Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language

Robert DeLine
rdeline@microsoft.com
Microsoft Research

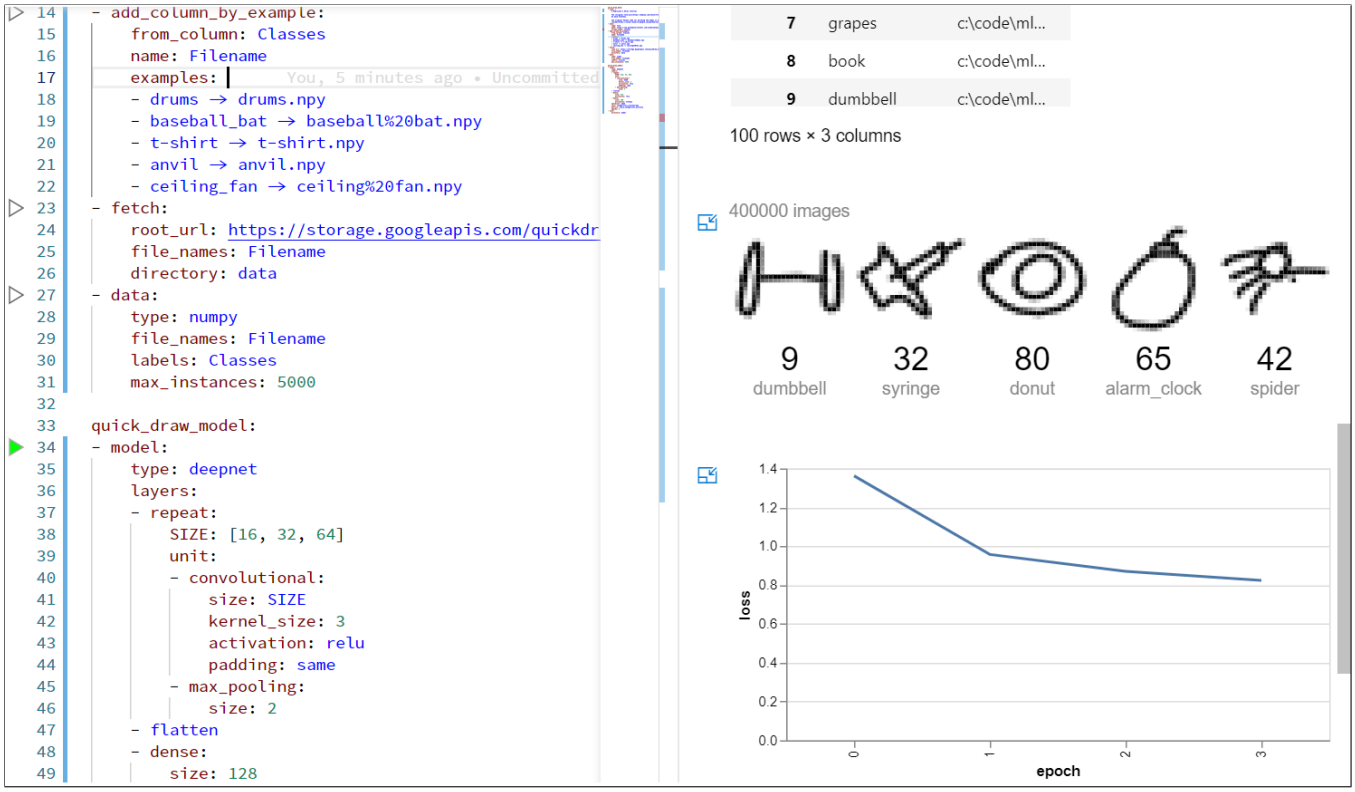

Figure 1: Glinda provides live programming for a domain-specific language for data science workflows, including reading, cleaning, and transforming data and building models.

ABSTRACT

Researchers have explored several avenues to mitigate data scientists' frustrations with computational notebooks, including: (1) live programming, to keep notebook results consistent and up to date; (2) supplementing scripting with graphical user interfaces (GUIs), to improve ease of use; and (3) providing domain-specific languages (DSLs), to raise a script's level of abstraction. This paper introduces Glinda, which combines these three approaches by providing a live programming experience, with interactive results, for a domain-specific language for data science. The language's compiler uses an open-ended set of "recipes" to execute steps in the user's data science workflow. Each recipe is intended to combine the expressiveness of a written notation with the ease-of-use of a GUI. Live programming provides immediate feedback to a user's input, whether in the form of program edits or GUI gestures. In a qualitative evaluation with 12 professional data scientists, participants highly rated the live programming and interactive results. They found the language productive and sufficiently expressive and suggested opportunities to extend it.

CHI '21, May 8–13, 2021, Yokohama, Japan

ACM ISBN 978-1-4503-8096-6/21/05…$15.00
https://doi.org/10.1145/3411764.3445267

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

data science, exploratory programming, domain-specific language, live programming

ACM Reference Format:
Robert DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3411764.3445267

1 INTRODUCTION

Computational notebooks have become a popular tool for data scientists to analyze data and build models. Nonetheless, notebook users experience a variety of frustrations [4], which have inspired researchers to enhance the notebook user experience. One frustration is due to the notebook's flexible execution model, in which a user organizes script code into separately executable units (cells). While this allows careful control over the order of execution, users complain about "messy" notebooks with dead code, missing code, and stale results [11] and about the rigor required to ensure that results are replicable [4]. In response, researchers have invented tools for sense-making over execution history [16, 18] and for automatically cleaning notebooks [11]. Other tools, like Tempe [6] and Streamlit[1], propose live programming as an alternative to cell-based execution. Live programming ensures that a script and its results are consistent and up to date, but provides less control over execution.

Beyond execution, a second problem with notebooks is the reliance on scripting to accomplish most steps in a user's workflow, including basic plots. This is in contrast to data tools like spreadsheets, Tableau [27], and PowerBI, whose graphical user interfaces (GUIs) allow users to explore datasets and produce plots and dashboards. Scripting and GUIs represent a well-known trade-off between expressiveness and ease-of-use [12]. To combine the benefits of both, researchers have enhanced notebooks with GUIs that write script code on a user's behalf. This generated code makes library calls that are equivalent to a user's GUI gestures [19], transforms data through programming by example [7], or exposes a user's GUI selections as a scriptable dataset [31]. Converting a user's GUI actions into script code ensures that the actions are a persistent, replicable part of the notebook.

Another way to simplify scripting is to raise the level of abstraction with a domain-specific language (DSL). Some DSLs for data science are visual; some are textual. A visual language like Lobe[2] or Azure ML Studio[3] presents a palette of drag-and-drop components, which a user composes by forming connections between them. By chaining components, a user can create entire workflows, from data collection, cleaning and labeling, to model training, evaluation and deployment. Textual DSLs, like Vega-Lite [25] for visualization and Tea [14] for statistical analysis, provide a declarative, high-level notation, with a compiler that intelligently fills in details. The Vega-Lite compiler, for instance, chooses rendering details; the Tea compiler chooses appropriate statistical tests. Both GUIs and DSLs employ a similar tactic of presenting a user with fewer, higher-level choices, with the goal of increasing productivity and reducing errors.

This paper introduces Glinda (GUI and Language for Interactive Data Analysis), which combines live programming, GUIs, and a DSL into a novel user experience for data science work. Glinda is an extension to Visual Studio Code that provides a live programming experience, with interactive results, for a domain-specific language for data science (Fig. 1). The language is written in YAML, a popular markup language with a lean syntax. A Glinda file describes a set of named *workflows*, each with high-level, declarative *steps* for reading, cleaning, transforming, and visualizing data and for building, testing and deploying ML models. Executing a step in the language produces an interactive visualization in the right-hand pane, which provides immediate feedback about the step's results and progressive updates about ongoing computation. The example in Fig. 1 trains a deep net on image data. After each training epoch, the training curves update to show the latest loss and accuracy, in the style of TensorBoard.[4]

All the visualizations are interactive and allow a user to browse data in detail, for example, by sorting and filtering table columns or by hovering over plot points to see tooltips. Some visualizations also allow GUI-based input, which provides a second way to edit the YAML content. One example is a step for learning string transformations by example (Fig. 2). When a user enters this step, the visualization pane shows an editable table, which allows the learned values to be corrected in place (A). This in-place correction causes the new example to be added to the YAML (B), which in turn causes the live programming algorithm to re-run. As this example illustrates, the GUI provides an intuitive editing experience, the DSL captures the high-level intent, and live programming keeps these two consistent and up to date.

To accommodate the variety of current and future workflows, Glinda's compiler has an extensible architecture. To run a Glinda file, the compiler translates YAML to Python, using an open-ended set of *recipes*. Syntactically, a recipe is a Python function whose docstring contain a YAML template. If a step's YAML matches a recipe's template, that step is compiled as a call to the recipe's function. At a design level, a recipe simultaneously provides (1) a syntax for capturing the intent behind a high-level operation and (2) a user interface for providing feedback about computational progress and results and (where advantageous) taking GUI-based input to affect the computation. That is, each recipe is intended to provide a thoughtful user experience for accomplishing one step in a data science task.

This paper makes the following contributions:

- We introduce a user experience that extends an existing development environment with interactive, exploratory features to support data science work.
- We provide a declarative, domain-specific language for data science workflows, with an extensible compiler architecture.
- We report the results of a qualitative evaluation with 12 professional data scientists doing exploratory data analysis.

[1] https://streamlit.io/
[2] http://lobe.ai
[3] https://studio.azureml.net
[4] https://www.tensorflow.org/tensorboard/

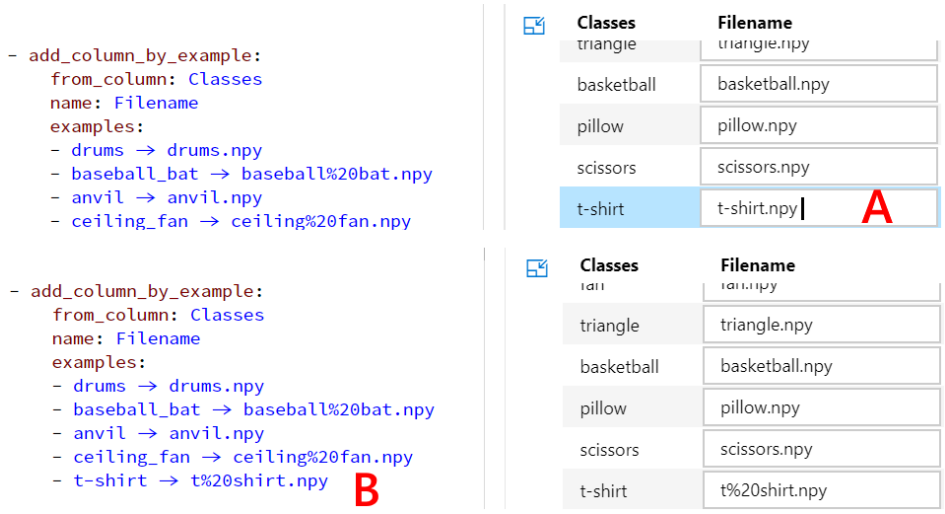

Figure 2: Many Glinda steps combine text editing with GUI interactions. Here, using the `add_column_by_example` recipe produces an editable table. When a learned value is wrong, the user can correct it in place (A). The user's correction is then added as a new example to the YAML (B).

2 BACKGROUND AND RELATED WORK

2.1 Computational Notebooks

Computational notebooks have become a popular environment for data science, particularly the exploratory stages [22, 24]. This user experience allows a data scientist to create an arbitrary number of small editors, called cells, each of which submits script code to an interpreter session. Any visual response to a cell's executed code is shown directly below the cell. As such, a typical notebook is a kind of literate program [21] that interleaves code and visual results, as well as documentation written in markdown.

Combining an interpreter session with visual results make notebooks an effective environment for iterative data exploration [17]. However, current notebooks environments, like Jupyter Lab, Colab, and Databricks, are not without their problems [4]. Enhancing the notebook experience has become a popular research topic. One issue is the problem of "messy" notebooks. The cell model is flexible but also problematic for producing clear, replicable results: cells can be executed in any order, leading to a "spaghetti" organization of cells; cells are the only way to get feedback, leaving behind many cells with short-term value; and cells can be overwritten or deleted, which allows inconsistencies between the cell contents and the interpreter execution state. Verdant [18] provides a light-weight versioning system for notebooks to allow data scientists to understand and retrieve alternatives they have explored. Gather [11] helps data scientists "clean" notebooks by using program analysis to produce a subset of cells that are needed to produce a chosen result. Glinda's live programming explores a different point in the design space where these inconsistencies cannot occur, but the user gives up direct control over execution.

Another shortcoming of notebooks is that the work is done through scripting, with visualizations merely reflecting the scripting code. That is, scripting is primary; visualization is secondary. Recent research aims to improve parity between the two. Mage [19] provides an API to facilitate the creation of task-specific GUIs within the notebook, for example, for editing images, doing test/train splits, or exploring confusion matrices. Wrex [7] allows a data scientist to provide examples of data transformations inside a table tool in the notebook. Wrex then synthesizes readable code from these examples, for the data scientist to run in a cell. B2 [31] records interactions with notebook visualizations to turn the user's data selections and filtering into script variables that can be computed over. This parity between scripting and interactive visualizations is also an ingredient of Glinda.

2.2 GUI-based Data Science Tools

Another approach to data science tools are environments with graphical user interfaces. Commercial tools like Tableau [27] or PowerBI and academic tools like Voyager [30] allow users to load, manipulate, explore, visualize and create reports and dashboards about data. While these tools' user interfaces provide a wide range of functionality, they lack open-ended nature of programming environments. Visual programming languages for data science, like Azure ML Studio and `lobe.ai`, combine some of the benefits of programming environments with the direct manipulation style of GUI-based tools. These environments provide an extensible palette of parts that can be dragged onto a canvas and connected to create pipelines. For example, a box with a live camera feed could be fed to an image processing box, which in turn is fed to a deep net model to form a prediction. These environments are essentially as general as programming environments, but with a GUI that scaffolds program construction to avoid certain classes of errors. Glinda is similar in spirit to these visual programming languages, but designed to fit within a textual programming environment. Code completion and code snippets act as a palette of parts to aid construction, and its declarative syntax is higher-level and less technical than scripting languages.

2.3 Domain-specific Languages

Another technique for making programming more accessible is to design a domain-specific language (DSL), which aims for a directness of mapping between domain concepts (for example, finance) and the syntax and semantics of the language. There are two kinds of DSLs: an external DSL uses a traditional compiler infrastructure and therefore has full control over syntax and semantics; an internal DSL is a stylized API in an existing general-purpose language (often one with flexible syntax and expressive type systems, like Haskell or F#) [9]. There are many DSLs for data science work, particularly data queries, which Makrynioti and Vassalos have recently surveyed [23]. Many of these are designed not so much to support the end user, but for a high-performance implementation, for example ML2SQL [26] or SystemML [2].

Some tools have DSLs for data science workflows, with the goal of automating batch-style execution, particularly for replicability. The Worflow Description Language [5] chains together the steps of a data science workflow (which are often shell scripts), both to make them understandable and to automate execution. The deep net toolkit Caffee uses JSON as a declarative notation for describing deep-net architectures to automate training [13]. Glinda is similar to these, but with an emphasis on the editing experience, rather than batch execution.

2.4 Live programming

Live programming (also called reactive programming) is a user experience in which a programmer's edits have an immediate and visible effect [28]. This immediacy is intended both to increase productivity and to improve comprehension [15]. The technique is particularly useful in visual domains, like the design of user interfaces [3, 10] and visualizations in data science (Tempe [6], Observable[6]). In the context of data science, live programming eliminates some of the inconsistency problems that the notebook cell model creates. Glinda re-implements Tempe's algorithm for live programming, but without the emphasis on live streaming data [5]. Tools like Tempe and Observable track data dependencies to avoid unnecessary computation as values change. On the other hand, Streamlit, a live-programming framework for creating data apps, uses a naive implementation (continuous whole-program re-execution, with user-provided hints about which computation to memoize), but is available as a Python library, rather than an environment to adopt.

3 DESIGN OVERVIEW

Glinda provides two complementary ways for data scientists to express their intentions about their workflow. The first is through a domain-specific language (DSL), which breaks down the workflow into a series of named steps. Our design goal is for each step to represent a logical unit of work, expressed in plain-spoken terms, with useful defaults to allow the omission of unknown or uninteresting details. The second way is through interactive visualizations, whose GUI style of interaction complement the DSL's text editing interactions. That is, the design goal is for each step to be expressed through a thoughtful combination of text editing and GUI interactions. Live programming is the engine that continually keeps these two mutually consistent and up to date.

In addition, as an extension to Visual Studio Code, Glinda is intended to fit comfortably within a development environment. The Glinda user experience is launched whenever the user edits a file with the extension `.gda`. Opening such a file creates a second document tab that contains the interactive visualizations. Glinda synchronizes the scrolling in these two documents, so that a selected DSL construct and its corresponding GUI are both highlighted and visible. Glinda also uses familiar editor features, like code completion, error diagnostics, and margin icons, to aid in program construction and feedback. A Glinda file sits side by side with other source files in a project. Features like source control and real-time collaborative editing work on `.gda` files like any other text files.

To give a sense of Glinda's live programming, we describe some key activities our user study. This section describes several of the steps that our participants completed while following the study's tutorial. The tutorial analyzes a small dataset about passengers aboard the Titanic. As a first step, a participant enters the following YAML text into an empty `.gda` file:

```
titanic:
- data:
  type: file
  path: /data/titanic.csv
```

A Glinda file consists a collection of named *workflows*, where each workflow is a list of *steps* that read data, transform data, visualize data, and build models from data. In the text above, the participant defines a single workflow named `titanic`, with a single step that begins with `data`.

To give this YAML a computational meaning, Glinda keeps an extensible collection of *recipes*. A recipe is a Python function with a special docstring that contains a YAML template. To compile a step, the Glinda compiler searches all recipes to find one whose template matches the step's YAML. If a matching recipe is found, the step is compiled into a call to the recipe function. If no match is found, the compiler reports an error. The matching algorithm is described in Section 4.1. In this first step, the YAML matches a recipe that uses the Pandas library to read a CSV file into a dataset. A few seconds after entering the YAML text, a table appears in the visualization pane (Fig. 3, top right). This table shows the size of the dataset, its schema, and a sample of its rows (in this case, the entire dataset). To allow the user to browse the data, the table provides standard features for resizing, sorting and filtering columns.

Continuing the tutorial, a participant next uses code completion to enter a second workflow step (Fig. 3, top). The code completion menu shows all available recipes. The participant chooses `model (decision_tree)`, which causes the following code snippet to be inserted:

```
- model:
  type: decision_tree
  target:
  uses:
```

To fill in the snippet placeholders (gray boxes), the participant can again use code completion. The key `target` expects a column name

[5] https://github.com/openwdl/wdl
[6] http://www.observablehq.com

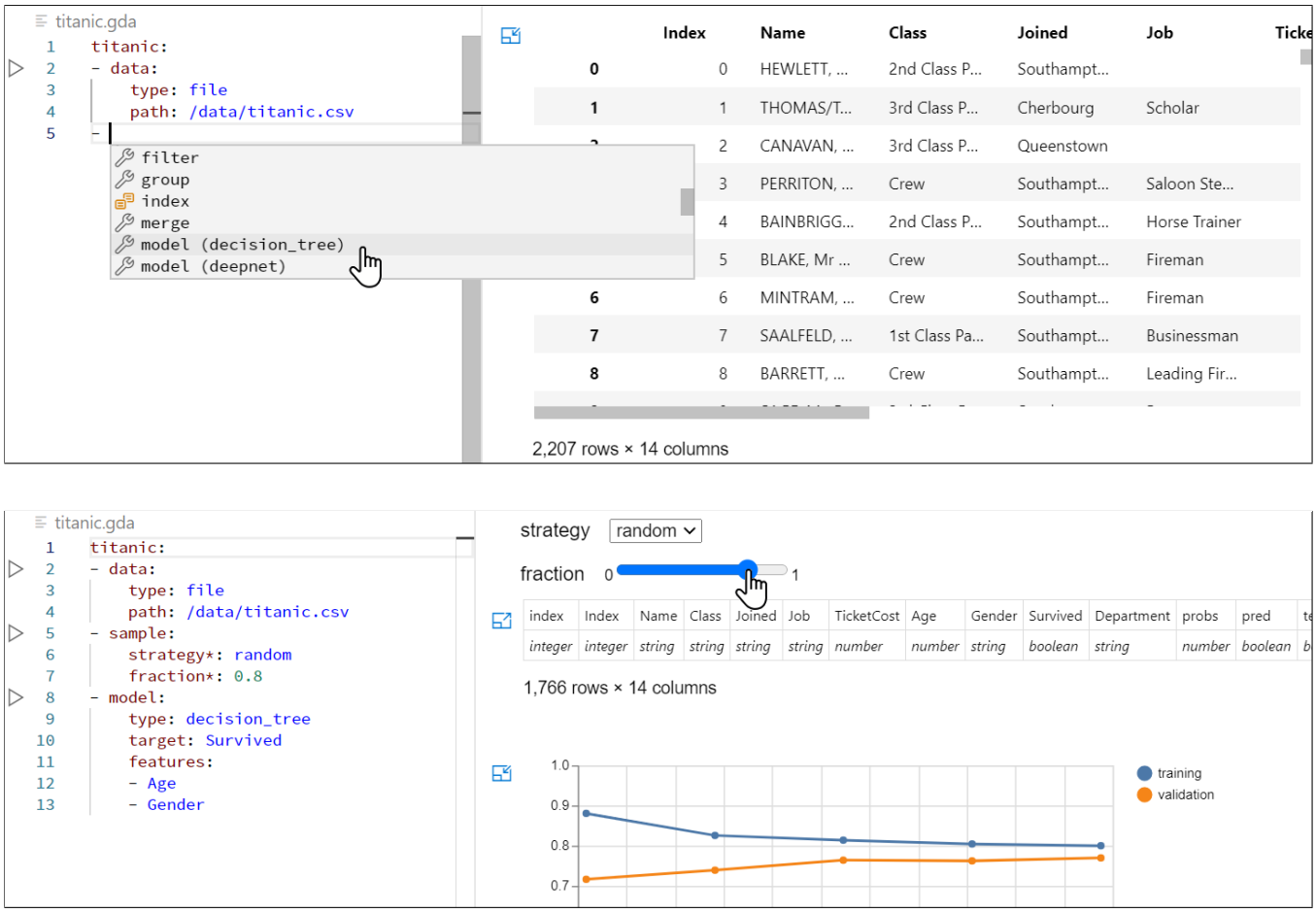

Figure 3: (top) After a user enters the text on lines 1–4, the table visualization on the right immediately appears because of Glinda's live programming. Glinda provides code completions, which generate YAML snippets. (bottom) Steps in Glinda chain together. Adjusting the slider in the `sample` step changes the data sample flowing into the `model` step. This causes the `model` step to re-run and the learning curve to update.

as a value, so invoking code completion there brings up a list of the active columns. The key `uses` expects a list of columns, which can be similarly filled in with code completions. The combination of a declarative language and code snippets makes writing Glinda programs less like programming and more like filling in a form.

Once the participants has chosen a `target` column and at least one `uses` column, live programming invokes the `model` recipe to build a decision tree. Within a few seconds, the model's training curves appear in the visualization pane (Fig. 3, bottom right). The participant sees that the model achieves a final accuracy around 80%. The tutorial then encourages the participant to comment and uncomment the line containing `Age`, which alternately adds and removes that feature from the model. The participant can watch the training curve alternately shift between 75% and 80%.

A workflow's steps are combined through function composition: the output from one step is the input to the next step. (The tidyverse APIs in R use the same concept, with an explicit piping syntax.) So far, the participant has two steps: a data-reading step that flows into a model-training step. The tutorial next asks the participant to insert a sampling step between these two (Fig. 3, bottom). To change the sampling strategy or size, the participant can edit the corresponding key values in the text. Alternatively, by adding an asterisk next to the key name, Glinda adds an appropriate GUI element for updating that key's value, in this case, a drop-down list for `strategy` and a slider for `fraction`. (These elements are similar to Juxtapose's tuning variables [10].) Interacting with these elements updates the YAML text and triggers live programming. The newly updated sample flows into the model-building step, which then rebuilds the model and updates the learning curve.

In short, the participants have several means for updating the YAML content—text operations, code completions, code snippets, GUI elements, and interactive visualizations. All of these changes trigger the live programming algorithm (described in Section4.4) to recompute results as necessary, creating a responsive exploratory experience.

4 IMPLEMENTATION

Data science today encompasses a wide variety of workflows: traditional statistics and estimation, mining relational data, time series analysis of logs, signal processing on sensor data, training deep nets on unstructured data, and on and on. For Glinda to cover these use cases, as well as future ones, requires extensibility. Here we describe Glinda's novel architecture to provide live programming for a domain-specific language in an extensible way. We also cover a few additional details about the language, which provide context for the participants' feedback in our study.

4.1 Matching recipes

As mentioned earlier, a recipe is a Python function with a special docstring that contains a YAML template (Fig. 4-A). When the user's YAML (B) matches the YAML template, the user's YAML is compiled into a call to the recipe function (C). To understand the matching algorithm, a quick overview of YAML is helpful.

YAML has essentially three constructs: scalars, dictionaries and lists. Scalars are Booleans, numbers, and strings (which are typically not written inside quotes unless they contain certain characters). Dictionaries are written with a colon (`:`) between the key and the value; list items begin with a hyphen (`-`). Like Python, whitespace in YAML is significant and reduces the need for syntactic bracketing. Specifically, indentation is used for nesting, and newlines signify the end of constructs like list items and key-value pairs in dictionaries. (There is also a second, compact syntax for dictionaries and lists, which surround them with curly and square brackets, respectively.) Scalars, dictionaries, and lists can be composed to describe hierarchical data.

The matching algorithm is defined inductively over the YAML's structure. For scalars, the YAML template can either specify an exact value to be matched (e.g. the string `scatter_plot`) or can allow the user to choose a scalar value, subject to some constraints. These constraints are written as a dictionary with dollar-sign keys:

- `$type` for the type of scalar expected, for example `columnname` for the name of a column;
- `$min` and `$max` for the range of a numeric value;
- `$default` for the default value to be used if the key is omitted; and
- `$optional` as shorthand to say the `$default` is the Python value `None`.

For dictionaries, every key in the template must have a corresponding key in the step YAML, unless the the value specifies `$default` or `$optional`. For lists, there are two ways for a YAML template to constrain the list items. For the common case of lists of scalars, the template can use a `$type` specification with brackets to specify the item types. For example, the template `$type: [columnname]` matches a list whose items are all column names. The second way to constrain list items is by reference to other recipes, as described next.

To support compositionality, a template can specify its content by reference to another template. Every YAML template has a `$kind`, which is implicitly kind `step`, if omitted. A YAML template may specify that part of its content must be of a certain `$kind`. For example, Glinda has a set of recipes of kind `layer` for creating layers in a deep net. The recipe for training a deep net specifies that items in its `layers` list must match recipes of `$kind layer`. In short, these dollar-sign keys allow a template writer to set expectations about the YAML syntax, but without the need for a formal grammar or parsing expertise.

Anyone with the skills to write a Python function can contribute a new recipe to the collection. Glinda currently has recipes for reading and manipulating tabular data (using Pandas), visualizing data (using Altair [29]), transforming data by example (using PROSE Code Accelerator [7]), training predictive models (using Scikit Learn), and training deep nets (using Keras). In most cases, the recipe functions are thin wrappers for framework API calls. Some recipes supplement the framework APIs with conveniences or useful defaults. For example, the `data` recipe in the previous section uses the provided path's extension (`.csv`, `.json`, `.xlsx`) to choose an appropriate Pandas function for reading the data. As another example, the `model` recipes, by default, will choose encodings for columns that cannot be used directly in training, for example, one-hot encoding for categorical columns. For experienced data scientists, these default encodings make common cases less verbose. For those new to data science, these defaults scaffold the learning process to avoid initial pitfalls.

YAML's declarative, hierarchical syntax lends itself to a design principle of allowing the user to omit details to rely on defaults. For example, the simplest plot requires only a column name:

```
plot: Age
```

This plots the column's distribution as a histogram with default binning, colors, axis labels, etc. To override these defaults, the user can add details:

```
plot:
  type: histogram
  column: Age
  bin_size: 10
  sideways: true
```

A second design principle is that similar functionality should have similar YAML. Creating a scatter plot or line chart is similar to creating a histogram:

```
plot:                          plot:
  type: scatter_plot             type: line_chart
  x: Job                         x: Job
  y: Age                         y: Age
```

These steps use different recipes whose templates were designed to coordinate. Note that using the key `type` with a literal string (`histogram`, `scatter_plot`, `line_chart`) is simply a convention that exploits the recipe matching algorithm. We call any key with a literal value, like `type`, a discriminator, in that it helps the user discriminate among similar choices. Also, because code completion lists are flat in VS Code, we add discriminators to the completion labels to distinguish similar recipes (Fig. 3, top).

4.2 Compiling recipes

When a matching recipe is found for a step, Glinda compiles the step into a call to the recipe's function. Most of the arguments passed in the call come from the YAML key values. Their position in the call is determined by matching the names of the function's formal parameters. For example, in the function `scatterplot` (Fig. 4-A), the YAML template has keys `x`, `y`, `color`, `size`, and `tooltip`, and

A

```
def scatterplot(input, x, y, color, size, tooltip):
  """
  plot:
    type: scatter_plot
    x: { $type: columnname }
    y: { $type: columnname }
    color: { $type: columnname; $optional: true }
    size: { $type: columnname; $optional: true }
    tooltip: { $type: [columnname]; $optional: true }
  """
  args = {"x": x, "y": y, "tooltip": [x, y]}
  if color in input:
    args["color"] = color
    args["tooltip"].append(color)
  if size in input:
    args["size"] = size
    args["tooltip"].append(size)
  if tooltip:
    args["tooltip"] = tooltip
  chart = alt.Chart(input).mark_circle().encode(**args)
  respond_chart(chart)
```

B

```
spotify:
- data:
    type: file
    path: /data/spotify.csv
- sample:
    strategy: random
    size: 1000
- plot:
    type: scatter_plot
    x: energy
    y: tempo
    color: playlist_genre
```

C

```
spotify1 = pandas_read_file('/data/spotify.csv')
spotify2 = pandas_sample(spotify1, 'random', 1000)
scatterplot(spotify2, 'energy', 'tempo',
            'playlist_genre', None, None)
spotify = spotify2
```

Figure 4: The Glinda recipe for scatter plots (A), a Python function whose doc string contains a YAML template. When the recipe is used (B), the values supplied for keys `x`, `y`, `color`,`size`, and `tooltip` are passed as the corresponding arguments in a call to `scatterplot` (C). When a recipe function has a parameter named `input`, then the passed argument is the output of the previous step (`spotify1`).

the function has parameters with those same names. If the user's YAML is missing a key marked `$default: V`, then the compiler passes value V for that parameter; if it is marked `$optional: true`, then the compiler passes `None`.

To combine steps into an overall computation, the Glinda compiler threads the calls together using assignments to intermediate variables (Fig. 4-C). The result of the first step is assigned to a variable (`spotify1`). This variable is then passed to the second step's function whose result is assigned to a second variable (`spotify2`). Some recipes, like those for plots, do not produce results, so their function calls are not bound to variables, for example, the call to `scatterplot` in Fig. 4-C. The result of the final step that produces an output is bound to the workflow's name. Hence the final assignment in Fig. 4-C assigns `spotify2` to `spotify`.

This threading of calls is determined by two special parameters to recipe functions, `input` and `parent`. If a recipe has a parameter called `input`, then the output of the previous step is passed for this parameter. For example, since the `scatterplot` recipe takes an `input` parameter, the call to `scatterplot` is passed `spotify2`, the output from the previous step (Fig. 4-C). As mentioned earlier, recipes can be hierarchically composed from other recipes by using `$kind`. For example, the `deepnet` recipe takes a list of layers whose recipes have `$kind: layer`. The special parameter `parent` gives a recipe access to the containing recipe. For example, a convolutional layer recipe needs access to the input shape from the containing deep net. Finally, Glinda provides a recipe called `use` to insert the result of one workflow into the computation of another. With this ability, Glinda computations can be arbitrary directed acyclic graphs. Glinda also provides compile-time loops to avoid repetition in the YAML structure (for example, repetitive deep net layers) and limited run-time loops for experiments like hyperparameter sweeps.

4.3 Expressions

Steps like filtering data and defining new columns require the use of predicates and arithmetic. While it is feasible to encode these expressions in YAML, the result would be tedious and read like a syntax tree. Instead, Glinda provides an expression language based on Python's syntax, for example:

```
- filter:
    keep: Age > 0
- add_column:
    NormTicket: TicketCost / max(TicketCost)
```

Here, the first step filters the dataset to those rows whose `Age` column is positive; the second defines a new column called `NormTicket` whose value for a given row is that row's `TicketCost` divided by the maximum value of the `TicketCost` column. Treating column names as though they were scalar values is similar to Pandas in Python, tidyverse in R, and SQL.

4.4 Live programming

Glinda uses DeLine and Fisher's live programming algorithm [5], adapted for Python rather than C#. Briefly, every time the user pauses more than 500 ms, Glinda compiles the user's YAML to Python, as described in the previous section. If there are no compilation errors (for example, syntactic errors in the YAML), the compiler gives the Python code to the live programming algorithm. This algorithm compares the latest Python code to the previous version. All of the statements that have changed form an initial set

of statements to run. The algorithm then computes a dependency graph for the new code. Any statement that depends on statements in the set is then added to the set. This process of adding dependencies is repeated until a fixed point is reached. The statements in the final set are then run using a Python interpreter. (As an aside, computing a dependency graph for a Python program presents many technical difficulties not present in C# programs, mostly because Python is a dynamically typed language. The live programming algorithm is conservative in the face of these challenges: it assumes a dependency exists unless the program analysis is certain it does not. In practice, the algorithm typically runs the minimal amount of code.)

Glinda uses the Jupyter Server as its Python interpreter for several reasons. First, the Jupyter Server implements a socket-based protocol for executing code, which means that the interpreter can run locally or in the cloud. Second, the Jupyter Server's protocol provides a side-channel for reporting visualization results as the Python code runs. This is what Jupyter Notebook and Jupyter Lab use to display their visualizations, and Glinda does the same. Finally, Jupyter Server supports many languages, not just Python. Currently Glinda files compile only to Python. However, using the Jupyter Server opens the possibility of compiling to other languages or even multiple languages within a file. A major downside to using Jupyter Server is that interrupting the Python kernel is slow and unreliable. Ideally, as the user updates the YAML, any Python code currently executing would be stopped in favor of running the most recent code. Unfortunately, due to Jupyter Server's limitations, Glinda's live programming runs Python code to completion.

5 QUALITATIVE EVALUATION

Ultimately, we would like to evaluate Glinda with multiple types of roles on software development teams: data scientists; developers; operations engineers; and non-programming roles like program managers. For our initial study, we begin with data scientists. After all, if data scientists feel that Glinda is inadequate or inappropriate for doing data science work, there is little point in trying to convince the other roles. Evaluating first with data scientists also allows us to incorporate their expert feedback before testing with other roles.

5.1 Participants

We recruited participants through an invitation email sent to a random sample of 200 data scientists at a large, USA-based software company. The random sample was selected based on job title, position in the organization (Human Resources and Legal were excluded), and location (for convenience of running the study). The invitation yielded 23 volunteers (11.5% response rate), of which 12 participated in the study. The participants (10 male, 2 female) report a mean of 8.9 years of professional experience and 7.2 years of working as a data scientist. They report spending between 10–50 (mean 28.2) hours per work week on data science activities.

5.2 Protocol

We conducted each session remotely over a video chat system, with the ability to share the desktop and record the session. Each session lasted 60 minutes, for which the participants were compensated $25 USD. After signing a consent form, each participant connected to a virtual machine in the cloud, which we set up in advance with the necessary software and data. The rest of the session had four parts: a data science task with a computational notebook (10 min); an interactive Glinda tutorial (15 min); a data science task with Glinda (20 min); and a final questionnaire (5 min). The purpose of the notebook task was to warm them up for the creative task of analyzing data and to allow the participant to contrast the experiences of using a notebook versus Glinda. For the computational notebook, we used VS Code Notebooks, which provides a user experience nearly identical to Jupyter Notebooks, but within the VS Code editor. We chose VS Code Notebooks to minimize the differences between the notebook and Glinda experiences. That is, we did not want to give Glinda an advantage when participants compared the two experiences, simply due to its integration into a development environment.

For both the notebook and Glinda tasks, the participant chose from six Tidy Tuesday[7] datasets, used to promote best practices in the R community. The six datasets in the study were on the following topics (with sizes given as rows × columns): Spotify song genres ($32,833 \times 24$); hotel bookings ($119,390 \times 33$); cocktail ingredients (2104×14); coffee bean ratings ($1,339 \times 44$); National Football League games, teams, and attendance ($10,846 \times 9$, 5324×20, 638×16); and cracked passwords (507×10). Each dataset had a local file path and a link to a data dictionary on the Tidy Tuesday site. For both the notebook and Glinda tasks, we invited the participant to choose their own goal, like data cleaning, exploratory data analysis, creating visualizations, or building models. In addition to the Glinda tutorial described in Section 3, we also provided a "cheat sheet" poster to describe all the available Glinda recipes, including those not covered in the tutorial. Throughout all the tasks, the experimenter was available to answer questions.

6 RESULTS AND DISCUSSION

Overall, the data scientists' response to Glinda was quite positive, as reflected in both their remarks and responses to the usability questionnaire (Fig. 5). The majority asked for an installer so they could use the tool again after the study, and several requested follow-up meeting with their teams to discuss accommodating their data sources and practices.

The questionnaire ended with four open-response questions about their experiences:

- What aspects of using Glinda do you like?
- What aspects of using Glinda do you dislike?
- Are there aspects of Glinda that are better than using notebooks like Jupyter Notebook, Colab, Databricks, etc?
- Are there aspects of notebooks that are better than using Glinda?

Their responses to these items, plus a transcription of their remarks during the tasks, are the sources of the quotes below.

6.1 Liveness

The participants responded the most positively to live programming and its immediate, interactive visualizations. The initial step of the tutorial, which pops up a table display, often elicited a spontaneous

[7] https://github.com/rfordatascience/tidytuesday

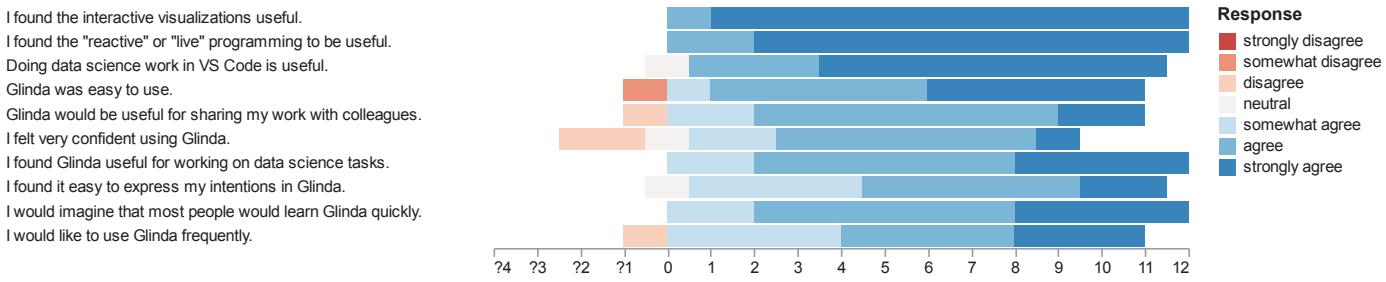

Figure 5: The twelve participants' responses to a usability questionnaire were overall quite positive. They particularly appreciated the responsiveness of interactive visualization and live programming.

"oh", "wow", or "cool". Many commented that live programming's visualizations automated a task they do explicitly in notebooks. One participant wrote, "The idea of workflow and execution under the hood were very fascinating. I tend to call display at intermediate steps to see if the data looks ok or the plot looks ok. The fact that the workflow did that for me saved me a lot of time." Another said, "I liked the interactive visualizations quite a lot. I found it easier to use than the notebook equivalents where I'd have to arduously define what I want visualized instead of just giving the name of a column." Consistently, a similar moment of delight was adjusting the slider in a `sample` step and seeing the downstream computations update: "Oh my god, this is wild. It's like a complete GUI experience."

That said, several participants were concerned about how live programming would cope with large datasets or slow operations like training a deep net: "I guess that is one of my worries with this kind of always-updating thing is when your datasets start to get large, when you have a lot of columns or a lot of features...I don't know what a good solution to that is, because I really like it. Maybe you could always be sampling by default." (In fact, researcher have previously used continuous sampling for incremental visualization [8].) Several participants suggested that Glinda should provide a way to pause computation temporarily. One participant wrote "reactive is too aggressive", and a few expressed a preference for the explicit execution control in notebooks.

6.2 High-level, Declarative Language

The participants also felt that expressing their intentions in a high-level declarative notation made them more productive: "I would have taken maybe an hour to write all this by myself in Pandas. It took like three seconds here....I think this tool has a lot of potential. I mean the time saving is insane." Another participant similarly wrote, "it lets me skip a lot of work." Indeed, during the warm-up notebook task, half the participants opened a web browser to search for details about the APIs they were using. Their feedback also made it clear that high-quality code completions are a crucial aspect of this productivity. This is partly due to the feature's familiarity: "I really like the code completions. Everything just blended in with what I expect from VS Code." More importantly it scaffolded their use of an unfamiliar language. Many named code completion as a favorite feature, for example, "Loved the way how the code completion worked and the way in which the code completion was also able to suggest the features of my dataset after initializing it."

In addition to productivity, some participants also felt that the high-level notation improves reusability: "Declarative syntax. YAML is simple. Provides a framework that encourages packaging code for reuse." In particular, the high level of abstraction, particularly the implicit chaining of steps, reduces the bureaucratic details that can interfere with code reuse. Several participants were also excited by the possibility of writing their own recipes to capture common team practices for reuse: "Code also feels way more reusable when defining recipes than in isolated Notebooks"

A few participants were hesitant to give up low-level control over their work, for example, "recipes are abstract, black-boxed making it harder to debug". Another wrote: "I like having access to the 'lower' level objects. I can imagine Glinda making hard things harder while making moderate to easy things easier." One area of for further work is the recipe authoring experience, for example the ability to "go to definition" from a recipe instance in YAML to the corresponding recipe function in Python. Similarly, a user should be able to write the YAML for a non-existent recipe and then create from it a new recipe function in Python. Better support for recipe authoring may help open the black box that worried some participants.

6.3 Language Design

Overall, participants liked having a high-level, declarative notation and found it productive. However, a few spotted design flaws in the current Glinda language. For example, one participant noticed than Python-like expressions could appear when defining a new column, but not when choosing the size of a data sample, which requires an integer constant: "One thing that's confusing to me is when I'm allowed to use Python expressions in place. I clearly wasn't allowed [in the sample size]. I wonder if it's a typing problem or an expression-not-allowed problem." Another asked: "Can I have a list of columns as a variable? If I want to something more sophisticated...I want to do the same training on a list of different feature sets." They tried to define a top-level name for a list of columns and to use the variable name for the model's `uses` key. This does not work because Glinda interprets the variable name as a column name.

The heart of the issue is that Glinda currently has two competing notions of compilation: one based on lexical scoping (workflow names, `use` steps, and expressions); and one based on template

matching. More work is needed to reconcile these two. One potential solution would be a phased compilation: first resolving lexical names, then doing the recipe matching.

Most of the participants were aware of YAML, but had not worked with it before. While all of the participants were able to write YAML productively from the start, there were a few common pitfalls. The most frequent problem is that a space character is compulsory after both the hyphen for list items and the colon for dictionary keys. For example, `- data` is parsed a list item, while `-data` is parsed as a five-character string. A more serious issue is due to YAML's simplicity: small syntactic changes create unintended differences in structure. For example, several participants accidentally omitted the initial hyphen when forming a list of steps, for example:

```
titanic:
- data
  type: file
  path: /data/titanic.csv
plot: Age  # missing initial hyphen
```

Here, the participant intended `plot` to be a step in the workflow `titanic`. Instead, the text above defines a top-level constant called `plot` with the string value `Age`. Similarly, had `plot` been indented by two spaces (without a hyphen), it would be key of the `data` step, rather than its own step. In short, because the syntax is so simple, mistaken programs can still be legal programs. Addressing this issue requires a trade-off between expressiveness and checking. For example, forbidding top-level constants from having the same names as recipes would allow the program above to be reported as erroneous.

6.4 Supporting Data Science

Many participants noticed feature gaps in the current recipes. For example, several participants wanted greater control over the appearance of visualizations, like axis bounds, axis labels, and titles. They wanted the ability to edit these in place through GUI controls, but to have their choices recorded in the YAML. Similarly, some participants wanted explicit control over the hyperparameters of model training, be able to override implicit choices about data encoding, and the ability to choose different metrics besides accuracy. Participants consistently complained about explicitly listing the columns to use as features during training. They wanted abbreviations for lists of columns, for example a star notation to mean all columns except the prediction target. All of these suggestions would require only small changes to existing recipes.

Many participants wanted the ability to export the Python code that the Glinda compiler produces. Glinda has an item on the context menu called "Show Compiled Markup", which we showed participants when they brought up this topic. The menu item opens a read-only Python file containing the Glinda compiler output, which updates automatically with live programming. In a similar vein, several participants asked for a feature to export the visualization pane (or the pane and YAML together) as HTML. Their goal was to share a report with other roles on the team, particularly program managers and executives. This communication use case was why they wanted careful control over appearance of visualizations.

7 CONCLUSION

The professional data scientists in our study were surprisingly enthusiastic about Glinda, given that they have entrenched work practices with notebooks, Python, and its libraries. The combination of a declarative language, live programming, and interactive visualizations created an iterative exploratory experience that several participants described as more productive than writing Python in notebooks. In short, these data scientists found Glinda not only a plausible way, but a desirable way to do data science work.

Data science has become a routine part of software development, both in the design of the user experience and in support of the development process [1, 20]. Many roles now involve data: developers incorporate ML models alongside their hand-written logic; operations engineers use models to predict failures and resource usage; and program managers use data to understand customer behavior. Glinda's use of a DSL and GUIs might make data science work more approachable to these other roles. After incorporating the data scientists' feedback, the next step of the research agenda is to evaluate Glinda with two additional user groups: developers and operations engineers (who have programming skills) and program managers (who often do not). Given that there are already so many tools and notations for data science, one could reasonably ask whether there is room for another.[8] We believe that the goal of democratizing data science across all roles on software teams could be a sufficient "killer app" to justify room for a new notation.

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, New York, NY, USA, 291–300.

[2] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: declarative machine learning on spark. *very large data bases* 9, 13 (2016), 1425–1436.

[3] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's alive! continuous feedback in UI programming. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 95–104.

[4] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–12.

[5] Robert DeLine and Danyel Fisher. 2015. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 111–119.

[6] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Michael Barnett, James F Terwilliger, and John Wernsing. 2015. Tempe: Live scripting for live data.. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Vol. 15. IEEE, New York, NY, USA, 137–141.

[7] 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*. ACM, New York, NY, USA, 1–12.

[8] Danyel Fisher, Igor Popov, Steven Drucker, and MC Schraefel. 2012. Trust me, I'm partially right: incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1673–1682.

[9] Martin Fowler. 2010. *Domain-specific languages*. Pearson Education, London, UK.

[8] https://xkcd.com/927

[10] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st annual ACM symposium on User Interface Software and Technology* (Monterey, CA, USA) *(UIST '08)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1449715.1449732

[11] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Rob DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, NY, USA, 1–12. https://microsoft.github.io/gather/

[12] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct Manipulation Interfaces. *Human–Computer Interaction* 1, 4 (1985), 311–338.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[14] Eunice Jun, Maureen Daum, Jared Roesch, Sarah Chasins, Emery Berger, Rene Just, and Katharina Reinecke. 2019. Tea: A High-Level Language and Runtime System for Automating Statistical Analysis. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 591–603. https://doi.org/10.1145/3332165.3347940

[15] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 737–745.

[16] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[17] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 25–29.

[18] Mary Beth Kery and Brad A Myers. 2018. Interactions for untangling messy history in a computational notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 147–155.

[19] 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 ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 140–151.

[20] Miryung Kim, Tom Zimmermann, Rob DeLine, and Andrew Begel. 2017. Data Scientists in Software Teams: State of the Art and Challenges. *IEEE Transactions on Software Engineering* 44, 11 (September 2017), 1024–1038.

[21] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[22] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, New York, NY, USA, 1–11.

[23] Nantia Makrynioti and Vasilis Vassalos. 2019. Declarative data analytics: a survey. *IEEE Transactions on Knowledge and Data Engineering* Early, Access (2019), 1–1.

[24] Bernadette M Randles, Irene V Pasquetto, Milena S Golshan, and Christine L Borgman. 2017. Using the Jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, New York, NY, USA, 1–2.

[25] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* 23, 1 (2017), 341–350. http://idl.cs.washington.edu/papers/vega-lite

[26] Maximilian Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, Konstanz, Germany, 562–565. https://doi.org/10.5441/002/edbt.2019.56

[27] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 52–65.

[28] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, New York, NY, USA, 31–34.

[29] Jacob VanderPlas, Brian E Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive statistical visualizations for python. *Journal of open source software* 3, 32 (2018), 1057.

[30] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* 22, 1 (2016), 649–658. http://idl.cs.washington.edu/papers/voyager

[31] Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, New York, NY, USA, 152–165.