
When Alice inspects VegaProf's pulses and dataflow graph visualizations, she finds out that both the initial rendering and every interaction trigger a request to the CDW, blocking the entire dataflow graph. Since all the data processing is done as a pre-process step in SQL, each interactive selection re-executes the whole data pipeline. In turn, Vega is blocked waiting for data that could have been cached. Knowing that the transforms parameterized by interactions are fast enough to be executed in the browser, she moves these interaction-based data transformations into the Vega specification, keeping only the underlying computational-intensive operations on the backend. These time-consuming operations are only executed once and cached in the dataflow graph. As a result, the initial rendering time for the chart is acceptable, while interactions are significantly smoother.

## 8. Expert Interviews

To further evaluate the usability of VegaProf, we conducted a qualitative user study with five visualization engineers at Sigma Computing. In the following, we describe the study setup and then report our observations gathered during these interviews. Finally, we summarize feedback on the usability of VegaProf elicited from participants through a post-study questionnaire.

### 8.1. Interview Setup

We now provide details on the setup of our study, including our participant pool, and the procedure and data used.



**Figure 4:** During our evaluation, we used a scatter plot with binned aggregation as an example. At the beginning, the rendering time was about 600 ms. The individual point marks did not add substantial value to the visualization; in fact, they even obstructed the aggregated heatmap visualization. Thus, removing them did not undermine the message the visualization aims to communicate while reducing the mark rendering time to almost zero. Furthermore, the data transformation was specified in a suboptimal way that required Vega to copy data. Restructuring the data transformation further saved about 200 ms without changing the visualization.

**Participants.** Our interviewees worked with Vega-generated visualizations daily, although they had different levels of background knowledge of Vega’s internal dataflow. As such, they well represented VegaProf’s target audience. While we had to collect our data opportunistically because of the limited availability of our participants, field studies like ours excel at capturing how visualization engineers actually work.

**Procedure.** To understand the affordances and limitations of VegaProf, we had 30-minute long think-aloud sessions with each interviewee individually. During the interviews, we first gave a quick tutorial of VegaProf, before participants could experiment with the profiler themselves. For this experimentation, our participants got access to VegaProf with a visualization specification preloaded. Our participants were asked to explore the profiler based on two guiding questions, namely *how can the mark rendering be improved without harming the message of the visualization?* and *how can the data processing be improved?*. We also encouraged them to share a specification from their recent work and show us how they would use VegaProf to inspect it. Finally, after the main think-aloud session, we sent interviewees an online questionnaire to evaluate the usefulness of a visualization profiler to their job in general, and their ratings of each VegaProf’s specific feature.

**Data and specification.** The specification we used for this evaluation renders a scatter plot with binned aggregation (cf. Figure 4) as described in Section 7.1. It includes three million data points and supports panning and zooming to re-calculate the aggregation with new buckets. Naturally, rendering or aggregating a large number of data points is prone to performance issues.

## 8.2. Study Observations and Discussion

In the following, we outline the main findings of our interviews.

**Initial performance improvements.** With the help of the flame graph, all participants correctly identified the point marks as the most time-consuming components. They located the relevant part in the specification by hovering on the flame graph and stated that

they found the feature useful *“it is impressive that you can highlight the spec [from the flame chart]”* (P3). Based on the highlighted region in the specification, all participants recognized that a scatter plot might not be the most efficient visually and computationally and removed it. Highlighting relevant parts of the visualization specification was one of the most well-received features of VegaProf as the current way of debugging slow specifications is to *“just guess which part is the cause and modify it so see if it solves the issue or not”* (P1). VegaProf greatly simplifies this laborious process as it connects profiling measurements back to the visualization specification. One participant underlined the importance of the flame graph to their workflow, since without VegaProf *“we could separate the data transform out and profile it programmatically, but there was no way to do that for everything else, [including] the marks, the rendering, etc.”* (P5).

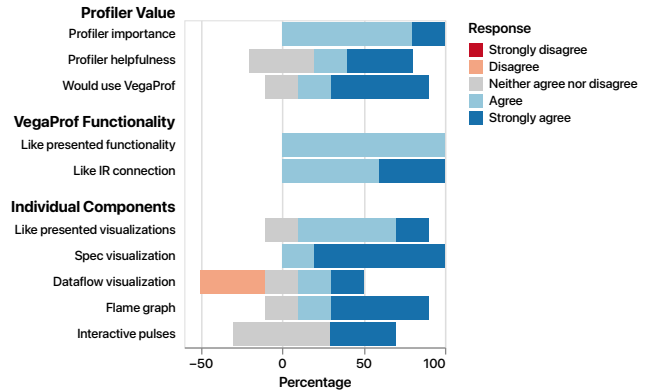
**Data transformation performance.** As participants inspected the resulting performance after this first edit, they discovered that the runtime for rendering the marks was greatly reduced. Subsequently, they recognized that, with the modified specification, the most time-consuming operations came from the transformations that were used for processing the data. Participants were able to select the relevant dataflow nodes responsible for the performance bottleneck, however, most of them lacked the background knowledge about how nodes are instantiated from the specification through parsing and compilation. Specifically, they were not able to infer from the node name *relay* that the performance bottleneck was caused by a unnecessary data copy operation. P5 managed to solve the task by removing unnecessary operations, although it required a hint from our side: *“Now that you told me a transform in the spec can be expanded to multiple operations, I can see it in the flame chart and everything makes sense to me”* (P5). Some participants couldn’t come up with a solution addressing this performance bottleneck. After we explained how to reconstruct the data transformation pipeline, our participants acknowledged that knowing the Vega internals and inspecting the dataflow graph would help optimize specification authoring: *“I’m surprised that doing this can save so*

*much [execution] time!*” (P1). We believe that the above findings also suggest that the data pipeline development in Vega can be further optimized, for example, to avoid unnecessary data copying and streaming regardless of how users structure the specifications.

**Visualization usage.** Overall, we observed that participants spent most of their time exploring the flame chart *“because it exactly tells you what part of rendering is taking up all the time”* (P2). When our participants decided to temporarily focus on part of the flame chart after initial exploration, they typically inspected the highlighted segments of the specification. Participants spent less time inspecting the dataflow graph. This might be partly due to the fact that most participants had a lack of understanding of the node names and *“so far have just been using Vega as a black box [...] assuming that it would work well”* (P2).

**Dataflow graph usability.** Initially, even participants who had frequently debugged Vega visualizations with the dataflow graph before found the flame chart more helpful than the dataflow graph. They focused more on the dataflow graph only after being reminded of the connection between the flame chart and dataflow graph. While this underlines the importance of our flame graph visualization, it also raises questions about the usefulness of the dataflow graph for debugging purposes. One of our participants noted that *“we used a lot of the Chrome Devtools and their entire interface is basically only the flame chart”* (P3), attributing their focused view partly to previous habits. However, they also mentioned that *“the connection between the spec and dataflow graph, and the structural features [in the dataflow graph] could be really helpful to understand what goes on behind the scenes for Vega”* (P4). Further research targeted directly at visualizing dataflow systems in a more understandable way, including explanations of individual nodes, might help visualization engineers make better use of the dataflow graph as an IR for debugging.

**Participant-provided specifications.** After the guided exploration, three participants asked us to directly explore specifications they recently worked on in VegaProf. We observed how they used VegaProf to validate or reject their assumptions about a given specification. P2 showed us a scatter plot with categorical data. At first, they were surprised that *“the axes took the longest to render and then the marks were comparatively shorter [...] that’s not what I would have guessed initially”* (P2). Then, they realized that the dataset they used was relatively small while the categorical variables being mapped to the axes had a high cardinality. P3 shared a Sankey diagram that they have been working on for Sigma Computing’s product with us. During the development, they frequently inspected the dataflow graph to understand and debug the connection between nodes. Concluding that *“I’m not surprised that the “linkpath” and “datajoin” operations took the most time”* (P4), they verified that the performance conformed with their mental model for such diagram. Finally, P5 wanted to explore a visualization automatically generated from a Sigma Workbook [GSU\*22]. They exported the specification of a simple chart to test their mental model of how it was implemented. As expected, it was implemented well, such that *“it’s so fast that the axis take half of the rendering time”* (P5).



**Figure 5:** After the main think-aloud session, our study participants rated VegaProf on a five-point Likert scale. We separated these questions into three main topics. First, we asked about the general value of a visualization profiler and VegaProf specifically. Second, participants gave feedback about the perceived usefulness of VegaProf’s functionality. Third, we evaluated individual components of VegaProf. Overall, our participants saw great value in VegaProf and its visualizations.

### 8.3. Usefulness Questionnaire

Upon completion of the main interview study, we sent our participants a link to an online form with ten questions to be answered. Participants were able to provide feedback on their experience and takeaways from our interview session on a 5-point Likert scale. The results of this evaluation can be seen in Figure 5.

Questions one (*I think a visualization profiler is an important tool*), two (*A visualization profiler would be helpful to my work*), and three (*If I ever have to profile visualizations, I would use the presented profiler*) were targeted at the general value of a visualization profiler and VegaProf, specifically. Overall, participants found visualization profilers useful and noted that they might be helpful to their work. This confirms the findings from our formative interviews. Furthermore, most of them would like to use VegaProf for their visualization profiling needs, affirming the usefulness of our profiler implementation.

Questions four (*Overall, I like the functionality of the presented profiler*) and five (*The fact that different levels of profiling are linked is very helpful*) were to evaluate the functionality of VegaProf. Regarding the way VegaProf functions, our participants all liked its functionality, indicating that it indeed provides tooling that was not available before. Our participants also liked the way we coordinate Vega’s intermediate representations via visual interaction, supporting our architectural design choices.

Finally, questions six to ten were designed to assess the usability of VegaProf’s individual components (*Overall, I like the visualizations used to present profiling results, Visualizing which part of the specification can be attributed to profiling results is very helpful, Visualizing profiling results in the dataflow graph is very helpful, The flame graph is very helpful, and I like the fact that I can select pulses and this way debug interactive visualizations*). Overall, our



participants also liked the individual visualizations that VegaProf provides. They were especially fond of the visualization of performance results mapped to the DSL specification and the flame graph, which can provide an overview of performance results. Participants were torn on the usefulness of interactive pulses. One reason for that might be that we did not focus on them for the interactive evaluation session, however. The most controversial of the visualizations was the dataflow graph. While some found it valuable, others did not see it as beneficial for their workflows. We discuss potential reasons for this discrepancy in the previous subsection.

Overall, our participants rated VegaProf very positively, underlining the importance of this line of work, its architectural design choices, and most of the visualization decisions we made.

## 9. Discussion

VegaProf is the first profiler for the Vega visualization grammar. Through its multi-level performance tracing approach, VegaProf provides previously unavailable background for performance problems, making performance debugging tractable and actionable. In the following, we will discuss our findings made during the development and evaluation of VegaProf. Lastly, while our studies and experiments show that VegaProf can be helpful for profiling Vega visualizations, we also identified several promising directions for future research.

**Direct visualization connection.** Surprisingly, our study participants did not interact much with the rendered visualization. However, there might still be a case to be made for dataflow graph nodes to be connected to the scene graph that renders the visualizations. With such a connection, components of the resulting visualization could be linked to the visualizations included in our profiler.

**Profiling dataflow systems.** While VegaProf focuses on the Vega DSL, the underlying approach and visual interaction design can readily apply to other DSLs. Many DSLs go through a similar lowering process and use a dataflow graph as an intermediate representation. Further generalization of our approach could enable performance profiling for a wide array of these systems, broadening the availability of accessible profiling even further.

**Performance at scale.** Our evaluation shows how our approach enables performance improvements for individual visualizations. However, DSLs are also frequently used for visualization generation at scale. As a result, thousands of users can use automatically generated visualizations, e.g., in web applications under different browser and cloud configurations. Future research could extend our work to help visualization engineers profile the distributed performance of visualizations.

**Improvement recommendations.** While mapping performance issues to the DSL proved to be an important step towards providing visualization engineers with more powerful profiling tools, the next step would be to suggest ways of improving performance. Here, knowledge from other visualizations or performance improvement sessions could be used to provide suggestions proactively. Using approaches such as the described *performance at scale* in combination with machine learning methods could enable such recommendations.

**In-browser profiling.** Based on the insights of our formative interviews, we provide our profiler as an extension to the Vega Editor. While this is a common place for Vega visualization experiments, debugging is even more commonly done in the browser. We developed our visualizations in the Vega Editor because of its extensibility and flexibility in accessing Vega internals and providing rich visualizations. If the same tooling could be provided directly in browser profiling tools, fixing performance issues might become even more accessible.

**Scope of evaluation.** While our studies were conducted with professional visualization engineers, they all work at the same institution. In this context, they work with a spectrum of visualizations targeted at the business intelligence domain, creating visualizations for large data sets and user bases. This is a common use case for DSL-based visualization grammars; however, we recognize that our findings might not transfer to all usage scenarios, such as one-off visualizations, small data sets, and specific user bases. Future work might evaluate the usefulness of our approach in such settings.

**Offloading decisions.** A common pattern we saw during our evaluation was that visualization engineers are willing to offload certain aspects of the data transformation to a dedicated backend. However, they often do not know which parts are worth the effort. With VegaProf, such offloading decisions become much easier, as data transformation performance can be inspected in detail and tailored for specific designs. In turn, VegaProf could be combined with tools like VegaPlus [YJY\*22], VegaFusion [KMM22], and other backends or data processors to move time-consuming parts of the visualization process to dedicated services.

## 10. Conclusion

We introduced VegaProf, the first profiler enabling in-depth analysis of visualization performance bottlenecks. The design of VegaProf is informed by formative interviews that surfaced the difficulty of Vega visualization performance debugging. VegaProf brings visual profiling affordances to Vega’s different IRs by hooking into the lowering process. This way, the presented visualizations surface profiling results directly on the dataflow graph and visualization specification. We demonstrated the usefulness of VegaProf through two use cases and reported feedback elicited from five visualization engineers, utilizing one of the use cases as a probe. In this evaluation, visualization engineers were able to locate and address performance bottlenecks through our linked visualization of Vega’s IRs. VegaProf replaces the state of the art of either debugging Vega’s performance through trial and error procedures or unnecessarily limiting data set sizes.

While our work marks the first endeavor into the domain of visualization profiling, we hope that future research will broaden the applicability of our approach. In particular, our instrumentation for bidirectional mapping and corresponding visual interaction design coupling IRs can benefit developer tools for dataflow systems at large. Finally, we advocate for designing future visualization DSLs with their developer tools, such as debuggers and profilers, in mind. This co-design approach would simplify and accelerate the development of introspection tools for DSLs, further enhancing the developer experience in using them.

## References

- [BFD\*16] BATTLE L., FISHER D., DELINE R., BARNETT M., CHANDRAMOULI B., GOLDSTEIN J.: Making sense of temporal queries with interactive visualization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (2016), pp. 5433–5443. [2](#)
- [BKB\*21] BEISCHL A., KERSTEN T., BANDLE M., GICEVA J., NEUMANN T.: Profiling dataflow systems on multiple abstraction levels. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 474–489. [3](#)
- [BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D<sup>3</sup> data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309. [1, 2](#)
- [CLC\*22] CAO J., LI M., CHEN X., WEN M., TIAN Y., WU B., CHEUNG S.-C.: Deepfd: Automated fault diagnosis and localization for deep learning programs. *arXiv preprint arXiv:2205.01938* (2022). [3](#)
- [CLRG19] CITO J., LEITNER P., RINARD M., GALL H. C.: Interactive production performance feedback in the ide. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), IEEE, pp. 971–981. [3](#)
- [CSX\*21] CHEN Q., SUN F., XU X., CHEN Z., WANG J., CAO N.: Vizlinter: A linter and fixer framework for data visualization. *IEEE transactions on visualization and computer graphics* 28, 1 (2021), 206–216. [2](#)
- [GGGP21] GEOFFREY X. Y., GAO Y., GOLIKOV P., PEKHIMENKO G.: Habitat: A {Runtime-Based} computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 503–521. [3](#)
- [GKRS11] GRUST T., KLIEBHAN F., RITTINGER J., SCHREIBER T.: True language-level sql debugging. In *Proceedings of the 14th International Conference on Extending Database Technology* (2011), pp. 562–565. [2](#)
- [GLB20] GATHANI S., LIM P., BATTLE L.: Debugging database queries: A survey of tools, techniques, and users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), pp. 1–16. [2](#)
- [GSU\*22] GALE J., SEIDEN M., UTKARSH D., FRANTZ J., WOOLLEN R., DEMIRALP Ç.: Sigma workbook: A spreadsheet for cloud data warehouses. *arXiv preprint arXiv:2204.03128* (2022). [8](#)
- [HCS20] HOPKINS A. K., CORRELL M., SATYANARAYAN A.: Visualint: Sketchy in situ annotations of chart construction errors. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 219–228. [2](#)
- [HHN85] HUTCHINS E. L., HOLLAN J. D., NORMAN D. A.: Direct manipulation interfaces. *Human-computer interaction* 1, 4 (1985), 311–338. [2](#)
- [HSH15] HOFFSWELL J., SATYANARAYAN A., HEER J.: Debugging Vega through Inspection of the Data Flow Graph. In *EuroVis Workshop on Reproducibility, Verification, and Validation in Visualization (EuroRV3)* (2015), Aigner W., Rosenthal P., Scheidegger C., (Eds.), The Eurographics Association. doi:10.2312/eurorv3.20151144. [2](#)
- [HSH16] HOFFSWELL J., SATYANARAYAN A., HEER J.: Visual debugging techniques for reactive data visualization. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 271–280. [2](#)
- [KMM22] KRUCHTEN N., MEASE J., MORITZ D.: Vegafusion: Automatic server-side scaling for interactive vega visualizations, 2022. URL: <https://arxiv.org/abs/2208.06631>, doi:10.48550/ARXIV.2208.06631. [9](#)
- [KWHH17] KIM Y., WONGSUPHASAWAT K., HULLMAN J., HEER J.: Graphscape: A model for automated reasoning about visualization similarity and sequencing. In *Proceedings of the 2017 CHI conference on human factors in computing systems* (2017), pp. 2628–2638. [2](#)
- [MHHH15] MORITZ D., HALPERIN D., HOWE B., HEER J.: Perfoticon: Visual query analysis for distributed databases. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 71–80. [2](#)
- [MK18] MCNUTT A., KINDLMANN G.: Linting for visualization: Towards a practical automated visualization guidance system. In *Vis-Guides: 2nd Workshop on the Creation, Curation, Critique and Conditioning of Principles and Guidelines in Visualization* (2018). [2](#)
- [MWN\*18] MORITZ D., WANG C., NELSON G. L., LIN H., SMITH A. M., HOWE B., HEER J.: Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 438–448. [2](#)
- [SHH21] SCHOOP E., HUANG F., HARTMANN B.: Umlaut: Debugging deep learning programs using program structure and model behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021), pp. 1–16. [3](#)
- [SMWH16] SATYANARAYAN A., MORITZ D., WONGSUPHASAWAT K., HEER J.: Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350. [2](#)
- [Veg22a] VEGA: Vega & vega lite visualization grammars, 2022. URL: <https://vega.github.io/>. [1, 2](#)
- [Veg22b] VEGA: Vega editor, 2022. URL: <https://vega.github.io/editor/>. [3](#)
- [WCLR22] WARDAT M., CRUZ B. D., LE W., RAJAN H.: Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 561–572. [3](#)
- [Wic16] WICKHAM H.: *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. URL: <https://ggplot2.tidyverse.org/>. [1](#)
- [Wil12] WILKINSON L.: The grammar of graphics. In *Handbook of computational statistics*. Springer, 2012, pp. 375–414. [1](#)
- [WQM\*17] WONGSUPHASAWAT K., QU Z., MORITZ D., CHANG R., OUK F., ANAND A., MACKINLAY J., HOWE B., HEER J.: Voyager 2: Augmenting visual analysis with partial view specifications. In *Proceedings of the 2017 chi conference on human factors in computing systems* (2017), pp. 2648–2659. [2](#)
- [YGP20] YU G. X., GROSSMAN T., PEKHIMENKO G.: Skyline: Interactive in-editor computational performance profiling for deep neural network training. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (2020), pp. 126–139. [3](#)
- [YJY\*22] YANG J., JOO H. K., YERRAMREDDY S. S., LI S., MORITZ D., BATTLE L.: Demonstration of vegaplus: Optimizing declarative visualization languages. In *Proceedings of the 2022 International Conference on Management of Data* (2022), pp. 2425–2428. [9](#)