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

Sora Kanosue

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-92
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-92.html

May 10, 2024

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

by

Sora Kanosue

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science, Plan II

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

,
Professor Sarah E. Chasins
Professor Alvin Cheung

Spring 2024

The thesis of Sora Kanosue, titled HiLT: A Framework for Generating Human-in-the-Loop Data Transformation GUIs, is approved:

Date May 10, 2024

Alvin Cheung (second reader)

Date May 8, 2024

Date

University of California, Berkeley

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

Abstract

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

by

Sora Kanosue

Master of Science, Plan II in Electrical Engineering and Computer Science

University of California, Berkeley

,

Domain experts often find themselves performing repetitive tasks restructuring large amounts of data. General-purpose data analysis tools like Tableau can help experts accomplish these tasks, but they have steep learning curves, and their flexibility and open-ended interaction model can be overwhelming for users. Task-specific interfaces can circumvent this issue by guiding users through each phase of a data transformation task, but authoring bespoke tools for each task is time-consuming and difficult. In this paper, we present HiLT, a domain-specific language embedded in Python which facilitates the creation of task-specific data transformation GUIs. Programs written in HiLT generate human-in-the-loop data transformation GUIs which walk users through the process of a given data transformation task. We conducted a formative user study with 17 participants who were tasked with constructing data transformation interfaces using HiLT and other existing frameworks in order to explore HiLT's learnability and usability relative to other tools in the same space. Our findings from the formative study suggest directions for future tools in this space.

Acknowledgments

Firstly, I would like to thank my advisor, Professor Sarah E. Chasins. Without her guidance and mentorship, this project would have been impossible. Her insight and advice have shaped both this work and my time at Berkeley, and for that, I am deeply grateful. I would also like to extend my heartfelt gratitude to my collaborators on this project, Parker Ziegler and Xiaorui Liu. Parker's friendship and expertise have been invaluable to both this work and myself, and Xiaorui's assistance and patience were essential in bringing it to fruition. I would also like to give thanks to Allison Husain, who was an essential part of the early stages of this project. A huge shoutout goes to the user study's pilot participants and their invaluable feedback, as well as the anonymous participants who volunteered to make this work possible. I would like to thank all the folks of the PLAIT Lab and my friends at Berkeley. Beyond Berkeley, I would like to thank my parents, Yu and Carlos, my siblings, Saia and Sona, and my partner, Su-Ann. Their support and encouragement have been a boundless resource for me, and I am eternally grateful to them.

Introduction

Data scientists and other domain experts frequently find themselves tackling data transformation tasks which require human insight and intervention, but at scales too large to be comfortably accomplished by purely manual means. Custom GUIs that walk users through a particular human-in-the-loop data transformation task hold the promise of reducing the burden of these undertakings. Especially for tasks that require human inputs at multiple stages, the human-in-the-loop nature makes custom GUIs a more practical technological intervention than attempts to automate them end-to-end.

Previous work has documented how domain experts in these situations resort to measures such as delegating the tasks to spread the workload involved, engaging "surrogate user" to perform the task instead [43], or even forgoing the task altogether [48]. The other primary alternative is code-free data transformation software like Tableau and Google Query. However, using these tools to accomplish a given data transformation task can require a great deal of expertise. Their open-ended nature makes them flexible, but also difficult for novices to navigate.

Another alternative, available to practitioners who have access to experienced GUI developers, is to solicit a bespoke GUI for each task, which practitioners may use to accomplish their data transformations. Challenges associated with this approach include (i) identifying relevant abstractions in the tasks, so that tools do not become obsolete via software rot as data sources and shapes evolve over time, and (ii) the high engineering cost associated with implementing data transformation GUIs, which are typically more complex than the more common data display GUIs. For example, an entity matching tool written by the authors of this project using the Django web framework took over 1100 lines of code. With current tools, writing custom human-in-the-loop data transformation GUIs is a difficult process.

1.1 Contributions

We believe there is an opportunity for a tool in this space that supports programmers in writing custom GUIs for human-in-the-loop data transformation tasks. The GUIs produced by such a tool must allow programmers to bring and transform their own data. We have designed a framework for constructing this class of GUIs, centered on meeting the following design goals:

1. **Design Goal 1: Guiding the User** The output GUI must walk the user through the process of their data transformation task, rather than leaving them to explore in an open-ended way, as Excel or Tableau do.

2. **Design Goal 2: Importing Data** The system must make it possible for an output GUI to accept different input data shapes, so users can bring their own diverse data.

3. **Design Goal 3: Transforming Data** The system must make it possible for an output GUI to change the data shape based on users' interactions with it.

In this paper, we present the initial design of HiLT, a Python-embedded domain-specific language (DSL) designed to output GUIs which guide users through human-in-the-loop data transformation tasks. HiLT's abstractions allow programmers to: (i) build customized interfaces by combining a variety of UI components, (ii) specify how UI interactions map to data transformations in the underlying database, and (iii) run the generated GUI as a fully fledged database-driven web application. Because it is implemented as a Python library, HiLT is compatible with other Python libraries that are also useful for data transformation tasks, such as Numpy, Pandas, and PyTorch.

We conducted a user study with 17 Python programmers in order to explore the usability and learnability of HiLT's initial design compared to existing alternatives. Each participant completed a series of programming tasks in HiLT. They also completed this same series of tasks in either Streamlit or Django. We found that participants were generally more successful with HiLT than Django, but that Streamlit outperformed HiLT in certain aspects, although participants did not succeed with Streamlit in two of the core goals of this work—supporting *transformation* of data and guiding users through tasks. From the qualitative results obtained from this study, we plan to iterate upon HiLT and run further user studies to evaluate both HiLT and the GUIs its programs produce.

Overall, this work contributes:

- A set of abstractions that allow programmers to implement GUIs that display, manipulate, and transform data without knowing the physical structure of the data in advance.

- HiLT, a Python DSL that implements these abstractions, and a framework for building custom GUIs that guide users through human-in-the-loop data transformation tasks.

- A formative study exploring HiLT's learnability and usability.

- A discussion of the formative study's results and the connection to directions for future work.

Related Work

In this section, we survey both human-in-the-loop data transformation systems and programming frameworks for building human-in-the-loop 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 [23] and Tableau [37] are popular open-ended human-in-the-loop data transformation systems. Examples of guided systems include survey software like Qualtrics [35] and Google Forms [10].

2. **Data Display vs. Data Transformation.** Many data work 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. In contrast, data work GUIs that permit *data transformation* allow users to: (i) insert their own tables, which may not match a previously fixed schema, and (ii) engage in processes involving querying and changing the shape of that data. This imposes a requirement that 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, although of course we can use general-purpose tools like vanilla web programming to author guided data transformation GUIs. Thus, we focus a majority 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), and briefly discuss the intersection of systems featuring guided data transformation tasks.

Data Analytics Software: Open-Ended User Interaction, Support for Data Transformation

Prior work both in academia and industry has contributed a large number of GUI-based, exploratory data transformation and analysis systems [37, 25, 1, 24, 26, 38, 18, 16, 9]. Many of these systems aim to (1) support flexible, open-ended data transformation and cleaning

and (2) emphasize low- or no-code interactions with data. For example, business intelligence (BI) tools like Tableau [37, 25], Trifacta [9], and Microsoft PowerBI [24] allow users to filter, aggregate, and visualize data from multiple sources without programming or database management expertise. OpenRefine [30] 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 (e.g., Data Analysis Expressions (DAX) [22] in PowerBI and General Refine Expression Language (GREL) [29] in OpenRefine), their core interaction models are GUI-based. Also ubiquitous in information workplaces are spreadsheet-based technologies like Microsoft Excel [23] and Google Sheets [11].

Another category of tool leverages programming-by-demonstration and programming-by-example to infer and synthesize data transformations from user interactions [19, 12, 7]. For example, Wrangler [19, 12] pairs direct manipulation with an inference engine to suggest and preview data transformations while a user interacts with their data in a table interface. Wrex [7] augments computational notebooks with table components allowing users to provide input-output examples of desired transformations as a specification to a program synthesizer. 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 the primary task. However, while these systems orient users toward open-ended, exploratory analyses, a user of a GUI generated by a HiLT program is guided through a *specific* transformation task defined by the HiLT programmer.

GUI-Building Software: Guided User Interaction, Support for Data Display

Web Programming Frameworks 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 data transformation systems. Django (Python) [6], Ruby on Rails [36] (Ruby), and Next.js (JavaScript, TypeScript) [45] 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 a data transformation interface to offer users the opportunity to:

- add their own tables, with arbitrary column structures
- engage in interactions that require querying those tables, linking tables, generating new tables, and generally transforming the shape 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 goals above requires more expertise and more complex codebases.

Frameworks for Interface Generation A number of web frameworks trade degree of control for ease of development in order to lower the complexity of building a functioning web application. One such framework in this space is Streamlit [39], which pairs a library of pre-made components with a web application engine to allow users to build simple web-apps using Python scripts. Although libraries in this space allow for rapid development of interfaces, the challenges around supporting data transformations in the interface backends (rather than only data display) remain. Other entries in this space which aim at rapid development of data visualization interfaces include Dash [33], Panel [14], and Gradio [42].

Beyond these frameworks, researchers have developed languages and libraries that aim to lift the level of abstraction even higher to simplify the end-user programming experience. Mavo [46] is a declarative HTML superset that adds data persistence, interactivity, and inferred data schemas to web applications via new language constructs applied to HTML elements. In this way, Mavo's abstractions support complex CRUD operations and user interactivity without requiring programmers to have JavaScript expertise or experience with relational data modeling. Again, these tools focus on display rather than transformation.

HiLT shares many of the core ideas and philosophies of these tools. Like Mavo, HiLT adds domain-specific constructs to a host language (Python) with abstractions for entities like database tables. Similarly, HiLT's `Component`s and `Processor`s mimic the data parameters and view descriptors of the mage API. However, unlike these systems, HiLT tailors its abstractions even more closely to the specific domain of data transformation GUIs, rather than data display GUIs. For example, programs in HiLT make no assumptions about the shape of the data which users input, allowing tools written in HiLT to be reused across multiple contexts and by users with different data sources. Also in this space is DIG [4], which abstracts data analysis interfaces into a formal grammar. By delineating the parameters of data analyses, DIG can be used to outline the requirements of an interface for a given analysis task. HiLT takes a similar approach by allowing programmers to use `Components` to give users degrees of freedom over the data transformation taking place. However, whereas DIG grammars encode that all parameters be resolved by the end users of an interface, HiLT gives programmers explicit control over which decisions the end users will make and what the programmers will decide for them in advance.

Augmenting Computation Notebooks with Data Displays Some prior works have used computational notebooks as a platform to support programmers in authoring custom notebooks which guide users through data-related tasks. For example, iPython [17] exposes a feature called Widgets, which allows developers to augment Jupyter notebooks with "interactive browser controls" [34]. mage [20] provides a framework for developers to create data widgets which can be represented simultaneously in code and within a GUI, facilitating interactions like data selection and exploration. PI2 [5] synthesizes data visualization interfaces from queries, allowing users to explore SQL query spaces. EDAssistant [21] provides users with an array of data visualizations and suggests code for subsequent analyses, thus guiding developers through exploratory data analysis tasks. Although these systems support

interactions with data within computational notebook systems, these interactions are either limited to data visualizations or intended to support programmers within the open-ended interface of a computational notebook.

Easing GUI Authoring Without Data Emphasis More broadly, a number of prior works feature systems which facilitate easier authoring of GUIs. As discussed previously, PI2 [5] synthesizes data visualization interfaces from sequences of queries. Bespoke [44] uses programming by demonstration to wrap command-line applications in GUIs. Varv [3] presents programs as data structures which allow for the authoring of user-extensible interactive software systems, including interfaces. A number of systems allow for the creation of visual programming environments, including Sandblocks, Blockly, Engraft, and livelit [2, 15, 27, 8]. Although these tools aim to make GUI authoring easier in a variety of domains, these domains do not include data transformation tasks.

Domain-Specific Custom GUIs: Guided User Interaction, Support for Data Transformation

Above, we describe both (i) open-ended tools that can be used for data transformation and (ii) frameworks that can be used to *generate* guided-interaction tools that support data display. We are not aware of systems that specifically target *generating* guided-interaction tools that support data *transformation*. However, even with tools primarily designed for other purposes, individuals and teams have created a variety of custom GUIs for walking users through data transformation tasks in particular domains. For example, Statsplorer [47] is a tool which walks novices through inferential statistical tests. The scope of interactions is limited, described by its authors as "[an] explicit, narrow, and deep decision tree," and, as required by our data transformation definition, the system accepts tabular data regardless of schema. Tools in this category are examples of the kind of human-in-the-loop guided data transformation UIs which users can author with a tool like HiLT, a category of tool that is still difficult to author with modern GUI authoring approaches. Unlike HiLT, these tools are not themselves aimed at generating interfaces.

HiLT Interaction Model

In the HiLT interaction model, programmers define interfaces which can be used to capture user inputs and perform data transformations parameterized by these inputs. The outputs of these data transformations can also be surfaced for inspection and approval by the users before being committed. HiLT programs can thus be used to produce human-in-the-loop data transformation pipelines usable by non-programmers.

In this section, we provide an overview of this interaction model by working through a motivating example. We consider the case of a lawyer, Marin, trying to perform entity matching across multiple years' of police roster data encoded in spreadsheets. Marin's challenge is that these spreadsheets each contain several thousand entries and have column names which differ across years. To this point, Marin has tried to surface the matches by manually inspecting the spreadsheets, an approach which has been inefficient at best. Frustrated, Marin reaches out to Navi, a colleague on her team with programming experience, who chooses to use HiLT to construct an interface which Marin can use to accomplish this data transformation task.

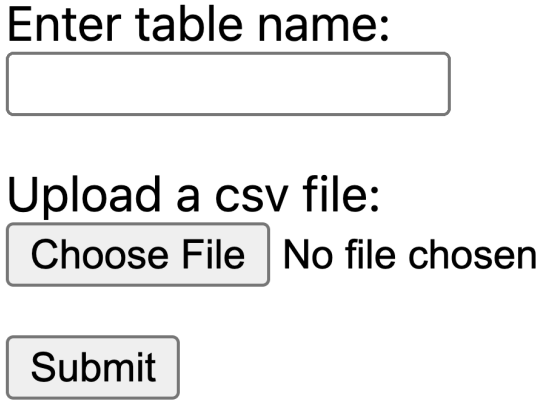


Figure 3.1: The resulting interface for a file upload `Stage`

3.1 Programming in HiLT

Navi begins by instantiating a HiLT `Tool`. Each program in HiLT defines a `Tool`, which is composed of `Stage`s that guide users through the different steps of a human-in-the-loop data transformation task. Each `Tool` has an underlying database, which allows its `Stage`s to read and modify a shared set of datasets.

```
1  tool = Tool('demo')
2
3  def csv_upload_and_name():
4      table_name = UserInputComponent(str, label="Enter table name:")
5      csv_file = FileUploadComponent('csv', label="Upload a csv file:")
6
7      def create_table():
8          return create_table_from_csv(table_name, csv_file, get_user_approvals=True)
9
10     created_table = LambdaProcessor(create_table).result
11     results.show_results([results.Result(created_table, "Created table: ")])
12     approvals.get_user_approvals()
13
14 tool.add_stage('csv_upload_and_name', csv_upload_and_name)
15
16 tool.run()
```

Figure 3.2: Code for a file upload `Stage`

Having instantiated the `Tool`, Navi now begins constructing the first `Stage`, which Marin will use to upload the datasets of interest. To define a `Stage`, Navi needs to specify a *stage-defining function* (SDF) which specifies:

1. The `Component`s that comprise the user interface which Marin will use to upload the datasets,
2. The `Processor`s that will run on Marin's inputs,
3. The `Result`s to show to Marin once she finished providing input.

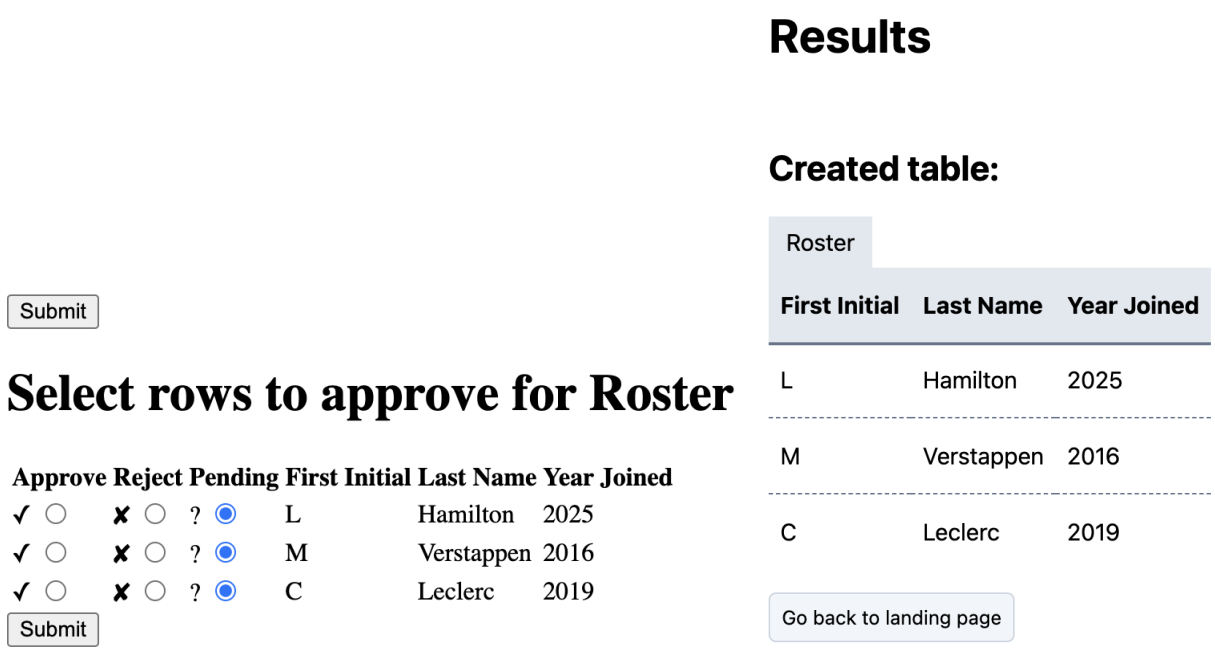
In HiLT, programmers define the interface of each `Stage` using one or more `Component`s, each of which specifies a separate interface element. Navi chooses to use a `UserInputComponent` and a `FileUploadComponent` to populate the interface for the dataset upload `Stage` (Figure 3.2), which render as a text entry box and a file upload input (Figure 3.1).

Having defined the user interface for the `Stage`, Navi moves on to defining a `Processor` in order to specify how to handle Marin's inputs. The main `Processor` class available is `LambdaProcessor`, which is instantiated using a Python function. To programatically access the `Component`s she just defined, Navi can pass a function nested in the SDF and use it to instantiate a `LambdaProcessor`, as seen in lines 7-10 Figure 3.2. She has the function passed

to the `Processor` return the created table, so that it can be accessed via the `Processor`'s `result` attribute.

Before committing Marin's uploaded data to the `Tool`'s underlying database, Navi may want Marin to inspect the uploaded data to give her the opportunity to filter out any malformed data. Navi can achieves this using the `get_user_approvals` argument of the `create_table_from_csv`, which caches the changes await approval. Then, the call she makes to the `approvals` module's `get_user_approvals` function surfaces these cached changes for inspection by Marin.

All that remains now is for Navi to specify what Marin should see after the `Processor` has run and the approved changes have been committed. To do this, Navi can use the `show_results` function available in HiLT's `results` library. As seen in Figure 3.2, Navi chooses to display the table created from the dataset uploaded by Marin, along with a label describing the displayed results.



(a) The approvals interface

(b) Interface for a table `Result`

Figure 3.3: The approvals and results interface for the file upload `Stage`

3.2 Using a HiLT-programmed `Tool`

To begin interacting with the `Tool` defined by Navi, Marin begins by navigating to the `Tool`'s landing page. To upload a new dataset, she navigates to the `csv_upload_and_name Stage`, which presents her with the interface shown in Figure 3.2. Upon uploading a dataset, Marin sees the approvals interface, which allows her to approve and reject various rows of the uploaded table, as seen in Figure Since Navi has directed the committed table to be shown as a `Result`, Marin sees exactly that after clicking the Submit button, as seen in Figure 3.3b.

HiLT API Description

. In this section, we describe the core APIs available to HiLT programmers for constructing `Tool`s. Specifically, we provide the Python type-annotated signatures for the methods and fields of:

- the `Tool` class,
- a sample of `Component` classes (`FileUploadComponent`, `TableSelectorComponent`, and `ColumnSelectorComponent`),
- and the `LambdaProcessor` sub-class of `Processor`.

We also describe a few key components of HiLT's standard library, including the modules that handle database creation and modification (`db_utils`), user approval of modifications (`approvals`), and displaying results (`results`).

4.1 `Tool`

HiLT programmers construct `Tool`s by progressively adding `Stage`s to them using the `add_stage` method. (See Table 4.1.) Using a `Tool` object's `run` method creates the initial webpage. Additional webpages are generated throughout the `Tool`'s run based on the combination of (i) the end user's interactions with webpages and (ii) each `Stage`'s SDF.

Table 4.1: The `Tool` Class

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.

Table 4.2: The `FileUploadComponent` Class

Method/Field Signature	Description
`expected_ext: str`	The expected file extension for uploaded files, used to validate user input.
`value: str`	The local path to the uploaded file.

4.2 Components

`Component`s form the building blocks of each `Stage`'s associated web page. HiLT provides a `Component` class, and a number of subclasses which define a number of pre-specified interface elements which HiLT programmers can use to build interfaces. Currently, HiLT provides seven pre-defined `Component` subclasses (`ColumnSelectorComponent`, `FileUploadComponent`, `SelectorComponent`, `SubmitComponent`, `TableSelectorComponent`, `TextComponent`, and `UserInputComponent`). Most `Component` classes are `InputComponent`s, which are designed to capture some form of user input. `InputComponent`s provide a `label` attribute, to allow programmers to prompt users for input, and a `value` attribute, which can be used to programatically access the provided input later, such as in a `Processor`. `InputComponent`s also provide overloading for Python's magic methods where appropriate, allowing each to be programmatically treated like its `value`. We provide details for three of the `Component` sub-classes in the following paragraphs.

`FileUploadComponent`

`FileUploadComponent`s provide an interface element for users to upload files. These can be used to solicit input datasets like CSV files being stored locally by users.

```
def column_selector():
    table = TableSelectorComponent("Pick a table")
    col = ColumnSelectorComponent("Pick a column from the same table")

    def select_col():
        col_names = col.value
        col_name = col_names[0]
        values = []
        for row in table:
            value = row[col_name]
            values.append(value)
        return values
    processor = LambdaProcessor(iterate_over_table, num_return_vals=1)
    results.show_results([results.Result(processor.result, "Values in column: ")])

tool.add_stage("column_selector", column_selector)
```

Figure 4.1: A simple `Stage` to demonstrate how a table selected via a `TableSelectorComponent` can be iterated over and its rows indexed into. The `column_selector` function is used to define a `Stage` via the `add_stage` function. The `Stage` uses a `TableSelectorComponent` and `ColumnSelectorComponent` to solicit a choice of table and column from the user. The `Stage` uses a `LambdaProcessor` to process the user's choice of table and column. The `iterate_over_table` function iterates over each row of the selected table to accumulate a list of values that appear in the user-selected column. Note that the HiLT standard library offers a simpler way of selecting column values; this example is for demonstration purposes.

TableSelectorComponent

`TableSelectorComponent`s provide an interface element which allow users to preview and select a table from the database underlying the associated `Tool`. The `TableSelectorComponent`'s `value` attribute contains a representation of the selected table (Table 4.3). It implements Python's `__iter__` interface, facilitating iteration over the rows of the selected table. These rows can additionally be indexed into using column names or iterated over to retrieve cell values. An example of `TableSelectorComponent` and `ColumnSelectorComponent` usage appears in Figure 4.1.

Table 4.3: The `TableSelectorComponent` Class

Method/Field Signature	Description
`value: Table`	A programmatic representation of the selected table.
`append(other: dict, get_user_approvals: bool = False)`	Method to append a row to the selected table. The new row is specified by a dictionary mapping column names to cell values. `get_user_approvals` specifies whether the user should be asked to verify the append before committing the change.

Table 4.4: The `ColumnSelectorComponent` Class

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

Table 4.5: The `LambdaProcessor` Class

Method/Field Signature	Description
`func: Callable`	The function to be run to handle user input.
`num_return_vals: int`	The number of values expected to be returned from `func`.
`result`	The value, or tuple of values, returned by `func`.

`ColumnSelectorComponent`

The `ColumnSelectorComponent` provides an interface element which allows users to to select a predetermined number of columns from that table (Table 4.4).

4.3 Processors

Once users have provided input, a `Processor` controls how the input is used to transform data in the `Tool`'s backend. The main `Processor` sub-class provided is the `LambdaProcessor`. `LambdaProcessor`s (Table 4.5) allow HiLT programmers to handle user input by passing a Python function that will be run when the associated `Stage` receives user input. HiLT requires that the passed function does not accept arguments. By defining a `Processor` within a `Stage`-defining function, programmers can keep variable-bound `Component`s in scope, as demonstrated in Figure 4.1. Specifically, note that the `iterate_over_table` function is

Table 4.6: The `db_utils` module

Function Signature	Description
`create_table_from_csv(csv_file: FileUploadComponent)`	Parses the uploaded file and creates a new table in the underlying database.
`create_table_from_lists(data: list[list])`	Creates a new table populated with the passed data.
`get_table_from_table_name(table_name: str)`	Returns an iterable representation of the queried table.

Table 4.7: The `approvals` module

Function Signature	Description
`get_user_approvals()`	Collects changes requiring user approval and presents them in a single view to the user.

defined within the `table_iterator` SDF. Since `iterate_over_table` is defined within the `table_iterator Stage` function, it has access to `table` and `col`, which it uses to make a list of the values that appear in the user-selected column of the table.

4.4 db_utils

The `db_utils` module, included in HiLT's standard library, provides programmers with a number of utility functions to interact with a `Tool`'s underlying database. Three of these functions are described in Table 4.6.

4.5 approvals

HiLT also contains an `approvals` module, which allows programmers to solicit user confirmation for changes being made to the database backing a `Tool`. HILT API calls which make stateful changes expose a `get_user_approvals` option, allowing programmers to cache changes they would like users to confirm. Programmers must then call the `approvals` module's `get_user_approvals` function in order to confirm that they would like to seek user confirmation, and direct HiLT to display the approvals interface. An example of this flow can be seen in lines 7-12 of Figure 3.2.

Table 4.8: The `results` module

Function Signature	Description
`show_results(results: list[Result], results_title: str = '')`	Displays a list of `Result`s, after `Processor`s have been run and approvals handled.
`Result(value: Any, label: str = '')`	Instantiates a new `Result` object to be passed to `show_results`

4.6 `results`

The `results` module allows programmers to specify the view to be shown to the user after input processing has taken place. This is done by passing a list of `Result` objects to the module's `show_results` function. E.g., after creating a new table, a programmer may provide the created table as input to `show_results`, as in line 13 of Figure 3.2, which will produce a web page that displays the created table.

System Implementation Key Insights

While running ,Tools switch back and forth between two operating modes: *interface-construction* and *handling-POST*. At a high level, a `Tool` running in interface-construction mode is in the process of synthesizing the web page comprising the interface of a `Stage`. SDF calls to `Processors`, the `approvals` module, or the `results` module are ignored while in this mode. A `Tool` running in handling-POST mode is in the process of handling user input and SDF calls to `Processors`, the `approvals` module, or the `results` module are live and capable of modifying state.

When a programmer runs a `Tool`, it launches a locally running web server, allowing users to interact with the `Tool` via a web browser. By default, `Tools` show users a landing page linking them to each available `Stage`; an example landing page appears in Figure 5.1.

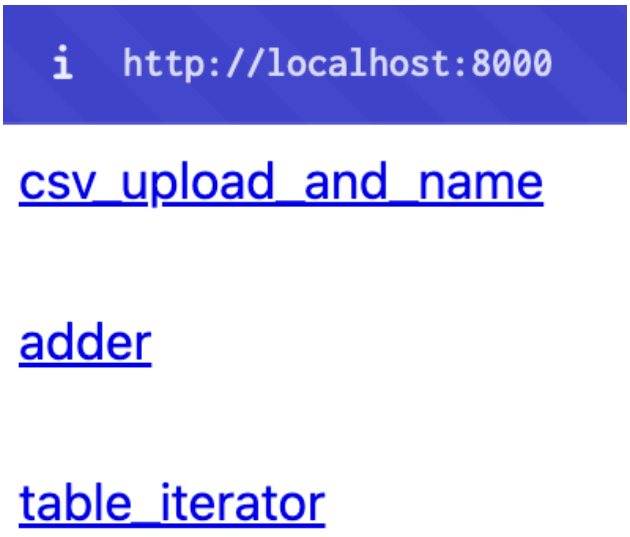


Figure 5.1: An example of a `Tool`'s landing page, accessed via the Arc web browser

When a user clicks on the link to a `Stage`, the `Tool` enters interface-construction mode and runs the `Stage`'s SDF associated with the `Tool` (Section 3.1). When an SDF run is triggered in this way, HiLT dynamically generates the user interface for that `Stage` and serves it as the response to the GET request generated by the user's click.

5.1 How API calls produce output UIs

`Stage` interfaces are generated dynamically each time a user clicks on the link for a particular `Stage` from a `Tool`'s landing page. When this happens, HiLT issues a GET request to the locally running web server, causing the `Tool` to run the `Stage`'s SDF in interface-construction mode. While in this mode, the SDF constructs the set of `Components` that will ultimately

```
def csv_upload_and_name():
    if len(db_utils.get_table_names_from_tool(tool)) == 0:
        table_name = UserInputComponent(str, label="Enter table name:")
        csv_file = FileUploadComponent('csv', label="Upload a csv file:")

tool.add_stage('csv_upload_and_name', csv_upload_and_name)
```

Figure 5.2: Code demonstrating how `Component` rendering can be controlled dynamically based on `Tool` state

be rendered in the user interface. Note that the set of `Components` associated with a `Stage` is *not* fixed pre-deployment. Rather, the SDF constructs the appropriate set dynamically at run-time; as a result, the programmer can design the HiLT-generated interface to vary based on prior user inputs. See Figure 5.2 for a simple example of varying the rendered `Components` based on prior user inputs.

While in interface-construction mode, HiLT ignores any `Processor` instantiations or API calls to the `results` or `approvals` modules which are made in the SDF being run.

Once the SDF has terminated, HiLT renders each `Component` instantiated in the SDF by converting it into an HTML snippet using the Jinja templating system [31]. Programmers can modify `Component` object attributes to alter the rendered `Components`' appearances and functionality—e.g., the programmer can provide a string to use as a label for a `Component`, or specify that user input to a `UserInputComponent` must have a given type. HiLT collates these HTML snippets into a single HTML document, again using a Jinja template. The resulting document comprises the output user interface for the `Stage`.

In total, there may be up to three web pages associated with a single `Stage`: the initial web page, created through the `Component` accumulation process described above; optionally, an approvals web page, if the programmer wants to give the user an additional opportunity for feedback as to which data transformations are propagated to the backend; and finally, again optionally, a results web page, which displays the results of the data transformations by showing part of the backend data. Figure 5.3 summarizes this topology.

5.2 How API calls and user interactions shape database interactions

When the user finishes providing inputs to the interface and clicks the submit button, HiLT issues a POST request to the running web server. This puts the `Tool` in handling-POST mode, and re-runs the SDF, which in turn re-instantiates each `Component`. While in handling-POST mode, `Components` that capture user input parse the relevant input from the POST

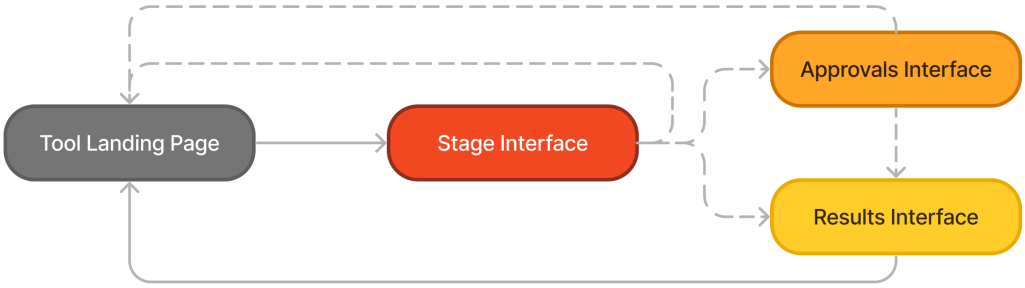


Figure 5.3: How users may move between different interfaces for a single `Stage`. Dashed lines indicate possible moves, and solid lines indicate moves that *always* happen from a given interface.

```
def adder():
    input1 = UserInputComponent(int, "Enter the first addend:")
    input2 = UserInputComponent(int, "Enter the second addend:")

    def add_inputs():
        return input1 + input2
    processor = LambdaProcessor(add_inputs, num_return_vals=1)
    result = processor.result
    results.show_results([results.Result(result, "Result of addition")], "Results: ")
```

Figure 5.4: SDF for a `Stage` adding user inputs

request, thus facilitating programmatic access to these values. Although these values are available in each `Component`'s `value` field, one goal we had when designing `Components` was to reduce boilerplate by allowing programmers to programatically treat `Component` variables *as if they were the values themselves*. In order to accomplish this, we overloaded many of Python's magic methods for various `Component` classes, allowing programmers to do things like iterate over the rows of a table (Figure 4.1) or add the values of two user inputs together (Figure 5.4).

When `Processors` are instantiated on the second run of the SDF, HiLT executes any associated transformations. In the case of `LambdaProcessors`, this amounts to running asso-

ciated programmer-defined function. Within these functions, programmers can manipulate the inputs associated with the `Stage`'s `Components`, and propagate changes to the `Tool`'s backing database. Each time a programmer defines and instantiates a new `Tool`, a fresh SQLite [41] database is created and wrapped inside the SQLAlchemy Python library [40]. Programmers can manipulate the state of these databases using functions from the `db_utils` module (Table 4.6). For example, a programmer can call the `create_table_from_csv` function to create a new table in the backing database. They can also modify existing tables via a number of `Component` methods (e.g., using the `TableSelectorComponent`'s `append` method to add new rows to a table).

By default, once the SDF terminates in handling-POST mode, the system redirects the user to the `Tool`'s landing page. If a call to the `results` module's `show_results` function is made while in handling-POST mode, then the passed `Result` objects are translated into HTML snippets and collated into a Jinja template in a process similar to interface construction during the initial run of the SDF. This result view is shown to the user once the SDF has terminated in handling-POST mode.

During interface-construction mode, a call to `get_user_approvals` sets a `get_approvals` flag on the running `Tool`. This flag remains active during handling-POST mode, and causes any API calls with *their* `get_user_approvals` option set to cache any changes they make, rather than propagating them to the underlying database. Then, when the SDF returns while in handling-POST mode, an interface prompting the user for confirmations is generated based on the cached changes. Once the user submits their inputs, the `Tool` goes into a `handling_approvals` mode, during which it propagates the confirmed changes, and afterwards returns the results view or landing page to the user.

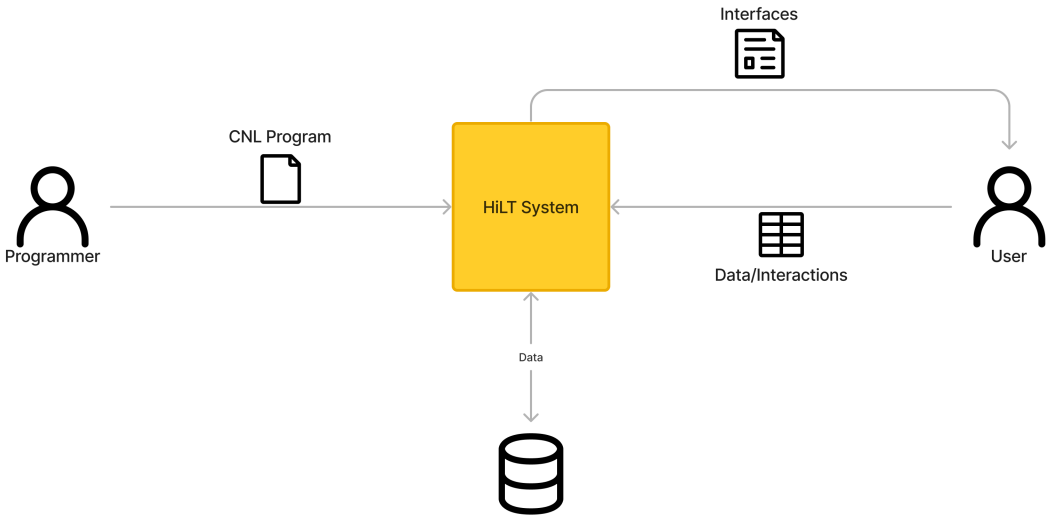


Figure 5.5: HiLT's system architecture

Formative User Study

To evaluate HiLT's usability and effectiveness in its stated design goals and to identify directions for future improvements to HiLT, we conducted a within-subjects user study with 17 participants. As comparison baselines, we used Django [6] (n = 9), and Streamlit [39] (n = 8). Each participant used HiLT and either Django or Streamlit.

6.1 Participants

We recruited participants with at least a basic level of prior experience with Python (the host language for HiLT, Django, and Streamlit). In all, we recruited 17 participants via UC Berkeley's Experimental Social Science Laboratory and by word of mouth. 12 participants had between two and six years of programming experience, two participants had at least six years of programming experience, and three participants had less than two years of programming experience. As compensation, each participant received $20 for each hour of participation, in the form of an Amazon gift card.

6.2 Tasks

We asked each participant to complete three tasks with each of two frameworks. Specifically, we asked participants to make a custom GUI for exploring information about recipients of the National Science Foundation's Graduate Research Fellowship Program (GRFP), by completing the following three tasks:

Task 1: File Upload: Let users upload a CSV file, give the uploaded dataset a name, and view the table represented in the CSV. Users should be able to view the datasets without having to re-upload them.

Task 2: Entity Matching: Let users create a table of applicants who were offered awards in both years. Users should be able to specify what criteria to match by; for example, by choosing the column containing names from each table and returning all applicants whose names appeared in both years. Users should also be able to specify which datasets they want to match. The matched data should be persisted in the underlying data store.

Task 3: Dataset Filtering: Let users filter a previously uploaded table by a value from a given column. For instance, if users select a table, a column from that containing school

names, and enter the name of a school, they should see a list and count of all applicants who attended that school for that year.

6.3 Protocols

We began each user session with an introduction and the informed consent process. Then, we assigned the participant two frameworks with which to complete the tasks: HiLT, and one of either Django or Streamlit. To mitigate learning effects, we balanced the order in which they used each framework across participants (i.e., half saw HiLT first, and half saw the alternative tool first). We then asked them to complete the three tasks using each framework. Each participant was given an hour total for each framework, during which we asked them to spend the first 15 minutes familiarizing themselves with the framework's documentation and tutorials. Throughout all parts of the sessions, participants were free to consult external resources like search engines or large language models (LLMs) to assist them.

After the hour they spent with each tool, each participant filled out a post-task survey. This survey asked them what they did and did not like about the framework they had just used. The survey also included a set of Likert-scale questions asking them to rate the framework's usability and learnability, as well as six NASA Task Load Index questions [13]. After attempting the tasks with both frameworks, they completed a final survey which asked which framework they preferred and collected demographic information.

Formative Study Results

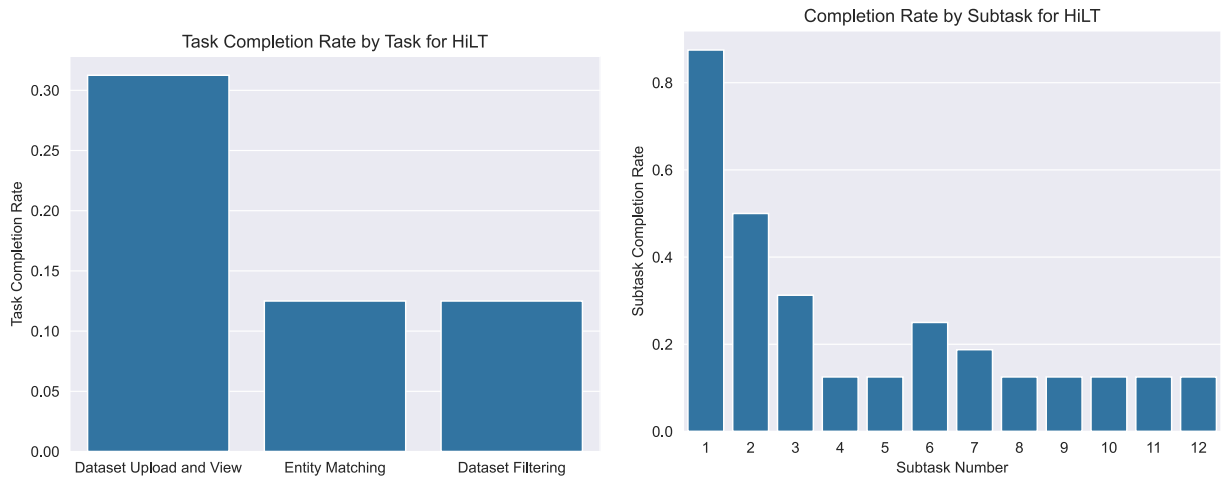
In order to evaluate each user's success at each task, we divided them into 12 total subtasks each with their own criteria, described in Table 7.1. Subtasks 1 through 3 form the criteria for Task 1, subtasks 4 through 8 the criteria for Task 2, and the remaining subtasks the criteria for Task 3.

7.1 Can users build real tools with HILT?

In general, participants struggled to use HiLT to complete all the tasks within the 45 minutes they were given (Figure 7.1). Only one user completed all three tasks in the allotted time. Overall, we observed that many participants were unaware of the existence of HiLT's `db_utils` module. Instead, these participants spent much of their time trying to wire up their own backend for their HiLT programs using libraries such as pandas [32]. See Section 8 for more discussion.

Table 7.1: Subtask Criteria

Subtask No.	Description
1	Can a user upload a CSV file?
2	Can a user name the CSV file they're uploading?
3	Can a user select and view a CSV file they're previously uploaded?
4	Can a user view matches across two datasets?
5	Are the matches correct?
6	Can a user specify what two datasets to match?
7	Does a user have some degree of control over the criteria used to match across two datasets?
8	Are the matches persisted in the underlying data store?
9	Can a user view the result of filtering over a single dataset?
10	Is the result correct?
11	Can a user specify what dataset to filter over?
12	Does a user have some degree of control over the criteria used to filter over a dataset?



(a) The completion rate for each task for HiLT users.

(b) The completion rate for each subtask for HiLT users.

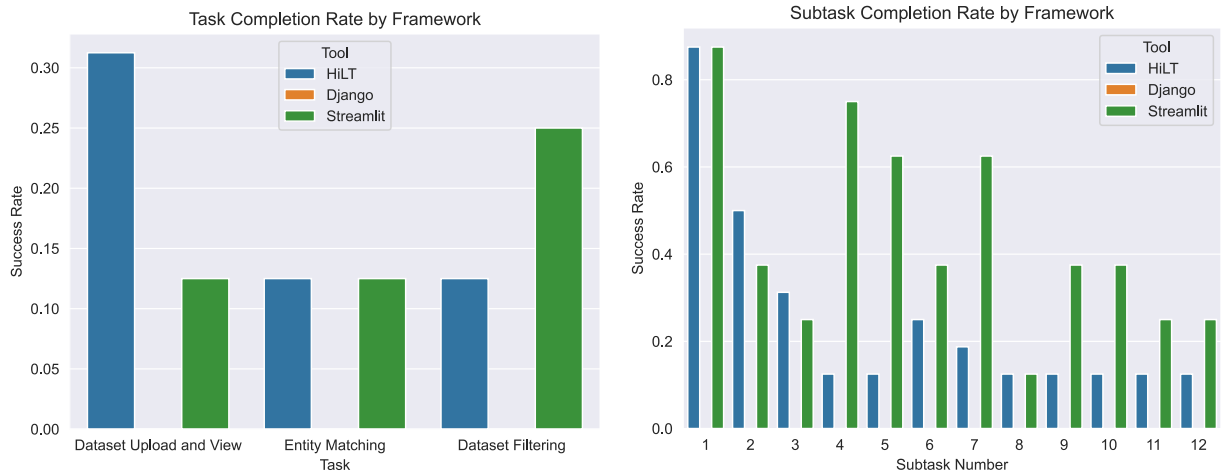
Figure 7.1: Participants did not usually complete the first task, and those who did often did not have the time to complete the second and third tasks (Subfigure 7.1a). Only one user completed all three, while two users completed Task 1 and only one of Tasks 2 or 3. Most participants were able to complete subtask 1, but subsequently struggled to complete subtask 3, which entailed saving uploaded datasets to the `Tool`'s underlying state (Subfigure 7.1b).

7.2 Can users complete more tasks and subtasks with HILT versus with a competitor tool like Django or Streamlit?

All participants who were assigned Django as an alternative baseline framework struggled to complete tasks. None of this subset of participants were able to complete any of the subtasks using Django (Figure 7.2). Participants using Streamlit were markedly more successful with it, with 2 of the 8 participants completing the tasks in full. We discuss the differences in their processes with HiLT and Streamlit in depth in Section 8.

7.3 What is the workload (TLX) of HILT vs. Django?

Figure 7.3 shows the average TLX responses grouped by Tool. Across all indices but Effort, the reported subjective workload was highest for participants using Django. In general, participants gave lower responses for Streamlit than HiLT, indicating that they felt that Streamlit was lower-workload than HiLT.



(a) Completion rate for each task, grouped by framework used

(b) Completion rate for each subtask, grouped by framework used

Figure 7.2: Across the board, participants did not complete any subtasks using Django in the time allotted. As seen in Subfigure 7.2a, more participants completed Task 3 with Streamlit than Task 1, as one participant did not allow users to name their uploaded datasets. No participants completed Task 2 with Streamlit, as none of them persisted the results of matches. This is also reflected in Subfigure 7.2b, under Subtask 8.

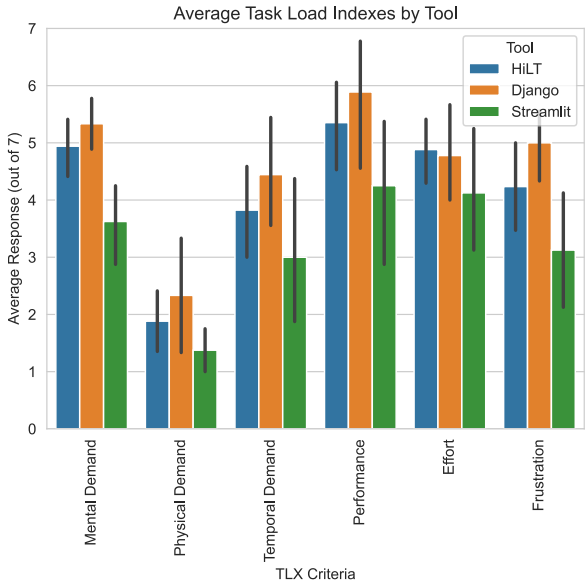


Figure 7.3: Average TLX Responses by Tool. Error bars show 95% confidence intervals.

Discussion

In this section, we reflect on the patterns we observed in the formative study, and suggest directions for how HiLT might be improved for the future. We plan to conduct a qualitative analysis of the data collected during the user study in order to more thoroughly explore potential areas of improvement.

8.1 LLM Support

Throughout the user study, we observed almost all participants at some point querying an LLM for assistance. Streamlit was first introduced in 2019, which gave it a marked advantage in this regard, as LLMs like ChatGPT [28] were able to output functional programs when prompted by participants. Indeed, one participant using Streamlit finished most subtasks in 20 minutes after copying the task prompt into ChatGPT and using the resulting program. As a side note, we observed that participants were unable to prompt LLMs to output functional Django programs, most likely due to the extensive boilerplate the framework requires. They were also unable to output functional HiLT programs, despite even the best efforts of one participant who copied the entirety of the HiLT documentation into ChatGPT.

8.2 Reasoning About First-Class Functions

To author HiLT programs, programmers must pass functions as arguments to other functions; for example, they author separate SDFs for each `Stage`, and pass functions nested in SDFs to `LambdaProcessors`. First-class functions may be a fairly sophisticated concept, especially for participants with limited programming experience, and we believe that this posed a conceptual hurdle to many of our participants. We observed in particular that participants with limited Python experience struggled to reason about Python's use of indentations for function scopes. In contrast, Streamlit allows interfaces to be generated and backend processing defined through function calls to Streamlit's API, without the use of first-class functions. Indeed, programmers can use Streamlit without defining any functions themselves.

8.3 Generalizing Programs Beyond Fixed Data Files

Another possible differentiating factor between Streamlit and HiLT is that HiLT is intended to author interfaces which can accomplish data transformation tasks *in which users bring their own data*. However, we gave participants only two sample datasets for testing their programs. As a possible consequence, 6 of the 8 participants who authored Streamlit programs hard-coded for those datasets in some way—e.g., asking users to upload exactly two datasets or hardcoding the expectation that they would receive a dataset named "dataset2022" and another "dataset2023". In contrast, participant-authored HiLT programs that successfully completed at least one task were always agnostic to the shape of user-uploaded data.

8.4 Data Transformation

More broadly, no Streamlit participants persisted data about matches (Task 2 of the formative study). Rather, they completed the tasks that could be completed via data display alone. Although our formative study has suggested important pathways for improvement of HiLT, the fact that the Streamlit participants did not in fact produce data transformation GUIs but rather data display GUIs suggests that any future evaluative study will need fresh task design to effect a head-to-head comparison with HiLT specifically in the space of custom data transformation GUIs.

Conclusion

Domain experts performing data transformation tasks and lacking programming experience often have their needs unmet by the current ecosystem of open-ended data analysis tools. At the same time authoring bespoke, guided data transformation tools for them to use on a case-by-case basis is difficult in the current landscape of tools. Only 3 of the 17 participants successfully produced HiLT GUIs which persisted the outputs of users' data transformations, and 0 of 16 participants achieved this using non-HiLT alternatives. This suggests an open need for improved tools for authoring custom human-in-the-loop data transformation GUIs.

This paper presents the preliminary design of HiLT, a domain-specific language for authoring data transformation GUIs which aims to fill this gap. We conducted a formative user study with 17 participants in order to identify directions for future development. For future work, we plan to conduct a qualitative data analysis of our user sessions, and iterate on HiLT before conducting evaluative user studies of programmers using HiLT and end users using the output GUIs.

Bibliography

[1] Alteryx. *Alteryx Designer Cloud*. en-US. 2024. URL: https://www.alteryx.com/products/designer-cloud (visited on 05/02/2024).

[2] Tom Beckmann et al. "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. New York, NY, USA: Association for Computing Machinery, Apr. 2023, pp. 1–16. ISBN: 978-1-4503-9421-5. DOI: 10.1145/3544548.3580785. URL: https://dl.acm.org/doi/10.1145/3544548.3580785 (visited on 05/09/2024).

[3] Marcel Borowski et al. "Varv: Reprogrammable Interactive Software as a Declarative Data Structure". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–20. ISBN: 978-1-4503-9157-3. DOI: 10.1145/3491102.3502064. URL: https://doi.org/10.1145/3491102.3502064 (visited on 05/09/2024).

[4] Yiru Chen, Jeffrey Tao, and Eugene Wu. "DIG: The Data Interface Grammar". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '23. New York, NY, USA: Association for Computing Machinery, July 2023, pp. 1–7. ISBN: 9798400702167. DOI: 10.1145/3597465.3605223. URL: https://dl.acm.org/doi/10.1145/3597465.3605223 (visited on 05/09/2024).

[5] Yiru Chen and Eugene Wu. "PI2: End-to-end Interactive Visualization Interface Generation from Queries". In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 1711–1725. ISBN: 978-1-4503-9249-5. DOI: 10.1145/3514221.3526166. URL: https://dl.acm.org/doi/10.1145/3514221.3526166 (visited on 05/09/2024).

[6] Django Software Foundation. *Django*. en. 2024. URL: https://www.djangoproject.com/ (visited on 03/10/2024).

[7] Ian Drosos et al. "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. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12. ISBN: 978-1-4503-6708-0. DOI:

10.1145/3313831.3376442. URL: https://dl.acm.org/doi/10.1145/3313831.3376442 (visited on 03/10/2024).

[8] Google. *Blockly — Google for Developers*. en. 2024. URL: https://developers.google.com/blockly (visited on 05/10/2024).

[9] Google. *Dataprep by Trifacta*. en. 2024. URL: https://cloud.google.com/dataprep (visited on 05/06/2024).

[10] Google. *Google Forms: Online Form Creator — Google Workspace*. en. 2024. URL: https://www.google.com/forms/about/ (visited on 05/09/2024).

[11] Google. *Google Sheets: Online Spreadsheet Editor — Google Workspace*. en. 2024. URL: https://www.google.com/sheets/about/ (visited on 05/09/2024).

[12] Philip J. Guo et al. "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. New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 65–74. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047205. URL: https://dl.acm.org/doi/10.1145/2047196.2047205 (visited on 03/10/2024).

[13] Sandra G. Hart and Lowell E. Staveland. "Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research". In: *Advances in Psychology*. Ed. by Peter A. Hancock and Najmedin Meshkati. Vol. 52. Human Mental Workload. North-Holland, Jan. 1988, pp. 139–183. DOI: 10.1016/S0166-4115(08)62386-9. URL: https://www.sciencedirect.com/science/article/pii/S0166411508623869 (visited on 05/02/2024).

[14] Holoviz. *Overview — Panel v1.4.2*. 2024. URL: https://panel.holoviz.org/ (visited on 05/09/2024).

[15] Joshua Horowitz and Jeffrey Heer. "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. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 1–18. ISBN: 9798400701320. DOI: 10.1145/3586183.3606733. URL: https://dl.acm.org/doi/10.1145/3586183.3606733 (visited on 05/09/2024).

[16] Kevin Hu, Diana Orghian, and César Hidalgo. "DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 1–7. ISBN: 978-1-4503-5827-9. DOI: 10.1145/3209900.3209910. URL: https://dl.acm.org/doi/10.1145/3209900.3209910 (visited on 03/11/2024).

[17] iPython. *Jupyter and the future of IPython — IPython*. 2024. URL: https://ipython.org/ (visited on 05/10/2024).

[18] JMP. *Statistical Software*. en. 2024. URL: https://www.jmp.com/en_us/home.html (visited on 03/11/2024).

[19] Sean Kandel et al. "Wrangler: interactive visual specification of data transformation scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 3363–3372. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979444. URL: https://dl.acm.org/doi/10.1145/1978942.1979444 (visited on 03/10/2024).

[20] Mary Beth Kery et al. "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. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 140–151. ISBN: 978-1-4503-7514-6. DOI: 10.1145/3379337.3415842. URL: https://dl.acm.org/doi/10.1145/3379337.3415842 (visited on 03/10/2024).

[21] Xingjun Li et al. "EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In Situ Code Search and Recommendation". In: *ACM Transactions on Interactive Intelligent Systems* 13.1 (Mar. 2023), 1:1–1:27. ISSN: 2160-6455. DOI: 10.1145/3545995. URL: https://dl.acm.org/doi/10.1145/3545995 (visited on 05/10/2024).

[22] Microsoft. *Data Analysis Expressions (DAX) Reference - DAX*. en-us. 2024. URL: https://learn.microsoft.com/en-us/dax/ (visited on 03/10/2024).

[23] Microsoft. *Free Online Spreadsheet Software: Excel — Microsoft 365*. en-US. 2024. URL: https://www.microsoft.com/en-us/microsoft-365/excel (visited on 05/09/2024).

[24] Microsoft. *Power BI - Data Visualization — Microsoft Power Platform*. en-US. 2024. URL: https://www.microsoft.com/en-us/power-platform/products/power-bi (visited on 05/02/2024).

[25] Kristi Morton et al. "Dynamic workload driven data integration in tableau". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 807–816. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213961. URL: https://dl.acm.org/doi/10.1145/2213836.2213961 (visited on 03/08/2024).

[26] Mashaal Musleh et al. "CoClean: Collaborative Data Cleaning". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, May 2020, pp. 2757–2760. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3384698. URL: https://dl.acm.org/doi/10.1145/3318464.3384698 (visited on 03/10/2024).

[27] Cyrus Omar et al. "Filling typed holes with live GUIs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 511–525. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454059. URL: https://dl.acm.org/doi/10.1145/3453483.3454059 (visited on 05/09/2024).

[28] OpenAI. *Introducing ChatGPT*. en-US. Nov. 2022. URL: https://openai.com/index/chatgpt (visited on 05/02/2024).

[29] OpenRefine. *General Refine Expression Language — OpenRefine*. en. Feb. 2024. URL: https://openrefine.org/docs/manual/grel (visited on 03/10/2024).

[30] OpenRefine. *OpenRefine*. en. 2024. URL: https://openrefine.org/ (visited on 05/09/2024).

[31] Pallets. *Jinja — Jinja Documentation (3.1.x)*. 2007. URL: https://jinja.palletsprojects.com/en/3.1.x/ (visited on 05/02/2024).

[32] pandas. *pandas - Python Data Analysis Library*. 2024. URL: https://pandas.pydata.org/ (visited on 05/02/2024).

[33] Plotly. *Dash Documentation & User Guide — Plotly*. 2024. URL: https://dash.plotly.com/ (visited on 05/09/2024).

[34] Project Jupyter. *Jupyter Widgets — Jupyter Widgets 8.1.2 documentation*. 2023. URL: https://ipywidgets.readthedocs.io/en/stable/ (visited on 05/10/2024).

[35] Qualtrics. *Qualtrics XM - Experience Management Software*. en. 2024. URL: https://www.qualtrics.com/ (visited on 05/09/2024).

[36] Rails Foundation. *Ruby on Rails*. en. 2024. URL: https://rubyonrails.org/ (visited on 03/10/2024).

[37] Salesforce. *Business Intelligence and Analytics Software — Tableau*. en-US. 2024. URL: https://www.tableau.com/ (visited on 05/02/2024).

[38] SAS. *SAS: Analytics, Artificial Intelligence and Data Management*. en. 2024. URL: https://www.sas.com/en_us/home.html (visited on 03/11/2024).

[39] Snowflake. *Streamlit • A faster way to build and share data apps*. en. Jan. 2021. URL: https://streamlit.io/ (visited on 05/01/2024).

[40] SQLAlchemy. *SQLAlchemy*. en. 2024. URL: https://www.sqlalchemy.org (visited on 05/02/2024).

[41] SQLite. *SQLite Home Page*. 2024. URL: https://www.sqlite.org/ (visited on 05/02/2024).

[42] Gradio Team. *Gradio*. en-US. 2024. URL: https://gradio.app (visited on 05/09/2024).

[43] Carol Traynor and Marian G. Williams. "Chapter 6 - End Users and GIS: A Demonstration Is Worth a Thousand Words". In: *Your Wish is My Command*. Ed. by Henry Lieberman. Interactive Technologies. San Francisco: Morgan Kaufmann, Jan. 2001, pp. 115–VI. ISBN: 978-1-55860-688-3. DOI: 10.1016/B978-155860688-3/50007-5. URL: https://www.sciencedirect.com/science/article/pii/B9781558606883500075 (visited on 05/01/2024).

[44] Priyan Vaithilingam and Philip J. Guo. "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. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 563–576. ISBN: 978-1-4503-6816-2. DOI: 10.1145/3332165.3347944. URL: https://dl.acm.org/doi/10.1145/3332165.3347944 (visited on 05/09/2024).

[45] Vercel. *Next.js by Vercel - The React Framework*. en. 2024. URL: https://nextjs.org/ (visited on 03/10/2024).

[46] Lea Verou, Amy X. Zhang, and David R. Karger. "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. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 483–496. ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984551. URL: https://dl.acm.org/doi/10.1145/2984511.2984551 (visited on 03/10/2024).

[47] Chat Wacharamanotham et al. "Statsplorer: Guiding Novices in Statistical Analysis". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 2693–2702. ISBN: 978-1-4503-3145-6. DOI: 10.1145/2702123.2702347. URL: https://doi.org/10.1145/2702123.2702347 (visited on 05/10/2024).

[48] Rachel B. Warren and Niloufar Salehi. "Trial by File Formats: Exploring Public Defenders' Challenges Working with Novel Surveillance Data". In: *Proceedings of the ACM on Human-Computer Interaction* 6.CSCW1 (Apr. 2022), 67:1–67:26. DOI: 10.1145/3512914. URL: https://dl.acm.org/doi/10.1145/3512914 (visited on 04/29/2024).