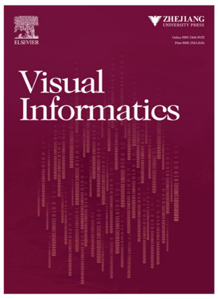


Codelin: An in situ visualization method for understanding data transformation scripts

Xiwen Cai, Kai Xiong, Zhongsu Luo, Di Weng, Shuainan Ye, Yingcai Wu

PII: S2468-502X(25)00008-7
DOI: https://doi.org/10.1016/j.visinf.2025.03.002
Reference: VISINF 233

To appear in: *Visual Informatics*

Received date : 28 October 2024
Revised date : 28 February 2025
Accepted date : 15 March 2025

CodeLin: An In Situ Visualization Method for Understanding Data Transformation Scripts

Xiwen Cai[a], Kai Xiong[a], Zhongsu Luo[a], Di Weng[b,*], Shuainan Ye[a], Yingcai Wu[a]

[a] *The State Key Lab of CAD&CG, Zhejiang University*
[b] *School of Software Technology, Zhejiang University*

Abstract

Understanding data transformation scripts is an essential task for data analysts who write code to process data. However, this can be challenging, especially when encountering unfamiliar scripts. Comments can help users understand data transformation code, but well-written comments are not always present. Visualization methods have been proposed to help analysts understand data transformations, but they generally require a separate view, which may distract users and entail efforts for connecting visualizations and code. In this work, we explore the use of in situ program visualization to help data analysts understand data transformation scripts. We present CodeLin, a new visualization method that combines word-sized glyphs for presenting transformation semantics and a lineage graph for presenting data lineage in an in situ manner. Through a use case, code pattern demonstrations, and a preliminary user study, we demonstrate the effectiveness and usability of CodeLin. We further discuss how visualization can help users understand data transformation code.

Keywords: Data Transformation, Program Visualization, In Situ Visualization

*Corresponding author

Email addresses: `xwcai@zju.edu.cn` (Xiwen Cai), `kaixiong@zju.edu.cn` (Kai Xiong), `zhongsuluo@zju.edu.cn` (Zhongsu Luo), `dweng@zju.edu.cn` (Di Weng), `sn_ye@zju.edu.cn` (Shuainan Ye), `ycwu@zju.edu.cn` (Yingcai Wu)

CodeLin: An In Situ Visualization Method for Understanding Data Transformation Scripts

Xiwen Cai[a], Kai Xiong[a], Zhongsu Luo[a], Di Weng[b,*], Shuainan Ye[a], Yingcai Wu[a]

[a]*The State Key Lab of CAD&CG, Zhejiang University*
[b]*School of Software Technology, Zhejiang University*

Abstract

Understanding data transformation scripts is an essential task for data analysts who write code to process data. However, this can be challenging, especially when encountering unfamiliar scripts. Comments can help users understand data transformation code, but well-written comments are not always present. Visualization methods have been proposed to help analysts understand data transformations, but they generally require a separate view, which may distract users and entail efforts for connecting visualizations and code. In this work, we explore the use of in situ program visualization to help data analysts understand data transformation scripts. We present CodeLin, a new visualization method that combines word-sized glyphs for presenting transformation semantics and a lineage graph for presenting data lineage in an in situ manner. Through a use case, code pattern demonstrations, and a preliminary user study, we demonstrate the effectiveness and usability of CodeLin. We further discuss how visualization can help users understand data transformation code.

Keywords: Data Transformation, Program Visualization, In Situ Visualization

*Corresponding author

Email addresses: `xwcai@zju.edu.cn` (Xiwen Cai), `kaixiong@zju.edu.cn` (Kai Xiong), `zhongsuluo@zju.edu.cn` (Zhongsu Luo), `dweng@zju.edu.cn` (Di Weng), `sn_ye@zju.edu.cn` (Shuainan Ye), `ycwu@zju.edu.cn` (Yingcai Wu)

1. Introduction

Many data analysts write scripts (e.g., Python and R scripts) to process tabular data. These scripts are responsible for transforming raw data into meaningful insights, making them crucial for various tasks such as data cleaning, integration, and feature engineering. The accuracy and efficiency of these transformations directly impact the quality of the resulting data, which in turn affects decision-making processes across industries, such as energy [1, 2], traffic [3, 4, 5], and sports [6, 7]. Given the importance of data transformation scripts in data analysis, it is imperative that analysts not only write but also thoroughly understand and check these scripts to ensure data integrity and reproducibility.

However, understanding data transformation scripts, especially those written by others or retrieved from public repositories, remains a challenge. Data analysts need to trace variables to comprehend the overall data processing pipeline, and dissect the code to understand the purpose of each code block, and print the intermediate result to grasp the specific operations performed by each line of code. It requires considerable effort to obtain the critical information about the data transformation process by reading code. Moreover, the tasks of understanding data transformation code is further complicated by the lack of comprehensive documentation or comments in many scripts. While recent advancements in large language models (LLMs) offer means for code understanding, these models are not yet a well-established solution. The generated content such as comments would require manual verification. Therefore, the need for effective methods to facilitate the understanding of data transformation scripts remains a critical and unsolved problem, essential for ensuring the robustness and reliability of data transformation processes.

Some visualization-based methods have been proposed to help analysts understand code and data transformations. Traditional code visualizations help users in debugging and communicating. Recent methods [8, 9, 10, 11] have explored different visualization techniques for revealing data transformation semantics (i.e., the key information such as data tables, function types and parameters, and table changes) and pipelines. These methods commonly require an independent view to help users gain an overview of the data transformation process. When using an independent view to visualize data transformation code, there are two limitations. First, the view and the visual representations require considerable space, which can disturb the presentation of code. Second, the connection between code and visualizations is indirect, and it may require efforts to match them and distract users from focusing on the code [12, 13].

To address these two limitations, we propose using in situ program visualization [14, 13] to visualize data transformation scripts. In our work, in situ program visualization refers to the visualization embedded within the original programming context (e.g., code editors). We are motivated by the user feedback in our previous work [11], where users found that though the visualizations were helpful in understanding data transformation scripts, switching between code and visualizations was inconvenient and distracting due to the gap between them. A promising solution may be in situ program visualization. By embedding visual representations into code or placing them alongside code, in situ program visualizations can reveal crucial information, such as variable status and program behavior, while narrowing the gap between code and visualizations. Although in situ visualizations have been used to help users understand programs, visualizing the data transformation process in an in situ manner has rarely been explored.

In this work, we explore how to use in situ program visualization to help data analysts understand how a data transformation script works with actual data. Informed by the interviews with data analysts and the summaries of design requirements from previous work, we derived a set of design goals and considerations. We present a new visualization method, CodeLin, which includes a backend model and a visualization interface. Taking the data transformation script and its corresponding dataset as input, the backend model parses the code and obtains key information for each line of code and the intermediate tables it produces. The visualization interface includes word-sized glyphs for presenting data transformation semantics and a lineage graph for presenting data lineage, both working in an in situ manner. Through code pattern demonstrations, a use case, and a preliminary user study, we demonstrate the effectiveness and usability of CodeLin. Based on the results and user feedback from the preliminary user study and its follow-up study, we further discuss how to use visualization to help users understand data transformation code.

The major contributions of this work are as follows:

- The problem characterization and design considerations for in situ data transformation visualization.
- A novel visualization method for supporting the interpretation of data transformation code in an in situ manner.
- A use case and a preliminary user study to evaluate the effectiveness and usability of the proposed method, and the insights and implications derived from them.

2. Related work

2.1. Visualization for Revealing Data Transformations

Data wrangling is a common practice for data analysts. In the past decades, programming languages and libraries, as well as interactive tools have been developed to help data analysts transform data [15]. The process of data transformation could be difficult to comprehend, and the need for understanding data transformation processes is growing [16]. To address this issue, numerous techniques have been utilized to facilitate the understanding of data transformation processes. For example, some tools (e.g., Wrangler [17]) use natural language to describe data transformation operations and provide visual previews of table changes to show the effects of operations. In our work, we focus on visualization techniques that reveal the data transformation semantics of a single step and entire data processing pipelines or data lineage.

Traditionally, icons have been used to present data transformation operations such as sorting and filtering in tools for data wrangling (e.g., Microsoft

Excel [18] and Google Sheets [19]). Recent research has proposed more semantically rich visualization techniques to show how actual datasets are transformed through data transformations. Data Tweening [8] and Datamations [10] use animation to explain data transformations. TACO [9] uses matrices to represent data changes in tables. SOMNUS [11] leverages grid-based glyphs to depict data transformations. Our work explores in situ visualization for presenting data transformation semantics, aiming to shorten the gap between visualization and code through juxtaposition. To accommodate the context of in situ visualization, we propose some new considerations for visualization design. Based on these considerations, we present new designs of glyphs and lineage graphs, which save space, present information more concisely, use color and layout to enhance lineage representation, and are aligned with the code.

Due to their diagrammatic nature, data transformation pipelines are often represented as node-link diagrams [20]. Typical examples include VisTrails [21], AVOCADO [22], and SOMNUS [11]. In addition to node-link diagrams, Sankey diagrams [23] and other visualization forms [24, 25] are also used to present data transformation pipelines. Generally, in order to present data transformation pipelines, existing work requires an independent view (such as a graph view). When used for visualizing code, the connection between the graph views and the code is often weak. Our work places the lineage graph next to the code and aligns visual representations with code lines, thus tightening the connection between the pipeline and the code and reducing the cognitive load for users.

2.2. In Situ Program Visualization

Program visualization helps users understand necessary information such as the structure and runtime behavior of a program [26] and is commonly used for debugging and educational purposes [27]. There are a number of visualization methods [28, 29, 30, 31, 32, 33] for facilitating the understanding of programs and code. We focus on the in situ program visualization methods, which are more relevant to our work.

In situ program visualization techniques have been proposed to explore a range of programming topics [13]. CODERCHROME [14] uses code coloring to enhance code reading in Eclipse. Swift [34] utilizes visual annotations to provide contextual information in cyberphysical programming. Word-sized visualizations embedded into code are leveraged to monitor value changes of numeric variables [35] and to understand performance bottlenecks of a program [36]. Hoffswell et al. [13] have described a design space for the placement of embedded visualization in code. Besides, graphical tools (e.g., GitLens [37]) inside integrated development environments (IDEs) are also used to help with version and branch management of source code. These works provide useful information about programs, but none of them reveal the semantics of data transformations. Unravel [12] is an exception, which uses visualizations to highlight table changes, with a focus on single-table data transformation operations that use fluent interface (method chaining) in R. To the best of our knowledge, none of the existing in situ visualization techniques is able to present data transformation pipelines. Our work explores how to reveal data lineage and code structure in an in situ manner, aiming to better assist users in understanding the process of data transformation.

3. Problem Description and Design Goals

This work was motivated by user feedback from previous work [11]. When using previously provided visualization tool for understanding data transformation scripts, users felt that visualization has several benefits. For example, visualization could help them quickly grasp the usage of an unfamiliar function, and thus saves the time for looking up documentation. In addition, visualization can provide an overview of a script and reveal the actual data changes caused by data transformations, which were difficult to see through reading the code. However, due to the gap between the view showing the code and that showing the visualizations, users felt that switching back and forth between code and visualizations was inconvenient and could easily distract their attention. They hoped that we could provide a more efficient solution. To meet their demands, we thought of using in situ program visualization. Yet, it is not easy to show the semantic information of data transformations and help users understand the pipeline in the limited space of the code. To achieve this, we explored the design possibilities.

Our goal is to develop an in situ visualization method to help users better understand and manage data transformation scripts. To achieve this goal, we first conducted iterative interviews with four data analysts (DA1-DA4) to understand their needs. They all have more than five years of data analysis experience and are not co-authors of this paper. DA1 and DA2 are data analysts working in a research lab. They have cooperated with us for more than two years and participated in our previous work as domain experts. DA3 is a Ph.D. candidate and DA4 is a full-time researcher, both with research interests in data science.

In the first round of interviews, our goal was to figure out their basic workflow in handling unfamiliar data transformation scripts. We conducted semi-structured interviews, the seed questions included "how did you handle unfamiliar data transformation scripts", "what is the basic workflow", "did you encounter any problems", etc. We summarized the common tasks based on the interviews, and then developed preliminary design goals. Then, we proposed some design prototypes.

To further clarify the design goals and determine some key design choices, we conducted a second round of interviews, which were unstructured and focusing on a topic: how visualization can help them understand data transformation scripts. In this round, we showed them our prototype designs, based on which we had further discussions. Through the second round of interviews, we determined the design goals. At the same time, their opinions helped us determine some design choices. For example, DA3 explicitly rejected inserting visualizations into code (e.g., "df{glyph} = ...") because he believed that "this seriously affects the format of code and makes it hardly readable". We clarify how the analysts guided us when introducing the specific design goals.

3.1. Terminologies

A data transformation script contains one or more data transformation **pipelines**, which consist of data transformation operations. Data transformation **operations** are the basic units in data transformation scripts, each of which changes the structure or the content of data. From the perspective of data lineage, a pipeline is a directed acyclic graph (DAG) with tables as nodes and operations as links, and contains one or more linear **chains**. Operations consecutive in processing (i.e., the output of one operation is the input of another) constitutes a chain.

In Python and R scripts, each data transformation operation is typically one **line** of code. Multiple consecutive lines of code form a **snippet** (block). The arrangement (e.g., segmentation and order) of code lines usually follows a certain coding style, which can be based on the data transformation chain (i.e. putting the consecutive operations together), based on the function of the code (e.g., putting data filtering operations together), etc. We refer to the arrangement of and relations between lines of code as **code structure**.

In the data transformation code, the most important variables are **table variables** (e.g., *df*), since the essence of data transformation is to make changes to tables. A table variable appears as an input or output in different lines of code, with its **lifespan** spanning from its initial definition to its last occurrence.

3.2. User Tasks

T1 **Understanding Individual Operations.** To understand what the script is exactly doing, data analysts need to understand individual operations. They will read the code line by line, identify the specific functions applied, and infer how tables change through each operation. They may not read every line in detail but focus on certain key operations.

T2 **Distinguishing Code Snippets.** Data analysts often distinguish code snippets in a data transformation script for ease of management. They break the script into more comprehensible components, such as major data transformation stages (e.g., input, cleaning, and integration stages) or different sub-flows (e.g., data transformation chains for different outputs). Although many data transformation scripts are already snipped, some long snippets may need further subdivision, and some short ones can be combined to reveal a higher-level purpose.

T3 **Identifying Data Flow.** When handling an unfamiliar data transformation script, identifying data flow is important. Data analysts need to identify the inputs, outputs, and the sequence of steps that transform the raw data into the final outputs. If they modify a part of the code without fully considering the impact on the subsequent data flow, it could lead to incorrect results or unexpected consequences.

3.3. Design Goals

Based on the expert interviews, we proposed four design goals (G1-G4). G1-G3 correspond to how to assist users in their tasks (T1-T3). G4 is about how to help users make better use of visualization and reduce the costs of using visualization. We further align these goals to the rationales and findings from previous works [13, 11, 9, 38, 12] and discuss their similarities and differences.

G1 **Reveal Table Changes.** For experienced data analysts, the code itself can provide them with a lot of semantic information (such as input/output tables and data transformation functions). However, they sometimes encounter cases where they cannot judge the table changes by code alone. For example, by performing a *filter* operation on *date*, DA1 expected to remove the rows that had a date earlier than "2020/2/20" from a table. However, the table had not been changed since the time format of the filtering condition was invalid. Table changes reveal important semantics about data transformations and are emphasized by a number of works [9, 11]. Similar to them, we also aim to facilitating the understanding of data transformation code through revealing table changes.

G2 **Present Code Structure.** Many data transformation scripts have their own organization logic, and understanding this logic can affect how data analysts maintain them and complete downstream tasks. For example, DA2 is accustomed to placing operations for reading files or filtering data together. However, the scripts he took over did not always follow this paradigm, and he would modify the code according to his own needs. In order to modify code effectively, he needed a certain understanding of how the code was organized originally. Furthermore, in cases where downstream tasks involve code modification, understanding the original code structure becomes crucial for avoiding errors. DA3 mentioned a case where he modified several lines of code without knowing their relationship to the rest of the code, which led to unexpected errors. We attempt to present the code structure within the code to facilitate code maintenance and downstream tasks.

G3 **Connect Transformation Pipeline.** Data transformation scripts may contain complex data processing pipelines with multiple branches and chains. In a script, multiple lines of code for the same chain may be scattered in different locations. During the interviews, both DA1 and DA2 reported difficulty in connecting the code before and after in some scripts, which requires repeated interactions, memory, or annotations. They believed that visualization has a great advantage in making the data transformation process more intuitive. Xiong et. al [11] also mentioned the need for data analysts to understand the data flow and listed "present table provenanc" as one of their design requirements. However, they did not consider revealing the provenance relationships between lines of code within the code itself. In our work, we attempt to use in situ visualization help them view the data flow within the code.

G4 **Enhance Code-Visualization Connection.** In most code understanding tasks, code is the focus of users, and the task of reading is emphasized [13]. Visualization can help understand data transformation code, which was acknowledged by the four data analysts. However, the solution of placing visualization in a separate view is not what they exactly wanted. They are concerned about the cost of switching contexts, which may reduce efficiency. Like existing work [12, 13], we also hope to reduce such costs by embedding visualization into code. Furthermore, we aim to strengthen the connection between code and visualization through the visual designs such as consistent visual encoding.

4. Overview

The architecture of CodeLin consists of a backend model and a frontend interface. The backend model uses the code adaptor from SOMNUS [11] to parse the semantics of the code and uses the approach of TACO [9] to judge the changes of the table. Since these backend modules are not our original work, this paper only provides a brief introduction for readers to understand the system, and the details can be referred to the original papers. Additional details such as the categorization of transformations and supported functions can be found in the supplemental materials.

4.1. Backend Modules

We employ the program adaptor from SOMNUS [11] to extract the semantics from the code. The program adaptor takes a script and corresponding data files as input, and obtains the input/output tables/columns/rows and data transformation function names and parameters of each code line through program execution and code parsing. Then, it infers the transformation type of each line using transformation inference. According to the framework proposed in Table Scraps [39], there are 15 main types of transformations: the combinations of 5 operation classes (create (0:1), delete (1:0), transform (1:1), combine (N:1), and separate (1:N)) and 3 data object types (tables, columns, and rows). In this paper, we use *operationclass_dataobject* (e.g., *create_tables*) to represent a certain type of data transformation operation.

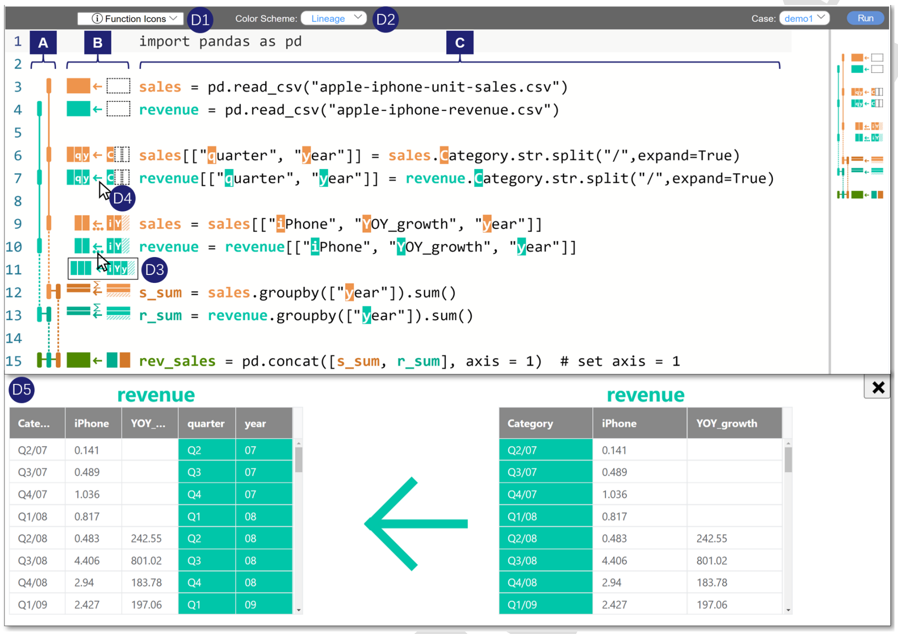
According to the relationships of the input and output tables, we obtain the table-level data lineage. We further expand the adapter of SOMNUS to map the function names (e.g., "sum" in Pandas) to the function types (e.g., *Aggregate*). Note that we support operations without an explicit function name (such as lines 9 and 10 in Fig. 1). Meanwhile, we categorize the data transformation operations into **table/column/row-oriented** based on the data object type. For those single-table (one-table-to-one-table) data transformations, which include *transform_tables* and column/row-oriented data transformations, we leverage TACO's approach [9] to obtain the table changes caused by each operation. We mainly consider the changes in the number of columns/rows in single-table data transformations, including increase, unchanged, and decrease.

4.2. Visualization Interface

The visualization interface (Figure 1) of CodeLin consists of three major parts: the lineage graph (Figure 1A), the glyphs (Figure 1B), and the code editor (Figure 1C). According to the data lineage, we assign colors to different table variables. Then, we generate a lineage graph, followed by drawing glyphs and coloring the code. Additionally, there is a minimap containing the lineage graph and the glyphs, which provides an overview of the entire data transformation pipeline and helps users gain information beyond the present context when the script is long.

4.3. Implementation

To demonstrate the effectiveness of the proposed method, we implemented a prototype system. The system is a web application with a backend server built with Flask [40] and an interface built in Vue.js [41]. In the backend, we leverage the program adaptor from SOMNUS [11] for parsing scripts, extracting parameters, and caching intermediate tables, and implement a change detector using Pandas [42]. Presently, we support 25 functions in dplyr (R) and tidyr (R) and 23 functions in Pandas (Python). In the frontend, we utilize Monaco Editor [43] for presenting the code and D3.js [44] for drawing the visual representations.



Color Scheme: Lineage

Case: demo1

```
import pandas as pd

sales = pd.read_csv("apple-iphone-unit-sales.csv")
revenue = pd.read_csv("apple-iphone-revenue.csv")

sales[["quarter", "year"]] = sales.Category.str.split("/",expand=True)
revenue[["quarter", "year"]] = revenue.Category.str.split("/",expand=True)

sales = sales[["iPhone", "YOY_growth", "year"]]
revenue = revenue[["iPhone", "YOY_growth", "year"]]

s_sum = sales.groupby(["year"]).sum()
r_sum = revenue.groupby(["year"]).sum()

rev_sales = pd.concat([s_sum, r_sum], axis = 1)  # set axis = 1
```

Cate...	iPhone	YOY_...	quarter	year
Q2/07	0.141		Q2	07
Q3/07	0.489		Q3	07
Q4/07	1.036		Q4	07
Q1/08	0.817		Q1	08
Q2/08	0.483	242.55	Q2	08
Q3/08	4.406	801.02	Q3	08
Q4/08	2.94	183.78	Q4	08
Q1/09	2.427	197.06	Q1	09

Category	iPhone	YOY_growth
Q2/07	0.141	
Q3/07	0.489	
Q4/07	1.036	
Q1/08	0.817	
Q2/08	0.483	242.55
Q3/08	4.406	801.02
Q4/08	2.94	183.78
Q1/09	2.427	197.06

Figure 1: The interface of our prototype system. The main body is a code editor, with line numbers on the left side. Our visual designs include (A) a lineage graph to present data lineage, (B) word-sized glyphs to present data transformation semantics, and (C) code colored in the same visual style. In the minimap on the right, we display the lineage graph and glyphs instead of the code. We support different interactive features. Users can choose to display icons for some common data transformation functions (D1) and select different color schemes (D2). When users hover on the circles below the arrow, they could view the complete glyph (D3). When users click the glyph (D4), they can see the details of table changes (D5).

5. Visual Design

In this section, we introduce our visual design and interactions, as well as how we could achieve our design goals through these designs. The current algorithms for color encoding in subsection 5.3 and the layout of the lineage graph in subsection 5.4 are only a preliminary implementation for the completeness of the work and are not intended as independent contributions. We have placed the details of the algorithms in the supplemental materials.

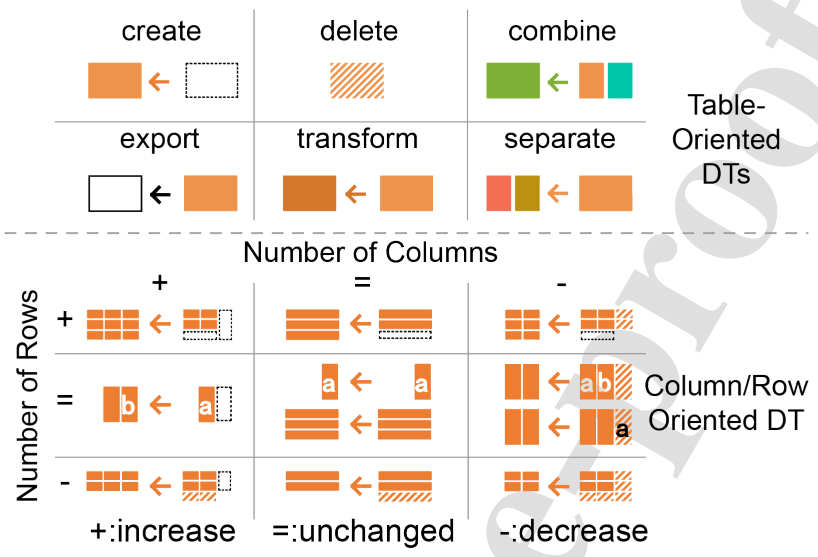


Figure 2: Glyphs for showing table-oriented DTs (top) and changes in numbers of rows/columns in column/row-oriented DTs (bottom).

5.1. Glyph Design

Our design of glyphs is based on the design of SOMNUS [11]. We further consider how to design effective word-sized glyphs and how to connect them with code. In the glyphs, we use color for encoding data lineage, which will be discussed in subsection 5.3.

Data Objects. We use rectangles 🟧 to represent tables , vertical bars with the first character of the column names inside ab to represent columns, and horizontal bars ☰ to represent rows.

DT Operations. Similar to SOMNUS, we use grid-based glyphs to show the semantics of DTs (Figure 2). For table-oriented DTs (Figure 2 top), we depict the operation type. For column/row-oriented DTs, we present changes in the numbers of columns/rows (Figure 2 bottom). Compared with SOMNUS, we further considered how to apply the glyphs in an in situ manner. The following are how our approach differs from SOMNUS and our design considerations.

First, to clearly reveal table changes (G1), we focus more on characterizing the data objects (tables, columns, and rows) and the types of changes in number (increase, unchanged, and decrease), rather than the types of DTs. Changes in the number of data objects are probably the most important semantics in DTs, which reveal how table shape changes and help users make judgments in many cases. In the limited space, we need to present important information in a concise and explicit way. Due to this design choice, there is an issue that our glyph design may not effectively distinguish some DT operations. To address this issue, we use icons shown above arrows to represent the function types.

Second, we do not use the color channel to represent changes in rows and columns. In our work, color is mainly used to encode table variables according to data lineage, for better illustrating DT pipeline (G3). At the same time, we believe that though using hue can better express the type of DT in a single operation, it probably causes inconsistent color encoding among multiple steps of data transformation operations.

Third, we maintain the consistency between visual representations and code to enhance their connection (G4). For consistency, the same color is applied to a data object in code and its corresponding visual element. In addition, considering that in an assignment statement, the output table is on the left side and the input table is on the right side, we place the corresponding representations in the same position ([output table] ← [input table]).

Fourth, we simplify the information to save space and to enhance readability. Considering the limited space, we only displayed the most general information. We limit the space of glyphs by restricting the number of visual elements displayed by default and their standard widths. For operations that are table-oriented, we display no more than two tables on each side (otherwise, the width of the table will be less than or equal to the width of the columns, causing confusion) and do not depict the columns and rows in the table. For operations that are column-/row-oriented, we display no more than three columns and three rows on each side and do not depict the specific number of columns and rows. When the visual elements are not displayed completely, we draw three dots below the arrow (Figure 1 D3) as a prompt, and the complete information

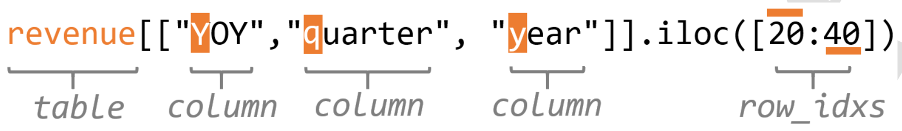


Figure 3: Visual embellishments are applied to data objects in code. We change the color of table name and the background color of the first character of the column name/index, and use overlines/underlines to indicate the starting/ending row indexes.

can be viewed when the user hovers over the circles.

5.2. Code Style

We change the code style to facilitate distinguishing different information in the code and to connect visualization with the code (G4). We divide the information in code into two categories: data-related and function-related. We only apply visual encoding to the data-related information to help users distinguish data objects from function names and data-irrelevant parameters (Figure 3).

Data-related information includes table names, column names/indexes, and row indexes. For a table name, we change its color to match the color used in the visual representation. For column names/indexes, we change the background color of the first character to indicate that it is a column name/index, which is consistent with the column in the glyphs. For rows, we use overlines/underlines to indicate the starting/ending row indexes.

5.3. Color Encoding

We use color to encode table variables. We hope that the tables within the same lineage have similar colors, while the colors between different lineages have a certain degree of distinction. To achieve this goal, we use a modified version of the subtractive Tree Colors algorithm [45].

Tree Colors [45] is a coloring method used to create color maps for tree structure data based on the Hue-Chroma-Luminance (HCL) color model. Due to the fact that data lineage is essentially a compound graph rather than a tree, Tree Colors cannot be directly applied to data lineage. In a lineage graph, some

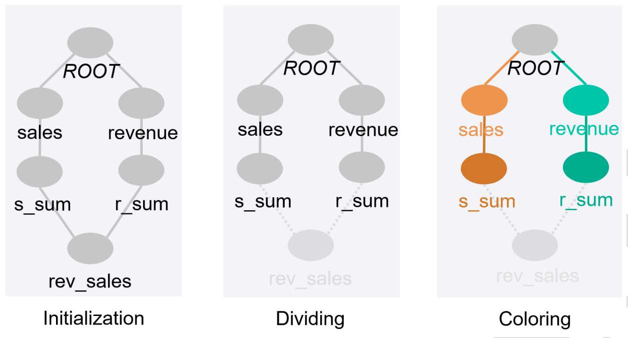


Figure 4: The illustration of one round of the coloring algorithm. We first convert the lineage graph into a compressed graph (initialization). Then, we divide the graph into a tree without compound nodes and branches with compound nodes (dividing), and colorize the tree (coloring).

nodes may have two or more parent nodes, which we call compound nodes. We need to handle these compound nodes and their generations specially. Therefore, we modify the original algorithm. Using the process for coloring Figure 1 A as an example, our algorithm consists of the following steps:

Initialization. Converting the lineage graph into a compressed graph (Figure 4 left), where a node represents a table, without considering it appears multiple times in the code.

Dividing. Dividing the compound graph into two parts: a tree without compound nodes and branches with compound nodes (Figure 4 middle). Each of the branches with compound nodes is regarded as a compound graph, which would be divided in the following rounds.

Coloring. Coloring the tree without compound nodes in this step (Figure 4 right). In the first round (after the first division), we use the original Tree Colors algorithm for coloring. In the following rounds, for each tree to be colored, we first color the root according to the colors of its parents and then color the tree according to the Tree Colors.

Our colors are based on data lineage, so table variables with similar lineage will naturally have similar colors. When users may want to distinguish variables

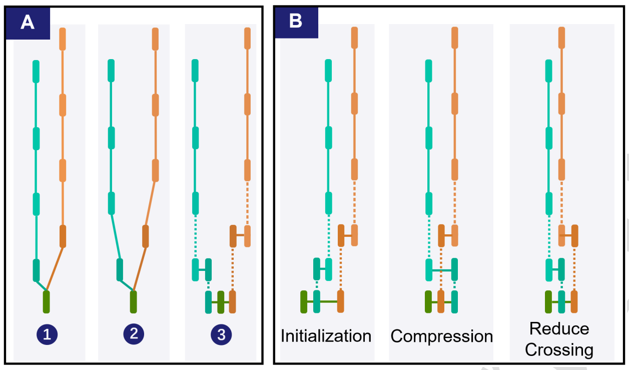


Figure 5: Different design choices of the layout of lineage graph (A) and the algorithm for refining layout in our work (B).

through colors, they could choose the "Category" color scheme (Figure 1 D2) which uses common categorical color scale to encode tables.

5.4. Lineage Graph

To reveal the data pipeline (G3), presenting data lineage is crucial. However, visualizing data flow directly in glyphs or code leads to overlap and disrupts alignment or formatting. An alternative is to route links around them, but this tends to cause crossover and reduces readability. To avoid the aforementioned issues, a standalone lineage graph connected to glyphs and code through alignment is chosen for clarity.

Visual Representation. We use node-link diagrams to present data lineage. Each table in each code line is represented as a node that has the same height and vertical position as that in the glyph. We reduce the widths of the nodes to decrease the distance between the visual representations in the lineage graph and the code (G4) as well as to save space. To avoid confusion between the tables in the lineage graph and the columns in the glyphs, we apply round corners to the nodes in the graph. We use solid lines to indicate tables that have been transformed and dashed lines to indicate tables that have not been transformed.

Layout. When using traditional graph layout techniques (e.g.,[46, 47]), it is not easy to distinguish between table variables (Figure 5 A1). To solve this problem, we could differentiate the variables in terms of their horizontal positions (Figure 5 A2), but it is still difficult to present table lifespans. For example, with Figure 5 A2, it is not intuitive to see that *sales* appears as an input in the third to last code line in Figure 1. To clearly present table lifespans, we linearize the graph according to table variables (Figure 5 A3) rather than paths (which would be like Figure 5 A1). Meanwhile, given the choice of linearization, the existing graph layout techniques do not work well for us. Thereby, we use a heuristic algorithm to generate the layout (see Figure 5 B). The layout algorithm consists of the following steps:

Initialization. Calculating the weight of each table and assign an initial horizontal position to it (Figure 5 B left). The hierarchical structure is preserved to better reveal data lineage and the branches with higher weights would be placed closer to the code to reduce the total distance between code and visual representations (G4).

Compression. Compressing the lineage graph horizontally to save space and make the visual elements closer to code (Figure 5 B middle), thus enhancing their connection (G4).

Reducing Crossings. Moving the tables horizontally to reduce the number of crossings (Figure 5 B right).

5.5. Placement

Hoffswell et al. [13] have identified placement techniques for in situ program visualizations and discussed five trade-off metrics: reflow, spacing, occlusion, width, and alignment. Based on these metrics, we clarify our considerations:

Reflow. During interviews with data analysts, it was found that they prefer that the inline structure of the code remains intact and not be disrupted by the visualization. As a result, it is important to avoid reflow.

Spacing. When visualizations are placed above or below the code, it often results in a reduction in the number of code lines being displayed. To optimize space utilization, we avoid occupying vertical space.

Occlusion. Occlusion may impact the reading experience. We avoid introducing occlusion unless users trigger interactions.

Width. We restrict the number of data objects displayed by default and apply a standard width to the data objects, to constrain the maximum width of a glyph by default as well as to facilitate alignment and visual comparison.

Alignment. To enhance the connection between visualizations and code, we aim to align visualizations horizontally with the code lines. Furthermore, we aligning the glyphs vertically to facilitate visual search and enable users to quickly identify snippets and patterns in the code (G2).

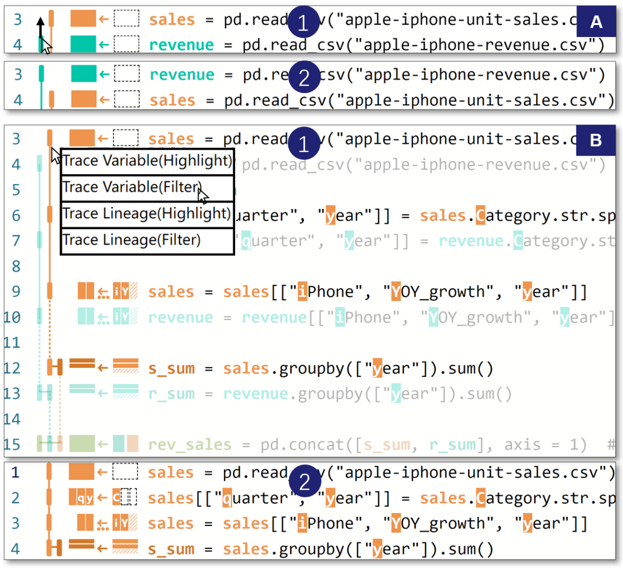
In our work, we place the lineage graph on the left, followed by the glyphs, and the code on the right. Considering that users typically read from left to right, placing the lineage graph on the left allows users to first notice the overview of the transformation pipeline. Placing the glyphs on the left to the code allows users to first notice the blocks and patterns of the code before reading it, and prompts users of the actual impact of the code on the data. Although incorporating visualizations into code can enhance the connection between visualizations and code, our domain experts do not endorse this approach due to two concerns: first, the display of code would be affected by the visualization (e.g., it would result in the undesired reflow); second, the task of reading the code may be influenced by the visualization (e.g., they would naturally read the visualization while reading the code). Therefore, we place the visualization alongside the code rather than inserting glyphs into the code.

5.6. Interactions

Viewing Table Information. Users can click on glyphs (Figure 1 D4) to view details of data tables in a DT operation (Figure 1 D5). In the data tables, we use colors consistent with the glyphs to highlight relevant rows and columns.

Adjusting Code. Users can adjust the position of a code line by dragging and dropping it (Figure 6 A1-A2).

Tracing Lineage and Variables. Users can also right-click on the table and select “Trace Variable” or “Trace Lineage” from the popup menu. Depending



```
sales = pd.read   v("apple-iphone-unit-sales.c
revenue = pd.read_csv("apple-iphone-revenue.csv")
```

```
revenue = pd.re   csv("apple-iphone-revenue.csv")
sales = pd.read_csv("apple-iphone-unit-sales.csv")
```

```
sales = pd.read   v("apple-iphone-unit-sales.c
                    pd.read_csv("apple-iphone-revenue.csv")

                   uarter", "year"]] = sales.Category.str.sp
                   'quarter", "year"]] = revenue.Category.st

  sales = sales[["iPhone", "YOY_growth", "year"]]
  revenue = revenue[["iPhone", "YOY_growth", "year"]

s_sum = sales.groupby(["year"]).sum()
r_sum = revenue.groupby(["year"]).sum()

rev_sales = pd.concat([s_sum, r_sum], axis = 1)  #
```

```
sales = pd.read   v("apple-iphone-unit-sales.csv")
sales[["quarter", "year"]] = sales.Category.str.sp
  sales = sales[["iPhone", "YOY_growth", "year"]]
s_sum = sales.groupby(["year"]).sum()
```

Figure 6: By dragging nodes (A1), users can adjust the position of the code (A2). Users can interactively trace variables and lineage (B1), including highlighting (B1) and filtering (B2) the lifespan or the lineage of variables.

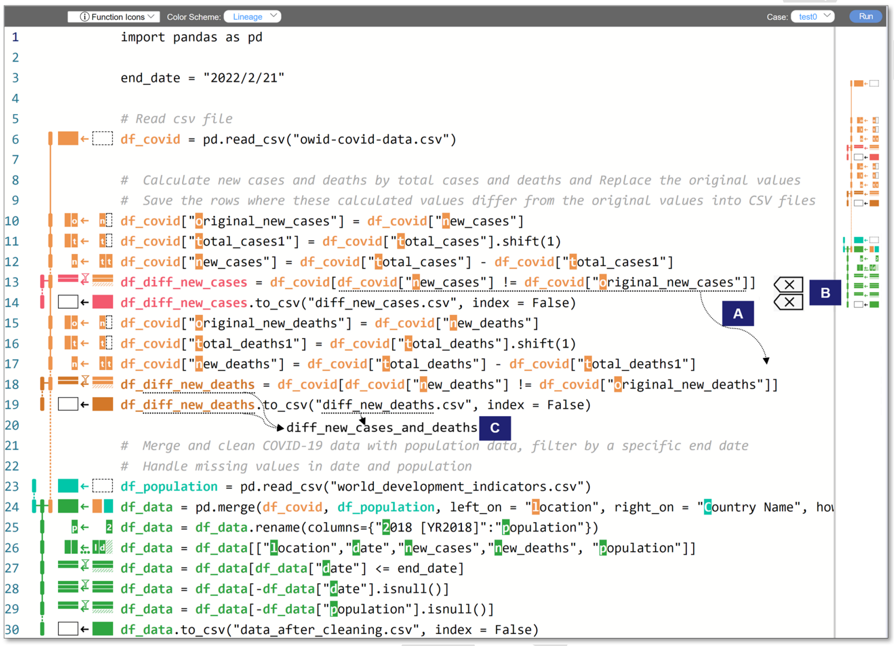
on their selection, we will highlight the relevant code lines (Figure 6 B1) or filter out unrelated code lines (Figure 6 B2).

6. Evaluation

In this section, we provide a use case, several code patterns, and a user study to show the usability and effectiveness of CodeLin. To avoid occupying too much space, we have a more complicated example which may better reveal the scalability of CodeLin in the supplementary materials.

6.1. Use Case

This demo showcases how a data analyst utilized CodeLin to understand a script inherited from a colleague and accomplished a new data analysis task through modification. He was told that the script produces three output tables, and two of them, “diff_new_cases.csv” and “diff_new_deaths.csv”, store records



```
1   import pandas as pd
2
3   end_date = "2022/2/21"
4
5   # Read csv file
6   df_covid = pd.read_csv("owid-covid-data.csv")
7
8   #  Calculate new cases and deaths by total cases and deaths and Replace the original values
9   #  Save the rows where these calculated values differ from the original values into CSV files
10  df_covid["original_new_cases"] = df_covid["new_cases"]
11  df_covid["total_cases1"] = df_covid["total_cases"].shift(1)
12  df_covid["new_cases"] = df_covid["total_cases"] - df_covid["total_cases1"]
13  df_diff_new_cases = df_covid[df_covid["new_cases"] != df_covid["original_new_cases"]]
14  df_diff_new_cases.to_csv("diff_new_cases.csv", index = False)
15  df_covid["original_new_deaths"] = df_covid["new_deaths"]
16  df_covid["total_deaths1"] = df_covid["total_deaths"].shift(1)
17  df_covid["new_deaths"] = df_covid["total_deaths"] - df_covid["total_deaths1"]
18  df_diff_new_deaths = df_covid[df_covid["new_deaths"] != df_covid["original_new_deaths"]]
19  df_diff_new_deaths.to_csv("diff_new_deaths.csv", index = False)
20                   diff_new_cases_and_deaths
21  #  Merge and clean COVID-19 data with population data, filter by a specific end date
22  #  Handle missing values in date and population
23  df_population = pd.read_csv("world_development_indicators.csv")
24  df_data = pd.merge(df_covid, df_population, left_on = "location", right_on = "Country Name", hov
25  df_data = df_data.rename(columns={"2018 [YR2018]":"population"})
26  df_data = df_data[["location","date","new_cases","new_deaths", "population"]]
27  df_data = df_data[df_data["date"] <= end_date]
28  df_data = df_data[-df_data["date"].isnull()]
29  df_data = df_data[-df_data["population"].isnull()]
30  df_data.to_csv("data_after_cleaning.csv", index = False)
```

Figure 7: A user used CodeLin for code modification. After understanding the overall pipeline with the help of lineage graph and identify the output files that are relate to his task in line 14 and 18, he decided to focus on the code in line 10-19. By reading the glyphs, he further distinguished the code snippets and understand their purposes. To complete his task, he copied the filter condition in line 13 and pasted in line 18 (A). Then, he deleted line 13-14 (B). Finally, he changed the identifier and file name in line 18-19 (C).

with calculated values of new cases and new deaths that were different from the original ones, respectively. Due to changes in downstream data analysis requirements, one CSV file that contained both types of differential information was needed instead of two separate tables. His task was to modify the script to meet this requirement. To complete this task, the data analyst uploaded the script to CodeLin and clicked the "Run" button (Figure 7).

By reading the code, it would be difficult for him to have an intuitive understanding of the data flow, as the relationship between the lines of code is not clear. With our method, he could get a rough grasp of the overall data transformation process by reading the visualization . The script first imports a data table (*df_covid*) in line 6. The first snippet (lines 10-19) mainly processes *df_covid*, and has two short branches which start from lines 13 and 18 and end with exporting files in lines 14 and 19 respectively. Then, the script merges *df_covid* with the table imported in line 23 (*df_population*), resulting in a new table (*df_data*). The code after line 23 all processes *df_data*.

There were three output tables in the entire pipeline, produced by the operations at lines 14, 19, and 30, among which the tables output at lines 14 and 19 were related to the task of the data analyst. Traditionally, in order to locate the relevant code of the two output tables related to his task, the data analyst needed to search for the file name (if known) or function names (e.g., *to_csv*) to locate the code that outputs them, and then double-click the variable names to find out where they appear in the code. To confirm the code related to them, the data analyst needed to repeatedly interact with different variables, in which process he can easily lose contextual information. Using CodeLin, he could find the related operations by viewing the glyphs and find out the code related to them using the lineage graph or the interactions. The code after line 20 had no impact on the two tables and would not use the corresponding variables, so he only needed to focus on the code before line 20. The two tables mainly depended on *df_covid*, and the transformation process for *df_covid* starts from line 10. Therefore, he decided to focus on the code from lines 10 to 19.

Next, he planned to break the code into different snippets according to subtasks for easier management and modification of the code. Traditionally, he needed to understand each line of code or obtain some prior knowledge before he could break down the code. Although comments exist, he could not directly associate them with the lines of code. With our glyphs, he could directly see the composition of this section of code: lines 10-12, performing column-oriented data transformation operations on *df_covid*; lines 13-14, filtering specific rows of *df_covid* to get a new table; lines 15-17, performing column-oriented data transformation operations on *df_covid* again; lines 18-19, filtering specific rows of *df_covid* to get a new table again. At the same time, by observing the glyphs, he could see that the operations performed by lines 10-12 are similar to those performed by lines 15-17, and the operations performed by lines 13-14 are similar to those performed by lines 18-19. Combining comments in lines 8-9, the user could infer the functionalities of these code snippets: lines 10-12 and lines 15-17 are used for calculating new cases and deaths by total cases and deaths and replacing the original values; lines 13-14 and lines 18-19 are used for filtering the rows where these calculated values differ from the original values and saving them into CSV files. If there were no comments or if he wished to obtain some detailed information, he could read through lines 10-12 and 13-14 of the code and infer the purpose of lines 15-17 and 18-19 based on the similarity.

After understanding the code, the user decided to start his task. He needed to merge *df_diff_new_cases* and *df_diff_new_deaths*. One way is to merge the two tables and then filter out the columns he wants. But based on his experience, this is not the most efficient way. From the lineage graph, it can be seen that neither *df_diff_new_cases* nor *df_diff_new_deaths* are used in the subsequent code. Therefore, lines 13-14 and lines 18-19 can be safely deleted or modified. He only needed to keep the data that meets both the filtering conditions of line 13 and line 18. After ensuring this, he decided to delete lines 13-14 and modify lines 18-19 to achieve his goal. He copied the filter condition in line 13, pasted it in line 18, and used the logical OR operator ("|") to combine the two filter conditions (Figure 7 A). He then deleted lines 13-14 (Figure 7 B) and changed the identifier (from "df_diff_new_deaths" to "df_diff_new_cases_and_deaths") and the name of the output CSV file in line 18-19 (Figure 7 C). The resulting *df_diff_new_cases_and_deaths* table would contain rows where either *new_cases* or *new_deaths* differs from the original values. With this modification, he successfully fulfilled the task.

After the modification of code, he ran the code to ensure the script produces desirable outputs. While reviewing the post-run code, he noticed that line 26 (line 28 in Figure 7) had executed a filter operation. However, the number of rows in the table had not changed. He inspected the data transformation process and found that line 25 (line 27 in Figure 7) also filtered *date*. After consulting the documentation, he confirmed that this operation indeed served to filter out null values, so line 26 was redundant and should be deleted.

In this case, CodeLin provides the following assistance:

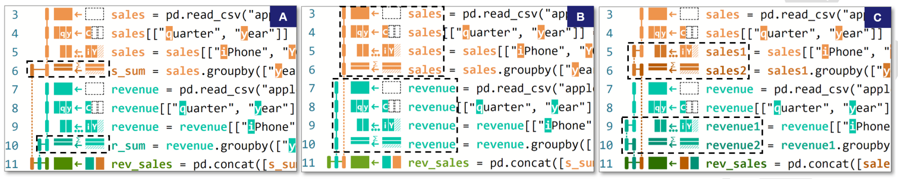
- It helps in understanding the pipeline within the code (G3) and unraveling the relationships between different code lines, which provides a context for modifying the code.
- It helps in comprehending the structure of the code (G4) and how it is organized based on its functionalities, which facilitates distinguishing code snippets and enables users to concentrate on key snippets.
- It helps in understanding the changes in tables caused by the code (G1), facilitating the identification of the key operations and the discovery of where the changes caused by the code deviates from expectations.

6.2. Demonstration: Code Patterns

Our work can help users understand the data transformation pipeline and code structure. Figure 1 shows a common pattern that the code is organized according to data transformation operation type. Another common pattern is organizing code by data lineage (Figure 8 A). Similar to Figure 8 A, the code snippets in Figure 8 B and C are organized based on data lineage, but there are differences. In Figure 8 B, each chain before line 11 involves only one variable. Although the data lineage becomes simpler, it is less intuitive to notice that *sales* in line 6 (and *revenue* in line 10) has gone through a critical transformation and has been transformed into the sum of sales. It becomes less intuitive to distinguish different stages claimed by variable names (in lines 6 and 10). In Figure 8 C, the code frequently utilizes short-lived variables, which may impact the readability of the code. To our knowledge, some automated models tend to generate data transformation code in this style.

6.3. Preliminary User Study

We conducted a preliminary user study to evaluate the effectiveness of our method. Unravel [12] and SOMNUS [11] are the most relevant works. We



```
3  sales = pd.read_csv("ap
4  sales[["quarter", "year"]]
5  sales = sales[["iPhone", "Y
6  s_sum = sales.groupby(["yea
7  revenue = pd.read_csv("appl
8  revenue[["quarter", "year"]
9  revenue = revenue[["iPhone"
10 r_sum = revenue.groupby(["y
11 rev_sales = pd.concat([s_su
```

```
3  sales = pd.read_csv("ap
4  sales[["quarter", "year"]]
5  sales = sales[["iPhone", "Y
6  sales = sales.groupby(["yea
7  revenue = pd.read_csv("appl
8  revenue[["quarter", "year"]
9  revenue = revenue[["iPhone"
10 revenue = revenue.groupby([
11 rev_sales = pd.concat([s_su
```

```
3  sales = pd.read_csv("ap
4  sales[["quarter", "year"]]
5  sales1 = sales[["iPhone", "
6  sales2 = sales1.groupby(["y
7  revenue = pd.read_csv("appl
8  revenue[["quarter", "year"]
9  revenue1 = revenue[["iPhone
10 revenue2 = revenue1.groupby
11 rev_sales = pd.concat([sale
```

Figure 8: Three different code patterns revealed by CodeLin organized. A shows a variant of Fig. 1 and its code is organized by data lineage. B and C are variants of A. In B, each chain before line 11 involves only one variable. In C, there are more short-lived variables.

use SOMNUS as the baseline since Unravel is limited to fluent code in R and cannot present data lineage, which differs from ours in scope. Our user study has two primary objectives: 1) to compare the advantages and disadvantages of two designs (independent views in SOMNUS and in situ visualization in CodeLin), and 2) to verify whether our design has achieved the proposed design goals. Considering the limitations of the study (small sample size and the lack of strict experimental control) as well as the unsurprising results, we focus on reporting the qualitative analysis based on our observations and the feedback from the users rather than conducting a quantitative analysis in the following subsection.

6.3.1. Participants and Design

Participants. We recruited 12 participants (8 males and 4 females, aged 22 to 31, and all reported normal or corrected-to-normal vision). Among them, 9 were graduate students and 1 was an undergraduate student, majoring in computer science, data science, or artificial intelligence. 2 were researchers from a university with research interests in computer science and data science. All participants reported having experience using Pandas for data processing (3 had less than 1 year of experience, 2 had 1-3 years of experience, 4 had 3-5 years of experience, and 3 had more than 5 years of experience). Each participant received 50 Chinese Yuan as a reward.

Tasks and Materials. We designed six tasks (Table 1) to compare the effectiveness of the two systems. To complete these tasks, users can utilize the interactions that are integrated within the code editor (e.g., *Ctrl + F* for finding strings and *double-clicking* for highlighting the same variable in different places). We disabled the interactions for tracing data lineage and variables in CodeLin since our focus is not on evaluating these interactions. We adopted a within-subject design using two datasets (D1 and D2), each containing a script with 9 lines of data transformation code and the corresponding source data. We restricted the number of lines since too many lines of code caused users to repeatedly zoom in and out of the view when using SOMNUS. We denote the two systems as C(odeLin) and S(OMNUS). In order to balance the effects of learning and sequence, the participants were divided into groups of four and each group covered the following conditions: [D1C, D2S], [D1S, D2C], [D2C, D1S], [D2S, D1C].

Procedure. We adopted a within-subjects design and the procedure of the study is as follows (A and B represent the two systems): [Training A] > [Task A] > [Break] > [Training B] > [Task B] > [Exploring Both] > [Questionnaire] > [Interview]. Prior to completing tasks on each system, we introduced the visual designs to the participants and provided several training tasks. Once they were ready, we initiated the formal tasks. We recorded the completion time (excluding that for reading instructions) when they completed the tasks. After the participants completed both sets of tasks, to facilitate the comparison of the two systems, we presented the same cases on both systems and allowed user for free exploration. Then, they were asked to rate the two systems via a questionnaire (Table 2), which used a seven-point Likert scale. Finally, we conducted interviews to collect their feedback. For each participant, the user study lasted approximately 40 minutes in total.

6.3.2. Result

Given that all participants were able to complete the tasks correctly, we report their completion times, which are shown in Figure 9. Overall, users completed tasks on CodeLin in less time than on SOMNUS. The ratings are presented in Figure 10. These ratings indicate that participants had a higher

Table 1: Tasks.

T1	Find the corresponding code line based on the given comment.
T2	Find the code lines that reduce/increase the number of rows/cols.
T3	Find the places where data processing is discontinuous in code (operations performed are not in the same chain before and after).
T4	In order to organize the code according to data provenance, which line of code should Line X be placed after?
T5	Find the lifecycle of each table variable.
T6	Find the code blocks that involve the same type of transformation operation in each line.

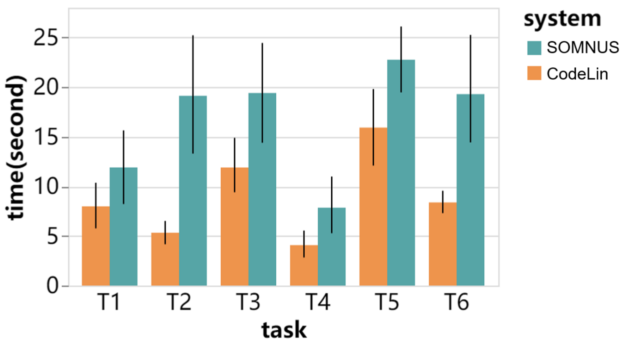


Figure 9: Average times of with 95% confidence intervals.

preference for CodeLin in terms of intuitiveness (Q1), ease of learning (Q3), the connection between visualization and code (Q6), and revealing the structure of the code (Q8).

By analyzing user behavior, we found that many participants (6 and 8 when using CodeLin and SOMNUS, respectively) relied on the search function when completing T1. This may suggest that the visualizations are not particularly helpful for locating code with specific semantics. To complete T3-5, participants frequently (an average of 7.33 times across all three tasks) relied on the *double-clicking* function of the code editor when using SOMNUS, whereas they rarely (an average of 0.50 times across all three tasks) utilized this function in CodeLin (note that we did not disable the functions of code editor in CodeLin). Considering that the time taken to complete these tasks using CodeLin was shorter,

Table 2: Questionnaire.

Q1	The visual design is intuitive.
Q2	The visual design is aesthetic.
Q3	The system is easy to learn.
Q4	The system is easy to use.
Q5	The visual representations well reveal the transformation semantics.
Q6	The visual representations and the code have a strong connection.
Q7	The system helps me figure out the transformation logic.
Q8	The system helps me figure out the code structure .

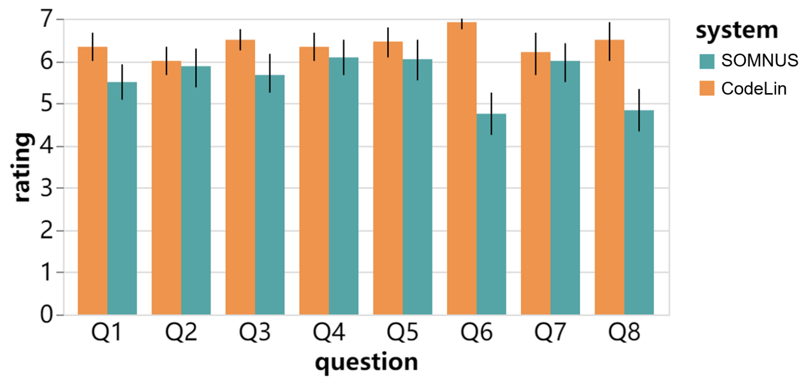


Figure 10: Average ratings with 95% confidence intervals.

it may suggest that our visualizations help users understand the data flow and the lifespan of tables, making users no longer need to rely on interactions with the editor.

6.3.3. Follow Up Study

During the process of conducting user study and analyzing results, we also generated several new questions, mainly on comparing our work with code editors, identifying the shortcomings of our work, and collecting their requirements and suggestions. We conducted a follow-up study to collect more feedback. We invited the participants and successfully found 8 of them (6 males and 2 females; 6 graduate students, 1 undergraduate student, and 1 researcher; we denote them as P1-8). We reintroduced them to the functions of CodeLin and let them freely explore based on our cases. Afterwards, we conducted interviews with them.

6.3.4. Feedback

Visual Design. In the initial interviews, 4 (out of 12) participants specifically mentioned that our visual design is simpler and easier to learn. In the follow-up study, all of the 8 participants mentioned that our visualization helps in viewing data transformation pipeline and conducting downstream code modification tasks. The frequently mentioned benefits of our method include: the intuitiveness of the data flow (6/8); the clear relationship between lines of code (6/8); highlighting of code blocks and data transformation operations (5/8). Based on the feedback, it appears that the participants were more satisfied with the role of visualization in explaining high-level information such as the data flow, relationships between code, and code structure, rather than its role in explaining low-level information such as a single line of code. We cite P4's viewpoint to illustrate the advantages of our visual design.

> *I've encountered the problem before. When taking over a script, not knowing where to start. To be honest, I really didn't want to go through it line by line, but there seemed no other choice... It is quite helpful to see an overall picture at the beginning. It (CodeLin) shows the general workflow intuitively. I can see how different parts are related to each other, then focus on some specific code and ignore the irrelevant ones. -P4*

Code-Visualization Connection. Our system differs significantly from SOMNUS in terms of how visualization and code are associated. Based on the ratings given by the participants for the connection between code and visualizations (Q6), CodeLin performs notably better in this regard. This may significantly influence the users' utilization of the visualization. In the initial interviews, only one participant mentioned that SOMNUS was useless for understanding code. The majority of the participants' complaints were focused on the poor association between code and visualization (4/12), difficulty in use (4/12), or high switching costs (3/12). We cited P6's comments to illustrate the impact of the connection between code and visualization on user experience.

> *The charts (of SOMNUS) seem disconnected from the code. When completing tasks, I need to look at both the charts and the code, but it's hard for me to combine the two. The other one (ours) is more user-friendly. It requires less effort. Even the colors (of the code and glyphs) correspond to each other. -P6*

Color. In the initial interviews, half (6/12) of the participants mentioned that our color scheme is better than that of SOMNUS. There are two color issues CodeLin generates: the distinctiveness (P1, P4, and P8) and its conflict with the default color scheme of code editor (P3 and P5). While the distinctiveness could be partially solved by using a more distinguishable color scheme, the color compatibility (adaption) remains an issue. In the follow-up study, we specifically inquired about this issue and asked the participants to judge our scheme. The majority of them (6/8) supported our color scheme. Among these participants, P1, P4, and P7 mentioned that the adaption of color schemes is not a notable issue; P2, P6, and P8 noted that taking its role in the visualization into account, our color scheme is better as an overall solution. On the opposite side, P3 and P5 believed that the convention of color scheme in code editors is important or the adaption is a big issue. Although they showed negative attitude, they also acknowledged the role of color in our method.

> *I'm not saying the color scheme is bad. It is unique, different from common code editors. I'm sure it has negative impact on reading code, since the convention is important... I appreciate the colors in the graph, and I understand the consistency (between code and visualization) is important. Therefore, I can't recommend using editor colors (in CodeLin). -P3*

Compatibility with Editor. In the initial interviews, 4 participants mentioned that our method could serve as a plugin or be integrated into an editor. In the follow-up study, we further collected participants' opinions on this issue. Through in-depth discussions with some participants, we also identified some existing problems. Except for the issue of color conflict, 3 participants (P2, P4, and P6) mentioned that our method seems to provide little help to write code. The matter also lies in the disparity between our focus (understanding unfamiliar code and making modifications) and the focus of a code editor (writing and debugging code). The following comments reveal this point:

> *The tool seems quite impressive, but I don't feel it offers much help for coding. I don't think presenting written code by a flowchart will aid me in writing subsequent code. As a code review tool, it seems acceptable. Its form of in-place visualization is quite good. However, as a plugin (for a code editor), I'm unclear what it could provide with coding. -P2*

7. Discussion

In this work, we propose an in situ visualization method for understanding data transformation scripts. Unlike previous methods where data transformation pipeline is presented in a separate view, we adopt an in situ visualization design to reduce the switching cost between code and visualization and enhance their connection. Through the evaluation, we have shown the usability and effectiveness of our proposed method. The design idea can be potentially applied to a wide range of tools.

In the process of iterative design of the system and conducting user experiments, we have also been inspired by our users. In this section, we will discuss the implications we have obtained and the limitations of current work.

7.1. Implications

Our research highlights the value of in situ program visualization in the context of data wrangling. Our experiment has provide evidence that in situ visualization (CodeLin) is more effectively utilized and preferred by users compared to independent view visualization (SOMNUS). It probably reduces the cost for switching context and associating code and visualization. We have shown that in situ program visualization for data wrangling is worth exploring.

This also implies that the effectiveness of program visualization may not solely depend on how the visual representations are designed, but rather on how they are presented to users and how they are integrated into original code environment. According to users' feedback, the differences in task performances and preference are probably due to the fact that in situ visualizations are easier to work with and require less cognitive cost for switching contexts. In visualization communication, there is a significant amount of work focused on revealing the complex processes or structures implemented by code (e.g., neural networks [48]), but there is seldom exploration into how to integrate visualization and code more closely. In addition to designing more powerful visualizations, better integrating visualization with code may also enhance the effectiveness of visualization methods, which is a direction worth exploring.

Another implication is that using in situ visualization to present high-level information is a promising direction. According to the users' feedback, one of the main advantages of in situ visualization is that it helps users understand the data flow within the code and how the code is organized. This high-level information is often not immediately apparent to users at a glance of a script; rather, it requires repeated reading and comprehension of the code. CodeLin can provide users with important overview information right from the start, helping them better accomplish downstream tasks (e.g., they can quickly identify the interested areas of code).

7.2. Limitations

Generalizability. Currently, our work supports scripts consisting of flat chains of data transformation code written in Python and R, and can be extended to similar programming languages (e.g., Scala). It does not support code with addition complexities (e.g., control flow and defined functions) and other programming language in very different form (SQL). Due to these limitations, our method is still far from a mature product. It is possible that the proposed method can serve as a modular solution that is integrated into other tools to display the data transformations performed by a piece of code and the logical connections between different parts of the code. However, we have not yet identified the requirements and challenges a in wider range of cases. We would like to conduct research on more real-world cases and explore how to support different programming languages and complex code structures in future.

Compatibility. In our research, we utilized color coding to represent data provenance. According to user feedback, this encoding is effective in enhancing the understanding of data lineage. However, it also has significant limitations. Besides the potential for confusion when encoding multiple categories, color encoding can be particularly challenging in environments like code editors, which already have their own color schemes. Since this overlap in color usage can lead to visual confusion, it results in the cost of compromising either the effectiveness of visualization or the habits that users have developed. Considering the importance of color coding, other in situ visualizations may also face this dilemma. Although we have not yet resolved this issue, we hope that future work can be inspired to develop better solutions.

Evaluation. While we believe our evaluations provide valuable insights into the effectiveness and usability of our proposed method, a statistically rigorous controlled study showing robust performance improvements is currently lacking. There is a need for further validation with a larger and more diverse user group in future studies. We have encountered two major challenges. First, recruiting a sufficient number of experienced data analysts is not an easy task. Second, high-level tasks require contextual information (e.g., that the code was used for processing Covid data), and participants' prior knowledge would significantly influence their performance. There are significant individual differences among data analysts, which presents considerable challenges to our experimental design. In the future, we hope to apply our methods in actual products and to obtain data on how real users use our methods.

Algorithms and Intelligence. For a comprehensive solution, we have implemented preliminary algorithms for coloring and generating lineage layout.

However, there is still large room for improvement. Users expressed a desire for more intelligent interaction methods for modifying code structure. Recent research [49, 50, 51, 52, 53, 54, 55] has shown that large language models (LLMs) can be applied to different scenarios to facilitate visual analytics. In terms of code understanding, LLMs can provide promising features, such as natural language explanations with no learning cost and specific answers to users' specific questions. Compared with LLMs, visualization has its own advantages, such as intuitiveness and information density. We believe that a promising idea is to combine in situ visualization with LLMs to help data analysts understand data transformation scripts. Though LLMs may help users in different tasks, designing features that transparently communicate code modifications and build user trust is still challenging. We plan to conduct deeper research into user needs and propose more intelligent methods to address these needs.

8. Conclusion

In this work, we explore the use of in situ program visualization for assisting data analysts in understanding data transformation scripts. We derive design goals and considerations and present a new visualization method that includes word-sized glyphs and a lineage graph which could be embedded in code editors. Our code pattern demonstrations, use case, and preliminary user study demonstrates the effectiveness and usability of our proposed method. Based on the result, we discuss how visualization can help users understand data transformation code. Overall, in situ program visualization can enhance the workflow of data analysts and facilitate their work on data transformation scripts.

In the future, we hope to enhance our approach so that it can support a wider range of languages and more complex code, while also being integrated as a plugin into code editors. Besides, we would like to explore the use of intelligent methods for adjusting code structure and the combination of code comments with visualizations.

References

[1] S. Liu, D. Weng, Y. Tian, Z. Deng, H. Xu, X. Zhu, H. Yin, X. Zhan, Y. Wu, Ecoalvis: visual analysis of control strategies in coal-fired power plants, IEEE TVCG 29 (1) (2022) 1091–1101.

[2] J. Wang, K. Kucher, A. Kerren, Visual analysis of power plant data for european countries, in: EuroVis, Eurographics, 2024.

[3] G. Moreira, M. Hosseini, M. N. A. Nipu, M. Lage, N. Ferreira, F. Miranda, The urban toolkit: A grammar-based framework for urban visual analytics, IEEE TVCG.

[4] J. Chen, Q. Huang, C. Wang, C. Li, Sensemap: Urban performance visualization and analytics via semantic textual similarity, IEEE TVCG.

[5] Z. Deng, D. Weng, Y. Wu, You are experienced: Interactive tour planning with crowdsourcing tour data from web, Journal of Visualization 26 (2) (2023) 385–401.

[6] Y. Fu, J. Stasko, Hoopinsight: Analyzing and comparing basketball shooting performance through visualization, IEEE TVCG.

[7] L. Cheng, H. Jia, L. Yu, Y. Wu, S. Ye, D. Deng, H. Zhang, X. Xie, Y. Wu, Viscourt: In-situ guidance for interactive tactic training in mixed reality, in: Proc. of Annual ACM Symposium on UIST, 2024, pp. 1–14.

[8] M. Khan, L. Xu, A. Nandi, J. M. Hellerstein, Data tweening: incremental visualization of data transforms, Proc. of the VLDB Endowment 10 (6) (2017) 661–672.

[9] C. Niederer, H. Stitz, R. Hourieh, F. Grassinger, W. Aigner, M. Streit, TACO: visualizing changes in tables over time, IEEE TVCG 24 (1) (2018) 677–686.

[10] X. Pu, S. Kross, J. M. Hofman, D. G. Goldstein, Datamations: Animated explanations of data analysis pipelines, in: Proc. of the SIGCHI, 2021, pp. 1–14.

[11] K. Xiong, S. Fu, G. Ding, Z. Luo, R. Yu, W. Chen, H. Bao, Y. Wu, Visualizing the scripts of data wrangling with SOMNUS, IEEE TVCG (2022) 1–1.

[12] N. Shrestha, T. Barik, C. Parnin, Unravel: A fluent code explorer for data wrangling, in: Proc. of the ACM Symposium on UIST, 2021, pp. 198–207.

[13] J. Hoffswell, A. Satyanarayan, J. Heer, Augmenting code with in situ visualizations to aid program understanding, in: Proc. of the SIGCHI, 2018, pp. 1–12.

[14] M. Harward, W. Irwin, N. Churcher, In situ software visualisation, in: Proc. of Australian Software Engineering Conference, 2010, pp. 171–180.

[15] S. Liu, G. Andrienko, Y. Wu, N. Cao, L. Jiang, C. Shi, Y.-S. Wang, S. Hong, Steering data quality with visual analytics: The complexity challenge, Visual Informatics 2 (4) (2018) 191–197.

[16] K. Kagkelidis, I. Dimitriadis, A. Vakali, Lumina: an adaptive, automated and extensible prototype for exploring, enriching and visualizing data, Journal of Visualization 24 (2021) 631–655.

[17] S. Kandel, A. Paepcke, J. Hellerstein, J. Heer, Wrangler: Interactive visual specification of data transformation scripts, in: Proceedings of the SIGCHI, 2011, pp. 3363–3372.

[18] Excel, `https://www.microsoft.com/en-sg/microsoft-365/excel`.

[19] Google, `https://www.google.com/sheets/about/`.

[20] Y. Chen, Y. Zhao, X. Li, J. Zhang, J. Long, F. Zhou, An open dataset of data lineage graphs for data governance research, Visual Informatics.

[21] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, H. T. Vo, Vistrails: visualization meets data management, in: Proc. of the ACM SIGMOD, 2006, pp. 745–747.

[22] H. Stitz, S. Luger, M. Streit, N. Gehlenborg, Avocado: Visualization of workflow-derived data provenance for reproducible biomedical research, in: Computer Graphics Forum, Vol. 35, Wiley Online Library, 2016, pp. 481–490.

[23] R. Hoekstra, P. Groth, Prov-o-viz-understanding the role of activities in provenance, in: International Provenance and Annotation Workshop, Springer, 2014, pp. 215–220.

[24] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, H. Pfister, Evaluation of filesystem provenance visualization tools, IEEE TVCG 19 (12) (2013) 2476–2485.

[25] P. Alvaro, J. Rosen, J. M. Hellerstein, Lineage-driven fault injection, in: Proc. of the ACM SIGMOD, 2015, pp. 331–346.

[26] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, W. Chen, A systematic literature review of modern software visualization, Journal of Visualization 23 (2020) 539–558.

[27] G.-C. Roman, K. C. Cox, Program visualization: The art of mapping programs to pictures, in: Proc. of the International Conference on Software Engineering, 1992, pp. 412–420.

[28] J. Sundararaman, G. Back, Hdpv: Interactive, faithful, in-vivo runtime state visualization for c/c++ and java, in: Proc. of the ACM Symposium on Software Visualization, 2008, pp. 47–56.

[29] R. Faust, K. Isaacs, W. Z. Bernstein, M. Sharp, C. Scheidegger, Anteater: Interactive visualization of program execution values in context, arXiv preprint arXiv:1907.02872.

[30] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, S. Z. Guyer, Heapviz: Interactive heap visualization for program understanding and debugging, in: Proc. of the International Symposium on Software Visualization, 2010, pp. 53–62.

[31] R. Schulz, F. Beck, J. W. C. Felipez, A. Bergel, Visually exploring bbject mutation, in: IEEE Working Conference on Software Visualization, IEEE, 2016, pp. 21–25.

[32] T. Ohmann, R. Stanley, I. Beschastnikh, Y. Brun, Visually reasoning about system and resource behavior, in: Proc. of the International Conference on Software Engineering Companion, 2016, pp. 601–604.

[33] P. Gestwicki, B. Jayaraman, Methodology and architecture of jive, in: Proceedings of the ACM Symposium on Software Visualization, 2005, pp. 95–104.

[34] B. Swift, A. Sorensen, H. Gardner, J. Hosking, Visual code annotations for cyberphysical programming, in: Proc. of International Workshop on Live Programming, 2013, pp. 27–30.

[35] F. Beck, F. Hollerich, S. Diehl, D. Weiskopf, Visual monitoring of numeric variables embedded in source code, in: Proc. of IEEE Working Conference on Software Visualization, 2013, pp. 1–4.

[36] F. Beck, O. Moseler, S. Diehl, G. D. Rey, In situ understanding of performance bottlenecks through visually augmented code, in: Proc. of International Conference on Program Comprehension, 2013, pp. 63–72.

[37] GitKraken, GitLens, `https://www.gitkraken.com/gitlens`, accessed: Mar 21, 2023.

[38] P. Vaithilingam, T. Zhang, E. L. Glassman, Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models, in: SIGCHI Extended Abstracts, 2022, pp. 1–7.

[39] S. Kasica, C. Berret, T. Munzner, Table Scraps: An actionable framework for multi-table data wrangling from an artifact study of computational journalism, IEEE TVCG 27 (2) (2021) 957–966.

[40] Flask, `http://flask.pocoo.org/`.

[41] Vue.js, `https://vuejs.org/`.

[42] Pandas, `https://pandas.pydata.org/`.

[43] Monaco Editor, `https://microsoft.github.io/monaco-editor/`.

[44] M. Bostock, V. Ogievetsky, J. Heer, D^3 data-driven documents, IEEE TVCG 17 (12) (2011) 2301–2309.

[45] M. Tennekes, E. de Jonge, Tree colors: color schemes for tree-structured data, IEEE TVCG 20 (12) (2014) 2072–2081.

[46] K. Sugiyama, S. Tagawa, M. Toda, Methods for visual understanding of hierarchical system structures, IEEE Transactions on Systems, Man, and Cybernetics 11 (2) (1981) 109–125.

[47] W. Gatterbauer, C. Dunne, H. Jagadish, M. Riedewald, Principles of query visualization, arXiv preprint arXiv:2208.01613.

[48] N. Chotisarn, S. Gulyanon, T. Zhang, W. Chen, VISHIEN-MAAT: Scrollytelling visualization design for explaining siamese neural network concept to non-technical users, Visual Informatics 7 (1).

[49] Y. Ye, J. Hao, Y. Hou, Z. Wang, S. Xiao, Y. Luo, W. Zeng, Generative AI for Visualization: State of the art and future directions, Visual Informatics.

[50] P. Soni, C. de Runz, F. Bouali, G. Venturini, A survey on automatic dashboard recommendation systems, Visual Informatics.

[51] Y. Tian, W. Cui, D. Deng, X. Yi, Y. Yang, H. Zhang, Y. Wu, ChartGPT: Leveraging LLMs to generate charts from abstract natural language, IEEE TVCG.

[52] L. Shen, Y. Zhang, H. Zhang, Y. Wang, Data Player: Automatic generation of data videos with narration-animation interplay, IEEE TVCG.

[53] Y. Zhao, Y. Zhang, Y. Zhang, X. Zhao, J. Wang, Z. Shao, C. Turkay, S. Chen, LEVA: Using large language models to enhance visual analytics, IEEE TVCG.

[54] L. Cheng, D. Deng, X. Xie, R. Qiu, M. Xu, Y. Wu, SNIL: Generating sports news from insights with large language models, IEEE TVCG.

[55] J. Wang, X. Li, C. Li, D. Peng, A. Z. Wang, Y. Gu, X. Lai, H. Zhang, X. Xu, X. Dong, et al., Ava: An automated and ai-driven intelligent visual analytics framework, Visual Informatics 8 (2) (2024) 106–114.

Ethical Approval

Informed consent was obtained from all participants prior to their involvement in the study. Signed consent forms are on file and available upon request.

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for *[Journal name]* and was not involved in the editorial review or the decision to publish this article.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Yingcai Wu is an Editorial Board Member for Visual Informatics and was not involved in the editorial review or the decision to publish this article.