

Xavier: Toward Better Coding Assistance in Authoring Tabular Data Wrangling Scripts

Yunfan Zhou
State Key Lab of CAD&CG,
Zhejiang University
Hangzhou, Zhejiang, China
yunfzhou@zju.edu.cn

Xiwen Cai
State Key Lab of CAD&CG,
Zhejiang University
Hangzhou, Zhejiang, China
xwcai@zju.edu.cn

Qiming Shi
State Key Lab of CAD&CG,
Zhejiang University
Hangzhou, Zhejiang, China
qimingshi@zju.edu.cn

Yanwei Huang
State Key Lab of CAD&CG,
Zhejiang University
Hangzhou, Zhejiang, China
huangyw@zju.edu.cn

Haotian Li*
Microsoft Research Asia
Beijing, China
haotian.li@microsoft.com

Huamin Qu
The Hong Kong University of Science
and Technology
Hong Kong SAR, China
huamin@cse.ust.hk

Di Weng[†]
School of Software Technology,
Zhejiang University
Ningbo, Zhejiang, China
dweng@zju.edu.cn

Yingcai Wu
State Key Lab of CAD&CG,
Zhejiang University
Hangzhou, Zhejiang, China
ycwu@zju.edu.cn

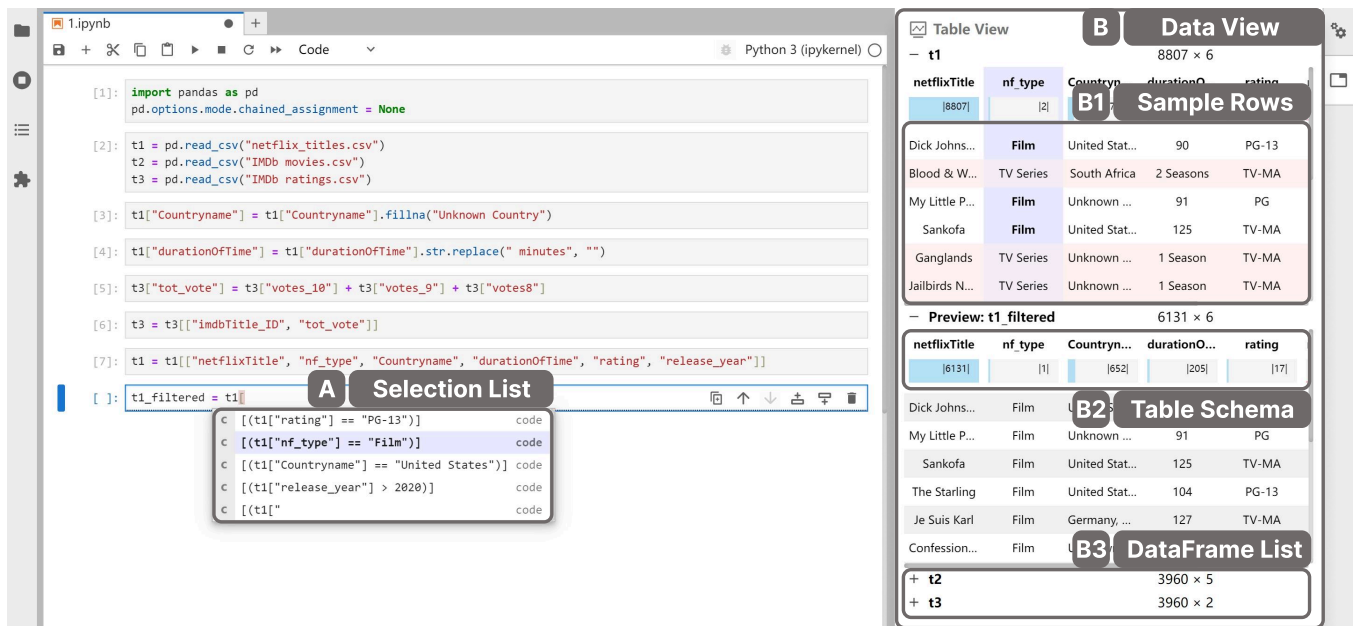


Figure 1: The interface of *Xavier* within the computational notebook. A) The selection list providing data context-aware code completions for users while typing. B) The data view displaying data contexts for user reference. During typing, *Xavier* automatically expands items by showing the table schema (B2) and sample rows (B1), highlighting the most relevant data based on user's code. Other irrelevant data contexts are listed in the data view and folded (B3).

ABSTRACT

Data analysts frequently employ code completion tools in writing custom scripts to tackle complex tabular data wrangling tasks.

However, existing tools do not sufficiently link the data contexts such as schemas and values with the code being edited. This not only leads to poor code suggestions, but also frequent interruptions in coding processes as users need additional code to locate and understand relevant data. We introduce *Xavier*, a tool designed to

*The work was done when Haotian Li was at HKUST.

[†]Di Weng is the corresponding author.

enhance data wrangling script authoring in computational notebooks. *Xavier* maintains users' awareness of data contexts while providing data-aware code suggestions. It automatically highlights the most relevant data based on the user's code, integrates both code and data contexts for more accurate suggestions, and instantly previews data transformation results for easy verification. To evaluate the effectiveness and usability of *Xavier*, we conducted a user study with 16 data analysts, showing its potential to streamline data wrangling scripts authoring.

CCS CONCEPTS

• **Human-centered computing** → **User interface programming**.

KEYWORDS

Interactive data wrangling, coding assistance

1 INTRODUCTION

Data wrangling is a crucial process in data science that involves tasks such as data cleaning, integration and format transformation for downstream analysis [31]. To perform complex tabular data wrangling tasks, data analysts with programming skills often write ad-hoc scripts. Given the tedious and time-consuming nature of scripting, the emergence of various code completion tools, particularly AI-powered ones including IntelliSense [53] and GitHub Copilot [18], has significantly enhanced coding efficiency in programming environments.

Despite their widespread application, AI-powered code completion tools primarily focus on code-related features (e.g., grammar and semantics), or *code contexts*, in the completion process. These tools often overlook the metadata of datasets people are working with, or *data contexts*, leading to issues when generating code. For instance, since the underlying Large Language Models (LLMs) may memorize training examples [5], AI-powered code completion tools often generate wrong columns or unreasonable values [80]. Some AI-powered code completion tools, like GitHub Copilot which has access to original data files in the project [66], consider datasets to a certain extent. However, they do not sufficiently link the metadata (e.g. table schemas and unique values of columns) to the specific code being edited. Therefore, these completion tools often fail to generate recommendations that require a strong understanding of datasets, such as correct column names to join two tables, or the prefix of a data value to be removed.

The limitations of existing tools motivate us to propose a novel coding assistance approach that links data contexts to the code being edited. However, developing such an approach presents several challenges according to our preliminary study. First, users are often frustrated at mistakes of AI-generated code and have to manually specify dataset information for intelligent code completion. However, dynamically linking data contexts to reduce the effort of data context specification remains unexplored. Additionally, locating specific data segments that are relevant and useful for users is challenging due to the dynamic nature of user focus during editing the incomplete code. Moreover, users have to spend considerable time validating the transformation result of AI-generated code [12, 13].

Existing live programming tools like Projection Boxes [41] can display information of variables (e.g. runtime values) in a real-time way as code is being edited, helping users validate the code to some extent. However, they do not link the most relevant information in the DataFrame variable to the code being edited, since users often focus on a small portion of information in the variable while editing data wrangling code. To tackle these challenges, we use Pandas [59] as an exemplary data transformation library, delineating a strategy combining code contexts and data contexts to provide data context-aware code completions. Additionally, we propose a method to dynamically link code and data, highlight the relevant data based on user's code, and allow instant preview of data transformation results for straightforward verification. Finally, an user interface is designed, leveraging visualizations and interactions to facilitate user's data sensemaking in the scripting process. On top of this, we take the computational notebook as an exemplary programming environment that facilitates trying and iterating data wrangling code [67] and introduce *Xavier*, a coding assistance extension for computational notebooks. To evaluate *Xavier*'s usability and effectiveness, we conducted a counterbalanced mixed-design user study on 16 participants, in which participants wrote data wrangling scripts to complete a task with *Xavier* and another task with a baseline tool. We find that users experienced significantly fewer context switches and errors during scripting by using *Xavier*. User feedback further confirms that *Xavier* helps users maintain data context awareness and author code more smoothly throughout the data wrangling scripting process. The major contributions are summarized as follows:

- A preliminary study that reveals user behavior patterns in authoring tabular data wrangling scripts and summarizes user requirements of AI-powered coding assistants.
- A computational notebook extension, *Xavier*, which aids users in maintaining awareness of data contexts during data wrangling code authoring.

2 RELATED WORK

In this section, we review research on data wrangling tools, coding assistants in computational notebooks, code completion tools and live programming tools.

2.1 Data Wrangling Tools

Data wrangling is a challenging task that involves change, rearrangement, and merging of data to prepare for various purposes such as visualization and analysis [31]. Several toolkits like Pandas [59] in Python, as well as dplyr [56] and tidyr [79] in R, have been developed to facilitate this process. These toolkits offer data analysts with significant flexibility for effective data manipulation. Nevertheless, analysts unfamiliar with Python or R may find it time-consuming and demanding to learn a new toolkit. To address these challenges, numerous interactive tools have been proposed to lower the barriers associated with data wrangling.

Interactive tools for tabular data wrangling can be categorized as either *imperative* or *declarative*. *Imperative* approaches [20, 31, 62] focus on the wrangling procedures and typically provide users with a menu of various wrangling operations. In contrast, *declarative* approaches emphasize the specification of transformation

outcomes. For instance, many systems allow users to compose example target tables and automatically synthesize transformation programs [19, 30, 75]. Other tools enable users to specify the intended tasks through declarative mappings [7] or natural language [14, 28]. While these interactive tools help democratize the wrangling process, they are primarily designed for users with limited experience in data wrangling and programming. Writing custom wrangling scripts remains common among data analysts, as coding offers a more flexible and expressive way to specify the wrangling process [55, 83]. Hence, *Xavier* offers coding support for data analysts proficient in programming.

Additionally, earlier studies have focused on code recommendation for data wrangling, which can be categorized into three types according to the smallest granularity of the output code: *block-level*, *line-level* and *token-level*. *Block-level* code recommendation tools [3, 9, 35] generate blocks of data wrangling code that often include multiple transformation operations. For instance, AutoPandas [3] leverages program synthesis techniques to generate code from input-output examples. Stepwise [35] and Phasewise [35] decompose the entire data wrangling problem into steps or phases to facilitate steering and verification of AI-generated code. *Line-level* code recommendation tools like Auto-Suggest [81] focus on generating single-operation data wrangling code. *Token-level* code recommendation tools like SnipSuggest [37] support suggesting next few tokens such as table names or column names as users write part of the code.

Although *token-level* code recommendation tools basically incorporate data contexts into their recommendation models, users still face challenges in understanding and verifying the recommendations. In contrast, *Xavier* not only links relevant data contexts to the code being edited in its recommendation model, but also offers corresponding highlighting and previews to provide on-the-fly explanations for its recommendations.

2.2 Code Assistants in Computational Notebooks

A vast range of code assistants have been proposed for integration into computational notebooks, aiming to enhance the productivity of data workers in programming tasks. These assistants can be categorized based on stages of data analysis in which they provide recommendations: *exploration* (i.e. during the data exploration stage), *next-step* (i.e. suggesting the next steps before typing), or *typing* (i.e. offering assistance during typing).

Exploration code assistants [44, 61] help data workers discover workflows or analysis techniques during data exploration. For example, EDAssistant [44] aids users by finding code snippets similar to those in the current notebook through in-situ code search, thereby inspiring the workflows in their analysis. *Next-step* code assistants [8, 11, 29, 36, 52] specialize in recommending next steps before users start typing, guiding users to continue their workflows effectively. For instance, Wrex [11] provides spreadsheet-like interfaces for data manipulation by example. Jigsaw [29] leverages large language models to enable multi-modal inputs for data wrangling script generation. BISCUIT [8] scaffolds users understanding and refining the LLM-generated code by introducing ephemeral UIs. *Typing* code assistants [10, 18] offer real-time recommendations

while users are typing. For example, Glinda [10] combines live programming, GUI elements, and a Domain-Specific Language (DSL) to provide immediate feedback during programming.

However, few code assistants focus on real-time recommendations during *typing*. Although Glinda [10] incorporates a code completion feature, it is primarily designed for the rapid construction of DSL structures based on language grammar. GitHub Copilot [18] does not sufficiently link the data contexts to the specific code being edited, which will be discussed in Section 2.3. In contrast, *Xavier* focuses on code completions for data wrangling scripts, allowing users to remain aware of data contexts during typing.

2.3 Code Completion Tools

Code completion, which suggests candidate subsequent tokens for programmers, is a widely used feature in code editors that accelerates the programming process. It has been extensively investigated in the literature [46]. Early code completion approaches relied on heuristic rules, such as static type information [21, 27, 60], similar patterns from codebases [4, 24, 57], or usage frequency [40, 58, 65]. Building on the analogy between human-written code and natural language [25], subsequent works [23, 63, 64] explored statistical approaches that leveraged the repetitive and predictive properties of code. With the advancement of deep neural networks, language models based on neural networks, such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM), have also been investigated [33, 43, 71] to improve the suggestion accuracy.

More recently, as Generative Pre-trained Transformers (GPT) gained popularity in natural language processing, numerous GPT-based models and tools have emerged [6, 47, 70]. For example, fine-tuned on a large corpus of publicly available code on GitHub, Codex [6] demonstrated improved accuracy in code completion. Its production version, GitHub Copilot [18], is capable of providing longer code completions, ranging from individual tokens to entire functions, while maintaining a relatively high level of accuracy.

These approaches optimize the exploration and utilization of deeper semantic information in code, primarily supporting multilingual code completion and general tasks. Nevertheless, data contexts such as schemas and values are also crucial for enhancing accuracy in data wrangling code authoring. Besides, most code completion tools focus on coding assistance during code authoring, which can cause users to lose data contexts before or after code authoring, making it difficult to locate relevant data and verify code. Therefore, *Xavier* incorporates both code contexts and data contexts for more intelligent code completions, and provides assistance for awareness of data contexts throughout the coding process.

2.4 Live Programming Tools

Live programming provides real-time feedback (e.g. visualization of a program's runtime data) as code is edited [13], which enhances understanding of how code changes affect the running system [32]. General tools like Projection Boxes [41] and Engraft [26] offer configurable frameworks for live programming. Other tools focus on specific programming tasks and can be categorized into three types: *generation*, *debugging*, and *validation*.

Generation tools [15, 16] allow users to modify runtime values to synthesize code. For instance, SnipPy [16] supports small-step

program synthesis by changing the displayed runtime values. *Debugging* tools [32, 84] aid in program debugging by visualizing runtime values. For example, OmniCode [32] displays the entire history of all run-time values for all program variables all the time when users are writing Python code. *Validation* tools [13, 82] enhance validation of AI-generated code by providing continuous explanations adjacent to the code. For instance, Ivie [82] breaks up complex code into pieces and annotates them with textual explanations. These prior works have made efforts to improve code understanding in programming, inspiring the design of *Xavier*. However, when users are editing code to apply a new transformation on datasets, they typically focus on a relatively small portion of information in variables. For instance, while applying a string format transformation like `df["A"] = df["A"].str.replace(":", "")` on the variable "df" with many columns, users typically focus merely on the value format of the column "A" in the variable "df". Although existing *validation* tools can effectively display the necessary information of variable states (e.g. all elements in an array), they generally fail to link such a small portion of relevant information required to validate the current code that users are authoring. Our work offers a nuanced inspection by highlighting the relevant data contexts and showing instant previews, which further enhances live programming and code understanding.

3 PRELIMINARY STUDY

Before designing an AI-powered code assistant for data wrangling scripts, we would like to understand the requirements and challenges the users may meet. Prior work [2, 45, 54, 69, 73] has explored patterns of user interactions with AI-powered code assistants in general coding tasks and identified various challenges, such as code understanding, verification, and context switching. These efforts have pointed out the direction for improving AI-powered code assistants and have inspired our research. Although the findings from the existing work can be generalized to various domains, they do not fully cover the problems users encounter when authoring data wrangling scripts. Unlike tasks with clear objectives such as front-end and back-end development or web scraping, data wrangling tasks often involve a higher degree of uncertainty. Users need to fully explore and understand the characteristics of the dataset [48] and complete data transformation tasks iteratively through interacting with data [12]. In order to understand the needs and challenges users face when using AI-powered code assistants to complete data wrangling tasks, we conducted a preliminary study¹ where participants took part in: 1) an experiment, in which they were asked to complete data wrangling tasks using GitHub Copilot [18], and 2) a semi-structured interview, in which they shared the basic workflow of data wrangling and the issues encountered in the previous experiment. We identified common patterns in users' behavior and summarized requirements according to the interview.

3.1 Participants

We recruited 9 data analysts (denoted as P1-P9, 5 male and 4 female, $Age_{mean} = 26.44$, $Age_{std} = 5.88$, $Experience_{mean} = 5.00$ years, $Experience_{std} = 2.55$ years) by sending invitations via social media and word-of-mouth. They had diverse backgrounds such as

Control Engineering, Mathematics, Digital Media Technology, Data Governance, Geographic Information Systems and so on. They regularly programmed with Python Pandas in computational notebooks for data wrangling in their projects (at least once a week). Their demographic information is detailed in supplemental materials. Participants consented to having their voices and programming processes recorded.

3.2 Apparatus and Materials

Apparatus. We chose GitHub Copilot [18], a representative of AI-assisted code assistant, in our preliminary study. Copilot is the first code assistant based on Large Language Models (LLMs) to reach widespread usage and has been studied by a wide range of literature [2, 13, 50]. Participants authored data wrangling code in computational notebooks on standardized desktop devices. A slide with task descriptions, output examples and data dictionaries was prepared for users to refer to in the code authoring experiment.

Datasets. We selected *Covid-19* dataset including 3 tables and 20 columns based on a publicly available notebook². The dataset was chosen for its public familiarity [68] and coverage of two major types of tabular data, namely categorical and numerical. Data tables were slightly modified such as changes of column names and categorical values to ensure Copilot had not been trained on them.

Tasks. In our preliminary study, participants were asked to author a data wrangling script to calculate the ratio of confirmed cases and death cases for each country. They needed to complete approximately 20 transformation operations, involving common operations such as joining, sorting, and filtering, to produce an output table that includes 7 columns. Details of task description are left to the supplementary materials.

3.3 Procedure

We first informed participants about relevant information regarding the study, including the purpose, overall procedure, and compensation of the study, and then sought participants' consent to data collection. Then, the participants would take part in a code authoring experiment (60-75 minutes) and a semi-structured interview (15 minutes). We captured participants' behavior in coding process through video recordings and collected feedback in the interview by audio recordings for subsequent analysis and summarization. The entire study took around 90 minutes and each participant received 70 Chinese Yuan as compensation.

Code authoring experiment. Initially, we briefly introduced GitHub Copilot and allowed participants to try it on a warm-up task, in order to ensure participants grasped its usage. Participants were then given the task description, dataset and data dictionary to write a script in a computational notebook with Copilot's assistance. Participants were free to consult the data dictionary or view raw data tables as needed. Following the think-aloud protocol, participants were encouraged to share their thoughts on Copilot during the experiment.

Semi-structured interview. The semi-structured interview consisted of three parts. In the initial part, we inquired of participants regarding their basic workflow of data wrangling in their data analysis work. In the second part, participants were encouraged to

¹The study has been approved by State Key Lab of CAD&CG, Zhejiang University.

²<https://www.kaggle.com/code/erikbruin/storytelling-covid-19>

share the issues they encountered in the experiment, including the difficulties in completing the task and the pain points of interacting with Copilot. In the last part, we specifically asked participants about the issues we observed in the experiment.

3.4 Findings

We manually captured participants' behaviors in the code authoring experiment through video recordings. To analyze the user feedback from the semi-structured interview, we conducted a qualitative inductive content analysis [38]. Initially, two co-authors (data analysis experience: ≥ 3 years) read the recording transcripts to extract the comments related to issues and difficulties individually. Then, they grouped the similar comments and identified the challenges collaboratively. Finally, all co-authors discussed the user behavior and challenges to derive the key findings. In this subsection, we first introduce the definition of two kinds of participants' activities observed in the code authoring experiment. Then we summarize four findings according to user activities and feedback.

3.4.1 Definition of Two Kinds of Activities. We categorized the participants' behavior into two kinds of activities: **Data Inspection (DI)** and **Code Authoring (CA)**. **DI** involves examining datasets, which can be further divided into two types: profiling (**DI_1**) and verifying (**DI_2**). **DI_1** is focused on understanding datasets and gaining inspirations for subsequent wrangling steps, which can manifest in multiple ways such as reading parts of the data table. **DI_2** is about verifying the effectiveness and correctness of transformations, typically manifested as viewing the result table after running the transformation code. **CA** involves writing code, which can also be divided into creating (**CA_1**) and modifying (**CA_2**). **CA_1** is related to applying new transformations or profiling datasets, while **CA_2** involves minor code modifications, such as adjusting function parameters, often seen in debugging. We measured duration of activities for each participant and scaled the duration of activities by normalizing each participant's total time spent on code authoring. The activity timelines of each participant during code authoring experiment are shown in Figure 2.

3.4.2 User Behavior and Challenges. According to the user behavior and feedback, we summarized four findings:

Users frequently switched between code contexts and data contexts, commonly fixing coding mistakes by checking data. According to Figure 2 and our observations in the code authoring experiment, participants frequently switched between **DI_1** and **CA_1** or between **DI_2** and **CA_1**, indicating frequent context switches during the coding process. For instance, P1 encountered a relatively difficult subtask in the second half of code authoring experiment, and he needed to write code (**CA_1**) and review the data (**DI_1**) iteratively to adjust his coding approach. Ideally, users who author data wrangling scripts experience an iterative process involving profiling, creating, and verifying steps, which typically follow the cycle: **DI_1**, **CA_1**, **DI_2**, (**DI_1**), **CA_1**, **DI_2**,... However, the interleaving of **CA_2** and **DI_2** is common in Figure 2, indicating that participants often encountered errors and typically fixed mistakes by checking the result table. As a representative example, P6 encountered an intractable bug in her script in the last third of

the experiment, having to repeatedly modify the code (**CA_2**) and examine the result (**DI_2**).

Users complained about frequent mistakes brought by Copilot's code completion. When writing partial code and letting Copilot complete it, doubts arose about its data knowledge. In Section 3.2, all column names were changed (e.g. "ProvinceOrState" instead of "Province/State"), yet Copilot often returned incorrect completions like "Province/State". Such mistakes caused "non-exist column" errors which frustrated participants. "Why did Copilot still complete 'Province/State' even if I explicitly mentioned the 'ProvinceOrState' column in previous code?", P8 questioned. When preferring to write comments first for Copilot to generate code, participants complained about lengthy prompts. P2, P4 and P9 admitted their detailed dataset information in comments was mainly for Copilot, which echoed previous research findings [2]. P2 doubted Copilot would generate correct code without specifying operations and operators. Nearly all participants suggested Copilot should access data tables so as to correctly complete column names and values. *Users faced difficulties searching for the part of the data to which the code completion tool was referring.* Before applying a new transformation operation, Copilot recommended the next possible line of code for participants. However, participants usually had no idea about the part of the data Copilot was focusing on and they were not sure about the relevance between the next-step recommendation and their current wrangling subtasks. Hence, they tended to reject the recommendation and instead used various methods to search datasets. For example, before unifying country names, participants wrote additional code to filter relevant names (P1, P4 and P6) or opened original CSV files (P2, P3, P5, P7 and P8) to search globally (**DI_1**). However, they found it time-consuming to search, as P2 commented, "I was overwhelming when searching for 'United States' in the raw data file with so much irrelevant and interfering information". P5 noted, "It was troublesome writing code just to see different spellings of United States, such as 'US' and 'USA'." Interruptions in writing often occurred to observe intermediate variables during transformations. For example, P1 printed columns of tables `covid_19_data` and `country_codes` to recall the exact spelling of key names while merging the two tables.

Users spent considerable time reviewing the transformation result of AI-generated code. Due to frequent "non-exist column" errors, participants remained skeptical especially about constants like column names or data values, requiring careful verification against data. However, reviewing was sometimes inconvenient even when running code. For instance, P6 spent about 10 minutes fixing a line due to unawareness of a column's data type and a hidden spelling error. "I expected cardinality changes of the column after the string replacing transformation, but needed extra code to verify the prediction." P1's comment indicated the challenge in tracking data changes, as reported in [12, 74].

3.5 User Requirements

We identified three requirements for a code completion tool in authoring data wrangling scripts according to findings. In the remaining part of the paper, **R1-R3** refers to the requirements. The requirements are:



Figure 2: The timelines of observed activities of each participant during the code authoring experiment. DI and CA refer to Data Inspection and Code Authoring, respectively. Users may profile data (DI_1), verify results (DI_2), create new transformations (CA_1) or modify the written code (CA_2) during scripting. To facilitate comparison, we scaled the duration of activities by normalizing each participant’s total time spent on code authoring.

R1. Incorporating data contexts for intelligent code completion. In our code authoring experiment, users were frustrated at mistakes (e.g. non-existent columns) made by Copilot and had to write lengthy prompts to transfer the dataset information to Copilot. However, general code completion tools like Copilot typically lack data contexts, which limits the intelligence of suggestions. Thus, automatic data context provision can help alleviate the need to manually convey data contexts to the code completion tool.

R2. Assisting users in locating relevant parts of datasets. In our code authoring experiment, participants struggled to search for relevant data to be wrangled and they had to frequently print tables or refer to original CSV files, which caused a significant context-switch overhead. Therefore, the tool should offer a straightforward way to help users search for and focus on relevant data before they start writing code to apply a new transformation.

R3. Offering straightforward code verification. In our code authoring experiment, participants were confused by how Copilot suggested code completion and took much effort in verifying correctness of Copilot’s suggestions, which also lowered the efficiency of code authoring. Thus, the tool should also offer simple verification approaches to assist users in validating AI-generated code and enhance their trust.

4 XAVIER

In this section, we first present an overview of *Xavier* (Section 4.1). This is followed by a detailed discussion of the design of *Xavier*, which features data context-aware code completion (Section 4.2), automatic data context highlighting (Section 4.3), and real-time transformation preview (Section 4.4). Usage scenarios in Section 4.2, 4.3 and 4.4 are highlighted with a blue background.

4.1 Overview

Xavier is a coding assistance tool enabling users to stay aware of data contexts while authoring data wrangling scripts. Integrated as an extension for computational notebooks, it supports Python Pandas [59], with the potential for generalization to other data transformation libraries.

Figure 1 illustrates *Xavier* within the notebook interface. It consists of two components: the notebook interface and an always-on data view visualizing data contexts. In parallel to users’ writing scripts in the notebook, a selection list (Figure 1 A) will appear, providing completion suggestions for users. Meanwhile, the data view (Figure 1 B) displays data contexts for user reference. Initially, all active DataFrames in the notebook kernel are listed (Figure 1 B3). Expanding an item in the DataFrame list reveals the schema (Figure 1 B2) with profiles such as data type, cardinality and value range of each column, helping users rapidly recall the content of the DataFrame. Clicking the “Show sample rows...” button displays the first 15 rows (Figure 1 B1), aiding further understanding without repeatedly using additional code like `df.head()` to print DataFrames. It also facilitates efficient verification of analysis results by automatically highlighting and previewing real-time data contexts based on the current partial code in the editor and the selected item in the completion list. Since the visualization of data contexts may occupy a significant amount of screen space, the data view is kept in a split window on the right. This layout choice minimizes visual obstruction of the code editor and reduces repeatedly scrolling to view the code. These components cover two major kinds of activities (i.e. DI and CA) discussed in Section 3.4, keeping users aware of data contexts throughout the coding process. To support the functionalities of the interface, three computational components, namely *code context manager*, *data context manager* and *completion generator* are designed. Their relationship is shown in Figure 3. The *code context*

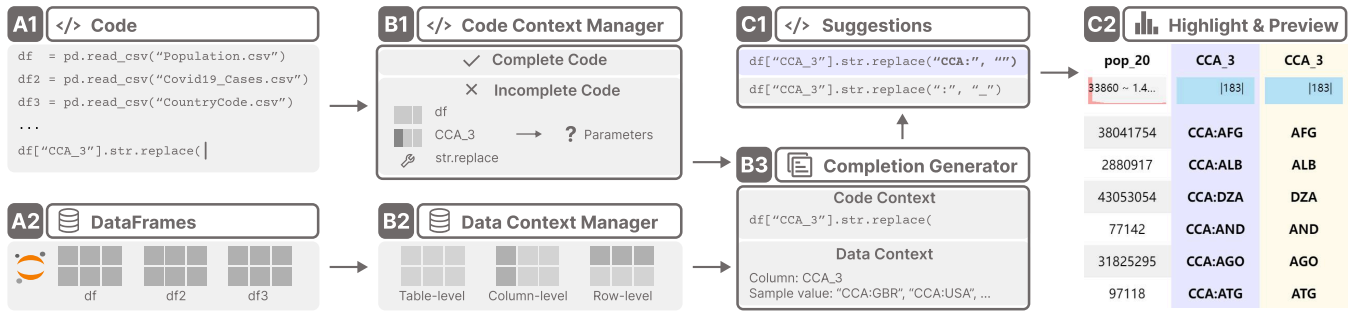


Figure 3: The workflow of Xavier. The input of *Xavier* consists of code in the editor (A1) and DataFrames in the notebook kernel (A2). The code is divided into the complete part and the incomplete part by the code context manager (B1) where the incomplete code is further parsed. Data contexts for each DataFrame are pre-calculated in the data context manager (B2) since the last run of code. The complete code, the parsing result and data contexts are transferred to completion generator (B3) for data context-aware code suggestions (C1). Meanwhile, *Xavier* highlights the most relevant data based on user’s code and the completion suggestions in the data view, previewing transformation results to facilitate code verification (C2).

manager (Figure 3 B1) splits the code in the current notebook (Figure 3 A1) into the complete part and incomplete part, parsing the partial code near the input cursor. The parsing result serves as clues to gather relevant data contexts for the subsequent workflow. The *data context manager* (Figure 3 B2) computes data contexts for all active DataFrames in the notebook kernel (Figure 3 A2). Both code and data contexts are then transferred to the *completion generator* (Figure 3 B3) which offers intelligent code completions (Figure 3 C1), real-time highlighting and preview (Figure 3 C2) upon invocation.

Data contexts are integrated into the design of *Xavier*. Inspired by a previous study [12], we categorize data contexts into three types according to three different types of data objects [34]: tables, rows and columns. Table-level data contexts contain basic information of a data table, including the table name, column name and shape. Column-level data contexts include detailed information of a column. Common column-level data contexts are the data type, null value count and sortedness. In addition to common column-level contexts, categorical columns have unique values, value frequency, cardinality and value format, while numerical columns have value range, sample data points and value format. Row-level data contexts include sample rows of a table, maintaining the spatial relationship between table cells. Different data contexts can be combined and enhance functionalities in *Xavier* like code completion. For instance, column names from table-level contexts and sample rows from row-level contexts are both required for the table join operation to complete parameters indicating the key columns. The same type of data contexts of different data objects can also be combined. For example, table-level data contexts from multiple tables are essential for the table concatenation operation.

4.2 Data Context-Aware Code Completion

Assume that Sarah, a data journalist, planned to transform a movie dataset into a table with movie titles, ratings and duration to analyze the relationship between the favourable rating and the duration of movies. After loading tables, Sarah found that the information about favourable ratings and the

movie duration was not in the same DataFrame, so she decided to join them first. As she wrote `joined=movies.merge(ratings)`, she forgot the exact spelling of the joined columns, though she remembered that they were about movie titles. Without manually searching column names in the “movies” and “ratings” DataFrames, *Xavier* suggested the completion `left_on="netflixTitle", right_on="title"`. Having a glimpse of corresponding highlight on the data view, Sarah found the code completion correct and immediately accepted the suggestion without typing verbose parameters.

Xavier incorporates both code contexts and data contexts to provide data context-aware code completion during typing (R1). The functionality is implemented by a completion generator (Figure 3 B3) which organizes code contexts from the code context manager (Figure 3 B1) and data contexts from the data context manager (Figure 3 B2) to compute completion results. The workflow consists of three steps, namely code context detection, data context organization, and code completion generation.

4.2.1 Code context detection. Leveraging the principles of lexical and syntax analysis [1], the code context manager (Figure 3 B1) detects the cursor position in the incomplete Python statement to determine missing parts of code. A set of configurable and extensible rules is manually constructed based on the Python grammar [17]. In the first step, the code context manager determines whether the user’s input cursor is currently positioned within the signature of a Pandas function (e.g. `pd.merge(df1, df2, left_on="col1")`) or out of the signature (e.g. `df[df["A"]]`). If the cursor is in a signature, the manager records the signature name and identifies missing parameters in the partial code. For example, in the partial code `pd.merge(df1, df2, left_on="col1"`, the signature name is “merge” and the filled parameters are the left table “df1”, the right table “df2” and the joined column “col1” in the left table. This indicates that a corresponding joined column name in the right table is required. If the cursor is out of the signature, the manager analyzes patterns of Abstract Syntax Trees (ASTs)

Figure 4 shows a Jupyter Notebook interface with the following code in the code cell:

```

[1]: import pandas as pd
pd.options.mode.chained_assignment = None

[2]: titles = pd.read_csv("netflix_titles.csv")
movies = pd.read_csv("IMDb movies.csv")
ratings = pd.read_csv("IMDb ratings.csv")

joined = titles.merge(movies, left_on="netflixTitle", right_on="title")

[3]: joined2 = joined[["netflixTitle", "durationOfTime", "nf_type"]]

```

The Table View on the right shows the schema of the 'joined' DataFrame:

item	nf_type	netflixTitle	director	durationOfTime
s8	Film	Sankofa	Haile Gerima	125 minutes
s6	TV Series	Midnight ...	Mike Flana...	1 Season
s16	TV Series	Dear White...	NaN	4 Seasons
s20	TV Series	Jaguar	NaN	1 Season
s25	Film	Jeans	S. Shankar	166 minutes
s30	Film	Paranoia	Robert Luk...	106 minutes

Annotations in the image:

- B1**: Points to the `joined` variable in the code cell.
- B2**: Points to the `durationOfTime` column in the Table View.
- C**: Points to the list of column names `["netflixTitle", "durationOfTime", "nf_type"]` in the code cell.

Figure 4: The usage scenario of automatic data context highlighting. A) Xavier detected the existing DataFrame “joined” and showed the corresponding schema. B) When Sarah was selecting the suggested column names for the partial code (B1), Xavier displayed sample rows of the DataFrame “joined” and highlighted relevant columns based on Sarah’s code and the selected suggestion (B2). C) Finally, Sarah selected three columns which were highlighted by Xavier.

Figure 5 shows a Jupyter Notebook interface with the following code in the code cell:

```

[1]: import pandas as pd
pd.options.mode.chained_assignment = None

[2]: titles = pd.read_csv("netflix_titles.csv")
movies = pd.read_csv("IMDb movies.csv")
ratings = pd.read_csv("IMDb ratings.csv")

[3]: joined = titles.merge(movies, left_on="netflixTitle", right_on="title")

[4]: joined2 = joined[["netflixTitle", "durationOfTime", "nf_type"]]

[5]: joined2 = joined2[joined2["nf_type"]=="Film"]

[6]: joined2["durationOfTime"] = joined2["durationOfTime"].str.replace("minutes", "")

```

The Table View on the right shows the schema of the 'joined2' DataFrame:

netflixTitle	durationOfTime	durationOfTime	nf_type
Sankofa	125 minutes	125	Film
Jeans	166 minutes	166	Film
Paranoia	106 minutes	106	Film
Paranoia	106 minutes	106	Film
Training Day	122 minutes	122	Film
King of Boys	182 minutes	182	Film

Annotations in the image:

- A**: Points to the transformation operation `joined2["durationOfTime"] = joined2["durationOfTime"].str.replace("minutes", "")` in the code cell.
- B1**: Points to the original `durationOfTime` column in the Table View.
- B2**: Points to the transformed `durationOfTime` column in the Table View.

Figure 5: The usage scenario of real-time transformation preview. When Sarah switched to a completion item about column format transformation (A), Xavier automatically computed the transformation result and added a preview column (B2) to the right of the original column (B1), with bold text in changed table cells.

to identify possible transformation operators and missing AST nodes. For instance, in the partial code `df[df["A"]]`, the manager identifies it as a filtering operation according to the AST pattern `df[]`. Based on the basic format of a filtering condition, at least an operator and corresponding parameters like `=="Value"` are required to complete the transformation. To ensure generalizability, the code context manager matches AST patterns from child

nodes up to parent nodes. Hence, for nested transformations like `pd.merge(df1, df2[df2["A"]])`, only the filtering transformation `df2[df2["A"]]` will be considered. Likewise, for chained transformations like `df1["A"].fillna("Unknown").str.replace()`, only the string replacement transformation function `replace()` will be taken into account. These code contexts will be utilized to determine the timing for triggering each category of code completion suggestions

(see Table 1 and Table 2 in the appendix) and to inform the content of those suggestions.

4.2.2 Data context organization. As discussed in Section 4.1, three types of data contexts in the data context manager (Figure 3 B2) have been pre-calculated since the last run and will be sent to the completion generator (Figure 3 B3) as the code completion is triggered. To save the computation cost and offer more targeted completions, the data contexts will be further filtered in two steps, namely type matching by operators and context selection by operands.

In type matching, data context types are determined based on transformation types classified in Pandas documentation [72]. This step excludes data context types that have weak relevance to the transformation. Since “DataFrame” and “Series” cover most of commonly used operators in Pandas, we classify all operators into three types: “DataFrame”, “Series”, and “Others”. “DataFrame” operators primarily rely on the table-level data contexts, with row-level and column-level data contexts as additional references. For instance, column names and sample values can provide useful clues to complete a joined table `country_code` and joined columns `left_on="Country"`, `right_on="Countryname"` in the partial code `df3 = covid_data.merge(`. For “Series” operators, only column-level data contexts are selected, since “Series” operators focus on single or multiple columns in a table. For example, the format of values in the Series `df["Country"]` is an essential clue to completing substrings to be replaced in `df["Country"].str.replace(`. For “Others” operators, the data contexts considered are default to table-level data contexts.

In context selection, data contexts of specific tables or columns are determined by missing parts of code detected in Section 4.2.1. This step further excludes weakly relevant data contexts from other tables or columns. For instance, in the partial code of a table filtering operation `df[`, row-level data contexts like sample rows of “df” are taken into account as a supplementary clue in addition to table-level data contexts, since the filtering condition is totally unknown. However, when completing the partial code `df[df["A"]`, column-level data contexts, such as sample values of column “A”, will take precedence over row-level data contexts, as the filtering condition is likely related to column “A”. As a result, within the same operator, the required data contexts will be dynamically adjusted according to the missing parts of the code, retaining only the most relevant data contexts to provide targeted code completion (see Table 1 and Table 2 in the appendix for details). Since the amount of filtered data contexts can be potentially large (e.g. unique values of primary key columns), the data context manager randomly samples values to control the size of data contexts. To reduce the sampling limitation, the data context manager also supports filtering data values by prefix, saving the effort of manually adding information about data values into comments before generating code. For instance, although at most 50 unique values are sampled for the “unique values” data context of a categorical column, users can still specify data values by typing the prefix of values.

4.2.3 Code completion generation. With the code context detected and data context organized, *Xavier* combines both to offer recommendations for subsequent code through the completion generator

(Figure 3 B3), supporting both single-token and multi-token completion. The single-token completion works like traditional code completion by completing the rest of a token based on its prefix, but it provides more comprehensive column name completions since writing column names is common in data wrangling scripts. For instance, for the partial code `df.sort_values(by="C"`, all column names starting with “C” in table “df” will be listed as suggestions. The multi-token completion leverages the capabilities of Llama3-70B [51], an open-source LLM with outstanding performance in various general tasks. Inspired by the prompt structure proposed in [28], we design a prompt template consisting of four components: *code context*, *data context*, *task instruction* and *format control*. The *code context* refers to all data wrangling code preceding the cursor. The *data context* is the textual representation of the organized data contexts discussed in Section 4.2.2. Column-level data contexts are grouped by columns, while table-level and row-level data contexts are grouped by tables. The *task instruction* includes the thinking steps to help derive the appropriate completions. *Format control* is employed to restrict the output format, increasing the likelihood of obtaining syntactically correct answers. Note that the prompt template is merely one of the feasible solutions to combining code and data contexts to offer data context-aware code completion. We encourage future researchers to explore alternative solutions.

4.3 Automatic Data Context Highlighting

After writing the code to join tables, Sarah ran it to compute the joined DataFrame. *Xavier* detected the change of the notebook memory and added profiles of the joined DataFrame to the data view with schemas automatically displayed, saving the need to manually write code to print the result DataFrame. Glancing at the schema, Sarah found that the joined DataFrame included 14 columns, with too many unrelated ones (Figure 4 A). She decided to select the related ones first for convenience of subsequent wrangling.

As she wrote `joined2=joined`, *Xavier* folded the schemas of other tables, making her focus on the current table (Figure 4 A). While selecting the suggested column names for the partial code like `joined2 = joined[["netflixTitle"` (Figure 4 B1), *Xavier* showed sample rows of the DataFrame “joined” and highlighted columns mentioned in the partial code (i.e. the column “netflixTitle”) as well as the currently focused completion item “durationOfTime”, as is shown in Figure 4 B2. Since the data view had limited width and could not display all columns of the DataFrame at the same time, *Xavier* anchored invisible highlighted columns (i.e. “durationOfTime” in Figure 4 B2) for Sarah to reference. This feature allowed her to review the selected columns while switching among different completion items, ensuring that these columns were what she wanted. Finally, Sarah typed the code `joined2 = joined[["netflixTitle", "durationOfTime", "nf_type"]]` (Figure 4 C) and executed it with confidence.

In order to help users rapidly locate relevant parts of datasets (R2), *Xavier* automatically unfolds profiles of the corresponding DataFrames and highlights columns in the side panel when users

A New Columns

pop_20	CCA_3	CCA_3
33860 ~ 1.4...	[183]	[183]
38041754	CCA:AFG	AFG
2880917	CCA:ALB	ALB
43053054	CCA:DZA	DZA
77142	CCA:AND	AND
31825295	CCA:AGO	AGO
97118	CCA:ATG	ATG

B Deleted Rows

Observati...	ProvinceO...	CName
[345]	[933]	[223]
2020-01-22	Anhui	China
2020-04-06	Algeria	Algeria
2020-04-06	Andorra	Andorra
2020-01-22	Beijing	China
2020-01-22	Chongqing	China
2020-01-22	Fujian	China

C Original Tables and New Tables

title	tot_vote	Countryn...
[2143]	6 ~ 1.7e+6	[308]
Sankofa	316	United Stat...
Jeans	1353	India
Paranoia	144	United Stat...
Paranoia	26	United Stat...
Training Day	240120	United Stat...
King of Boys	160	Nigeria

title	tot_vote	Countryn...
[2143]	6 ~ 1.7e+6	[308]
Inception	1703862	United Stat...
Pulp Fiction	1508986	United Stat...
Seven	1175284	Unknown ...
Django Un...	1067956	United Stat...
Schindler's ...	1029573	United Stat...
American ...	843384	United Stat...

Figure 6: Three preview forms of Xavier. A) For the column format transformation, a new column is created to the right of the original column. B) For the table filtering transformation, rows to be deleted are highlighted. C) For transformations that generate a new table or change the whole table (e.g. Sort movies by total votes. For movies having equal total votes, sort them by country names), both the original table and the result table are displayed.

are typing code or switching between different completion items. *Xavier* detects table and column names mentioned in the partial code as well as those in the currently focused completion item. When the partial code and completion item only contain table names, *Xavier* simply displays the schema of each table, with sample rows concealed. For example, when users are filling in the table names in a union transformation, such as `pd.concat([df1, and the currently focused completion item is df2, df3], the schemas of tables “df1”, “df2”, and “df3” will be automatically presented in the side panel, while other previously unfolded table profiles are collapsed. When column names are also present, sample rows of tables will be shown and the mentioned columns will be highlighted. For instance, as users are selecting columns of a table through df[["col1", "col2", and the currently focused completion item is "col3", all three columns (“col1”, “col2” and “col3”) will be marked using a different background color in the sample rows. Since the width of the side panel is limited and cannot simultaneously display all columns in a table, Xavier adopts a floating effect to anchor highlighted columns to the right side of the data view. Therefore, users do not have to frequently scroll horizontally across the sample row view to find all highlighted columns. Automatic data context highlight helps users rapidly recall tables or columns of interest, keeping users aware of data contexts in the current transformation although the code is not necessarily complete.`

4.4 Real-time Transformation Preview

Having selected relevant columns and obtained a new DataFrame `joined2`, Sarah wanted to standardize the format of the “durationOfTime” column in order to better analyze the relationship between the favourable rating and the duration of movies. When she wrote `joined2["durationOfTime"]=joined2["durationOfTime"]`, *Xavier* suggested multiple wrangling operations, including `.str.replace("minutes", "")` (Figure 5 A). Not sure about the effect of the operation, Sarah switched to the completion item and checked the preview in the data view. She found that a new column marked in yellow (Figure 5 B2)

was positioned beside the “durationOfTime” column (Figure 5 B1), showing the changes with bold text in modified table cells. Sarah immediately verified the suggestion and accepted it to standardize the column format with ease.

In order to facilitate understanding and straightforward verification of code completions (R3), *Xavier* provides preview in the data view for users to immediately examine the transformation results when required parameters for an operation are all filled, without explicit code execution by users. Inspired by Wrangler [31], *Xavier* supports three types of previews, as is shown in Figure 6. For column format transformation like substring replacement and null value filling, a new column marked in yellow is created to the right of the original column, with bold text indicating the changed cells (Figure 6 A). For table filtering transformation, deleted rows are marked in red, with bold text of filtering values mentioned in user’s code (Figure 6 B). For transformations that generate a new result table or change the whole original table like table sorting, aggregation and merging, *Xavier* displays both the original table and the new table (Figure 6 C). The preview visualization is similar to the design of TweakIt [39], although *Xavier*’s preview is intended for comprehending AI-generated code completions instead of pre-existing code snippets.

5 USER STUDY

To evaluate the effectiveness and usability of *Xavier*, we conducted a comparative user study³ in which participants were asked to complete data wrangling tasks using *Xavier* and a baseline tool. The setup of our user study is delineated in Sections 5.1-5.3 and the results are reported in Section 5.4.

5.1 Participants

In our user study, 16 data analysts (denoted as U1-U16, 11 male and 5 female, $Age_{mean} = 23.94$, $Age_{std} = 1.69$, $Experience_{mean} = 3.06$ years, $Experience_{std} = 1.77$ years) were recruited from a university through social media and word-of-mouth. This was a completely fresh sample with no overlap between participants in this

³The user study has been approved by State Key Lab of CAD&CG, Zhejiang University.

study and those involved in the preliminary study. They had diverse backgrounds including Software Engineering, Energy, Management Science and Engineering, Data Visualization, Physical Education and Training, and Computer Science. They all had programming experience in data wrangling ranging from one year to six years and regularly used Python Pandas in computational notebooks. Participants consented to having their voices and programming processes recorded.

5.2 Apparatus and Materials

Apparatus. The baseline tool we constructed was different from *Xavier* in two aspects: code completion and visualization view. For code completion, the baseline tool used the same backend LLM as that of *Xavier*, while data contexts were removed from the prompt to compare the effectiveness of data contexts in code suggestions. We chose Llama3-70B [51], an exemplary publicly-available LLM as the underlying model for both the baseline tool and *Xavier*, due to its relatively low latency in local deployment. For the fairness of tool comparison, the side panel view of AutoProfiler [12], a continuous data profiling tool in recent literature, was replicated for the visualization view. Similar to Section 3.2, we also prepared a slide with task descriptions and data dictionaries. In order to control the standardization of participants’ data transformation scripts and simultaneously reduce the cost of participants referring to APIs, we prepared a cheatsheet for them. Inspired by [49], the cheatsheet listed supported transformations by *Xavier* and corresponding API usage, categorizing APIs by functionality.

Datasets. We crafted two datasets, denoted as *Covid-19* and *Movies* dataset from two notebooks⁴ on Kaggle. We chose these notebooks since participants are generally familiar with the background of datasets [68] and typically do not spend much time understanding the data. The crafted datasets still covered two major types of tabular data (i.e. categorical and numerical). For the similar reason discussed in Section 3.2, data tables were slightly modified.

Tasks. We designed a data wrangling task for each dataset in a similar way to that described in Section 3.2. To avoid users being distracted by too much irrelevant data and spending much time exploring the dataset, we controlled the number of tables, columns, types of data transformation operators, and the number of data transformation steps involved in both tasks. For each dataset, we only selected 3 tables and 20 columns relevant to the corresponding task. In making these selections, we sought to preserve the logical and semantic relationships within the original dataset as much as possible, such as retaining primary and foreign key relationships. This ensures that participants can complete the tasks using only the provided data without requiring additional information. Both tasks share the same set of data transformation operators. Each task contains approximately 10 lines of code. For each task, participants were asked to construct a table with 4 attributes. In the remaining part of Section 5, *Movie Task* refers to the data wrangling task on *Movies* dataset, and *Covid Task* refers to the one on *Covid-19* dataset. Task details are left to the supplementary materials.

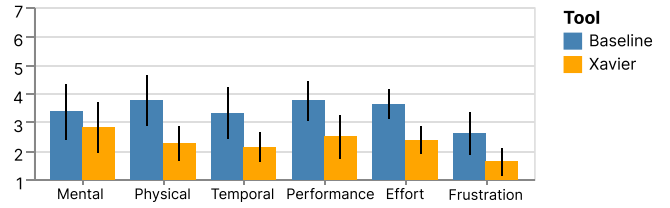


Figure 7: The perceived workload of participants using the baseline tool and *Xavier*.

5.3 Procedure

We opted for a counterbalanced mixed design to compare *Xavier* and the baseline tool. We denote the two systems as *X*(avier) and *B*(aseline) and the two datasets as *M*(ovies) and *C*(ovid-19). The participants were divided into groups of four. In each group, the participants covered the following four experimental conditions: [*MB*, *CX*], [*MX*, *CB*], [*CB*, *MX*], [*CX*, *MB*]. Such approach allowed each participant to experience both tools and reduced potential impact on experiment data brought by the sequence of tool using and tasks like learning effect.

Like Section 3.3, we informed participants about relevant information, conducted a code authoring experiment (50-60 minutes) and a semi-structured interview (10-15 minutes), and collected data from recordings. The entire study took around 75 minutes and each participant received 70 Chinese Yuan as compensation.

Code authoring experiment. Similar to Section 3.3, we initially introduced the tool before each task and allowed participants to try it on a warm-up task. To verify whether introducing data context can improve code completion and enhance user experience, we did not inform participants in advance about how these two tools worked. When participants were ready, they started to author data wrangling scripts assisted by *Xavier* or the baseline tool. Participants were permitted to consult the cheatsheet, the data dictionary and the task description at any time during authoring. After the experiment, they were asked to finish a questionnaire which assessed their perceived workload.

Semi-structured interview. Three parts consisted of the semi-structured interview. Initially, we asked participants to compare the effect of code completion between the two tools corresponding to **R1** in Section 3.5. Then, participants compared and discussed the coding assistance brought by the side panel, which corresponds to **R2** and **R3**. In the last part, we asked questions specifically targeting the issues observed during the experiment. The questionnaire included six NASA-TLX [22] questions to measure the perceived workload of participants when using Baseline and *Xavier*. All questions were measured using the 7-point Likert Scale.

5.4 Results

5.4.1 Quantitative Results. We collected and summarized the questionnaire results shown in Figure 7. On average, participants perceived relatively lower workload when using *Xavier* compared to that of the baseline tool regarding six aspects (mental demand, physical demand, temporal demand, performance, effort and frustration),

⁴<https://www.kaggle.com/code/erikbruin/storytelling-covid-19> and <https://www.kaggle.com/code/niharika41298/netflix-visualizations-recommendation-eda>

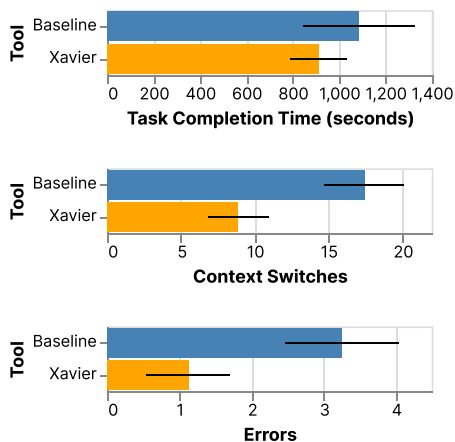


Figure 8: Quantitative results of our user study. We recorded task completion time in seconds, context switches and errors of participants by using the baseline tool and *Xavier*.

which suggests that the data context-aware design of *Xavier* has the potential to reduce workload in code authoring.

To figure out whether *Xavier* accelerated the coding process, we measured “task completion time” of each participant for each task. To assess whether *Xavier* facilitated locating and understanding relevant data (R2 and R3), we defined “context switches” to estimate the additional overhead of authoring data wrangling scripts. A context switch starts when users pause writing data wrangling scripts, during which users may manipulate the side panel by mouse, manually add code to profile the data, or leave the current editor window to review the original data tables. The context switch ends when users return to the editor and continue writing the wrangling scripts. To evaluate the suggestion correctness of *Xavier*, we defined “errors” to measure the errors users encountered in the code authoring experiment. The errors include code errors related to language grammar and data errors like accessing non-existent columns, unexpected transformation results and so on.

Since the raw data of these measurements did not obey either normal distribution or equal variance, we used the Scheirer-Ray-Hare test, a nonparametric substitute for ANOVA, for statistical tests. We tested the main effects of tools and tasks, as well as their interaction effects. We found that for all three measurements, the interaction effects and the main effects of tasks are not significant. For task completion time, the main effect of tools is not significant. However, for context switches and errors, the main effect of tools is significant (both $p < 0.001$). Considering that the interaction effects and the main effects of tool are not significant, we reported the means of task completion time, context switches, and errors of different tools, as shown in Figure 8. There are fewer context switches and errors when users using *Xavier*, which indicates that participants likely concentrated on data wrangling code authoring more easily and wrote data wrangling code with more accuracy with *Xavier*. However, we did not find a significant main effect of tools on task completion time. One of the possible reasons is that the task design is relatively simple (around 10 lines of code for each task) given the limited experimental duration. Hence, long-term

evaluation of the task completion time by using *Xavier* is required in the future work.

5.4.2 Qualitative Results. To analyze the user feedback from the interview, we conducted a qualitative inductive content analysis [38]. We focused on three aspects: 1) how the participants compared and evaluated the two tools; 2) the comments of the participants on *Xavier*; 3) the issues they encountered when using *Xavier* and their suggestions for improvement. In the open coding phase, two co-authors (data analysis experience: ≥ 3 years) read the recording transcripts and labeled the interview feedback individually. Then they collaboratively resolved disagreements through periodic meetings where the coders discussed the labeled content and reconciled differences in interpretation. Furthermore, they grouped the similar comments into higher-level themes. Finally, an additional co-author (data analysis experience: > 5 years) was involved to derive the consensus on the key findings, which are as follows:

User experience. Most participants (14/16) thought that *Xavier*’s code completions were “better”. Some participants even described the code completions of *Xavier* as “amazing” (U4), “useful” (U1, U6), or “smart” (U5). Similarly, most participants (14/16) mentioned the benefits of the automatic preview of *Xavier*, thinking that it is “helpful” (U1, U3, U12) and it can help them “avoid errors” (U5, U8, U14). The third feature of *Xavier* frequently mentioned by participants is automatic highlighting (13/16). Participants described the automatic highlighting as “intuitive” (U11, U15) and “convenient” (U3, U5, U14). As the live data view kept changing while coding, we additionally asked participants if they were distracted by frequent changes. However, none of the participants found it distracting. We identified two primary reasons from their responses. First, their attention primarily remained on the code editor during typing and they only saw the live data view when necessary (U2-U4, U6, U10, U11, U13, U16). Second, real-time updates of the live data view can bring them confidence (U1, U3, U8, U15). For example, “If it (the live data view) doesn’t change, I will worry if I have written something wrong”, U8 noted.

Code completion preferences. Participants seemed to prefer shorter code completions rather than longer ones. According to the participants’ feedback, they preferred completions like column names (U1-U5, U8, U11, U13-U16), data values (U3, U16) and parameters (U8, U13, U14), rather than completions like the whole transformation statement. We identified two primary reasons for the preference of shorter code completions. First, longer completions typically required more time to generate (U2, U6, U12). For instance, U6 skipped most of the completions (8/10) of the whole statement after the assignment operator. As she pointed out, “It (the code completion of *Xavier*) came out a little slower than I thought, making me hesitate whether to type it myself or wait for it to appear”. Second, participants believed longer completions would be less accurate (U12-U15). As U14 commented, “If it (*Xavier*) could really guess what I’m thinking and provide longer completions, that would be the best. But I probably don’t have such expectations because I think it’s very difficult to achieve.”

Transparency and trust. As previously discussed, the preview and highlight features of *Xavier* may help users more easily validate the completions and enhance their confidence in the tool. However, these features cannot fully address the issues of

transparency and trust. One issue is that the model mechanisms behind completion are not transparent. Specifically, U4, U6, U15 and U16 were curious about how the code completion is recommended. For instance, U4 was surprised by the accuracy of the completion `df["ConfirmedCases"] / df["pop_20"]` for the partial code `df["casesPer"] =`, which was exactly what he wanted. “How did it (Xavier) know the meaning of ‘pop_20’ is the population in 2020? Is there some information to tell it (Xavier)?”, U4 asked. We also noticed that initial few code completions may influence participants’ trust in its capabilities (U8, U11, U13, U15). For instance, during the code authoring experiment, U15 initially waited for the completion of the whole filtering transformation when she typed `selected =`, but did not receive a correct response. Eventually, she skipped most of the completions of this kind (6/7) and only waited for shorter ones as she specified the transformation operator. “Sometimes it (Xavier) completed a long line for me... I would not trust such completions”, she said in the interview.

6 DISCUSSION

This paper presents *Xavier*, a tool designed to enhance data wrangling script authoring in computational notebooks. The user study revealed that users encountered significantly fewer context switches and errors during scripting by using *Xavier* (Section 5.4.1). User feedback indicated that *Xavier* could help validate code and bring confidence to users (Section 5.4.2). In this section, we discuss the lessons learned from the development and evaluation of *Xavier* in Section 6.1. Besides, we identify limitations of our research that can be further improved in Section 6.2.

6.1 Lessons Learned

Throughout the development and the evaluation of *Xavier*, we have gained valuable insights and lessons:

Presenting contexts to users in an always-on side panel makes users context-aware while minimizes excessive interruptions. Prior work [50, 77, 78] has extensively discussed the user interface design of code assistants in computational notebooks. For instance, the display style can be on-demand for situational contexts or always-on for continuous contexts [77]. Since **DI** and **CA** are interleaved from our observation in the preliminary study (Section 3.4) and it is uncertain when users need to view data contexts, adopting the always-on display style is a suitable design to facilitate users to remain aware of data contexts in authoring data wrangling scripts. In comparison to displaying contexts inline, placing contexts in the side panel may help reduce disturbance while users are authoring code. Previous code assistants like Notable [42] and Auto-Profiler [12] adopt a similar design to reduce context switches while maintaining awareness of specific contexts like data facts or data profiles. In our user study, with the always-on display, participants could refer to the highlighted data and the preview result at any time, with less context switch overhead. Although the highlighted area kept changing, participants did not think the frequent change of the data view scattered their attention since they focused on the code editor while coding, which may suggest the feasibility of the always-on side panel design.

Controlling the length of code completions can improve user experience. According to Section 5.4.2, many participants

preferred shorter completions. One of the reasons is that they believed longer completions were less accurate. Another potential reason is that when completions did not exactly follow users’ intent (e.g., *Xavier* “might suggest variations of a function”), shorter completions can be more easily understood and verified. Given the current limitations in model performance, it is necessary to introduce new methods to control the generation of shorter completions, making it easier for users to verify and accept the suggestions. However, current suggestions on shorter completions could be only an interim solution. To better meet user expectations, it would be beneficial to delve into research on models with higher performance in longer completions. Meanwhile, it would be worthwhile for follow-up studies to develop new methods that not only foster user trust but also help assess the performance of models.

Improving model transparency facilitates evaluating the capability of code completion. Currently, *Xavier* incorporates preview and highlight features to assist users in validating AI-generated code completions. However, according to Section 5.4.2, these features cannot fully address the issues of transparency and trust, making participants unclear about the model’s capabilities in code completion. We suggest two possible reasons for the issues. First, to offer straightforward code verification and mitigate information overwhelming, *Xavier* did not provide explanations on what contexts were selected to complete the code. Hence, some participants (U4, U6, U15, U16) were curious about how the code completion worked in the user study interview. Second, users do not have a clear understanding of the performance of LLMs. Therefore, initial few code completions may influence participants’ trust in capabilities of the tool (U8, U11, U13, U15). To further address the transparency and trust issue, future improvements to the design of *Xavier* could include presenting the contexts used for code completion to users on demand. Additionally, quantitatively evaluating the model’s performance and informing users about the model accuracy will also help them better understand the capabilities of the code completion tool.

6.2 Limitations and Future Work

The limitations of our research can be observed in two main aspects: the functionalities and the evaluations of *Xavier*.

6.2.1 Functionalities. The functionalities of *Xavier* can be further improved from the three aspects: **Data context sampling.** To offer data context-aware code completions, *Xavier* selects relevant data contexts according to the partial code being edited. According to Section 4.2.2, if the amount of data contexts is large, only a sample will be collected to compute the code completions due to the limitation of LLMs. During the development of *Xavier*, we have attempted to improve the performance of code completions by adjusting different sampling sizes for data contexts, such as the different maximum number of unique values. However, due to the lack of benchmarks, it remains unclear what the optimal amount of data contexts is, necessitating future experiments for verification.

Response speed. While using *Xavier* for code authoring, users may experience delays in completion responses due to network latency, as *Xavier* frequently sends requests to the LLM behind the scene. Another reason for response delay is the relatively high computation cost, since *Xavier* combines both code contexts and

data contexts to generate intelligent code suggestions. However, response time is also an essential factor affecting the user experience of code completion tools [76]. We encourage future work to optimize the usage of both code contexts and data contexts to save the computation cost and improve response speed.

Supported libraries. *Xavier* uses Pandas as an exemplary data transformation library to offer data context-aware coding assistance. To provide intelligent code completions, highlight relevant data and preview the result, *Xavier* needs to parse the structure of Python Pandas code. The parsing rules need adjustment for generalization to other programming languages and data transformation libraries. Future work can explore a unified way like constructing intermediate domain specific language to adapt to different programming languages and libraries.

6.2.2 Evaluations. To assess whether *Xavier* facilitated locating and understanding relevant data, we defined a metric called “context switches” in Section 5.4.1 to estimate the additional overhead of authoring data wrangling scripts. These “context switches” assume that users pause typing data wrangling scripts while users are manually profiling datasets. However, this metric may not sufficiently capture attention shifts of users (e.g. users might simultaneously observe the live data view and type). As suggested by an anonymous reviewer, an interesting follow-up study could involve eye-tracking to understand how the live data view supports on-the-fly validation of code suggestions and how frequently users shift their attention between the code editor and live data view. Such a study could provide deeper insights to further optimize the design of *Xavier* and better evaluate the tool’s effectiveness.

7 CONCLUSION

During data wrangling code authoring, users have to constantly locate and understand relevant data while writing custom scripts. In this paper, we propose a novel coding assistance approach that prioritizes data contexts and allows users to remain aware of data contexts. We first conducted a preliminary study to identify common patterns from a code authoring experiment, deriving three user requirements according to the semi-structured interview. Then based on the requirements, we propose *Xavier*, a computational notebook extension designed to enhance data wrangling script authoring. *Xavier* integrates both code and data contexts for data context-aware code completion, automatically highlights the most relevant data and instantly previews data transformation results based on the user’s code. *Xavier* was overall appreciated by data analysts in the user study with 16 data analysts. The coding assistance of data context awareness has the potential to be generalized to other data wrangling libraries and programming languages. Furthermore, a long-term evaluation based on real-world datasets deserves exploration in the future work.

ACKNOWLEDGMENTS

The research was supported by National Key R&D Program of China (2022YFE0137800), NSFC (62402421), and HK RGC GRF grant (16210722). The authors would like to thank Qixin Liu, Qiaosong Ying and Kaicheng Shao for helping annotate the video data of the user study. We gratefully thank the anonymous reviewers for their valuable comments and all participants in our studies.

REFERENCES

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [2] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.
- [3] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
- [4] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, New York, NY, USA, 213–222. <https://doi.org/10.1145/1595696.1595728>
- [5] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, Virtual Event, USA, 2633–2650. <https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting>
- [6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* abs/2107.03374 (2021).
- [7] Ran Chen, Di Weng, Yanwei Huang, Xinhuan Shu, Jiayi Zhou, Guodao Sun, and Yingcai Wu. 2023. Rigel: Transforming Tabular Data by Declarative Mapping. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 128–138. <https://doi.org/10.1109/TVCG.2022.3209385>
- [8] Ruijia Cheng, Titus Barik, Alan Leung, Fred Hohman, and Jeffrey Nichols. 2024. BISCUI: Scaffolding LLM-Generated Code with Ephemeral UIs in Computational Notebooks. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, Los Alamitos, CA, USA, 13–23. <https://doi.org/10.1109/VL/HCC60511.2024.00012>
- [9] Bhavya Chopra, Anna Fariha, Sumit Gulwani, Austin Z. Henley, Daniel Perelman, Mohammad Raza, Sherry Shi, Danny Simmons, and Ashish Tiwari. 2023. CoWrangler: Recommender System for Data-Wrangling Scripts. In *Proceedings of Companion of the International Conference on Management of Data*. ACM, New York, NY, USA, 147–150. <https://doi.org/10.1145/3555041.3589722>
- [10] Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3411764.3445267>
- [11] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–12.
- [12] Will Epperson, Vaishnavi Gorantla, Dominik Moritz, and Adam Perer. 2023. Dead or Alive: Continuous Data Profiling for Interactive Data Science. *IEEE Transactions on Visualization and Computer Graphics* 30, 1 (2023), 197–207. <https://doi.org/10.1109/TVCG.2023.3327367>
- [13] Kasra Ferdowsi, Ruanqianqian Huang, Michael B James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–8.
- [14] Kasra Ferdowsi, Jack Williams, Ian Drosos, Andrew D. Gordon, Carina Negreanu, Nadia Polikarpova, Advait Sarkar, and Benjamin Zorn. 2023. COLDECO: An End User Spreadsheet Inspection Tool for AI-Generated Code. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, Los Alamitos, CA, USA, 82–91. <https://doi.org/10.1109/VL-HCC57772.2023.00017>
- [15] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 153 (2021), 29 pages. <https://doi.org/10.1145/3485530>
- [16] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 614–626. <https://doi.org/10.1145/3379337.3415869>
- [17] Python Software Foundation. 2024. Full Grammar Specification of Python. <https://docs.python.org/3/reference/grammar.html>. Last accessed on 2024-11-09.
- [18] GitHub, Inc. 2024. GitHub Copilot. <https://github.com/features/copilot>. Last accessed on 2024-11-09.
- [19] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 317–330.

- [20] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive Wrangling: Mixed-Initiative End-User Programming of Data Transformation Scripts. In *Proceedings of the ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 65–74.
- [21] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 27–38. <https://doi.org/10.1145/2491956.2462192>
- [22] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Human Mental Workload*. Advances in Psychology, Vol. 52. North-Holland, 139–183. [https://doi.org/10.1016/S0166-4115\(08\)62386-9](https://doi.org/10.1016/S0166-4115(08)62386-9)
- [23] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 763–773. <https://doi.org/10.1145/3106237.3106290>
- [24] R. Hill and J. Rideout. 2004. Automatic Method Completion. In *Proceedings of the 19th International Conference on Automated Software Engineering*. IEEE Computer Society, USA, 228–235.
- [25] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Zurich, Switzerland, 837–847.
- [26] Joshua Horowitz and Jeffrey Heer. 2023. Engraff: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, Article 72, 18 pages. <https://doi.org/10.1145/3586183.3606733>
- [27] Daqing Hou and David M. Pletcher. 2011. An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*. IEEE Computer Society, USA, 233–242.
- [28] Yanwei Huang, Yunfan Zhou, Ran Chen, Changhao Pan, Xinhuan Shu, Di Weng, and Yingcai Wu. 2024. Interactive Table Synthesis with Natural Language. *IEEE Transactions on Visualization and Computer Graphics* 30, 9 (2024), 6130–6145. <https://doi.org/10.1109/TVCG.2023.3329120>
- [29] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, New York, NY, USA, 1219–1231. <https://doi.org/10.1145/3510003.3510203>
- [30] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. FooFah: Transforming Data By Example. In *Proceedings of the ACM International Conference on Management of Data*. ACM, New York, NY, USA, 683–698.
- [31] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 3363–3372.
- [32] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 737–745. <https://doi.org/10.1145/3126594.3126632>
- [33] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, New York, NY, USA, 1073–1085. <https://doi.org/10.1145/3377811.3380342>
- [34] Stephen Kasca, Charles Berret, and Tamara Munzner. 2021. Table Scraps: An Actionable Framework for Multi-Table Data Wrangling From An Artifact Study of Computational Journalism. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 957–966. <https://doi.org/10.1109/TVCG.2020.3030462>
- [35] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving Steering and Verification in AI-Assisted Data Analysis with Interactive Task Decomposition. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, Article 92, 19 pages. <https://doi.org/10.1145/3654777.3676345>
- [36] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 140–151.
- [37] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. 2010. SnipSuggest: Context-Aware Autocompletion for SQL. *Proceedings of the VLDB Endowment* 4, 1 (2010), 22–33. <https://doi.org/10.14778/1880172.1880175>
- [38] Klaus Krippendorff. 2018. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, Thousand Oaks, CA.
- [39] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting End-User Programmers Who Transmogrify Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, Article 311, 12 pages. <https://doi.org/10.1145/3411764.3445265>
- [40] Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. 2013. Temporal Code Completion and Navigation. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE Press, San Francisco, CA, USA, 1181–1184.
- [41] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–7. <https://doi.org/10.1145/3313831.3376494>
- [42] Haotian Li, Lu Ying, Haidong Zhang, Yingcai Wu, Huamin Qu, and Yun Wang. 2023. Notable: On-the-fly Assistant for Data Storytelling in Computational Notebooks. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, Article 173, 16 pages. <https://doi.org/10.1145/3544548.3580965>
- [43] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, Stockholm, Sweden, 4159–4165.
- [44] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. 2023. EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In Situ Code Search and Recommendation. *ACM Transactions on Interactive Intelligent Systems* 13, 1, Article 1 (2023), 27 pages. <https://doi.org/10.1145/3545995>
- [45] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, New York, NY, USA, Article 52, 13 pages. <https://doi.org/10.1145/3597503.3608128>
- [46] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2022. A Unified Multi-Task Learning Model for AST-Level and Token-Level Code Completion. *Empirical Softw. Engg.* 27, 4 (2022), 91. <https://doi.org/10.1007/s10664-022-10140-7>
- [47] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2021. Multi-Task Learning Based Pre-trained Language Model for Code Completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 473–485. <https://doi.org/10.1145/3324884.3416591>
- [48] Zhongsu Luo, Kai Xiong, Jiajun Zhu, Ran Chen, Xinhuan Shu, Di Weng, and Yingcai Wu. 2025. Ferry: Toward Better Understanding of Input/Output Space for Data Wrangling Scripts. *IEEE Transactions on Visualization and Computer Graphics* 31, 1 (2025), 1202–1212. <https://doi.org/10.1109/TVCG.2024.3456328>
- [49] Irv Lustig and Princeton Consultants. 2014. Pandas Cheat Sheet. https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf. Last accessed on 2024-11-09.
- [50] Andrew M Mcnutt, Chenglong Wang, Robert A Deline, and Steven M. Drucker. 2023. On the Design of AI-powered Code Assistants for Notebooks. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–16. <https://doi.org/10.1145/3544548.3580940>
- [51] Meta. 2024. Llama 3. <https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/>. Last accessed on 2024-11-09.
- [52] Microsoft. 2024. Data Wrangler. <https://marketplace.visualstudio.com/items?itemName=ms-toolsai.datawrangler>. Last accessed on 2024-11-09.
- [53] Microsoft. 2024. IntelliSense in Visual Studio Code. <https://code.visualstudio.com/docs/editor/intellisense>. Last accessed on 2024-11-09.
- [54] Hussein Moazzar, Gagan Bansal, Adam Fournier, and Eric Horvitz. 2024. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, Article 142, 16 pages. <https://doi.org/10.1145/3613904.3641936>
- [55] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How Data Science Workers Work with Data: Discovery, Capture, Curation, Design, Creation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–15.
- [56] Kirill Müller, Romain François, and Hadley Wickham. 2018. dplyr. <https://dplyr.tidyverse.org/>. Last accessed on 2024-11-09.
- [57] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-Based Pattern-Oriented, Context-Sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Zurich, Switzerland, 69–79.
- [58] Takayuki Omori, Hiroaki Kuwabara, and Katsuhisa Maruyama. 2012. A Study on Repetitive Use of Code Completion Operations. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*. IEEE Computer Society, USA, 584–587.
- [59] The pandas development team. 2024. pandas-dev/pandas: Pandas 2.2.3. <https://doi.org/10.5281/zenodo.13819579>. Last accessed on 2024-11-09.
- [60] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-Directed Completion of Partial Expressions. In *Proceedings of the 33rd ACM*

- SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 275–286. <https://doi.org/10.1145/2254064.2254098>
- [61] Deepthi Raghunandan, Zhe Cui, Kartik Krishnan, Segen Tirfe, Shenzhi Shi, Tejaswi Darshan Shrestha, Leilani Battle, and Niklas Elmqvist. 2024. Lodestar: Supporting Rapid Prototyping of Data Science Workflows through Data-Driven Analysis Recommendations. *Information Visualization* 23, 1 (2024), 21–39. <https://doi.org/10.1177/14738716231190429>
- [62] Vijayshankar Raman and Joseph M Hellerstein. 2001. Potter’s Wheel: An Interactive Data Cleaning System. In *Proceedings of the VLDB Endowment*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 381–390.
- [63] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 731–747. <https://doi.org/10.1145/2983990.2984041>
- [64] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 419–428. <https://doi.org/10.1145/2594291.2594321>
- [65] Romain Robbes and Michele Lanza. 2008. How Program History Can Improve Code Completion. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, USA, 317–326.
- [66] Johan Rosenkilde. 2024. How GitHub Copilot is Getting Better at Understanding Your Code. <https://github.blog/ai-and-ml/github-copilot/how-github-copilot-is-getting-better-at-understanding-your-code/>. Last accessed on 2024-11-09.
- [67] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. Exploration and Explanation in Computational Notebooks. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–12. <https://doi.org/10.1145/3173574.3173606>
- [68] Bahador Saket, Alex Endert, and Çağatay Demiralp. 2019. Task-Based Effectiveness of Basic Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 25, 7 (2019), 2505–2512. <https://doi.org/10.1109/TVCG.2018.2829750>
- [69] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What Is It Like to Program with Artificial Intelligence?. In *Proceedings of the 33rd Annual Conference of the Psychology of Programming Interest Group*. 127–153.
- [70] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 1433–1443. <https://doi.org/10.1145/3368089.3417058>
- [71] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, New York, NY, USA, 2727–2735. <https://doi.org/10.1145/3292500.3330699>
- [72] The pandas development team. 2024. Pandas API Reference. <https://pandas.pydata.org/docs/reference/index.html>. Last accessed on 2024-11-09.
- [73] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Proceedings of Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–7. <https://doi.org/10.1145/3491101.3519665>
- [74] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the Loop: Supporting Data Comparison in Exploratory Data Analysis. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–10.
- [75] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J. Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 106:1–106:15.
- [76] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How Practitioners Expect Code Completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 1294–1306. <https://doi.org/10.1145/3611643.3616280>
- [77] Zijie J. Wang, David Munechika, Seongmin Lee, and Duen Horng Chau. 2024. SuperNOVA: Design Strategies and Opportunities for Interactive Visualization in Computational Notebooks. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, Article 304, 17 pages. <https://doi.org/10.1145/3613905.3650848>
- [78] Thomas Weber and Sven Mayer. 2024. From Computational to Conversational Notebooks. *arXiv preprint* (2024). <https://doi.org/10.48550/ARXIV.2406.10636>
- [79] Hadley Wickham, Davis Vaughan, Maximilian Girlich, and Posit Software. 2024. tidy. <https://tidyr.tidyverse.org/>. Last accessed on 2024-11-09.
- [80] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, Article 119, 14 pages. <https://doi.org/10.1145/3654777.3676374>
- [81] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 1539–1554. <https://doi.org/10.1145/3318464.3389738>
- [82] Lita0 Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight Anchored Explanations of Just-Generated Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, Article 140, 15 pages. <https://doi.org/10.1145/3613904.3642239>
- [83] Amy X. Zhang, Michael Muller, and Dakuo Wang. 2020. How do Data Science Workers Collaborate? Roles, Workflows, and Tools. In *Proceedings of the ACM on Human-Computer Interaction*. ACM, New York, NY, USA, 23 pages.
- [84] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODen: Live Programming for Neural Network Architecture Editing. In *Proceedings of the 27th International Conference on Intelligent User Interfaces*. ACM, New York, NY, USA, 392–404. <https://doi.org/10.1145/3490099.3511120>