HiLT: A Library for Generating Human-in-the-Loop Data Transformation GUIs

Sora Kanosue
Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California, USA
sorakanosue@berkeley.edu

Xiaorui Liu
Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California, USA
xiaoruiliu@berkeley.edu

Parker Ziegler
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
peziegler@berkeley.edu

Eric Rawn
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
erawn@berkeley.edu

Sarah E. Chasins
Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, California, USA
schasins@eecs.berkeley.edu

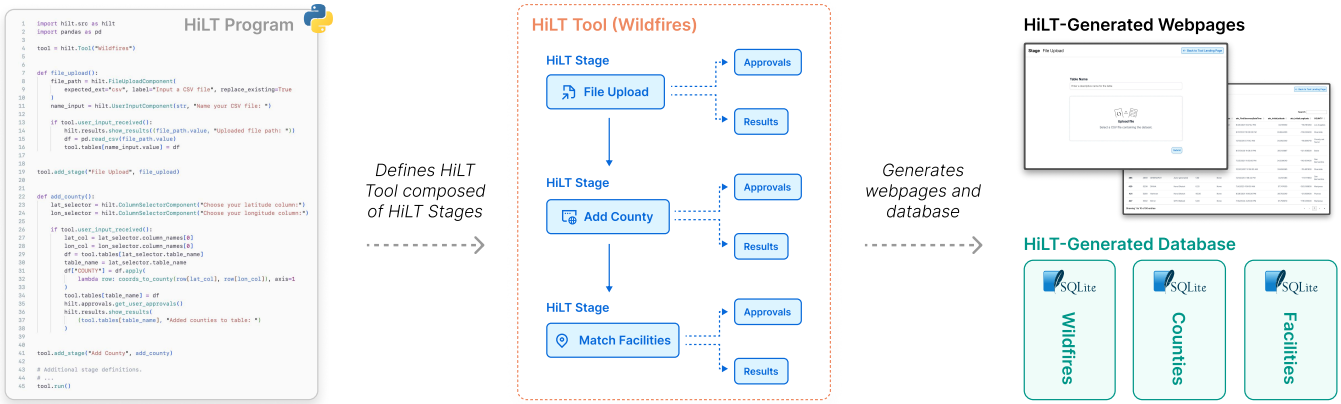


Figure 1: Programming with HiLT, a library for building human-in-the-loop data transformation GUIs. Programmers write HiLT programs in Python (left), using the library to define bespoke data processing `Tools` composed of one or more `Stages`. Using the graph of `Stages` (center), HiLT automatically generates webpages (and a backing database) for guiding end users through the programmer-defined data transformation task (right). HiLT makes it easier for programmers to build GUIs that let users upload and manipulate their own data, especially for settings where different users have data with different schemas.

Abstract

Many domain-specific data transformation tasks are best addressed with custom, task-specific interfaces, especially to accommodate users who are not comfortable with the programming capabilities of open-ended tools like Excel or Tableau. However, building a bespoke GUI for a custom data transformation task is often time-consuming and difficult. This paper presents HiLT, a Python library for authoring human-in-the-loop (HIL) data transformation GUIs for performing data transformation tasks on user-uploaded data. Programmers use HiLT to specify their custom data transformation steps and how they fit together in a set of *stages*; HiLT then generates the corresponding interfaces, backend, and underlying database. In a user study, we compared HiLT with a popular GUI component framework, Streamlit. Participants completed more tasks with HiLT than with Streamlit. We found that HiLT's `Stage` metaphor helped GUI programmers modularize their programs and avoid common state-related pitfalls. In particular, it led programmers to organize their code into clearly delineated data transformation steps, each encapsulated in a HiLT `Stage`, rather than focusing them on compositions of UI widgets, which is common in GUI libraries like React and Streamlit. Our findings suggest a role for GUI-building tools that focus on supporting data transformations by explicitly modeling the transformation steps, rather than centering GUI widgets.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

UIST '25, Busan, Republic of Korea

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2037-6/25/09
https://doi.org/10.1145/3746059.3747737

CCS Concepts

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

Keywords

GUI Programming, Tools for Building User Interfaces, Human-in-the-Loop, Data Work

ACM Reference Format:

Sora Kanosue, Xiaorui Liu, Parker Ziegler, Eric Rawn, and Sarah E. Chasins. 2025. HiLT: A Library for Generating Human-in-the-Loop Data Transformation GUIs. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28–October 01, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3746059.3747737

1 Introduction

Data transformation tasks that require human insight, rather than fully automatable data processing pipelines, are notoriously difficult for programmers to support. Building GUIs for these tasks means building the screens the user will see, but also the underlying data store, the logic for controlling how user inputs affect changes to the data store, and finally the logic for the data transformations themselves. It is even harder if the GUI developer does not know the schema of the user's data in advance, or if the data processing task requires changing the data schema during interactions.

Ideally, a UI for a single data transformation task should look simple. The UI requests inputs required for the task. The inputs vary from UI to UI. Consider a UI for extracting structured data from text [35]; the UI asks for examples of data to extract, or other inputs for refining extractors. Or a UI for feature discovery [36] requests information about dataset structure.

From the user's perspective, the UI may—in fact, should!—look simple. However, implementing such a UI is hard. The programmer must write the code for: (i) the UI, (ii) the underlying database, (iii) the data transformations themselves, and (iv) the logic linking user input to transformation code. Even restricting to tabular data, the schema of the user's data may not be known in advance—and the schema may change during data transformation.

This work addresses how to help programmers build GUIs for human-in-the-loop *data transformation* tasks—data work that involves modifying data schema. We highlight challenges programmers face when: (i) end users need to upload their own data, and the data schema is unknown, will change over time, or varies across multiple users; and (ii) end-users' transformations will change the schema of data in the underlying data store (e.g., a user's actions should trigger a join that produces a new table, or a user's input should be used for converting unstructured text into tabular data).

Why do end users want task-specific data transformation GUIs? For end users, performing data transformations manually is tedious and time-consuming. Prior work shows practitioners delegate these tasks to pools of volunteers, engage "surrogate users" [76], or even forego the task altogether [85]. General-purpose, multi-featured interfaces for data transformation like Trifacta [25], Tableau [68], and BI tools [3, 50, 52] have not solved this problem. These are data transformation GUIs designed for a wide range of tasks, rather than a single fixed task. These open-ended tools are a powerful aid for semi-technical users, but their learning curves are steep. In many contexts, users want "bespoke" GUIs specialized to a particular data task. Task- and domain-specific systems simplify transformations that would be too tedious or technically demanding in general-purpose, open-ended tools.

Why are bespoke data transformation GUIs hard to build? While tools for building data *display* GUIs are common, building data *transformation* interfaces is still hard. Frameworks like Django [17], Airtable [2], and Streamlit [71] let programmers quickly create UIs for viewing data, but we lack comparable support for letting users *upload*, *manipulate*, and *transform* data.

We see a design opportunity for tools that support programmers in writing custom GUIs for human-in-the-loop data transformation tasks. HIL data transformation GUIs are GUIs that satisfy the following three Design Constraints (DCs):

(1) **DC 1: Structured Interface**. The GUI walks the user through their data transformation task, rather than leaving them to explore in an open-ended way, as Excel or Tableau do.
(2) **DC 2: Diverse Data Schemas**. The GUI accepts different input data schemas, so users can bring their diverse data.
(3) **DC 3: Transformation of User-Uploaded Data**. The GUI transforms user-provided data—including by changing the schema or structure of data based on users' interactions—given programmer-defined data transformation code.

HiLT and the Stage Abstraction. To explore making it easier for programmers to author GUIs that satisfy these design constraints, this paper presents HiLT, a Python library for writing custom data transformation GUIs. HiLT's central construct is the `Stage`. Each `Stage` defines a screen the user will see, the inputs they provide on that screen, the logic that controls how inputs are fed into the next phase of a data transformation, and the logic of the transformation itself. This abstraction offers a structured way to decompose the overall implementation into modular components. Given `Stage` code, HiLT outputs a GUI that mirrors the step-by-step flow of the user's task—conceptually, one `Stage` in the program text corresponds to one stage in the user's process of cleaning, wrangling, or otherwise transforming their data.

We do not aim to make new GUIs that surpass the usability of old GUIs. In fact, we do not aim to make new GUIs at all—HiLT does not make previously impossible-to-build GUIs possible. Instead, HiLT makes an existing class of currently hard-to-build GUIs easier *from the programmer's perspective*. HiLT `Stages` support programmers by (i) enforcing that they explicitly decompose each transformation task, and (ii) colocating several related kinds of code together in the program text: specifically, code that collects user inputs from a given screen and the data transformation logic that uses those inputs. Our study revealed `Stages` helped participants reason about program structure, control flow, and state, steering programmers away from the pitfalls associated with standard widget-centric abstractions (Sections 5, 6.4, and 7).

HiLT can express many kinds of GUIs, but it is especially helpful for settings where Design Constraints 1–3 apply: the GUI guides users through a data task (DC1); users upload data with diverse and unpredictable data schemas (DC2); and the GUI backend transforms data during the interaction, even changing the data schema throughout. If a UI handles user input on the front end and data

schema changes on the backend, it is a good fit for HiLT. We demonstrate via case studies that HiLT can express human-in-the-loop data transformation GUIs published in the recent research literature, such as [35, 36]. HiLT is also a natural fit for UIs associated with crowdsourcing—these UIs often have the key traits of being specialized for a specific task, taking human input, and manipulating a database on the backend.

Our evaluative study compared GUI authoring with HiLT vs. a preexisting GUI-component library, Streamlit, focusing on whether programmers can accomplish tasks related to Design Constraints 1–3. For data transformation tasks (DC3), participants completed 2.8× as many tasks with HiLT as with Streamlit. Qualitative analysis of the GUI-programming process revealed that HiLT's *Stage* metaphor affected programmers in a surprisingly broad range of ways. For example, HiLT `Stages` seemed to improve programmers' mental models of the GUIs they built. We discuss implications for the design of future GUI-building tools, beyond data transformation settings.

Overall, this work contributes:

- A set of design criteria for tools that aim to support authoring human-in-the-loop data transformation GUIs.
- HiLT, a set of abstractions, with an accompanying Python library, for expressing GUIs that import, display, manipulate, and transform tabular datasets.
- A study of programmers, evaluating HiLT against Streamlit, a state-of-the-art GUI building framework.
- Qualitative analysis of HiLT's effect on how programmers view their programs. Discussion of how our findings can inform design of future GUI-building frameworks.

2 Related Work

In this section, we survey both human-in-the-loop data transformation systems and programming frameworks for building HIL data transformation tools. To situate HiLT in this prior work, we ground our discussion in a design space divided along two axes:

(1) **Guided vs. Open-Ended User Interaction.** Guided human-in-the-loop systems walk users through *specific* tasks, while open-ended systems provide toolkits for end-user directed analyses. For instance, Excel [51] and Tableau [68] are popular open-ended data transformation systems; users must understand what actions are possible and choose an available action at each step. Examples of guided systems include survey software like Qualtrics [65] and Google Forms [26], which walk users through adding one row of data to a table.

(2) **Data Transformation vs. Data Display.** Many data-backed GUIs are designed around data display rather than data transformation. Users can view and perhaps slightly tweak their data, or they may be able to enter additional rows into an existing programmer-designed table. For instance, making a post on a social media site may add a row to a database table with a fixed programmer-defined schema. In contrast, data work GUIs that permit *data transformation* allow users to: (i) **DT1** insert their own tables, which may not match a predetermined schema, and (ii) **DT2** engage in processes involving querying and modifying the schema of that data. For example, Excel users can bring their own data and change its structure. A data transformation GUI cannot be hard-coded to expect, for example, a particular column with a particular name.

We are most interested in tools for building custom GUIs that support *Guided User Interaction* and *Data Transformation*. To the best of our knowledge, there are no tools specialized for this task, so we focus most of this section on tools that specialize *either* in Guided User Interactions (but not Data Transformation) or in Data Transformation (but not Guided User Interactions).

2.1 Data Analytics Software: Data Transformation (But Not Guided User Interaction)

Prior work both in academia and industry has contributed a large number of GUI-based, exploratory data transformation and analysis systems [3, 25, 33, 37, 38, 42, 52, 54, 56, 68, 69, 95]. Many of these systems aim to support flexible, open-ended data transformation and cleaning and emphasize low- or no-code interactions with data. For example, business intelligence (BI) tools like Tableau [54, 68], Trifacta [25], and Microsoft PowerBI [52] allow users to filter, aggregate, and visualize data from multiple sources without programming or database management expertise. OpenRefine [60] supports sophisticated cell-, column-, and table-level operations on tabular datasets through GUI-based interactions. While these systems expose small data transformation DSLs or query languages to augment GUI functionality (like PowerBI's Data Analysis Expressions [49] and OpenRefine's General Refine Expression Language [59]), their core interaction models are GUI-based. Spreadsheet-based technologies like Microsoft Excel [51] and Google Sheets [27] also fall into this category.

Another type of tool leverages programming-by-demonstration and programming-by-example to infer and synthesize data transformations from user interactions [18, 29, 39]. In contrast to BI-style tools—which hide program representations of data transformations from users—these systems center the synthesized program as a primary output.

HiLT-generated tools share these systems' focus on data transformation as a primary task. However, these tools do not walk the user through a specific task; they rely on the user to be become an expert in the functions of a given tool—Excel or Tableau or another—and to decide how to apply those functions to achieve their data transformation goal. In contrast, a HiLT-generated GUI guides users through a *specific* transformation task defined by the HiLT programmer.

2.2 GUI-Building Software: Guided User Interaction (But Not Data Transformation)

Expert-Programmer GUI-Building Tools for Data Display. While not explicitly designed for data transformation, many web programming frameworks pair view-templating capabilities with abstractions for remote data access and persistence, making them popular choices for developing bespoke human-in-the-loop systems. Django [17], Ruby on Rails [66], and Next.js [80] are three popular examples; each provides APIs for managing data and application state across client, server, and database. These systems are capable of expressing a wide range of user interfaces, but recall that we expect

data transformation interfaces to offer users the opportunity to: (i) **DT1**: add their own tables, with arbitrary column structures and (ii) **DT2**: engage in interactions that require querying those tables, linking tables, generating new tables, and generally transforming the schema of their data. Django or Ruby on Rails can straightforwardly support authoring interfaces in which an end user can add a row to a fixed, programmer-designed table, but authoring interfaces that meet the above goals is difficult. Of the GUI-building tools, these are the most feature-complete and flexible, but building bespoke data transformation GUIs with them is still difficult.

Novice-Programmer GUI-Building Tools for Data Display. A number of tools aim to lower the complexity of building GUIs to make the task approachable to novice or intermediate-level programmers. One such framework is Streamlit [71], which offers a library of pre-made components for building simple web apps using Python scripts. Although tools like Streamlit support rapid development of data display interfaces, the challenges around supporting data *transformations* in an application backend remain. Other entries in this space which support rapid development of data display interfaces include Dash [64], Panel [31], Mavo [81, 82], and others [5, 12, 14, 16, 34, 40, 63, 75, 78, 90]. As with the tools above, these tools make it easier to author data *display* interfaces rather than data *transformation* interfaces.

Low-Code GUI-Building Tools for Data Display. Also in this space are commercially available low-code GUI-building tools like Retool [67], Budibase [10], and Airtable [2]. Some of them, such as Retool, meet subgoal **DT2**, by allowing GUI-builders to write arbitrary SQL and JavaScript to manipulate data. These tools fail to meet subgoal **DT1**, since they enforce that GUI-builders hardcode their interfaces and user interactions around fixed data schema. For example, Airtable enforces that GUIs funnel user data into programmer-designed relations with fixed, pre-specified schema, making the resulting applications inflexible to diverse user data formats. Hence, as with the tools above, these tools make it easier to author data *display* interfaces rather than data *transformation* interfaces.

Augmenting Computation Notebooks with Data Displays. Some prior works have used computational notebooks as a platform to guide data practitioners through data-related tasks. For example, mage [41] provides a framework for developers to create data widgets represented simultaneously in code and within a GUI, facilitating interactions like data selection and exploration. PI2 [15] synthesizes data display interfaces from example SQL queries. A number of other works augment computational notebooks to facilitate tasks like data exploration [13, 20, 21, 23, 47], data summarizing [46, 48], ML model development [11, 43, 44, 55, 61, 70, 84, 87, 94], and data analysis [28]. Although these systems support data transformation tasks, these interventions are targeted at programmers, and the end user ultimately has to write code. In contrast, HiLT-generated UIs are usable by non-programmers.

Easing GUI Authoring for Non-Data GUIs. More broadly, many works introduce GUI-building systems for GUIs without underlying data stores. A number of tools have applied program synthesis or machine learning to generate GUI designs, improve hand-written GUIs, provide design guidance, or test GUIs [8, 9, 45, 73, 74, 79, 91, 92]. Another line of work emphasizes allowing users to reprogram GUIs [6, 7, 19] or augment visual programming environments [4, 24, 32, 57, 93]. Although these tools aim to make GUI authoring easier in a variety of domains, these tools do not handle data transformation tasks.

3 HiLT Walkthrough and Interaction Model

In this section, we provide an overview of HiLT's interaction model (Figure 2) by working through a motivating example. Throughout this work, we will use the term **programmer** to refer to someone using HiLT to write a tool, and we will use the term **user** to refer to someone using a HiLT-generated tool. In this motivating example, Marin is a user, and Navi a programmer.

Marin, a journalist, is trying to perform entity matching across years of police roster data stored in spreadsheets. Each spreadsheet contains several thousand entries and has different column schemas. The spreadsheet structures are different from between departments and change annually even within a department. Marin has been struggling with manually inspecting the spreadsheets in Excel to figure out how individuals move between departments. In the past, she's tried working with a programmer to fully automate a similar data transformation task, but this took multiple months of back and forth emails, involving lots of manual work to confirm or deny particular matches, then getting back to the programmer to request new changes and restarting the process. She does not want to sign up for three months of back and forth this time around. In any case, the data is full of small typos, missing cells, and other errors, and many individuals in the data share the same name, so she is pretty sure she cannot use a fully automatic approach to make the decisions this time. She is happy to inspect the difficult cases one by one. Still, wrangling 40 different Excel windows is not getting her the desired output, a list of individuals who appear across multiple spreadsheets. Frustrated, Marin contacts Navi, the data scientist on her team, and the colleague with the most programming experience. Navi uses HiLT to construct an interface that Marin can use to accomplish this data transformation task going forward.

3.1 Programming in HiLT

To construct an interface using HiLT, Navi begins by instantiating a HiLT `Tool` (Figure 3, Line 2). HiLT programs define `Tools`, which are composed of `Stages`; each `Stage` guides users through one step of a human-in-the-loop data transformation task. She calls `tool.run` to launch the HiLT engine (Figure 3, Line 26).

Navi wants to allow Marin to upload her datasets as CSV files and give them names. She can do this by defining a `Stage` which accepts a CSV file and a name from Marin, and saves the contained data under the given name. To start the `Stage`'s *Stage Defining Function* (SDF), Navi defines a new Python function and registers it as a new `Stage` (Figure 3, Line 10). To construct an interface that accepts a file and a string, Navi instantiates a `FileUploadComponent` and a `UserInputComponent` (Figure 3, Lines 4-5).

Navi next wants to save uploaded datasets in the `Tool`'s underlying database. She reads the CSV file into a `pandas DataFrame` [62] (Figure 3, Line 7), using the `FileUploadComponent`'s value attribute to access the file. SDFs execute twice every time a `Stage` is used. In the first run, `tool.user_input_received` is false, and

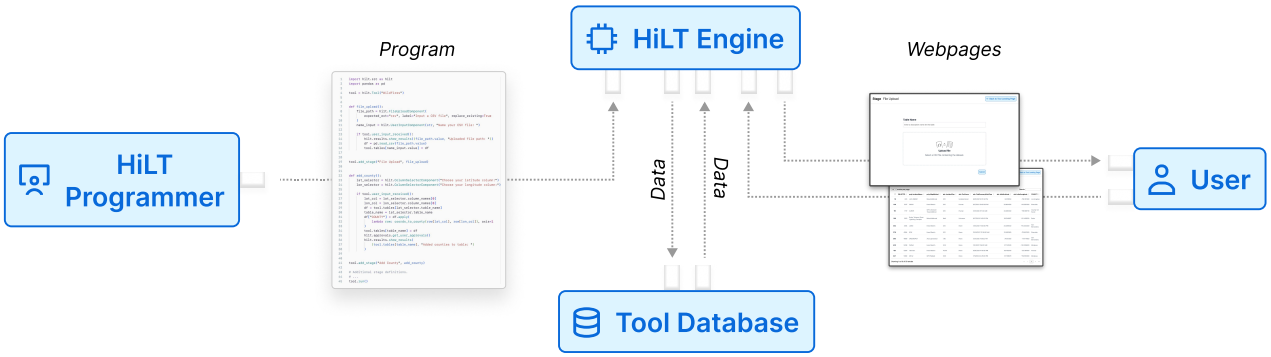


Figure 2: HiLT's interaction model. A programmer writes a HiLT program. When the HiLT Engine runs a HiLT program, it produces a set of webpages, which it displays to the user. The user interacts with the webpages to indicate how data in an underlying data store should be transformed. When these interactions are fed back into the HiLT Engine, the HiLT Engine and HiLT program are together responsible for generating the next webpage to show to the user. This is the core user interaction loop, with the tool generating webpages in response to user inputs, and the user generating new inputs in response to new webpages.

HiLT generates a webpage based on what `Components` are instantiated; in the second run, `tool.user_input_received` is true. Since Navi wants her program to access the file's data only *after* it has been uploaded, she gates the file access code with a call to `tool.user_input_received` (Figure 3, Line 6). HiLT ensures that any code which is marked as user input-dependent in this way is run only once user input has been received.

Now, Navi can save and display the `DataFrame`. She uses the `Tool`'s `tables` attribute to save the `DataFrame` under the name in the `UserInputComponent`'s `value` field (Figure 3, Line 8). `Tool`'s `tables` attributes are Python `dict`-like, and map table names to the table contained in the `Tool`'s underlying database. Navi wants Marin to be able to see the uploaded dataset committed to the `Tool`'s underlying database. Navi accesses the table using `tool.tables` and the table name, and passes it to `results.show_results`, along with a label (Figure 3, Line 9).

Navi next builds an interface to Marin through some basic entity matching (for simplicity, using just a single column). The interface will ask Marin to choose two tables, and a column from each table to use for matching; she uses two `ColumnSelectorComponents`. `ColumnSelectorComponent` `values` contain the chosen tables filtered to the chosen columns, so Navi uses `tool.tables` to retrieve the full `DataFrame` for each table. Finally, `pd.merge` joins the two tables, and the SDF calls `results.show_results` to display the identified matches.

Navi knows there are multiple individuals in Marin's datasets who share the same name. Since there is no automatic way to detect which individual is which, Navi knows she'll need Marin to make the final call for these cases. She opts to let Marin manually inspect the proposed matches before they are committed to the `Tool`'s underlying database, by calling `approvals.get_user_approvals` (Figure 3, Line 23). This causes the HiLT GUI to show Marin a webpage in which she can approve or reject new database rows, either in bulk or individually, before actually committing them to the database.

3.2 Using a HiLT-programmed `Tool`

Marin can now interact with Navi's `Tool` by navigating to the `Tool`'s landing page. To upload a new dataset, she navigates to the `csv_upload_and_name` `Stage`, a webpage with a file upload widget.

After uploading a few datasets, Marin moves on to the entity matching `Stage`. When Marin clicks on the `Stage`, she is prompted to choose two datasets and a column from each. She starts with the data from the last two years, choosing to match based on the name column from each dataset—the first column in the most recent table, and the third column in the older table. Once she submits her inputs, HiLT performs the join, then displays the matches for her to approve or reject. She finds a few cases where a match is suggested, but the individuals' employment start dates make it clear they are two separate individuals. After approving and rejecting based on the name matches, she enters the matching stage again. This time she chooses the start date column for each table. Based on the start dates, she finds a few matches where the names were listed differently for the different years—e.g., adding a middle initial—but confirms the matches should be added anyway.

Because of HiLT's focus on supporting interactions regardless of data schema, Marin can repeatedly use Navi's `Tool` for tables of data from the past ten years, even though they all use slightly different formats, without having to rely on Navi to adapt the interface to each one.

4 Case Studies

In this section, we briefly present five HiLT case studies, describing HiLT implementations of human-in-the-loop data transformation GUIs introduced in prior research. We also provide the source code for several additional HiLT examples in the supplemental materials, including a GUI for tracking entities of interest across collections of PDFs, a GUI for aligning rosters of varying formats, and a basic assignment-grading GUI (lightly inspired by GradeScope [77]).

```
import pandas as pd

tool = Tool('demo')

def csv_upload_and_name():
  table_name = UserInputComponent(str, label="Enter table name:")
  csv_file = FileUploadComponent('csv', label="Upload a csv file:")

  if tool.user_input_received():
    df = pd.read_csv(csv_file.value)
    tool.tables[table_name.value] = df
    results.show_results((tool.tables[table_name.value], "Created table: "))

tool.add_stage('csv_upload_and_name', csv_upload_and_name)

def simple_column_matcher():
  c1 = ColumnSelectorComponent("Pick column containing names from one dataset")
  c2 = ColumnSelectorComponent("Pick column containing names from another dataset")

  if tool.user_input_received():
    c1_full_table = tool.tables[c1.table_name]
    c2_full_table = tool.tables[c2.table_name]
    matches = pd.merge(c1.value, c2.value, how='inner', left_on=c1.column_names,
      right_on=c2.column_names)

    if len(matches) == 0:
      results.show_results("No matches found")
      return

    tool.tables["Matches"] = matches
    approvals.get_user_approvals()
    results.show_results(matches)

tool.add_stage('simple_matcher', simple_column_matcher)
tool.run()
```

Figure 3: Code for a simple HiLT `Tool` with a file upload `Stage` and a data joining `Stage`. The first SDF instantiates a `UserInputComponent` and a `FileUploadComponent` for accepting user inputs. The program saves user-uploaded CSV data to the underlying database using the `Tool`'s `tables` attribute. The call to `approvals.get_user_approvals` means that HiLT will intercept database modifications and present them to the user for approval before committing them to the database; the user can approve in bulk or row-by-row. (Some imports elided for brevity.)

4.1 WannaDB

WannaDB [35] is a human-in-the-loop system designed to extract information from unstructured text corpora into tabular formats. WannaDB first makes initial guesses about each document in a dataset. Next it launches an iterative feedback cycle in which the user provides feedback on individual guesses by either updating or accepting the proposed values. WannaDB uses this feedback to update the other guesses. Using HiLT, we have reimplemented this core interaction loop in 167 lines of code. Our program contains three `Stages`: corpus upload, attribute feedback, and attribute extraction. Note that our 167-line version does not include less central parts of WannaDB's UI, such as exporting found attribute values to CSV format and executing ad-hoc SQL queries over extracted attribute values; to extend the HiLT GUI with peripheral features like these, we would add additional `Stages` that call code from the existing WannaDB implementation—e.g., a new `Stage` with a single call to WannaDB's CSV export function. Our implementation already relies on WannaDB's existing implementation of text extraction, demonstrating that HiLT programs can interoperate with other systems, including tooling for manipulating non-tabular data.

4.2 HILAutoFeat

HILAutoFeat [36] leverages human-in-the-loop feedback to guide automated feature discovery for tabular data. It operates on collections of structured data; given a particular table and column of interest, it identifies potentially related columns throughout a collection of data tables for use in training machine learning models. A partial HiLT reimplementation of HILAutoFeat recreates the core human-in-the-loop process and takes 99 lines of code. Over four `Stages`, users can upload a repository of datasets, choose a table and column of interest, specify hyperparameters to compute join trees across the repository, and select and inspect a join tree for further external use. We describe the HiLT implementation as partial since it does not include interactions for removing features from chosen datasets or producing the resulting augmented tables; as in the WannaDB case, these would be simple additional `Stages` that call a single existing HILAutoFeat function.

4.3 NICER

NICER [22] is a neural network-based system designed to apply aesthetic edits to images. It leverages human-in-the-loop feedback in order to guide the edits being made to the image. We have reimplemented the NICER interface in a 179-line HiLT program that allows users to upload a photograph and run an epoch of NICER's underlying neural network, providing adjustments between iterations of the interaction. This example again showcases HiLT's ability to express GUIs that operate over non-tabular data types, such as images, text, or PDFs.

4.4 Coretab

Coresets [53] are algorithmically determined subsets of structured datasets chosen with the intent of speeding up certain machine learning tasks. Coretab [30] is a system built on a coreset selection algorithm with a human-in-the-loop feedback cycle which allows users to iterate on initially chosen coresets. Its interaction model centers on calling functions exposed by Coretab in order to initialize, visualize, evaluate, and modify coresets within Jupyter notebooks. We have wrapped Coretab in a HiLT interface that supports this same interaction model. This interface allows users to upload datasets and compute initial coresets, evaluate existing coresets' statistical performance, and generate new coresets by adding and removing features from existing coresets. Our reimplementation is written in 145 lines of code.

4.5 Interactive Table Search

A number of previous works in the database community [83, 86, 88, 89] investigate the problem of searching for a user's "favorite rows" from a particular table. These works typically present human-in-the-loop algorithms, wherein the user's preferences are learned by asking them to compare and choose between two or several table rows at a time. Using HiLT, we have implemented an interface for one of these systems [86] in 85 lines of code. Our interface reimplements a basic web interface that the original authors used to conduct a user study comparing four preference-learning algorithms. It allows the user to upload their dataset of interest, then repeatedly presents the user with pairs of rows from the dataset and learns their preferences until it has determined their preferred row. The HiLT interface uses only one of the four preference-learning algorithms, since the others require pre-trained machine learned models that the authors did not open-source; however, the user-facing interaction loop remains the same across the algorithms. This example again demonstrates how external Python libraries can be integrated directly into programmer-defined transformations within HiLT `Stages`.

5 HiLT Design and Implementation

In order for its output GUIs to align with Design Constraint 1 from Section 1, HiLT is designed to break data transformation tasks into individual *stages*, so that users are directed to separate GUI pages for each step of the data transformation process. The HiLT programmer specifies how the data transformation task is broken down into stages by defining a Stage-Defining Function (SDF) for each step. Within each SDF, the programmer specifies the content of the interfaces that the user will utilize for the given `Stage`, as well as how inputs provided by the user will be handled. Since HiLT is implemented as a Python library, programmers can write arbitrary Python to specify how data is transformed and use other libraries in their code, as demonstrated by the case studies in Section 4.

Under the hood, running a HiLT program generates a locally hosted AIOHTTP [1] web application, backed by a SQLite [72] database. This database is exposed to the programmer as a dictionary-like object mapping table names to `pandas DataFrames` [62], allowing programmers to store and manipulate user data (Design Constraints 2 and 3). Each programmer-defined `Stage` makes a number of webpages available to the programmer:

- An initial webpage that collects user input,
- An optional approvals page allowing the user to inspect the outputs of programmer-specified data transformation before they are committed to the underlying database,
- And an optional results page allowing users to see what changes have been propagated into the underlying data store.

The possible flows through these webpages are depicted in Figure 4. The HiLT engine generates the contents of a `Stage`'s initial webpage dynamically by executing the SDF and rendering any HiLT `Components` instantiated during execution (see Appendix Section A.2 for more). This allows HiLT programmers to construct GUIs that depend dynamically on a user's actions, current data, or other relevant inputs, as in the simple example in Figure 5. The HiLT programmer also specifies the contents of any results pages they choose to include (see Appendix Section A.5). In contrast, if the HiLT programmer chooses to make an approvals page available to the end user, the HiLT engine generates its contents by automatically intercepting changes to the database before they are committed.

As mentioned above, a HiLT programmer specifies data transformations by writing Python code to manipulate the data at hand within a stage's SDFs. Data transformations, however, are usually dependent on the schema or structure of the data being handled, which the programmer may not know ahead of time. For example, in order to fully specify a join between two tables, a programmer will need to know the column name or names being used as the criteria for the join, or even the names of the tables that they would like to join. A key part of HiLT's design is supporting cases like these: rather than expecting that the programmer can fully specify

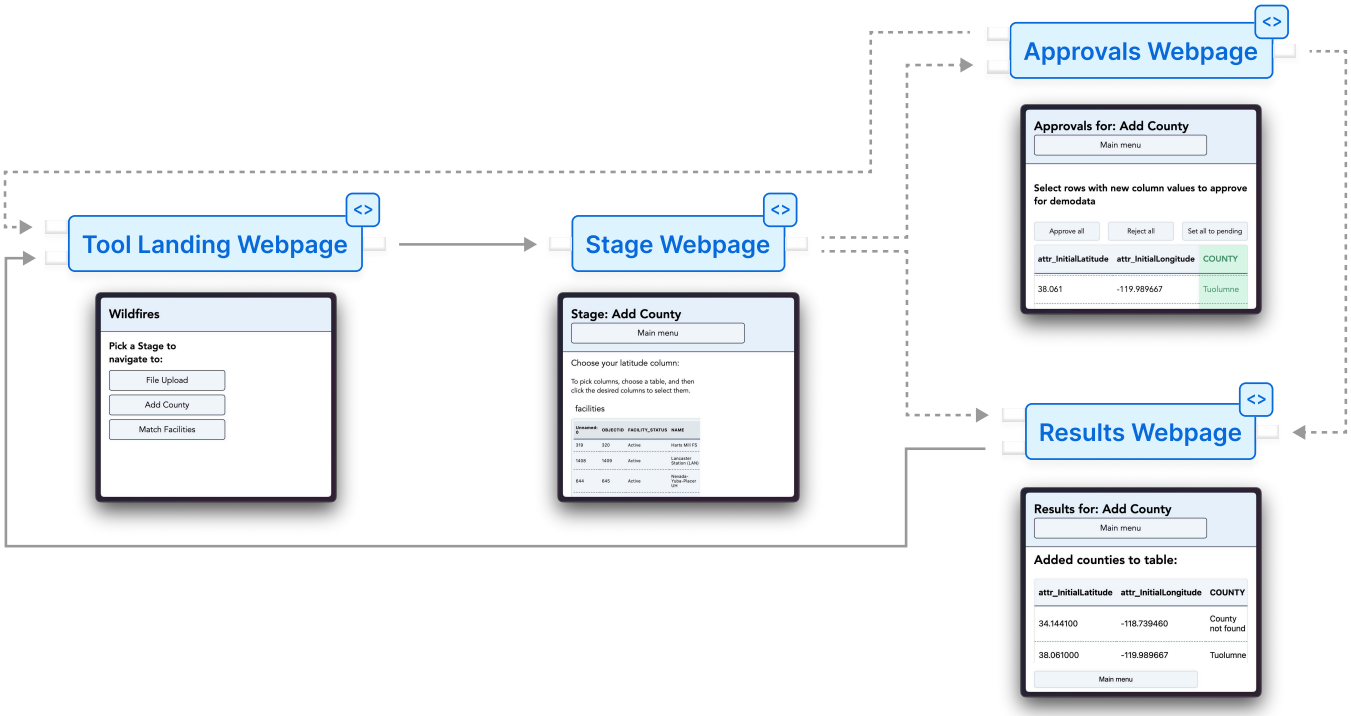


Figure 4: Webpages (optionally) associated with a single HiLT `Stage`. Above, a node represents a webpage, and an edge represents a transition between webpages. A solid line indicates there is only one outgoing edge from a node, while a dashed line indicates there are multiple outgoing edges. When there are multiple outgoing edges, the programmer controls which edge the user takes; the HiLT program can use user input to choose the appropriate outgoing edge. Because this decision is made at run time, there may not be a single, fixed flow diagram associated with any given HiLT `Stage`.

their data transformations, we allow them to parameterize them with user input provided to the generated HiLT GUI. In the above scenario, a HiLT programmer can use `ColumnSelectorComponents` to collect user input as to which columns to merge on and which to retain from each table being joined. This allows HiLT programmers to generate GUIs which can flexibly handle diverse data schemas, while still giving the programmer precise control over the nature of the transformations.

Any UI that accepts human input on the front end and handles database schema changes on the backend is HiLT-able. If a programmer can specify a custom data transformation in Python, they can feasibly use HiLT. This is because HiLT is not primarily a library of GUI widgets—it is an architecture that structures GUI codebases around the stages of the user's workflow. While one could implement a data transformation UI in Streamlit or other generic GUI-building tools as well, our argument is that the program may be easier to write in HiLT, because of how `Stages` enforce problem decomposition, modularity, and a clear mental mapping between parts of the program text and parts of the user workflow.

The HiLT implementation and documentation are open source and available at https://github.com/skberkeley/HiLT/.

6 Evaluation

Our evaluation assesses how effective HiLT is in supporting programmers in building GUIs that meet our three design constraints—guiding the user, importing data (in particular, varied user-inputted table structures), and transforming data (in particular, changing data schema based on user interactions). We conducted a within-subjects user study with 16 participants (P1 - P16). As a comparison baseline, we used Streamlit [71], based on findings in a formative study that participants were much more successful with Streamlit than with Django. All participants used both HiLT and Streamlit.

6.1 Participants

We recruited 16 participants with at least a basic level of prior experience with Python (the host language for both HiLT and Streamlit) via a participant recruitment service run by our institution, through mailing lists, and by word of mouth. The participants' reported years of programming experience ranged from 1–12 years, with a mean of ~5.3 years. Participants were compensated, and the study was approved by our institution's Institutional Review Board.

6.2 Tasks

We asked participants to build the same GUI with both HiLT and Streamlit. Half used HiLT first, and half used Streamlit first. All

```
def csv_upload_and_name():
    if len(tool.tables) == 0:
        table_name = UserInputComponent(str, label="Enter table name:")
        csv_file = FileUploadComponent('csv', label="Upload a csv file:")
    else:
        TextComponent("You've already uploaded a table!")
...
```

Figure 5: A simple HiLT snippet demonstrating that programs can control dynamically what widgets to render. Generating GUI pages at run time allows programmers to change the GUI structure in response to user input.

participants had one hour with each framework to make as much progress as possible in creating a custom GUI for exploring data about wildfires and wildland fire protection facilities in California. We provided each participant with 40 datasets with which they could test their output GUIs. To complete the GUI in full, participants would have to complete five tasks. We did not expect programmers to complete the full GUI—rather, we aimed to observe what portion programmers could complete with each framework in the hour available. The five tasks were:

(1) **File upload**: Allow users to upload CSV files, persisted in an underlying data store, allowing users to use the data in future interactions without re-uploading it.
(2) **Dataset augmentation—Add county name**: Allow users to add county name columns to wildfire datasets, based on the existing latitude and longitude columns, without making assumptions about the structure of uploaded data. The county column should be persisted with the rest of the dataset. We provided a snippet of Python code to convert from coordinates to county name.
(3) **User approvals of dataset augmentation**: Allow users to inspect and verify the columns created in Task 2. Users should be able to accept or reject each new value one by one or all at once, and only accepted values should be persisted with the dataset.
(4) **Dataset matching—Match fires and facilities by county name**: Allow users to match each fire protection facility with wildfires in the same county. Users should be able to select both the wildfire and fire protection facility datasets being matched. Users should also be able to choose the columns from each table to include in the table of matches.
(5) **User approvals of dataset matches**: Allow users to inspect and verify the datasets created in Task 4. Users should be able to accept or reject each new row one by one or all at once, with only accepted rows persisted.

All five tasks combine elements from two or more of the three design constraints outlined in Section 1—Guiding the User, Importing Data, and Transforming Data. We therefore divided each task into subtasks, each associated with only one of the design constraints. For example, subtask 4-4 is a subtask of Task 4: "Does the user have control over the criteria used to match the datasets?" This particular subtask connects to Design Constraint 3, Transforming Data. See the full table of subtasks in Table 3 of the Appendix.

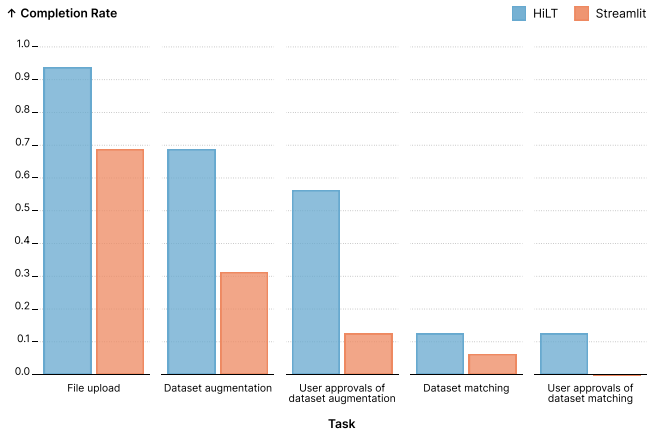


Figure 6: Completion rate for each task. For all tasks, participants were more successful with HiLT than with Streamlit. Two of the 16 participants successfully completed all five tasks with HiLT, while none finished with Streamlit. Task 1 was completed by four more users with HiLT than with Streamlit, and for every subsequent task, the number of successful HiLT completions was more than double the number of Streamlit completions.

6.3 Sessions

We began each user session with an introduction and the informed consent process. To mitigate learning effects, we balanced the order in which HiLT and Streamlit were used across participants. We then asked them to complete the five tasks using each framework. We gave participants links to the official documentation for each framework, and informed them that they were free to consult external resources like search engines or large language models (LLMs) to assist them. After attempting the tasks with both frameworks, they completed a survey collecting demographic information, and qualitative feedback on their experiences using each framework.

6.4 Results

Over 50% of HiLT users completed three GUI tasks, while only 11% completed three GUI tasks with Streamlit. Participants completed 2.8× more subtasks related to Transforming Data with HiLT than with Streamlit. Qualitatively, participants appeared to benefit from HiLT's Stage structure. It provided useful constraints on output GUI

structure and control flow, two factors that hampered participants when using Streamlit.

6.4.1 Quantitative Results.

Completion Rate By Task. Figure 6 shows the completion rates for each of the five tasks. Recall that we gave participants only an hour with each tool, so we did not expect participants to complete all tasks in the time available. Nevertheless, two participants still completed the entire GUI under the HiLT condition (none did under the Streamlit condition).

The furthest task any participant was able to complete with Streamlit was task 4. We find this particularly notable because, to support ecological validity, we did not restrict participants' access to online resources, including resources like LLMs that can generate programs. Because Streamlit was introduced in 2019, LLMs like ChatGPT [58] can output functional Streamlit code. Due to HiLT's novelty, LLMs cannot—to our knowledge—write HiLT code. Of the 16 participants, 8 participants successfully used an LLM to obtain functional Streamlit snippets, while none did so for HiLT.

Completion Rate By Subtask. Figure 7 shows the completion rates for each individual subtask. The first digit in each subtask name indicates its parent task (e.g., subtask 3-3 is the third subtask within Task 3). As with tasks, we see that subtask completion rates were higher with HiLT than with Streamlit. For subtask 1-2, participants completed it at the same rate with both tools; for all other subtasks, participants completed each subtask at a higher rate with HiLT than with Streamlit.

Completion Rate By Design Constraint. Figure 8 shows the subtask completion rates grouped by our three design constraints. Overall, participants accomplished 2.0× more Guiding the User subtasks with HiLT, 1.3× more Importing Data subtasks, and 2.8× more Transforming Data subtasks. These differences were statistically significant (Guiding the User: $p = 0.0019$, Importing Data: $p = 0.045$, Transforming Data: $p < 10^{-65}$).

6.4.2 Qualitative Results.

Method. The second author coded the post-session survey reflections, 28 hours of screen recordings, and the participants' programs. Written notes taken during the study were used for P13 and P16, who opted out of recording. These codes were formed into top-level themes through bi-weekly discussions with the whole author team.

HiLT's `Stage` *metaphor shaped the programming process.* Although we originally designed the `Stage` metaphor to enforce that generated GUIs must guide end users through a task's steps, many participants expressed that they appreciated how HiLT's `Stages` helped them organize their program structure around the structure of the underlying data transformations. Participants mentioned "modularity" (P4, P15), "organization of code" (P11, P10), and "division of workflow" (P13) when describing how `Stages` affected their programming. Our analysis revealed that participants used this language to refer to many aspects of their user experience.

`Stages` may have helped programmers develop a mental model of the output GUI's structure. Participants struggled to build a concrete understanding of how Streamlit produces GUIs. For example, P11's Streamlit GUIs often had erratic and unexpected behaviors, such as widgets disappearing during interactions or conditional rendering occurring in unexpected places, suggesting a lack of a full understanding of the relationship between the Streamlit program and the GUI. Much of their session was spent attempting to reorganize the program such that the GUI structure satisfied their goals.

In contrast, participants commented that `Stages` (and the associated flow of webpage connections) gave them a valuable framework to make predictions about how the output GUI would be structured. Returning to P11, this assisted in how they understood their program's behavior and its connection to what was displayed: "[With HiLT] I thought, [it's] very clear what will happen when, different features are separated into different places ... with the `Stage` system; Streamlit made me feel like 'everything is happening all at once' which made it hard to track when things would happen." P10 shared a similar perspective: "I liked how user friendly the interface it created was[.] ... The way it was organized ... was very straightforward and easy to navigate." Interestingly, P10—who saw HiLT before Streamlit—decided to structure their Streamlit program into HiLT-style stages, with separate webpages for each task. It is unclear if this was a case of learning effects or if they would have decomposed the tasks regardless of whether they used HiLT first or second.

`Stages` helped clarify control flow. Commenting on control flow and data manipulation in HiLT, P15 emphasized that "I liked ... the flow of render, then receive user input." While Streamlit has explicit support for receiving user input, its support operates at a per-widget granularity (rather than a stage granularity). P15 struggled throughout their session to reason about control flow and about how values get passed between different phases of a user's interaction. In an attempt to make state and control flow more transparent in Streamlit, they eventually resorted to adding a sidebar into the Streamlit output GUI that simply displayed the values they read from various GUI widgets. `Stages` may have helped users avoid these issues by imposing a fixed structure for control flow.

`Stages` allowed for the modularization of already written code, templates found in documentation, and LLM-generated code. All participants used prewritten code from documentation, the web, or LLMs in their sessions. Recall that HiLT `Stages` separate code used to render components from code used to handle user input. This separation of concerns made it easier for participants to integrate snippets from other sources. For example, P4 demonstrated this by slicing different sections of `Stages` from multiple different tasks, and composing the snippets together to create a new interaction.

Many participants (P1, P2, P4, P5, P6, P8, P9, P11, P14, and P16) opted to use LLMs for Streamlit. Participants often attempted to generate Streamlit programs to complete an entire task rather than more granularly, such as only generating the data transformation or only composing widgets. Even though, to our knowledge, no LLMs are trained on HiLT code or can generate HiLT code, participants P4, P9, P14, and P15 were successfully able to integrate framework-agnostic snippets of LLM generated code into their HiLT programs. In contrast to Streamlit programmers, HiLT programmers did request finer-granularity program components, such as requesting a specific data transformation. This last behavior may not be indicative of an underlying HiLT characteristic—if LLMs start being able to generate HiLT code, we cannot predict if this

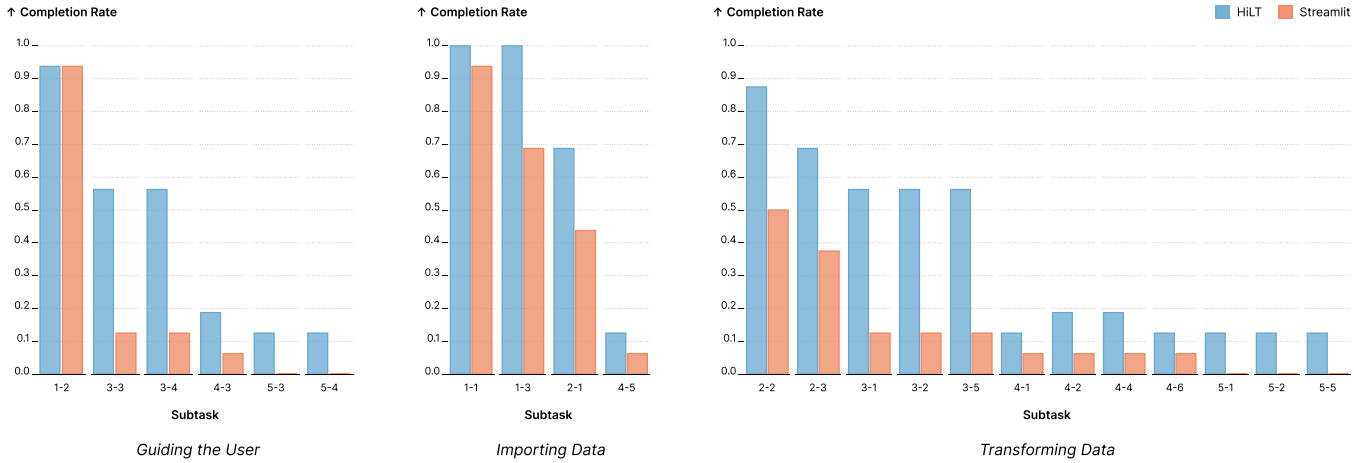


Figure 7: Subtask completion rates for each framework, grouped by our three design constraints. Participants were consistently equally or more successful with HiLT than with Streamlit. For Data Transformation subtasks in particular, participants completed 2.8× more subtasks with HiLT than with Streamlit. Subtasks and their descriptions can be found in Table 3 in the Appendix.

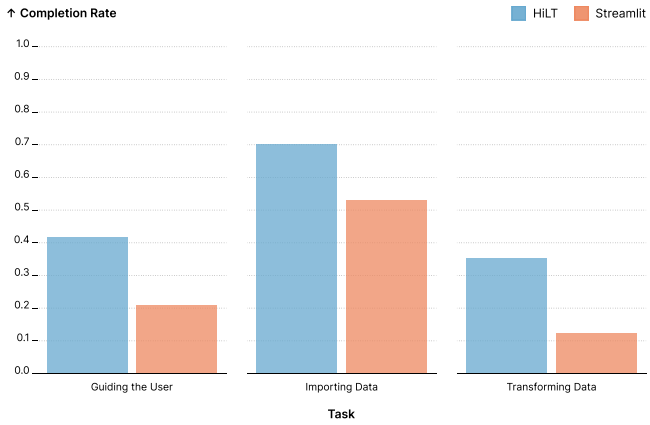


Figure 8: Subtask completion rate by design constraint. Overall, participants accomplished 2.0× more Guiding the User subtasks, 1.3× more Importing Data subtasks, and 2.8× more Transforming Data subtasks with HiLT compared to Streamlit.

behavior will persist. We do not speculate about whether modularizing prewritten code into HiLT stages' component rendering vs. input processing phases is desirable, but merely observe that programmers completed it successfully, and thus did not seem to find the two-phase model an obstacle to using prewritten code.

`Stages` meaningfully constrained programmers. While HiLT's `Stage` abstraction sometimes constrained programmers in desirable ways—e.g., ways that supported their mental model of their programs, or ways that made the output GUI easier to navigate—some participants exhibited confusion about the allowable structure for a `Stage`. The most common way that participants attempted to diverge from HiLT's prescribed stage structure was to nest a `Stage` inside of another `Stage`. Participants P5, P6, P12, and P15 each attempted to nest `Stages` at some point during their sessions. Because HiLT enforces that each `Stage` is associated with a single Stage Webpage (see Fig. 4), this produced a HiLT-specific error message. After the error message, all four participants successfully restructured their programs to resolve this issue.

Reflecting on the potential drawbacks of HiLT's `Stage` structure, P13 commented on cases where it could force programmers to reorient how they guide certain interactions: "[S]plitting rendering ... and waiting for user input can make some operations a lot more complex. ... I wanted to populate a select field with columns from a `DataFrame` to select the field to merge on, but would have needed to use a separate Stage to do this." Although P13 was able to complete the entire GUI with HiLT, this offers an illustrative example of how the structure imposed by the `Stage` metaphor constrained programmers both in useful ways—having to break tasks down into subtasks in a way that promoted modularity—and in confusing or undesirable ways—having to break a task down into subtasks even when they originally wanted an interaction to appear on a single page.

Participants' understanding of similar library constructs varied based on the surrounding context. In many ways, Streamlit's and HiLT's data persistence functionality was very similar. Both offered a dictionary-like data structure for storing variables. However, participants struggled significantly more to persist data with Streamlit. Quantitatively, 5/16 participants failed to produce a Streamlit program which persisted data in Task 1, while 0 failed using HiLT. Participants P7, P9, P10, P11, and P15 identified that the issue was not that they didn't know about Streamlit's storage construct, or even an issue with the storage construct itself; rather, they didn't understand *other* aspects of how Streamlit managed state, so even getting to the point of using the persistence functionality was a barrier. In particular, Streamlit reruns the entire program in response to any user interaction with a displayed widget. P9 commented: "The data store and refreshing was weird, I'm not sure how it worked

fully." Many participants expected that if a user interacted with input A, then with input B, then their program could check whether both A and B had associated values. In fact, the user's interaction with input B would trigger a re-run that would wipe out input A's value. This reinforces the idea that one of the core benefits of a GUI-building programming model is that it can (i) structure programmers' conceptual model of the underlying GUI-generation system in a way that supports programmers' intuitions about how to use the framework and (ii) structure programmers' code in a way that supports programmers' conceptual model of their particular GUI. Our findings about storage capabilities emphasize the other side of the coin—that a programming model that gives users the wrong intuitions can undermine their success. Above all, however, it suggests that the success of a particular GUI library construct is secondary to the importance of the overall mental model of GUI generation that the library teaches or encourages. This goes some way to explain why so many of our findings ultimately distill down to how Stages changed how HiLT programmers thought about GUI generation.

7 Discussion

GUI frameworks encourage programmers to think in terms of widget composition, but that may be better for GUIs where users consume information, rather than GUIs where users accomplish other tasks. Many current GUI-building frameworks—e.g., React, Streamlit—feature APIs designed to model the subcomponents of a given GUI screen. In our user study, we observed that this encouraged participants to structure their programs in ways that resemble the structure of the screens or pages being described. This design makes sense in settings where frameworks aim to be fully generic, with the only common denominator being the fact that they generate GUIs. However, our results suggest this may not be the most beneficial for *all* GUI-building frameworks to model. For example, we initially designed our `Stage` structure to help end users—not programmers—by allowing the end user to focus on each step of a data transformation task in sequence. However, our qualitative analysis suggests that in fact the `Stage` metaphor helped programmers write HiLT code, shifting their primary level of abstraction from GUI widget to data transformation subtask.

The Stage metaphor may generalize to other task-structured GUI domains. Although HiLT's `Stage` abstraction was designed for human-in-the-loop data transformation tasks, our findings suggest similar abstractions could benefit GUI-building frameworks in other domains where user workflows unfold in structured, sequential steps. Onboarding flows, multi-step configuration wizards, various kinds of data labeling tools, most of the interfaces one might build for a Mechanical Turk worker or for other crowdsourcing tasks, and interactive tutorials all share this step-by-step structure. Existing frameworks often provide abstractions for composing individual screens or widgets, but not for structuring an interface around a decomposition of the task. A generalized version of the `Stage` construct could give developers a way to organize code around the steps of the task itself, aligning program structure with user-facing structure. Is this actually how programmers want to modularize their GUI programs? Without more follow-on research, we cannot know for sure. However, as we found in our study, the stage structure enforced that programmers completed important problem decomposition work early in their process, and ultimately improved how they reasoned about control flow, modularity, and state. We believe this connection—between end-user stages and using stages to modularize the program structure—is worth exploring more broadly, across domains.

Results challenge the idea that general-purpose GUI frameworks can be embedded in a host language to fit task-specific needs. As framework designers, it is tempting for us to imagine that we can provide a set of generic abstractions, implement them in a library, and plan that programmers can augment our abstractions with host-language logic where needed. This is the typical "minimalist" embedded DSL (eDSL) strategy. This is even an important part of HiLT's own strategy—specifically, we plan for programmers to use other libraries like Pandas or OCR libraries. However, our results should make us question the minimalist eDSL hypothesis in this context. Concretely, our user study asked participants to stretch Streamlit beyond its primary use case of building data *display* GUIs. If the minimalist eDSL strategy was sufficient in this setting, we would expect programmers' behaviors with Streamlit to be comparable to behaviors with HiLT, with programmers supplementing Streamlit's generic GUI abstractions with Python logic. Instead we observed that Streamlit programmers accomplished only about a third as many data transformation-related subtasks as HiLT programmers, and participants indicated that their mental models of their programs were substantially different in the two tools. This echoes the insight from the findings that it was important that HiLT reoriented programmers' attention to the data transformation stage as the core building block of the program. In short, nudging programmers to view their programs through a different lens is not just another way of achieving what we achieve when we ask them to augment a minimal eDSL.

HiLT's benefits come not just from programmability but from shaping how programmers reason about their GUIs. Finally, why not Django or other high-programmer-control tools? If a tool like Streamlit was the wrong tool for the job because it locked users into a 'wrong' way of viewing the problem, perhaps the right answer is to give them a high-programmer-control tool in which they themselves control the lens they use for decomposing their program. Our user study does not take up this hypothesis, so this remains an open question. However, our attempt to produce a version of just one of the GUIs in our case studies using Django took 1,100 lines of code, as opposed to 47 with HiLT. Clearly it is possible to write human-in-the-loop data transformation GUIs with these high-expressivity tools, but the difference in the program sizes suggests this is a case where the Django program encodes some amount of incidental complexity, as well as the necessary complexity. Again, this points to the idea that existing GUI-building frameworks, both on the low-expertise and the high-expertise sides of the spectrum, lack the constructs that suit the particular class of GUIs we want to build when we target human-in-the-loop data transformation GUIs.

Beyond the Stage construct, should we use user-visible structure to structure the programmer's work too? We end on a slightly more speculative note. Outside of our particular setting, for GUI-building

frameworks focused on other tasks, should we expect that design choices that enforce a property that is desirable for end users will also enforce that programmers produce better organized mental models and, ultimately, better structured programs? Our `Stage` metaphor modularizes the user-facing GUI but also ultimately also modularized the program structure. Our user study let us observe that both HiLT and Streamlit program structures tended to echo the structure the GUI's end user would see, albeit in different ways across the two tools: in Streamlit programs, widgets often appeared in the same order in the program text as in the output pages; in HiLT programs, the program text was broken down into ranges associated with the user-visible data transformation `Stages`. We hypothesize that because of the special character of GUI-building tools—the fact that they interact with programmers on one side and GUI end users on the other—there may be a relationship between the properties that a framework enforces for end users and the structure that the framework enforces or suggests for programmers. We suggest that future work on GUI-building tools interrogate the connection between GUI structure and program structure, and explore how we can use this connection to change programmers' mental models and programming practices.

8 Conclusion

This paper focuses on a category of tool that is difficult to express with today's GUI-authoring frameworks: bespoke GUIs for completing a particular human-in-the-loop data transformation task. Based on three design constraints, we proposed a set of abstractions for implementing GUIs that display, manipulate, and transform data without knowing the data's structure in advance, and implemented the abstractions in HiLT, a Python library. We found that participants completed more subtasks—and especially more data transformation subtasks—with HiLT than with the prior state of the art. The HiLT 'stage' construct strongly influenced how programmers thought about the structure of their GUIs. More broadly, our user study demonstrated an interesting connection between the properties a GUI library enforces on its output GUIs and the properties it ends up enforcing on GUI programmers—and on their mental models of their programs.

Acknowledgments

We extend our gratitude to Allison Husain for her input in shaping early versions of HiLT. In addition, we would like to thank the participants in our user studies. This work is supported in part by NSF grant FW-HTF 2129008 and gifts from Google and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

References

[1] aiohttp contributors. 2025. Welcome to AIOHTTP — aiohttp 3.11.16 documentation. https://docs.aiohttp.org/en/stable/

[2] Airtable. 2025. Airtable: The platform to build next–gen apps. https://www.airtable.com/

[3] Alteryx. 2024. Alteryx Designer Cloud. https://www.alteryx.com/products/designer-cloud

[4] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3544548.3580785

[5] Edward Benson, Amy X. Zhang, and David R. Karger. 2014. Spreadsheet driven web applications. In *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 97–106. doi:10.1145/2642918.2647387

[6] Jeffrey P. Bigham and Richard E. Ladner. 2007. Accessmonkey: a collaborative scripting framework for web users and developers. In *Proceedings of the 2007 international cross-disciplinary conference on web accessibility (W4A) (W4a '07)*. Association for Computing Machinery, New York, NY, USA, 25–34. doi:10.1145/1243441.1243452 Number of pages: 10 Place: Banff, Canada.

[7] Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandsted Klokmose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–20. doi:10.1145/3491102.3502064

[8] Paul Brie, Nicolas Burny, Arthur Sluÿters, and Jean Vanderdonckt. 2023. Evaluating a Large Language Model on Searching for GUI Layouts. *Proc. ACM Hum.-Comput. Interact.* 7, EICS (June 2023), 178:1–178:37. doi:10.1145/3593230

[9] Lukas Brückner, Luis A. Leiva, and Antti Oulasvirta. 2022. Learning GUI Completions with User-defined Constraints. *ACM Trans. Interact. Intell. Syst.* 12, 1 (March 2022), 6:1–6:40. doi:10.1145/3490034

[10] Budibase. 2025. Budibase - Internal tools made easy. https://budibase.com/

[11] Alex Bäuerle, Ángel Alexander Cabrera, Fred Hohman, Megan Maher, David Koski, Xavier Suau, Titus Barik, and Dominik Moritz. 2022. Symphony: Composing Interactive Interfaces for Machine Learning. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3491102.3502102

[12] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating interactive web data applications with spreadsheets. In *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 87–96. doi:10.1145/2642918.2647371

[13] Chen Chen, Jane Hoffswell, Shunan Guo, Ryan Rossi, Yeuk-Yin Chan, Fan Du, Eunyee Koh, and Zhicheng Liu. 2023. WhatsNext: Guidance-enriched Exploratory Data Analysis with Interactive, Low-Code Notebooks. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Washington, DC, USA, 209–214. doi:10.1109/VL-HCC57772.2023.00033 ISSN: 1943-6106.

[14] Yiru Chen, Jeffrey Tao, and Eugene Wu. 2023. DIG: The Data Interface Grammar. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '23)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3597465.3605223

[15] Yiru Chen and Eugene Wu. 2022. PI2: End-to-end Interactive Visualization Interface Generation from Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1711–1725. doi:10.1145/3514221.3526166

[16] Matthew Conlen and Jeffrey Heer. 2018. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 977–989. doi:10.1145/3242587.3242600

[17] Django Software Foundation. 2024. Django. https://www.djangoproject.com/

[18] 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 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376442

[19] Oscar Díaz, Cristóbal Arellano, and Maider Azanza. 2013. A language for end-user web augmentation: Caring for producers and consumers alike. *ACM Trans. Web* 7, 2 (May 2013), 9:1–9:51. doi:10.1145/2460383.2460388

[20] 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 (Oct. 2023), 197–207. doi:10.1109/TVCG.2023.3327367

[21] Will EPPerson, Doris Jung-Lin Lee, Leijie Wang, Kunal Agarwal, Aditya G. Parameswaran, Dominik Moritz, and Adam Perer. 2022. Leveraging Analysis History for Improved In Situ Visualization Recommendation. *Computer Graphics Forum* 41, 3 (2022), 145–155. doi:10.1111/cgf.14529 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14529.

[22] Michael Fischer, Konstantin Kobs, and Andreas Hotho. 2020. NICER: Aesthetic Image Enhancement with Humans in the Loop. doi:10.48550/arXiv.2012.01778 arXiv:2012.01778 [cs].

[23] Takanori Fujiwara, Xinhai Wei, Jian Zhao, and Kwan-Liu Ma. 2022. Interactive Dimensionality Reduction for Comparative Analysis. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (Jan. 2022), 758–768. doi:10.1109/TVCG.2021.3114807 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[24] Google. 2024. Blockly | Google for Developers. https://developers.google.com/blockly

[25] Google. 2024. Dataprep by Trifacta. https://cloud.google.com/dataprep

[26] Google. 2024. Google Forms: Online Form Creator | Google Workspace. https://www.google.com/forms/about/

[27] Google. 2024. Google Sheets: Online Spreadsheet Editor | Google Workspace. https://www.google.com/sheets/about/

[28] Grace Guo, Ehud Karavani, Alex Endert, and Bum Chul Kwon. 2023. Causalvis: Visualizations for Causal Inference. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–20. doi:10.1145/3544548.3581236

[29] 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 24th annual ACM symposium on User interface software and technology (UIST '11)*. Association for Computing Machinery, New York, NY, USA, 65–74. doi:10.1145/2047196.2047205

[30] Aviv Hadar, Tova Milo, and Kathy Razmadze. 2025. Interactive Coreset Selection for Tabular Data: Fairness-Aware, Explainable, and User-Guided. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '25)*. Association for Computing Machinery, New York, NY, USA, 1–5. doi:10.1145/3736733.3736735

[31] Holoviz. 2024. Overview — Panel v1.4.2. https://panel.holoviz.org/

[32] Joshua Horowitz and Jeffrey Heer. 2023. Engraft: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, 1–18. doi:10.1145/3586183.3606733

[33] Kevin Hu, Diana Orghian, and César Hidalgo. 2018. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '18)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3209900.3209910

[34] David F. Huynh, David R. Karger, and Robert C. Miller. 2007. Exhibit: lightweight structured data publishing. In *Proceedings of the 16th international conference on World Wide Web (WWW '07)*. Association for Computing Machinery, New York, NY, USA, 737–746. doi:10.1145/1242572.1242672

[35] Benjamin Hättasch and Carsten Binnig. 2024. More of that, please: Domain Adaptation of Information Extraction through Examples & Feedback. In *Proceedings of the 2024 Workshop on Human-In-the-Loop Data Analytics (HILDA 24)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3665939.3665966

[36] Andra Ionescu, Zeger Mouw, Efthimia Aivaloglou, Rihan Hai, and Asterios Katsifodimos. 2024. Human-in-the-Loop Feature Discovery for Tabular Data. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 5215–5219. doi:10.1145/3627673.3679211

[37] JetBrains. 2024. Datalore: Collaborative Data Science Platform by JetBrains. https://datalore.jetbrains.com/index.jsp

[38] JMP. 2024. Statistical Software. https://www.jmp.com/en_us/home.html

[39] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. Association for Computing Machinery, New York, NY, USA, 3363–3372. doi:10.1145/1978942.1979444

[40] David R. Karger, Scott Ostler, and Ryan Lee. 2009. The web page as a WYSIWYG end-user customizable database-backed information management application. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST '09)*. Association for Computing Machinery, New York, NY, USA, 257–260. doi:10.1145/1622176.1622223

[41] 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 (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. doi:10.1145/3379337.3415842

[42] Tim Kraska. 2018. Northstar: an interactive data science system. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2150–2164. doi:10.14778/3229863.3240493

[43] Nahyun Kwon, Hannah Kim, Sajjadur Rahman, Dan Zhang, and Estevam Hruschka. 2023. Weedle: Composable Dashboard for Data-Centric NLP in Computational Notebooks. In *Companion Proceedings of the ACM Web Conference 2023 (WWW '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 132–135. doi:10.1145/3543873.3587330

[44] Michelle S. Lam, Zixian Ma, Anne Li, Izequiel Freitas, Dakuo Wang, James A. Landay, and Michael S. Bernstein. 2023. Model Sketching: Centering Concepts in Early-Stage Machine Learning Model Design. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–24. doi:10.1145/3544548.3581290

[45] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3623344

[46] 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 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3544548.3580965

[47] 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 (March 2023), 1:1–1:27. doi:10.1145/3545995

[48] Yanna Lin, Haotian Li, Leni Yang, Aoyu Wu, and Huamin Qu. 2023. InkSight: Leveraging Sketch Interaction for Documenting Chart Findings in Computational Notebooks. *IEEE Transactions on Visualization and Computer Graphics* 30, 1 (Oct. 2023), 944–954. doi:10.1109/TVCG.2023.3327170

[49] Microsoft. 2024. Data Analysis Expressions (DAX) Reference - DAX. https://learn.microsoft.com/en-us/dax/

[50] Microsoft. 2024. Database Software and Applications | Microsoft Access. https://www.microsoft.com/en-us/microsoft-365/access

[51] Microsoft. 2024. Free Online Spreadsheet Software: Excel | Microsoft 365. https://www.microsoft.com/en-us/microsoft-365/excel

[52] Microsoft. 2024. Power BI - Data Visualization | Microsoft Power Platform. https://www.microsoft.com/en-us/power-platform/products/power-bi

[53] Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. 2020. Coresets for data-efficient training of machine learning models. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20, Vol. 119)*. JMLR.org, Vienna, Austria, 6950–6960.

[54] Kristi Morton, Ross Bunker, Jock Mackinlay, Robert Morton, and Chris Stolte. 2012. Dynamic workload driven data integration in tableau. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 807–816. doi:10.1145/2213836.2213961

[55] David Munechika, Zijie J. Wang, Jack Reidy, Josh Rubin, Krishna Gade, Krishnaram Kenthapadi, and Duen Horng Chau. 2022. Visual Auditor: Interactive Visualization for Detection and Summarization of Model Biases. In *2022 IEEE Visualization and Visual Analytics (VIS)*. IEEE, Oklahoma City, OK, USA, 45–49. doi:10.1109/VIS54862.2022.00018 ISSN: 2771-9553.

[56] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. 2020. CoClean: Collaborative Data Cleaning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2757–2760. doi:10.1145/3318464.3384698

[57] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. doi:10.1145/3453483.3454059

[58] OpenAI. 2022. Introducing ChatGPT. https://openai.com/index/chatgpt

[59] OpenRefine. 2024. General Refine Expression Language | OpenRefine. https://openrefine.org/docs/manual/grel

[60] OpenRefine. 2024. OpenRefine. https://openrefine.org/

[61] João Palmeiro, Beatriz Malveiro, Rita Costa, David Polido, Ricardo Moreira, and Pedro Bizarro. 2022. *Data+Shift: Supporting Visual Investigation of Data Distribution Shifts by Data Scientists*. The Eurographics Association, Rome, Italy. https://doi.org/10.2312/evs.20221097

[62] pandas. 2024. pandas - Python Data Analysis Library. https://pandas.pydata.org/

[63] Santiago Perez De Rosso, Daniel Jackson, Maryam Archie, Czarina Lao, and Barry A. McNamara III. 2019. Declarative assembly of web applications from predefined concepts. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 79–93. doi:10.1145/3359591.3359728

[64] Plotly. 2024. Dash Documentation & User Guide | Plotly. https://dash.plotly.com/

[65] Qualtrics. 2024. Qualtrics XM - Experience Management Software. https://www.qualtrics.com/

[66] Rails Foundation. 2024. Ruby on Rails. https://rubyonrails.org/

[67] Retool. 2025. Retool | Build polished applications, fast. https://retool.com/apps

[68] Salesforce. 2024. Business Intelligence and Analytics Software | Tableau. https://www.tableau.com/

[69] SAS. 2024. SAS: Analytics, Artificial Intelligence and Data Management. https://www.sas.com/en_us/home.html

[70] Venkatesh Sivaraman, Yiwei Wu, and Adam Perer. 2022. Emblaze: Illuminating Machine Learning Representations through Interactive Comparison of Embedding Spaces. In *Proceedings of the 27th International Conference on Intelligent User Interfaces (IUI '22)*. Association for Computing Machinery, New York, NY, USA, 418–432. doi:10.1145/3490099.3511137

[71] Snowflake. 2021. Streamlit • A faster way to build and share data apps. https://streamlit.io/

[72] SQLite. 2024. SQLite Home Page. https://www.sqlite.org/

[73] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. 2018. Rewire: Interface design assistance from examples. In *Proceedings of the 2018 CHI conference on human factors in computing systems (Chi '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3174078 Number of pages: 12 Place: Montreal QC, Canada.

[74] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *Proceedings of the 2020 CHI conference on human factors in computing systems (Chi '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3313831.3376593 Number of pages: 13 Place: Honolulu, HI, USA.

[75] Gradio Team. 2024. Gradio. https://gradio.app

[76] Carol Traynor and Marian G. Williams. 2001. Chapter 6 - End Users and GIS: A Demonstration Is Worth a Thousand Words. In *Your Wish is My Command*, Henry Lieberman (Ed.). Morgan Kaufmann, San Francisco, 115–VI. doi:10.1016/B978-155860688-3/50007-5

[77] Turnitin. 2025. Gradescope | Save time grading. https://www.gradescope.com/

[78] Priyan Vaithilingam, Elena L. Glassman, Jeevana Priya Inala, and Chenglong Wang. 2024. DynaVis: Dynamically Synthesized UI Widgets for Visualization Editing. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3613904.3642639

[79] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 563–576. doi:10.1145/3332165.3347944

[80] Vercel. 2024. Next.js by Vercel - The React Framework. https://nextjs.org/

[81] Lea Verou, Tarfah Alrashed, and David Karger. 2018. Extending a Reactive Expression Language with Data Update Actions for End-User Application Authoring. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, New York, NY, USA, 379–387. doi:10.1145/3242587.3242663

[82] Lea Verou, Amy X. Zhang, and David R. Karger. 2016. Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. Association for Computing Machinery, New York, NY, USA, 483–496. doi:10.1145/2984511.2984551

[83] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2023. Interactive Search with Mixed Attributes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Anaheim, CA, USA, 2276–2288. doi:10.1109/ICDE55515.2023.00176 ISSN: 2375-026X.

[84] Zijie J. Wang, Alex Kale, Harsha Nori, Peter Stella, Mark Nunnally, Duen Horng Chau, Mihaela Vorvoreanu, Jennifer Wortman Vaughan, and Rich Caruana. 2021. GAM Changer: Editing Generalized Additive Models with Interactive Visualization. doi:10.48550/arXiv.2112.03245 arXiv:2112.03245 [cs].

[85] Rachel B. Warren and Niloufar Salehi. 2022. Trial by File Formats: Exploring Public Defenders' Challenges Working with Novel Surveillance Data. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW1 (April 2022), 67:1–67:26. doi:10.1145/3512914

[86] Weicheng Wang, Victor Junqiu Wei, Min Xie, Di Jiang, Lixin Fan, and Haijun Yang. 2025. Interactive Search with Reinforcement Learning. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Hong Kong, 1443–1455. doi:10.1109/ICDE65448.2025.00112

[87] Peter Xenopoulos, João Rulff, Luis Gustavo Nonato, Brian Barr, and Claudio Silva. 2023. Calibrate: Interactive Analysis of Probabilistic Model Output. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (Jan. 2023), 853–863. doi:10.1109/TVCG.2022.3209489

[88] Min Xie, Tianwen Chen, and Raymond Chi-Wing Wong. 2019. FindYourFavorite: An Interactive System for Finding the User's Favorite Tuple in the Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 2017–2020. doi:10.1145/3299869.3320215

[89] Min Xie and Yang Liu. 2022. Favorite+: Favorite Tuples Extraction via Regret Minimization. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 5049–5053. doi:10.1145/3511808.3557188

[90] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. 2008. WYSIWYG development of data driven web applications. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 163–175. doi:10.14778/1453856.1453879

[91] Faraz YazdaniBanafsheDaragh and Sam Malek. 2022. Deep GUI: black-box GUI input generation with deep learning. In *Proceedings of the 36th IEEE/ACM international conference on automated software engineering (Ase '21)*. IEEE Press, Melbourne, Australia, 905–916. doi:10.1109/ASE51524.2021.9678778 Place: Melbourne, Australia Number of pages: 12.

[92] Shengcheng Yu, Chunrong Fang, Xin Li, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Effective, platform-independent GUI testing via image embedding and reinforcement learning. *ACM Trans. Softw. Eng. Methodol.* 33, 7 (June 2024), 27. doi:10.1145/3674728 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[93] Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Artifact for "Live Pattern Matching with Typed Holes"* 7, OOPSLA1 (April 2023), 96:609–96:635. doi:10.1145/3586048

[94] Dan Zhang, Hannah Kim, Rafael Li Chen, Eser Kandogan, and Estevam Hruschka. 2022. MEGAnno: Exploratory Labeling for NLP in Computational Notebooks. In *Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances)*, Eduard Dragut, Yunyao Li, Lucian Popa, Slobodan Vucetic, and Shashank Srivastava (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates (Hybrid), 1–7. https://aclanthology.org/2022.dash-1.1

[95] Xiong Zhang and Philip J. Guo. 2017. DS.js: Turn Any Webpage into an Example-Centric Live Programming Environment for Learning Data Science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 691–702. doi:10.1145/3126594.3126663

A HiLT Core APIs

Here we briefly cover details of several of the key APIs available to HiLT programmers.

A.1 `Tool`

HiLT programmers construct `Tools` by adding `Stages` to them using the `add_stage` method (see Table 1.) Using a `Tool` object's `run` method creates an initial webpage, from which users can navigate to each `Stage`. Additional webpages are generated as the `Tool` runs based on the combination of (i) the end user's interactions with webpages and (ii) each `Stage`'s Stage Defining Function (SDF)—that is, the function used as input to the `add_stage` method. We will use the terms stage and SDF interchangeably. Programmers can use the `user_input_received` method to gate code in their SDFs which should only run once the `Tool` has received user input.

A.2 `Components`

`Components` allow programmers to specify the interface associated with a given `Stage`. Currently, HiLT provides seven classes of pre-specified interface elements which HiLT programmers can use to build interfaces (`ColumnSelectorComponent`, `FileUploadComponent`, `SelectorComponent`, `SubmitComponent`, `TableSelectorComponent`, `TextComponent`, and `UserInputComponent`). Most `Component` classes capture some form of user input, and provide a `label` attribute, allowing programmers to prompt users for input (see Design Constraint 1), as well as a `value` attribute, to programatically access the provided input. We show the APIs associated with three sample components in Table 2.

A.3 `Tables`

The `Tables` class allows programmers to manipulate the tables in the underlying database of a `Tool`. Each `Tool` maintains its own `Tables` instance, for accessing the `Tool`'s underlying database. The `Tables` class allows programmers to access individual tables using dictionary-like notation, with table names as keys (i.e. `tool.tables['myTable']`). This notation supports reading, storing, and updating tables in the underlying database (i.e. `tool.tables['myTable'] = pd.DataFrame(data)`). Tables can also be dropped using the `del` keyword. The `Tables` class also provides a number of convenience methods to access metadata like table or column names.

A.4 `approvals`

HiLT's `approvals` module allows programmers to solicit user confirmation for changes being made to a `Tool`'s database. When database changes are made via the `Tables` API (Section A.3), they are cached before actually being committed to the underlying database. Programmers can then call the `approvals.get_user_approvals` function to direct HiLT to display the approvals interface and seek user confirmation(e.g., line 23 of Figure 3). After the user interacts with the approvals interface, the user-approved changes are committed to the database, and non-approved changes are dropped.

A.5 `results`

The `results` module allows programmers to control what is displayed after input processing has taken place. The programmer passes a list of values, each optionally with a label, to the `results.show_results` function. For example, after creating a new table, a programmer may provide the created table as input to `show_results`, as in line 9 of Figure 3, which will produce a webpage that displays the created table.

Table 1: The `Tool` Class: A `Tool` is the parent object representing a complete GUI, which may have many component `Stages`. `add_stage` allows programmers to add `Stages` to their `Tools`, passing a Stage Defining Function (SDF) as an argument. `run` launches the `Tool`. `user_input_received` allows programmers to separate code in their SDFs that should only run once the user has provided input. `state` facilitates the sharing of non-tabular data between `Stages`. `tables` provides an API to manipulate the `Tool`'s underlying database.

Method/Field Signature	Description
`add_stage(stage_name: str, stage_func: Callable)`	Method to add a `Stage` to this `Tool`. `stage_func` is expected to be a function which defines a `Stage`.
`run()`	Runs this `Tool` as a local web server, allowing users to access interfaces through a browser.
`user_input_received() -> bool`	Returns true if this `Tool` is currently processing user input.
`state: dict[str, Any]`	A dictionary of values associated with this `Tool`, allowing variables to be shared across `Stages`.
`tables: Tables`	A view of this `Tool`'s underlying database.

Table 2: Component Classes: A sample of HiLT's available component classes. `FileUploadComponent` provides a widget for users to upload files. `TableSelectorComponent` provides a widget to select a table from a `Tool`'s underlying database. Once user input is received, the `value` attribute contains a pandas `DataFrame` with the chosen table. `ColumnSelectorComponent` provides a widget allowing users to first select a table from a `Tool`'s underlying database, and then a number of columns from that table.

The `FileUploadComponent` Class:

Method/Field Signature	Description
`expected_ext: str`	The expected file extension for uploaded files, used to validate user input.
`replace_existing: bool`	Whether to silently replace or raise an exception if a file with the same name was previously uploaded.
`value: str`	The local path to the uploaded file.

The `TableSelectorComponent` Class:

Method/Field Signature	Description
`value: pd.DataFrame`	A pandas `DataFrame` containing the selected table.
`table_name: str`	The name of the selected table.

The `ColumnSelectorComponent` Class:

Method/Field Signature	Description
`num_columns: int`	The number of columns the user is expected to select.
`value: pd.DataFrame`	A pandas `DataFrame` containing the chosen columns of the selected table.
`table_name: str`	The name of the table from which columns were selected.
`column_names: list[str]`	The names of the columns selected by the user.

Subtask No.	Description	Design Constraint
1-1	Can a user upload their choice of CSV files?	Importing Data
1-2	Does the interface ask the user to name the CSV file they're uploading?	Guiding the User
1-3	Is the CSV data persisted in an underlying data store?	Importing Data
2-1	Does the column addition work with varying structures of input CSVs?	Importing Data
2-2	Can the user select which columns of the data should be interpreted as latitude and longitude?	Transforming Data
2-3	Is the fresh column persisted in an underlying data store?	Transforming Data
3-1	If the user wants to batch accept or reject all the computed values, can they?	Transforming Data
3-2	If the user wants to individually approve or reject the computed values, can they?	Transforming Data
3-3	Does the interface show the user how to batch accept or reject all the computed values?	Guiding the User
3-4	Does the interface show the user how to individually approve or reject the computed values?	Guiding the User
3-5	Are the accepted values persisted in an underlying data store?	Transforming Data
4-1	Can a user specify which two datasets to match?	Transforming Data
4-2	Is the user instructed to select two datasets?	Guiding the User
4-3	Can a user construct a new data table based on identifying matches across two data sources?	Transforming Data
4-4	Does the user have control over the criteria used to match the datasets?	Transforming Data
4-5	Does matching work with varying structures of input CSVs?	Importing Data
4-6	Are the proposed matches persisted in an underlying data store?	Transforming Data
5-1	If the user wants to batch accept or reject all the computed matches, can they?	Transforming Data
5-2	If the user wants to individually approve or reject the computed matches, can they?	Transforming Data
5-3	Does the interface show the user how to batch accept or reject all the computed matches?	Guiding the User
5-4	Does the interface show the user how to individually approve or reject the computed matches?	Guiding the User
5-5	Are the accepted matches persisted in an underlying data store?	Transforming Data

Table 3: Each task in the evaluative study constituted a set of subtasks. In order for a participant to be marked as having completed a task, they must also have completed all component subtasks. The first digit of the subtask number indicates which task is associated—e.g., 1-3 with task 1. Also note that each subtask is associated with one of the design constraints outlined in Section 1.