Unravel: A Fluent Code Explorer for Data Wrangling

Nischal Shrestha
nshrest@ncsu.edu
NC State University
Raleigh, North Carolina, USA

Titus Barik
titus.barik@microsoft.com
Microsoft
Redmond, Washington, USA

Chris Parnin
cjparnin@ncsu.edu
NC State University
Raleigh, North Carolina, USA

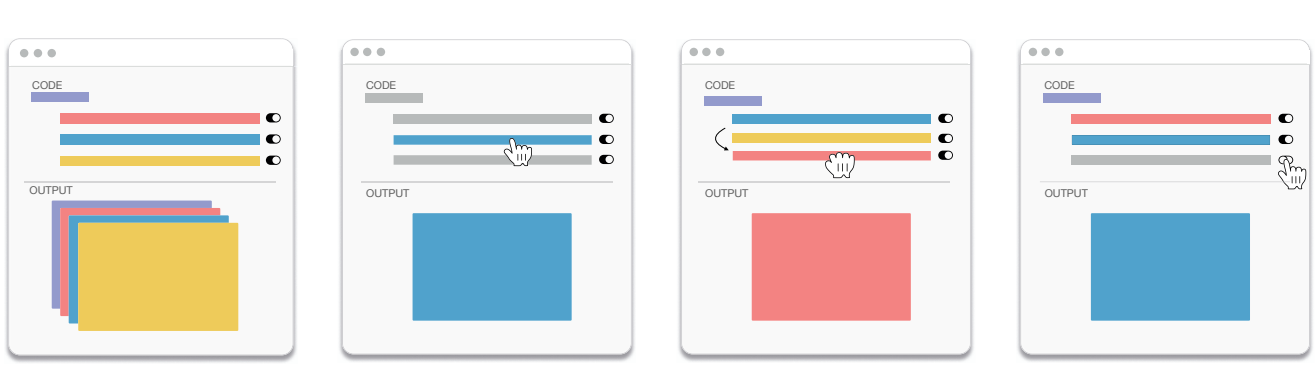

Figure 1: Unravel is a tool that helps data scientists understand and explore fluent code via structured edits using drag-and-drop and toggle switch interactions. The data scientist unravels fluent code to get access to intermediate outputs for each line. They can then inspect a particular line of code and its respective output. Data scientists can explore the code using drag-and-drop to reorder lines, and toggle switches to enable or disable lines and automatically produce new outputs to investigate.

ABSTRACT

Data scientists have adopted a popular design pattern in programming called the fluent interface for composing data wrangling code. The fluent interface works by combining multiple transformations on a data table—or dataframes—with a single chain of expressions, which produces an output. Although fluent code promotes legibility, the intermediate dataframes are lost, forcing data scientists to *unravel* the chain through tedious code edits and re-execution. Existing tools for data scientists do not allow easy exploration or support understanding of fluent code. To address this gap, we designed a tool called Unravel that enables structural edits via drag-and-drop and toggle switch interactions to help data scientists explore and understand fluent code. Data scientists can apply simple structural edits via drag-and-drop and toggle switch interactions to reorder and (un)comment lines. To help data scientists understand fluent code, Unravel provides function summaries and always-on visualizations highlighting important changes to a dataframe. We discuss the design motivations behind Unravel and how it helps understand and explore fluent code. In a first-use study with 14 data scientists, we found that Unravel facilitated diverse activities such as validating assumptions about the code or data, exploring alternatives, and revealing function behavior.

UIST '21, October 10–14, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8635-7/21/10…$15.00
https://doi.org/10.1145/3472749.3474744

CCS CONCEPTS

• **Human-centered computing** → **Graphical user interfaces**.

KEYWORDS

data science, data wrangling, programming

ACM Reference Format:
Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472749.3474744

1 INTRODUCTION

Data scientists apply a common programming design pattern—the fluent interface [13, 14]—when they transform and wrangle data tables, or *dataframes*. The distinguishing feature of the fluent interface is that it composes multiple operations into a chain, with each operator in the chain accepting data from the result of the previous operator, performing a computation on it, and passing its result on to the next operator. Advocates for the fluent interface suggest this style of programming improves readability by removing the need to assign intermediate results to variables. To illustrate how fluent interfaces are applied in practice, consider fluent code written in R (Figure 2).

```
penguins %>%
  select(species, flipper_length_mm) %>%
  group_by(species) %>%
  summarise(mfl = mean(flipper_length_mm))
#> Output:
#>   species    mfl
#> 1 Adelie      NA
#> 2 Chinstrap  196.
#> 3 Gentoo      NA
```

(a) Summarizing mean flipper lengths of penguin species.

```
penguins %>%
  select(species, flipper_length_mm) %>%
  # group_by(species) %>%
  # summarise(mfl = mean(flipper_length_mm))
#> Output:
#>   species flipper_length_mm
#>   ...                   ...
#>   ...                   ...
#> 4 Adelie                 NA
```

(b) Inspecting the line up to `select` with a dangling `%>%`.

Figure 2: An example of exploring fluent code in R, which outputs a dataframe of mean flipper lengths of different penguin species.

In the "tidyverse" [2] dialect of R, the pipe (`%>%`) operator forms the links in the chain of expressions by piping [1] results of function calls together, shown in Figure 2. In Figure 2a, the `penguins` variable containing the dataframe from the Palmer Penguins dataset [16] gets piped (`%>%`) or passed to a `select` function to select columns, `species` and `flipper_length_mm`. The result is piped to the function `group_by` function to group the data by the `species` column which is finally piped to the `summarise` function to calculate the mean of the flipper length (`flipper_length_mm`) according to each species and store it in a new column, `mfl`. Note how the final code is built up by solving smaller subproblems (selecting, grouping, and summarising), forming a single large chain. But, what if there is a problem with the final output?

Although fluent code is designed to be readable and concise, these advantages come with a cost: isolating problems within a broken chain becomes a clerical and cumbersome process. Because fluent code removes intermediate variables, one of its significant disadvantages arises when the data scientist needs to inspect an intermediate result. In Figure 2a, a data scientist might be surprised to find the mean of flipper lengths of the Adelie and Gentoo species are `NA`s or "not available." To hunt for clues, the data scientist is forced to "unravel" the chain and find the "broken" link (Figure 2b), where they have to modify and re-execute the code to discover `flipper_length_mm` contains missing values, an easy to miss detail. To verify the source of `NA`s, the data scientist had to comment lines, remove a dangling pipe operator, and execute the code up to the `select` function. They can then fix this issue by excluding rows with `NA`s for `flipper_length_mm` could be removed by inserting a function called `drop_na` before the `select` line.

We identified several limitations behind existing solutions to help data scientists explore fluent code. Prior research has focused on helping data scientists become more productive by managing messy code [15], keeping track of versions [18], exploring alternatives [31] or generating code [10] using programming-by-example techniques. However, these tools are designed to help manage entire scripts or notebooks, and do not provide affordances to easily understand and explore code at a finer-grain level. In the R community, data scientists have voiced a need to support introspection and debugging tools for fluent code, with several attempts at solutions.[1] Existing inspection and debugging tools attempt to solve parts of the problem but fall short in several ways. Current solutions require a data scientist to meticulously debug, log, and selectively execute of code. For example, a debugger [3] is a heavyweight solution for exploring fluent code and it forces the data scientist to linearly step through their code. Printing intermediate results to the console—for example, with `tidylog` [11]—is lightweight but generates noisy output. Other solutions require newer types of operators to debug fluent code which might introduce more issues.[2] This suggests a need for easily exploring and understanding fluent code in data science.

To address this need, we introduce Unravel, a tool that enables structured edits using drag-and-drop and toggle switch interactions with always-on visualizations to help data scientists explore and understand fluent code. Unravel is a web application that runs within the RStudio IDE. Using an interactive code overlay, data scientists can *unravel* a chain of fluent code in R to examine intermediate dataframes, understand the transformations of dataframes along the chain, and apply simple structural edits without typing. Data scientists can apply structural edits to fluent code by reordering lines using drag-and-drop, or enabling or disabling lines using toggle switches. To help data scientists understand each step in fluent code, function summaries describe the transformations on dataframes and always-on visualizations are used to highlight important changes to dataframes. We designed Unravel to help data scientists get clarity on data transformations, and reduce the burden of typing to manipulate fluent code.

The contributions of this paper are:

(1) A tool called Unravel that enables structured edits via drag-and-drop and toggle switch interactions with always-on visualizations to help data scientists explore and understand fluent code. We discuss the design motivations behind Unravel and how data scientists can use it to support a variety of tasks.
(2) Through a first-use study with 14 data scientists, we demonstrate that Unravel complements an IDE workflow, offers an interactive way to explore fluent code, and supports a variety of tasks related to understanding and writing data wrangling code.

2 DEMO OF UNRAVEL

Asha is a data scientist who is familiarizing herself with tidyverse R, a dialect of R that uses fluent code for data wrangling. Asha is analysing the `babynames` dataset [33], which holds popular U.S. babynames from 1880-2017, and she wants to calculate the percent of male babynames and the ratio of males to females. She is unfamiliar with how grouping works and wants to check her assumptions

[1] https://community.rstudio.com/t/whats-currently-the-recommended-way-to-debug-pipe-chains/14724
[2] https://win-vector.com/2017/01/29/using-the-bizarro-pipe-to-debug-magrittr-pipelines-in-r/

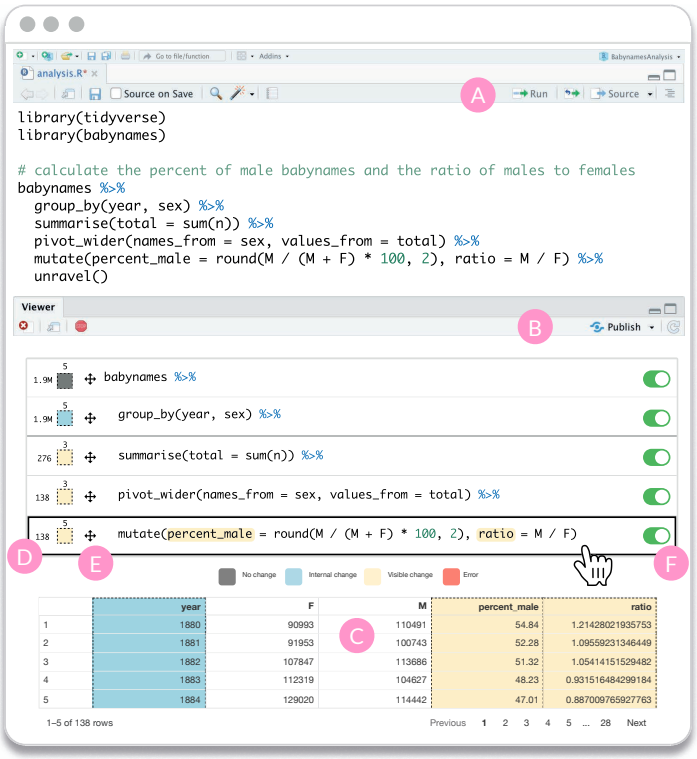

Figure 3: The workflow and interface of Unravel. The user writes their code in the editor (A) to `unravel()` the fluent code and invoke the web application with the code overlay in the Viewer Pane (B). Then, the user can click on a line to inspect its output (C), view information about the dataframe dimensions (D) and the type of change occurred. To explore the code, they can reorder (E) or enable / disable a line using the toggle switches (F).

regarding the behavior of operations on grouped dataframes. To explore her code snippet, Asha pipes (%>%) her code to `unravel()` which opens up Unravel in RStudio's Viewer Pane.[3]

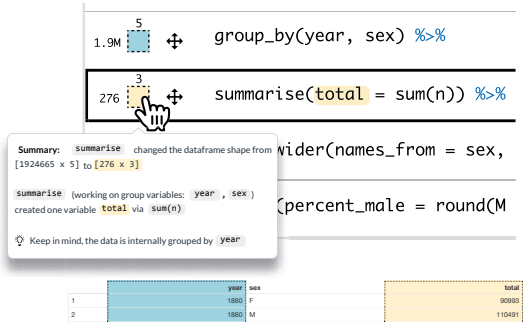

Figure 4: Visual highlights on the code, function summary, and output are applied when focusing on a line.

Asha begins her investigation on checking why the final dataframe was grouped in the final output (Figure 3 C). Unravel focuses on the final line with the `mutate` function automatically, and

[3]https://rstudio.github.io/rstudio-extensions/rstudio_viewer.html

Asha notices that the `percent_male` and `ratio` on the code and the dataframe output are marked as visible changes, but the `year` column is marked as an internal change. She had assumed `summarise` would have removed all group variables (columns) after calculating the sum of `year` and `sex` columns, but a group variable was kept. Puzzled, Asha investigates the summary of the `summarise` line to figure out how it handles group variables (Figure 4).

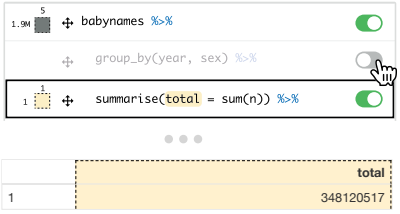

Figure 5: A line can be enabled or disabled using a toggle switch which automatically re-evaluates the remaining lines.

The new `total` column is marked as a visible change, while the `year` has stayed an internal change. To her surprise, Asha learns from the summary that `summarise` will drop the last grouping variable (`sex`), but keeps the rest of the group variables (`year`). This explains why she only saw the `year` column marked as internal change in the final output. Before moving on, Asha wants to confirm that `summarise` works on a grouped dataframe, so she temporarily disables the `group_by` line (Figure 5). The `summarise` line is automatically focused. Asha glances at the dimensions the dataframe output which is only one row and column. She confirms that without `group_by`, `summarise` will work on column `n` of the entire dataframe.

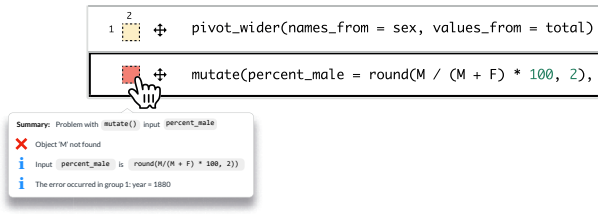

Figure 6: Clicking the summary box of the line with an error displays the error message.

Asha now wants to sample the first `year` and `sex` groups. She adds a line in the editor to select the the first group using a `slice` function, placing it before the `group_by` line. Upon running Unravel on her new code, she comes across an error (Figure 6). Reading the summary of the error on `mutate` line Asha learns that a column for male is missing. Examining the dataframe dimensions on the `slice` and subsequent functions, she realizes she had only selected the first row of the original dataframe. Asha fixes the issue by reordering the `slice` line below the `group_by`, which automatically updates the output to return the first row of each `year` and `sex` group (Figure 7). She also learns that `slice` will keep the groups `year` and `sex`.

3 RELATED WORK

Unravel builds on prior tools that help data scientists write and understand code. Our work is closely related to the research on interactive tools that enable exploratory programming.

Writing code for data science. In computational notebooks, researchers have developed tools that help data scientists write and modify code. For example, Gather [15] helps analysts find, clean, recover, and compare versions of code in cluttered, inconsistent notebooks. While Gather was designed for the notebook as a whole, our tool helps data scientists manage messes that arise within fine-grained, fluent code chains. To explore alternative code in notebooks, Fork It [31] introduces a technique to fork a notebook and directly navigate through decision points in a single notebook. We designed a more lightweight approach to explore code by allowing exploration through interactive overlays and structural edits on the code itself, for example, by enabling, disabling or reordering lines. To help data scientists generate data wrangling code, Wrex [10] uses programming-by-example. Similarly, mage [19] is a tool that helps users generate code based on the modifications made from interacting with dataframes. Unravel complements tools like Wrex

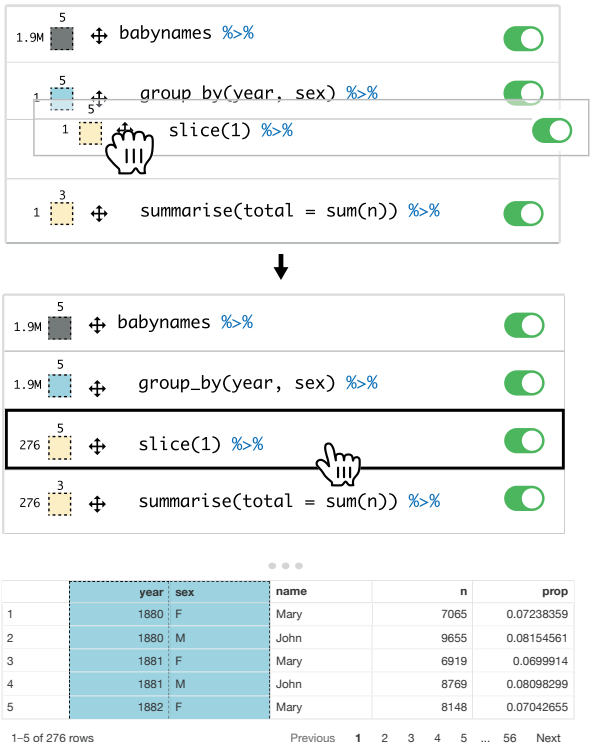

Figure 7: Drag-and-drop can be used to reorder a line, which automatically re-evaluates code to generate new outputs.

and mage by allowing data scientists to understand and iterate on the machine-synthesized code.

Understanding code for data science. Prior work has explored tools to help data scientists understand code. For example, Wrangler [17] is an interactive tool designed to ease the process of writing data transformation scripts. Wrangler takes a table-centric approach where data scientists manipulate the table to produce scripts; our approach assumes that data scientists are already working with code and supports their understanding and exploration through it. Lau et al. [21]'s TweakIt is a system designed to help end-user programmers collect, understand, and tweak Python code within a spreadsheet environment. Unravel shares similar design goals such as previewing outputs on different parts of the code, but it surfaces these capabilities through interactive visual overlays on the fluent code chains.

Closely related to our current work is Pu et al. [28]'s Datamations tool for animating dataframe wrangling and visualization pipelines in R. Datamations automatically animate fluent code in R using `tidyverse` [2] packages and provides a paired explanation and visualization of each step in the chain. Our work differs in multiple respects. First, Datamations provides visualizations of operations within the chain with summaries on the intention behind each step. Unravel is designed as an interactive tool that allows direct access to the intermediates throughout the chain for further inspection and exploration. While Datamations provide basic visual cues for tabular animations—such as highlighting a column for different grouping variables—we provide visual cues for additional

information such as the intermediate dataframe dimensions, and the type of change occurred at each step. Lastly, Unravel allows for exploration within the code, dynamically creating explorable code upon structural edits, whereas Datamations is focused on providing animated explanations.

Interactive affordances for code exploration. Researchers have investigated many useful interactive affordances for code explorations that we adapt to the data science context. Although these affordances were designed for different contexts, they are useful for addressing some of the pain points regarding fluent code.

Always-on visualizations can help data scientists understand and inspect code and data. Lieber et al. [24] built an IDE tool called Theseus which provides an always-on visualization to display the number of API calls made within the editor for JavaScript code. We adapt this visualization technique for fluent code to display dataframe properties like its row and column dimensions. Similarly, "projection boxes" [12, 23] are an always-on visualization technique for displaying runtime values of Python programs such as the contents of arrays. This can help minimize context-switches when writing data wrangling code since it requires constant verification between code and output. Instead of multiple projection boxes hold multiple dataframes, we display one dataframe at a time.

Interactive debuggers and steppers are another useful technique for exploring data wrangling code. For example, Whyline [20] introduced an *interrogative debugging* interface for asking "why" or "why not" questions about a program. Whyline visualizes answers in terms of runtime events connected to the questions. Although our tool does not directly support asking explicit questions, it can aid this type of investigation by facilitating inspection of each intermediate dataframe in fluent code for data wrangling. Timelapse [6] is a tool that helps web developers browse, visualize, and explore recorded program executions using debugging tools such as breakpoints and logging. We record program executions on fluent code to support investigation of all intermediate dataframes produced in the chain.

4 SYSTEM DESIGN AND IMPLEMENTATION

Unravel is a tool that is run within the RStudio IDE to support data scientists introspect and explore fluent code using R. We picked R because it is widely used in data science, and is a popular language.[4] The R language also provides metaprogramming capabilities which make it convenient for some stages of the implementation such as parsing and evaluation of intermediate expressions. It is built using the R Shiny Framework [7] and HTML/CSS/JavaScript.

4.1 Design Motivations

Fluent expressions are used by many programming languages in data science. For example, LINQ (Language-Integrated Query) is a fluent interface in C#, typically used as a convenient wrapper—known as an object-relational mapper (ORM)—around database query languages like SQL. Languages like Python use the fluent interface for data analysis code through the `pandas` library.[5] In the R community, the fluent interface is used in a collection of R packages called the "tidyverse" [2] to facilitate importing data, wrangling data, computing statistics, manipulating strings, and modeling data. In the RStudio Community Forums, a popular Q&A site for R, the tidyverse is the 3rd largest category suggesting users experience pain points with these packages on a daily basis. Among the various contexts where fluent code is used, we examined data wrangling as an important activity to support because it is one of the most time consuming and difficult aspects of analysis [8, 25].

We examined the R community to identify pain points expressed by data scientists when understanding and exploring fluent code. Data scientists have expressed the need for transparency about the data that they are transforming.[6] One data scientist expressed how "we aren't good at tracking state,"[7] and it's easy to miss whether or not a dataframe is grouped, where "working on a grouped [data] that you forgot is grouped can lead to 'unexpected' results."[8] To inspect issues in fluent code, a traditional debugger can be too heavyweight for exploring smaller code snippets. The data scientist also has to linearly progress through their code and cannot openly explore code at any step. The R community has explored special pipe operators to debug fluent code, but these can introduce more typing mistakes and confusion for data scientists by adding more syntax to remember.[9] `tidylog` [11] is a lightweight solution which prints the summaries of functions to the console output, but this can generate noise and it does not save intermediate dataframes for further inspection. During explorations of the code, data scientists have to constantly switch between the source editor and the console output to validate the effect of code on dataframes. This forces context switches. Altogether, we identified a need for an in-situ tool within an IDE—such as RStudio—that provides clarity on transformations, and reduces the burden of typing to manipulate fluent code. To address these needs, we arrived at the following design goals:

> **D1. Provide transparency about the dataframe in fluent code.** The code and the respective dataframe intermediate outputs should be accessible at all times. Users must be able to click the relevant part of the chain to view its intermediate dataframe and glean basic information like row and column dimensions, the types of changes occurred and a summary about the transformation.
>
> **D2. Allow just-in-time explorations of fluent code.** To help data scientists easily perform inspections on fluent code, they must be able to perform simple structural edits to the fluent code. Structural edits must instantly update the UI to easily explore the new intermediate dataframes.
>
> **D3. Minimize context-switching to unravel fluent code.** To minimize context-switching, the tool should be integrated into data scientists' workflow within the IDE. Users must be able to input their own code to explore the chain.

4.2 Implementation

We present the implementation of Unravel by describing the entire process from invoking the tool to exploring a code snippet in the

[4] https://www.tiobe.com/tiobe-index/r/
[5] https://pandas.pydata.org
[6] https://community.rstudio.com/t/whats-currently-the-recommended-way-to-debug-pipe-chains/14724
[7] https://twitter.com/mjskay/status/1367244873607249922
[8] https://twitter.com/aosmith16/status/1369689345335070732
[9] https://win-vector.com/2017/01/29/using-the-bizarro-pipe-to-debug-magrittr-pipelines-in-r/

web application. We discuss our design decisions for all of the features to support our design goals in Section 4.1.

4.2.1 Code Parsing and Trace Executions. Unravel initially parses the user's code (Figure 3 A) and splits the fluent code into multiple code snippets that represent each part of the chain. Unravel parses the code passed to the `unravel()` function. We make sure to check the abstract syntax tree (AST) to ensure that the code is fluent code using the pipe (`%>%`) operator and that it contains at least one line of code, a variable (or symbol in R) pointing to the dataframe. Unravel then splits the fluent code into intermediate expressions of the chain on the pipe operator. For each expression, we strip the pipe (`%>%`) operator at the end of an expression, and store a list of these expressions to be evaluated in the next step. A challenge we faced was graceful handling of parsing errors along the chain. We chose to perform a best effort at parsing syntactically correct lines and exclude the syntactically incorrect line and subsequent lines. This simpler implementation relies on the user to fix their code first instead of skipping to the lines after the error.

Trace executions. To produce the intermediate dataframes, we iterate through the list of intermediate expressions from the previous step, and evaluate each expression to create a new list holding all of the intermediate dataframes. For lines that throw an error, we store the error message and skip the rest of the lines that may follow. The message is presented in the function summary tooltip to give users feedback as they would receive it on the console output. An alternate implementation could be to skip the line which causes the runtime error, and keep evaluating the rest of the lines. We decided to rely on a simpler solution: storing the error message and displaying it when the user clicks on the summary box (Figure 3 D) next to the problematic line so that they can try to fix it. We also provide toggle switches to disable problematic lines. While tracing the executions, Unravel extracts and stores its row and column dimension information, as well as the type of change occurred. As before, these dataframes and their associated information are stored in a list for the UI to reference.

Summary generation. Unravel generates summaries using an extension of `tidylog` [11], a package that is designed to log function summaries of tidyverse R code onto the console output. By loading our extension of `tidylog`, we override transformation functions with custom logging functions using the same name and signature. For example, when the user calls functions like `group_by`, we use our own custom `group_by` function to introspect into the input dataframe and its arguments. We extended `tidylog` to capture summaries of each function instead of printing them to the console. There are numerous ways one could describe a function's effect on a dataframe such as warnings against certain parameters. However, we decided to focus on three simple pieces of information: 1) Mention the dataframe dimensions and if they have changed, 2) Highlight important column variables, and 3) Provide supplementary information about functions that have subtle changes like `summarise` (Figure 4).

4.2.2 Visual Cues. Before interaction is possible, Unravel constructs the GUI using information about the chain from previous steps. To provide transparency about the data and its lineage in fluent code (D1), Unravel uses the dataframe dimensions, types of changes, and function summaries to create visual cues. We first describe the design of the visual affordances below.

Data change schema. To help data scientists pay attention to subtle changes, we designed a simple data change schema which visualizes different types of changes . We analyze the difference between the incoming and the resulting dataframe when a transformation function is called. Different colors are used highlight changes within the summary box, code and the dataframe output. "No changes" indicates no changes occurred after an operation. "Visible Changes" indicates the dataframe was transformed (e.g. creating new columns, mutating existing columns). The "Internal Changes" indicates the dataframe has been marked as a grouped by variables or by rows, and there is no visible effect. Although an "Error" is not an explicit change, it indicates a runtime error.

Code, summary, and dataframe highlights. To help data scientists keep track of dataframes and their properties, Unravel highlights the code, function summaries, and the dataframes using the data change schema described above. We drew inspiration of this design by Wayne [30]'s strategy to use run-time information to highlight parts of the code. There are lots of properties one could access from a dataframe during runtime, such as the number of missing values, but we decided to highlight column variables of interest in the code, output, and function summary (Figure 4). For a particular line, Unravel compares the previous and the new dataframe to highlight column names if they were transformed, or used as a group variable. Unravel also highlights text related to the changed column variables within the function summary text. Finally, the output dataframe column(s) is also highlighted accordingly.

Always-on visual cues for data transparency. To achieve our design goal of providing transparency (D1) about the dataframe, Unravel uses always-on visualizations. Data scientists have to continually track properties about dataframes which can be cognitively demanding, especially in complex data wrangling code composed of many operations. TensorSensor [27] approaches the problem by improving the quality of exception messages around data dimensions, a particularly difficult task for novices. We were also inspired by Lieber et al. [24]'s always-on visualizations which tracked the number of api calls for web applications to help prevent misconceptions among students. Unravel's always-on visualizations consists of a summary box next to each line which displays the dataframe's row and column dimensions (Figure 3 D) and a color to indicate the type of change according to the data change schema. Visual diffs—a display of the differences between lines of code—could have been used to illustrate differences between two dataframes. However, data scientists might not always be interested in checking the change between operations and a diff visualization might be too disorienting. We decided on a simpler design to show a high-level snapshot state of intermediate dataframes in terms of relevant columns that were added or changed.

4.2.3 Fluent Code Interactions. Unravel constructs the GUI by incorporating the dataframe information captured by evaluating intermediate expressions from the previous steps. We also link communication between R and JavaScript to respond to user interactions. Once the setup is complete, users can start inspecting the fluent

code or apply structural code edits for exploration. Below, we discuss the design behind the structural drag-and-drop and toggle switch interactions to achieve D2.

Fluent code overlay. To help data scientists interact with fluent code, Unravel creates a web application within RStudio which overlays the code with a UI (Figure 3 B). Lerner [23] used the idea of live projection boxes—presenting runtime values in boxes as the user types—for live programming to keep track of changes in data types like lists and arrays. We considered adopting this idea for data wrangling with fluent code, but typing can be cumbersome and the continual visual updates could become distracting. To help data scientists focus, we designed Unravel as an *exploration mode* for data scientists to inspect and explore fluent code in isolation. Therefore, Unravel is presented in a separate window but within the IDE. Unlike projection boxes, Unravel only shows one dataframe at a time for a particular line in the fluent code. Information about the intermediate lines of code and their respective dataframes are used to populate and update UI elements on the fluent code overlay for displaying dataframe dimensions and types of changes occurred.

Structural edits via drag-and-drop and toggle switch interactions. To help data scientists easily edit fluent code, Unravel provides drag-and-drop to reorder lines and toggle switches to enable/disable them. The order of operations (lines) is important in fluent code because a dataframe is transformed by functions in sequence along the chain. Drag-and-drop interactions on code has been used previously to help users refactor or change code, and fix bugs [4, 22] We use drag-and-drop to explore the effects of function order. Using the move icon (Figure 3 E), a line can be dragged before or after another line. Upon dropping a line, Unravel automatically evaluates the code to produce new dataframes to explore. Unravel will also handle trailing pipe (`%>%`) operators for the last enabled line in the new code overlay. Another structural edit we implemented was enabling or disabling a line using toggle switches (Figure 3 F). We use toggle switches as another structural edit to help data scientists examine the effects behind the presence or absence of certain functions. Although a simple edit, this can be useful for isolating the exploration on certain lines of the chain, or disabling problematic lines temporarily.

4.3 System Scope and Limitations

We limited the scope of our tool in order to explore the usefulness of interactive exploration of fluent code. Here, we briefly describe the scope and limitations of Unravel.

Supported code. We scoped our tool to focus on single-table data wrangling functions in the `dplyr` [34] and `tidyr` [35] data wrangling packages that use the fluent interface. Certain non-fluent code like variables storing dataframes and other similar data types like lists could benefit from always-on visualizations. However, we chose to limit Unravel to fluent code since it is a prevalent pattern of coding in R. For highlights on code, Unravel is limited to simple function parameter values representing column names. One challenge with highlighting columns on the code snippet is that some functions can have parameters that also accept a function as its value. We chose to scope our tool to initially handle the simpler parameter values. In a future version, Unravel could handle arbitrary nesting of expressions to isolate column variables in code by analyzing the AST further. Finally, the output of an operation in fluent code could produce other types of data besides dataframes. However, we only support dataframes as outputs because data wrangling typically involves transformations of dataframes. Unravel can be extended to render other types of data to enable explorations on complex code.

Evaluation limitations. Unravel does not attempt to sanitize a valid fluent code for side-effect functions, guarantee deterministic outputs, or optimize for performance. Some functions in both base R and `tidyverse` R cause side effects instead of returning values. Unravel is not currently aware of such functions, which could cause unexpected results. Using our evaluation strategy to generate the intermediate outputs, if a line within the chain contains a function that generates random numbers (e.g. `runif`), we currently generate new numbers for each subsequent operation. This can be an unexpected result if programmers make use of such functions. Lastly, we did not optimize Unravel to handle large dataframes and opted to use smaller datasets for the study. Hence, Unravel will become sluggish once dataframes become too large and we would need to paginate the results in a more efficient way.

5 EVALUATION: FIRST-USE STUDY

To evaluate the usefulness of Unravel, we conducted a first-use study with 14 data scientists, with varied levels of experience. On a 5-point Likert scale, participants self-reported their experience in data science (μ = 3.6), data wrangling (μ = 3.7), R (μ = 3.7), and the fluent interface (μ = 4).

5.1 Methods

We conducted the studies over video conference using an online version of Unravel. We began each study by describing the tasks to participants. The tasks used built-in R datasets like `mtcars` and `iris`, some open datasets like `diamonds` (included in the `ggplot2` package [32]), `babynames` [33], and `gapminder` [5], as well as one hand-crafted dataset called `student_grades`. The participants were tasked with exploring several code snippets written in the tidyverse R dialect using the `dplyr` and `tidyr` packages. The code snippets were chosen to tease out how users would discover and explore prototypical types of data wrangling tasks like selecting, filtering, mutating, grouping, and summarising dataframes.

For each code snippet, the task began open-ended where participants could explore each code snippet with Unravel then focused on specific tasks tailored to each snippet. We wanted participants to start using Unravel with open-ended exploration to capture their initial interactions with the tool. We then focused on specific tasks related to probing certain lines, and performing actions like toggling lines on or off, reordering lines, and asking them to observe the effect of the functions on the dataframe. Finally, we also asked participants to explore their own code in the RStudio IDE to gauge how well Unravel could be integrated into their daily workflows. While interacting with the tool, we asked participants to think aloud and ask questions. After the completion of the study, we administered an exit survey to measure the usefulness of Unravel features, and to ask for additional feedback from participants.

5.2 Post-study Survey Results

On 5-point Likert scale, participants positively rated the usefulness of Unravel overall (μ = 4.6). Participants found the clickable lines for viewing intermediate dataframes (μ = 4.6) and toggle switches for enabling or disabling lines (μ = 4.6) were the most useful features. Similarly, the participants positively rated the usefulness of the summary boxes for viewing the dimensions and data change type (μ = 4.4), and drag and drop for reordering lines (μ = 4.3). However, there were less positive ratings for the usefulness of the data change color schema for visual highlights on code and dataframe outputs (μ = 4.1), and function summary tooltips (μ = 3.9). 93% of the participants responded that they would likely use Unravel to debug fluent code, while 79% of the participants responded that they would use it to understand fluent code.

5.3 Qualitative Results

We present our qualitative results from the user study, describing the interactions we observed, and the feedback participants provided throughout the tasks. The results of our first-use study suggest that Unravel addresses the design goals we formulated in Section 4.1. Participants found that Unravel provided transparency about the data (D1), allowed just-in-time exploration (D2), and minimized context switches between code and data (D3). In this section, we discuss our study results through the context of our design goals.

5.3.1 Visual Cues Helped Achieve Data Transparency. Participants relied on the visual cues to track transformations of a dataframe across the chain (D1). Summary boxes provided a useful visual cue for the basic properties of dataframe. Participants like P5, P6, P14, or P1 used the summary box dimensions to infer changes like adding columns or stripping rows from certain operations like `filter` or `summarise`. For example, P14 found that "it was very useful to have this at a glance information about the data shape and type of change at times because it supports quickly checking if the dataframes are correct." Upon discovering the row and column dimensions of the summary box, P5 thought "that feature of rows and columns numbers is I think one of the most powerful teaching things. It's really cool to see mutate is adding this column." The data change schema highlights were used by the participants to validate changes to the dataframe between steps. P4 expressed, "I like that you can flip between lines pretty quickly to see what changed, you have something that guides your attention. Being able to step through it and being able to walk through like look at this, look at that!" P5, a data science educator, commented how the internal change would be useful for teaching students about the behavior of grouped data: "I really like being able to show this internal change. This color scheme is really nice. Because I think students, even after you tell them they should expect it, they miss it."

5.3.2 Explorations on the Code and Data Enabled Checking Assumptions. Overall, we were able to achieve D2. Participants clicked on different parts of the chain, and explored the code using the drag-and-drop and the toggle switches to validate their assumptions about the code and output. We found that being able to click on arbitrary lines of the fluent code was helpful for data scientists to inspect intermediate dataframes without being constrained to a linear stepping interaction like debuggers. The toggle switches to enable or disable lines helped participants explore the influence of certain functions when used with other functions. For example, P2 expressed that "it's cool how it helps dispel goofy assumptions about what attribute persists versus not. It made me examine so many assumptions especially grouping." Participants like P12, P8, P4 or P10 tested hypotheses about the code behavior by applying structural edits to the fluent code like disabling lines or reordering them. When examining the role of a `group_by` function for example, P10 guessed that "if you don't group by species then that would just work on the entire dataset." P10 then toggled off the `group_by` line and confirmed "when you summarise the total now, it's applying this function across everyone in the dataset."

Participants explored and made use of the summary text to understand operations with visible but subtle changes. Some R experts (P4, P2, P6) found that toggling lines on or off especially useful for understanding pivoting operations: "being able to quickly flip between lines after toggling things on or off is nice for these pivot wider or longer functions." (P4). P2 found the summary text was useful for confirming their own summaries they made mentally: "I really like the pivot summary description because it can be difficult to visualize the pivot operations." Sometimes, order effects were explored by participants. For example, P6 was able to validate the importance of placing a function like `filter` line before running a summary function on the dataframe by reordering lines. Using the summary descriptions of row changes they confirmed the behavior: "There are rows outside of mass that are also being dropped. This makes sense because I missed the hair color column, so that's where I'm getting my counts of 28 versus 33 rows. This is interesting to be able to check your assumptions of getting the outputs."

5.3.3 Unravel Helped Minimize Context Switches Between Code and Output. Participants found the integration of Unravel into their IDE workflow to be useful. All participants commented that it was quite convenient that they could simply pipe (`%>%`) their own fluent code in R into the `unravel` function to open up the tool in RStudio (D3). Upon unraveling a complex tidyverse code snippet P4 commented, "Oh wow, it actually worked! I like how you pipe it in at the end and it gives you this awesome thing." However, other participants wanted tighter correspondence between the text editor and the explorable code. For example, P3 expected the structural edits on the text editor to automatically update the code overlay and suggested adding it as a feature since "it's so handy to not have to manually run from editor." Other participants (P4, P6, P9, P11) tried to edit the code on the interactive code overlay itself, suggesting a need for live updates to the editor to further reduce context switches, especially if they want to copy the edited code.

6 DISCUSSION AND FUTURE WORK

In this section, we discuss the broad implications of our findings and identify the ways in which Unravel could be adapted to various programming languages and contexts.

6.1 Unraveling Code in Educational Settings

One common thread from our study was the excitement around using Unravel as an educational tool. Data science educators make use of computational notebooks, which students use to engage

with content, code, and interactive elements like widgets or videos. These interactive documents enrich the learning experience by allowing learners to explore and understand code. P5 wanted to use Unravel as a teaching tool to author exercises in the spirit of Parson's Problems [9], asking students fix problematic fluent code using the structured drag-and-drop interactions. P9 commented that Unravel would have helped their tutoring sessions within the IDE because "it would've been really useful to walk through steps of how it starts getting data to how it ends." In a future version, Unravel could be used within interactive tutorials to explore code in RMarkdown using the learnr package [29] or extended for Jupyter Notebooks.

6.2 Data Scientists Can Benefit From in-Situ Learning Tools

Instead of offloading to external learning resources, we should provide learning tools within the IDE or computational notebook. Data scientists can avoid context switches by using in-situ tools to generate code [10, 19], version code [18], or manage messy code [15]. Unravel builds on this approach by offering an *exploration mode* for fluent code within the IDE. An interesting application of Unravel is to help data scientists understand code generated by programming by example techniques. Ferdowsifard et al. [12]'s study of a live programming tool for python found that "programmers do not try to understand the code generated by the synthesizer." To encourage programmers to understand, tweak, and trust synthesized code, providing a means to explore code interactively with descriptions of operations might be useful. The data scientists in our study found it useful to use Unravel within the IDE because they could understand and explore their own code. However, some participants wanted live updates between the code in text editor and the explorable code overlay. In the future version, Unravel could incorporate elements of live programming techniques, syncing the structural edits to the original code. Based on our participants' focused behavior using Unravel, we believe that constant updates of dataframes during typing could be too distracting, so there should be a careful balance between liveliness and focused explorations.

6.3 Unravel in Other Programming Contexts and Environments

Unravel can be adapted to other programming languages and contexts to help understand and explore fluent code. Language-Integrated Query (LINQ)[10] is a domain-specific language in C# that is used to query from various data sources such as relational databases (SQL). LINQ uses the fluent interface to filter on data:

```
List<int> numbers = new List<int>() {5, 4, 1, 3, 9, 8, 6,
↪  7, 2, 0};
var orderingQuery = numbers
    .Where(num => num < 3 || num > 7)
    .OrderBy(n => n);
```

Structured explorations on LINQ queries can help C# programmers inspect and debug the chain by allowing them to inspect each intermediate result, and explore variations with drag-and-drop or toggle interactions. Similarly, Python uses fluent code through method chaining to wrangle data using the `pandas` library:

```
df[['fl_date', 'tail_num', 'dep_time', 'dep_delay']]
    .dropna()
    .sort_values('dep_time')
```

Since each method returns a dataframe, we can analyse the code and split on method calls to store the intermediate dataframe for each operation (`[[`, `dropna`, and `sort_values`). Python programmers can then rely on the same visual cues to display dataframe dimensions and types of changes occurred, as well as explore the code via structural edits. Fluent code is also used in data processing frameworks like Spark[11], which is used for handling big data. Spark provides wrappers for many languages including Python and R. Programmers using Spark can use a tool like Unravel for interactively exploring fluent code that handles much larger data. Scaling Unravel to handle big data will require an effective way to summarise and visualize data transformations. Here, it might be useful to use Niederer et al. [26]'s strategy behind TACO, an interactive comparison tool that visualizes the differences between multiple tables at various levels of detail. Instead of showing the entire table of each step, it might be useful to initially provide an overview of data changes in terms of row and column differences, before selecting a particular dataframe to get more details.

7 CONCLUSION

We explored the usefulness of Unravel, a tool that enables structured edits via drag-and-drop and toggle switch interactions with always-on visualizations to help data scientists explore and understand fluent code. Through examination of the R community, we identified several needs related to exploring fluent code. To address those needs, we designed Unravel which integrates within data scientists' IDE, helps them gain transparency about data, and explore fluent code using simple structural edits like reordering and enabling or disabling lines. Through a first-use study with 14 data scientists, we found that Unravel facilitated diverse activities such as validating assumptions about the code or data, finding redundant or equivalent code, and learning about function behavior. Based on our results, we discussed some ways to generalize Unravel to other programming languages and contexts and identified future work to better support interactive exploration of fluent code.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant No. 2006947.

REFERENCES

[1] [n.d.]. *Pipe.* https://magrittr.tidyverse.org/reference/pipe.html
[2] [n.d.]. *Tidyverse.* https://www.tidyverse.org
[3] Stefan Milton Bache and Hadley Wickham. 2014. *magrittr: A Forward-Pipe Operator for R.* https://CRAN.R-project.org/package=magrittr R package version 1.5.
[4] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. 2016. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 211–221.

[10] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

[11] https://spark.apache.org

[5] Jennifer Bryan. [n.d.]. *gapminder: Data from Gapminder.* https://github.com/jennybc/gapminder, http://www.gapminder.org/data/, https://doi.org/10.5281/zenodo.594018.

[6] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Unterface Software and Technology*. 473–484.

[7] Winston Chang, Joe Cheng, JJ Allaire, Carson Sievert, Barret Schloerke, Yihui Xie, Jeff Allen, Jonathan McPherson, Alan Dipert, and Barbara Borges. 2021. *shiny: Web Application Framework for R.* https://shiny.rstudio.com/ R package version 1.6.0.9000.

[8] Tamraparni Dasu and Theodore Johnson. 2003. *Exploratory Data Mining and Data Cleaning*. Vol. 479. John Wiley & Sons.

[9] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth International Workshop on Computing Education Research*. 113–124.

[10] 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*. 1–12.

[11] Benjamin Elbers. 2021. *tidylog: Logging for 'dplyr' and 'tidyr' Functions.* https://github.com/elbersb/tidylog/ R package version 1.0.2.9000.

[12] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 614–626.

[13] Martin Fowler. 2010. *Domain-specific Languages.* Pearson Education.

[14] Martin Fowler and E Evans. 2005. Fluent interface. *martinfowler.com* (2005).

[15] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[16] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. 2020. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data.* https://doi.org/10.5281/zenodo.3960218 R package version 0.1.0.

[17] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372.

[18] 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*. 1265–1276.

[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 33rd Annual ACM Symposium on User Interface Software and Technology*. 140–151.

[20] Amy J Ko and Brad A Myers. 2004. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 151–158.

[21] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. TweakIt: Supporting end-user programmers who transmogrify code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[22] Yun Young Lee, Nicholas Chen, and Ralph E Johnson. 2013. Drag-and-drop refactoring: Intuitive and efficient program transformation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 23–32.

[23] Sorin Lerner. 2020. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[24] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2481–2490.

[25] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How data science workers work with data: Discovery, capture, curation, design, creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–15.

[26] Christina Niederer, Holger Stitz, Reem Hourieh, Florian Grassinger, Wolfgang Aigner, and Marc Streit. 2017. TACO: Visualizing changes in tables over time. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2017), 677–686.

[27] Terrence Parr. [n.d.]. *Clarifying exceptions and visualizing tensor operations in deep learning code.* https://explained.ai/tensor-sensor/index.html

[28] Xiaoying Pu, Sean Kross, Jake M Hofman, and Daniel G Goldstein. 2021. Datamations: Animated explanations of data analysis pipelines. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[29] Barret Schloerke, JJ Allaire, Barbara Borges, and Garrick Aden-Buie. 2021. *learnr: Interactive Tutorials for R.* https://rstudio.github.io/learnr/, https://github.com/rstudio/learnr.

[30] Hillel Wayne. [n.d.]. *Syntax highlighting is a waste of an information channel.* https://buttondown.email/hillelwayne/archive/syntax-highlighting-is-a-waste-of-an-information/

[31] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[32] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org

[33] Hadley Wickham. 2019. *babynames: US Baby Names 1880-2017.* https://CRAN.R-project.org/package=babynames R package version 1.0.0.

[34] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. 2021. *dplyr: A Grammar of Data Manipulation.* https://CRAN.R-project.org/package=dplyr R package version 1.0.4.

[35] Hadley Wickham and Lionel Henry. 2019. *tidyr: Tidy Messy Data.* https://CRAN.R-project.org/package=tidyr R package version 1.0.0.