Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration

Priyan Vaithilingam
UC San Diego
La Jolla, CA, USA
pvaithil@eng.ucsd.edu

Philip J. Guo
UC San Diego
La Jolla, CA, USA
pg@ucsd.edu

ABSTRACT

Programmers, researchers, system administrators, and data scientists often build complex workflows based on command-line applications. To give these power users the well-known benefits of GUIs, we created Bespoke, a system that synthesizes custom GUIs by observing user demonstrations of command-line apps. Bespoke unifies the two main forms of desktop human-computer interaction (command-line and GUI) via a hybrid approach that combines the flexibility and composability of the command line with the usability and discoverability of GUIs. To assess the versatility of Bespoke, we ran an open-ended study where participants used it to create their own GUIs in domains that personally motivated them. They made a diverse set of GUIs for use cases such as cloud computing management, machine learning prototyping, lecture video transcription, integrated circuit design, remote code deployment, and gaming server management. Participants reported that the benefit of these bespoke GUIs was that they exposed only the most relevant subset of options required for their specific needs. In contrast, vendor-made GUIs usually include far more panes, menus, and settings since they must accommodate a wider range of use cases.

Author Keywords

command-line interfaces, GUI synthesis, PBD

CCS Concepts

•Human-centered computing → Human computer interaction (HCI);

INTRODUCTION

GUIs have been the most visible form of human-computer interaction in the past four decades, but command-line applications are still frequently used by programmers, researchers, system administrators, DevOps, digital archivists, machine learning engineers, and data scientists. Look over the shoulder of any of these people at work, and they probably have multiple terminal windows open, each running a diverse array of command-line applications. These include widely-used tools for version control (e.g., Git [13]), software development (e.g., GCC, LLVM, npm [21], Webpack [26], Docker [8]), computational science (e.g., BLAST [27]), and media processing (e.g., ImageMagick [14], FFmpeg [12]).

Command-line apps are flexible and composable: experts can quickly tinker with parameters, connect app outputs via pipes and files, and write scripts to chain multiple apps together. But they exhibit classic usability problems that hinder novices, the most salient being a violation of *recognition over recall* [29]: Users must recall and type in the exact arcane syntax of parameters rather than being able to recognize items in a GUI. For instance, here is the top forum answer on how to use FFmpeg to overlay a new .wav audio track atop an existing .mp4 video [11]: "`ffmpeg -i v.mp4 -i a.wav -c:v copy -map 0:v:0 -map 1:a:0 new.mp4`"

How can we make these complex command-line applications more usable? It is hard to design a suitable GUI for these apps since they often have up to hundreds of complex command-line options, so a comprehensive GUI could have hundreds of selectors and input boxes, dozens of panes, and grow into a tangled mess. What about making a custom GUI for a particular app that exposes only the subset of its many options that meet a given user's needs? This may be feasible, but hand-coding a bespoke GUI takes a lot of software development effort, and it can be hard to alter UI details as user needs inevitably change over time. To eliminate this manual coding effort, we present a novel technique that automatically synthesizes a GUI for a set of command-line apps by having the user demonstrate how those apps are invoked. This way, a power user can quickly create GUIs without writing any code.

We prototyped our technique in a system called *Bespoke*. Figure 1 shows an example usage scenario: Alice is a senior scientist in a computational biology lab. She is adept at invoking the dozens of command-line apps required for her job, but she is not a programmer. Despite not coding, she still wants to create simplified GUIs for common computational workflows in her lab so that her junior labmates can get their work done without the complexities of command-line environments.

1. Alice starts Bespoke, which resembles a Jupyter notebook [23]. She creates a new notebook cell that contains an embedded terminal. There she types the first command in her workflow, `cutadapt` (a DNA sequence filter [59]),

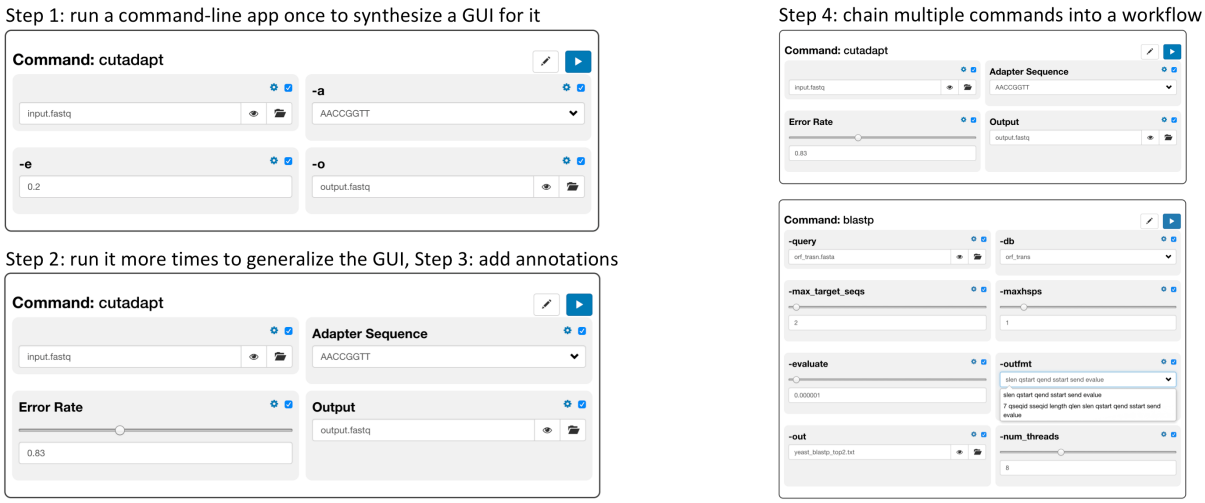

Figure 1: Bespoke lets users create GUIs by invoking command-line applications and annotating their workflows (see Introduction for step details).

with these arguments: `input.fastq -a AACCGGTT -e 0.2 -o output.fastq`. Bespoke runs that command and creates a simple GUI based on the inferred types of its arguments: an input text box for `-a`, a numerical spinner for `-e`, and file selectors for the input and output filenames.

2. Alice runs `cutadapt` a few more times with different arguments to provide more demonstrations for Bespoke. For instance, as she passes in different `-e` values, Bespoke infers that it should create a GUI range slider with reasonable bounds for that argument rather than an open-ended numerical spinner. She can override any undesirable inferences.
3. She adds human-readable labels to these GUI components, such as "Adapter Sequence" for `-a` and "Error Rate" for `-e`.
4. After she is happy with the synthesized `cutadapt` GUI, she continues creating more terminal cells for additional commands in her workflow, such as BLAST and FFmpeg. Bespoke synthesizes those GUIs by demonstration as well. In addition, it detects argument relationships *across* commands, such as the output file of one being passed into the subsequent one; this lets the GUI auto-propagate values.
5. Each cell also shows previews of text, image, audio, and video outputs from its command, along with an inline text editor that can be used to edit input files. The final command in Alice's workflow is an FFmpeg call that stitches DNA sequence images together into an animation; users can watch that animation in the cell to check for accuracy.
6. Alice sends this synthesized GUI to her labmates to use. To facilitate scientific experiment reproducibility, users can export their current GUI state as a runnable shell script.

The main benefit of Bespoke is that it allows power users like Alice to create GUIs for their command-line-based workflows without writing any code. The resulting GUIs are useful not only for novices but also for experts like Alice, who can quickly explore the parameter space of their experiments using a streamlined GUI. Since each research team likely has their own idiosyncratic workflows, it is critical to enable them to quickly build GUIs for their own needs. In this case, another biology lab may use those exact same tools (e.g., cutadapt, BLAST, FFmpeg) in completely different ways, so it is unlikely that someone could build a one-size-fits-all GUI that meets every biologist's unique computational needs.

Bespoke makes two main research contributions: 1) It unifies the two dominant forms of desktop human-computer interaction (command-line and GUI) via a new hybrid approach that combines the flexibility and composability of command-line interfaces with the usability and discoverability of GUIs. 2) It extends literate programming [44] to GUIs rather than solely text. Both classic (e.g., CWEB [44]) and modern incarnations (e.g., Jupyter notebooks [23]) of this long-standing idea are based on weaving text-based code and inline documentation. Bespoke extends the reach of literate programming by letting users 'write' command-line narratives, which it synthesizes into inline GUIs within notebook cells. This interaction style makes it easier for data scientists, researchers, and sysadmins to build interactive workflows intertwined with documentation. As command-line ecosystems (e.g., Docker, Kubernetes) grow more complex and as notebooks spread to fields far beyond science (e.g., Netflix uses them for orchestrating datacenter tasks [1]), we expect the ideas embodied by Bespoke to become more relevant to a broader set of domains.

To assess the versatility of Bespoke, we ran an open-ended study where six users created their own GUIs in domains that personally motivated them. They made a diverse set of GUIs for use cases such as cloud computing management, machine learning prototyping, lecture video transcription, integrated circuit design, remote code deployment, and gaming server management. Participants reported that the main benefit of these bespoke GUIs was that they exposed only the most relevant subset of options required for their specific needs.

The contributions of this paper are:

- The idea of unifying GUI and command-line interfaces via a demonstration-based technique that interactively synthesizes GUIs from command-line invocations.
- An instantiation of this technique in Bespoke, a computational notebook application that lets users create multi-stage workflow GUIs by running command-line applications on their computer and documenting their steps.

RELATED WORK

Bespoke continues the long lineage of systems that facilitate end-user programming, which Ko et al. define as the act of creating software for personal use rather than for public dissemination [45]. Bespoke is an end-user programming tool that synthesizes GUIs for personal use within, say, a particular scientific lab, rather than for widespread public use. Within this genre, Bespoke is most closely related to tools for programming by demonstration and custom UI generation.

Programming by Demonstration

Programming by demonstration (PBD) is an end-user programming technique where the system generates a program by watching a user demonstrate a set of actions on their computer rather than making the user write textual code [32, 48, 54]. (Some researchers call a closely-related technique "programming by example" [37, 40, 50].) The promise of PBD is that end-users can do programming without writing code.

One major class of PBD systems aims to synthesize task automation scripts so that users do not need to manually write scripting code. These span a variety of application domains including text manipulation [28, 49, 61, 64, 74, 75], file and directory management [39, 65, 66], image editing [46, 47, 53, 60], webpage task automation [51, 52, 58, 73], web data scraping [30, 50, 55], and data wrangling [38, 40, 43]. Although their domains vary widely, these systems all generate scripts or processed data as outputs. Bespoke differs in a fundamental way since it generates an interactive GUI as output.

More closely related to Bespoke are PBD systems that synthesize GUIs by example. Peridot [67] and Lapidary [68] enable users to draw UI elements (e.g., menus, radio buttons, progress indicators), their dynamic behaviors, and data constraints; the systems then generate code to control the layout and behaviors of those UI elements. Highlight [69] allows users to demonstrate actions on a webpage and then synthesizes a mobile version of the webpage's UI based on those observed actions. Bespoke shares the high-level goals of these systems but is the first, to our knowledge, to synthesize GUIs from user demonstrations of command-line apps. It brings classic PBD ideas to the novel domain of making modern command-line-based workflows more usable.

Generating Custom User Interfaces

Bespoke is more distantly related to systems that automatically generate UIs in domains such as accessibility and ubiquitous computing. SUPPLE [34] and SUPPLE++ [35] generate custom UIs for users to accommodate their motor and vision capabilities based on user-provided specs and an activity trace, respectively. Projects such as UNIFORM [70] and the Personal Universal Controller (PUC) generate custom UIs for appliances such as media consoles and printers that are customized for each user's individual preferences and interaction history. Huddle [71], built atop PUC, generates UIs to coordinate multiple electronic appliances such as those in home entertainment sets. Bespoke shares these systems' goals of making specialized UIs tailored to individual user needs but does so for command-line applications rather than by adapting the UIs of existing GUI apps.

Improving Usability of Command-Line Applications

Bespoke aims to make existing command-line applications more usable by wrapping them in lightweight GUIs. Prior systems have approached similar goals by designing a hybrid between command-line and graphical interfaces. For instance, Inky [62] embeds a command line into the web browser to enable power users to more efficiently perform common browsing actions. It supplements the command line with rich visual feedback and inline GUI widgets such as date pickers and parameter selection menus. The browser-shell extension to LAPIS [63] allows users to manually wrap an HTML GUI around existing command-line apps and then invoke them directly from a browser-based command line. Unlike these systems, which require users to manually create the GUI widgets that augment their command-line interfaces, Bespoke automatically synthesizes GUIs by demonstration.

Complementary projects make the command line more usable by providing richer help, error correction, and natural language UIs. Tutorons [41] shows contextually-relevant explanations of what command-line arguments mean by parsing the documentation of commands such as `wget` and adapting explanations to the user's inputted arguments. NoFAQ [33] provides suggestions for correct command-line invocation syntax based on a corpus of expert-curated pairs of faulty and fixed commands. NL2Bash [56, 57] implements a natural language interface to the Unix command line, translating English phrases such as *"display the 5 largest files in the current directory and its sub-directories"* into Bash shell commands. Bespoke differs from these systems by synthesizing GUIs instead of enhancing text-based command-line interfaces.

BESPOKE SYSTEM DESIGN AND IMPLEMENTATION

Bespoke is a cross-platform Electron [10] desktop app that enables users to run arbitrary command-line applications in embedded terminals and see inline GUIs synthesized from those runs. Figure 2 shows how users can create two types of cells: a) Markdown-formatted text cell [36] for exposition, b) *command cell* where they can run a single command-line app (often repeatedly with different arguments) and have Bespoke synthesize a GUI for it. This interface was inspired by computational notebooks like Jupyter [23].

The simplest use case is to make only one command cell to synthesize a GUI for a single command-line app (e.g., `git`, `docker`, `ffmpeg`). More realistic use cases involve making a series of command cells, each representing a stage in the user's workflow (e.g., Alice's computational biology workflow in Figure 1). Each command cell has a Run button, which runs that command and sends its output to the terminal at the bottom of the interface. Commands from all cells share one global Bash shell environment. (In this paper, we will use the term "command" as a shorthand for command-line app.)

Bespoke does not need to know about the implementation details of any specific commands; it should work on any command that the user runs in their terminal. It has been tested so far on macOS and Linux, whose commands usually follow Unix-style POSIX conventions [4]. It should also work on Windows with adjustments to account for its conventions.

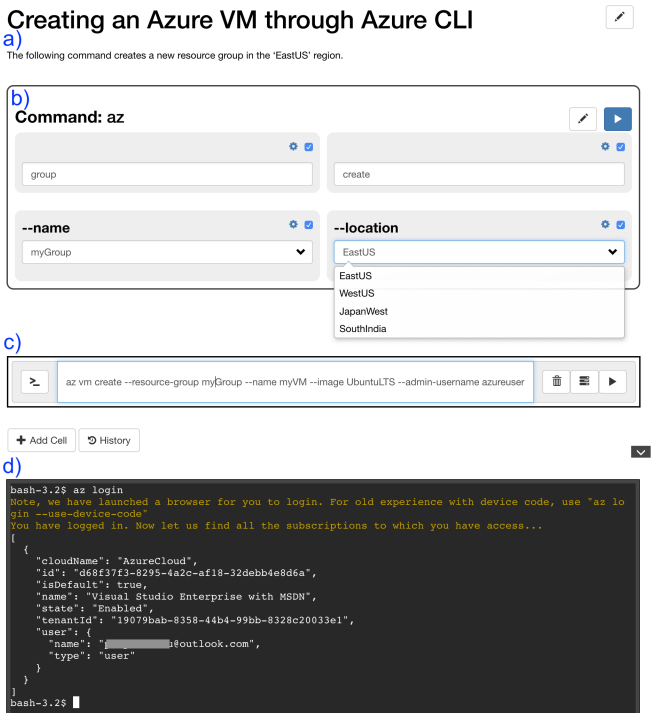

Figure 2: Bespoke allows users to create: a) Markdown documentation cells and b) command cells where GUIs are synthesized. c) new command cell before GUI is synthesized there; d) terminal showing outputs.

Within each command cell, Bespoke synthesizes a GUI in a three-step process whenever the user runs a command: 1) parsing its command-line arguments, 2) inferring types and value ranges for those arguments, 3) mapping those types and value ranges to standard GUI elements. Finally, Bespoke performs inter-cell value inferences to help users create multi-stage workflows. We now describe each synthesis step:

Step 1: Parsing Command-Line Arguments

When the user runs a command in a cell, Bespoke first parses its arguments using a grammar that we empirically derived from studying Unix-style conventions in the official POSIX standards guide [4], command-line parsing utility libraries in C and Python [3, 5, 7, 9], and a curated summary of help manuals from dozens of popular command-line apps [72]. While we cannot guarantee that *all* command-line apps will abide by these conventions, in our experience as long-time users of Unix-based systems, this grammar handles the most common use cases in a variety of domains.

We will explain this grammar using a slightly-modified version of the first command in Alice's workflow from Figure 1:

```
cutadapt in1.fq in2.fq -v --seq ACG -e 0.2 -o out.fq
```

The first string is always the command's name (`cutadapt` in this example). All subsequent strings are its arguments, each of which can be boolean, keyword, or positional:

- Boolean arguments start with - or -- and at least one non-digit character (numbers like -1.3 do not count). These are either followed by nothing or by another boolean or keyword argument. In this example, `-v` is a boolean to indicate whether `cutadapt` should print more verbose output.

- Keyword arguments start with - or -- followed by one or more strings (that do not start with - or --), which are their associated values. In this example, `--seq` is a keyword argument with an associated value of `ACG`, `-e` is a keyword argument with value `0.2`, and `-o` is a keyword argument with a value of `out.fq`. If a keyword argument is followed by more than one string (e.g., `-o out1.fq out2.fq`), they are stored in a list. Keyword arguments and their values can also be delimited with an equals sign instead of with whitespace: e.g., `--seq=ACG`

- Positional arguments are strings that do not start with - or -- and that do not immediately follow keyword arguments. In this example, `in1.fq` and `in2.fq` are positional arguments, which are stored in an ordered list.

Following standard shell conventions, strings with spaces in them (e.g., `"hello world"`) must be properly quoted.

Ambiguity: To eliminate ambiguity, we strongly recommend users follow style guide conventions of putting all positional arguments *before* boolean and keyword arguments [5]. However, users are free to put arguments in any order, such as positional ones (`in1.fq` and `in2.fq` here) at the end:

```
cutadapt -v --seq ACG -e 0.2 -o out.fq in1.fq in2.fq
```

Now Bespoke will incorrectly parse `-o` as a keyword argument with `out.fq in1.fq in2.fq` as its three values (since keywords can be followed by any number of strings). When it synthesizes the GUI in Step 3, the user can see the mistake and manually tell Bespoke that `-o` takes only one value (`out.fq`), so the other two are positional. Since Bespoke does not understand the semantics of individual commands, it cannot guarantee a correct parse if there is such ambiguity. But we assume an experienced user can recognize which arguments have been misparsed once they see the GUI.

Repetitions: If a keyword argument is listed multiple times, all of their values are appended together into a single list. For instance, `-o out1.fq -o out2.fq -o out3.fq` is identical to `-o out1.fq out2.fq out3.fq`. If a boolean is listed multiple times, it will turn into a counter: e.g., `-v -v -v` parses as `-v` with a count of 3. This is a common Unix idiom to specify output verbosity count levels.

Splits: Some commands let users group multiple boolean arguments together with a single - for convenience. For instance, `tar -zxvf` is often used instead of `tar -z -x -v -f`, and `ls -lhG` is shorter than `ls -l -h -G`. By default Bespoke parses each as a single boolean argument, but the user can manually override it in the GUI to split them into multiple single-letter arguments. (Note that `-vvv` will split into `-v -v -v` and parse as a counter of 3.)

Redirection operators: Bespoke parses basic shell redirection operators such as < (take input from a file), > (write standard output to a file), >> (append standard output to a file), and 2> (write standard error to a file), along with its filename.

Pipes: Bespoke parses Unix pipes | that split a command into multiple sub-commands connected by an input-output pipe. Each sub-command parses normally, and the GUI renders them as sub-cells within a single command cell (Step 3).

Special shell syntax: Glob wildcards [31], environment variables, and other special syntax are *not* expanded before parsing. For instance, a wildcard argument named `*.mp3` will appear to Bespoke as that literal string instead of as a list of all mp3 files in this user's current directory. The rationale for this design is that wildcards encapsulate user intent better than their expanded forms. In this case, the user intended to operate on all mp3 files, so a GUI rendered with `*.mp3` as an option will work more robustly on other users' machines, which likely contain mp3 files with different filenames.

Parser limitations: We designed this parser to handle a wide variety of command-line app invocations, including those with complex argument types, redirection operators, and pipes. However shells such as Bash are actually full programming languages with conditionals, control flow operators, and other special forms. Bespoke is *not* meant to parse the full scope of shell languages. We recommend users to write complex shell scripts in separate files and then invoke them within a command cell just like any other command so that Bespoke can generate GUIs for those script invocations.

Step 2: Inferring Types and Value Ranges

After the parser finishes, it produces a structure like the following one for our `cutadapt` example (in JSON notation):

```
{
  "--seq": "ACG",
  "-e": "0.2",
  "-o": "out.fq",
  "-v": true,
  "positional": ["in1.fq", "in2.fq"]
}
```

All values are strings by default (except booleans args). Bespoke infers more specific types from strings using heuristics:

- **Numbers**: Anything parsable as an integer or floating-point number, such as `0.2` in this example.
- **Dates and times**: We use the Moment.js library [20] to parse date and time strings written in a wide variety of formats, such as `2019-04-01` and `03:15:20`.
- **Filenames**: After Bespoke runs each command, it checks whether any string arguments match the names of files or directories that exist on disk. If there is a match, then it infers that value is a filename. In this example, `in1.fq` and `in2.fq` are input files, which presumably exist on disk prior to the run, and `out.fq` is an output file that is created after the command finishes. Bespoke will detect that these are filenames and give the user an option to expand them to full absolute paths if desired.

Although Bespoke can infer types from a single run, in practice users will run the same command multiple times with different argument values as they are experimenting within a single command cell. For instance, suppose that Alice ran `cutadapt` three times in a cell:

```
cutadapt in1.fq in2.fq -v --seq ACG -e 0.2 -o out.fq
cutadapt in3.txt --seq TA -e 0.5 -o out2.fq out3.fq
cutadapt in4.fq --seq GC -e -1.3 -o out4.fq
```

Bespoke refines its inferences for each argument based on values observed in these multiple runs. Here `-e` took on values of 0.2, 0.5, and -1.3[1], and the first positional argument was `in1.fq`, `in3.txt`, and `in4.fq`. Bespoke infers value ranges using the following heuristics:

- **Number ranges and step sizes**: Here `-e` took on values in the range of -1.3 to 0.5, with a likely step size of 0.1 since there is one significant digit of decimal precision. (If the user then passed in 0.25, it would infer a step size of 0.01.)
- **Date/time ranges**: Similar to numbers, Bespoke infers the ranges of dates and times along with likely step sizes based on the most granular resolution (e.g., days, hours, seconds).
- **Filename extensions**: In this example Bespoke infers that the first positional argument can have an extension of `*.fq` or `*.txt`, and that `-o` is likely `*.fq`.

Bespoke continually updates inferences as the user invokes the same command repeatedly with different argument values. If the user attempts to invoke a different command, Bespoke will prompt them to start a new cell, since each cell synthesizes a GUI for (multiple runs of) a single command.

If Bespoke infers different types for the same argument across multiple runs (e.g., a date then a filename), then it will simply default to marking it as a string. Finally, each cell shows a history of all past command invocations, so the user can delete any of them to prevent those from being used for inference.

Step 3: Synthesizing a GUI from Type+Range Inferences

Bespoke synthesizes a GUI for the command in each cell by mapping inferred types and ranges for its arguments into corresponding GUI elements. Figure 3 illustrates the mappings:

- Boolean values render as checkboxes.
- Strings render as text input boxes augmented with autocomplete suggestions based on observed past values. In addition, a dropdown menu shows the five most common values for each argument so the user can quickly choose amongst them. This is convenient for commands whose arguments have a small set of string values. For instance, the first positional argument to `git` is usually one of a few strings, such as `pull`, `add`, `commit`, `push`, or `status`.
- Dates render as a calendar date picker widget, and times render as a time spinner widget. The user can manually override these to select a different granularity.
- Numbers render as either a numerical spinner or a slider widget, depending on the user's preference, with the inferred step size. The user can also manually enter numbers.
- Filenames and directory names render as input text boxes augmented with a pop-up file selector widget. The file selector defaults to the inferred extension types to help filter.

The user can inspect the synthesized GUI and manually change the types and value ranges of any element. If the user

[1]Note that -1.3 is *not* misparsed as the name of a boolean or keyword argument since those strings must start with letters and not digits.

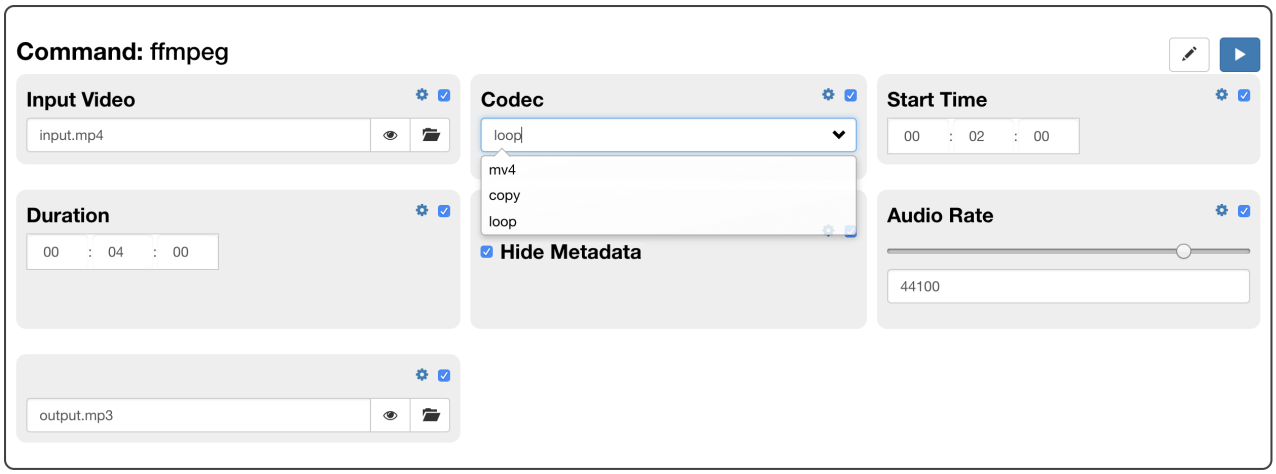

Figure 3: The main kinds of GUI elements that Bespoke synthesizes from observing the user's command-line app invocations: file selectors (e.g., Input Video), text box with dropdown menu (Codec), date/time picker (Start Time, Duration), checkbox (Hide Metadata), and numerical slider (Audio Rate).

makes any changes, Bespoke will no longer attempt to run inferences on those arguments since it assumes that the user wants to take manual control of them from now on. For instance, in a cell the user may always want the first positional argument of `git` to be fixed to the string `commit` in order to make a GUI to specify options for `git commit`.

To prevent incorrect inferences due to user typos, if a command run results in an error, as indicated by a non-zero exit status code, then Bespoke will ask the user whether they actually want to add it to their history for inference.

Element labels and tooltips: By default each GUI element is labeled with the name of its corresponding argument. Users can manually change labels to more human-readable ones, such as replacing "`--seq`" with "Adapter Sequence" in Figure 4. To provide additional guidance, Bespoke parses the man (manual) pages and `--help` pages of the given command, if available, to extract the lines of documentation for that argument (similar to Tutorons [41]). It displays that documentation as a pop-up tooltip on mouse hover. Users can also manually write tooltips to provide custom instructions.

Inline file viewer and editor: For all filenames that the user selects in the GUI, Bespoke shows an inline preview of those files in the cell if they are text, images, videos, or audio clips. We use the Unix `file` command [2] to detect file types, since it is more robust than file extensions. For unsupported types, Bespoke adds a button to open those files in the OS's default handler program. In addition, text files can be edited and saved in the cell. In our `cutadapt` example, the user can edit `*.fq` files within the cell since those are text files.

Being able to view output files makes it convenient for users to iterate effectively with instant visual feedback. For instance, when using the GUI to tune argument values for image processing algorithms in the ImageMagick [14] command, the user can see the processed output images in the cell and adjust accordingly. Also, being able to edit input text files makes it convenient for users to adjust configuration settings for certain commands, which are often specified in text files.

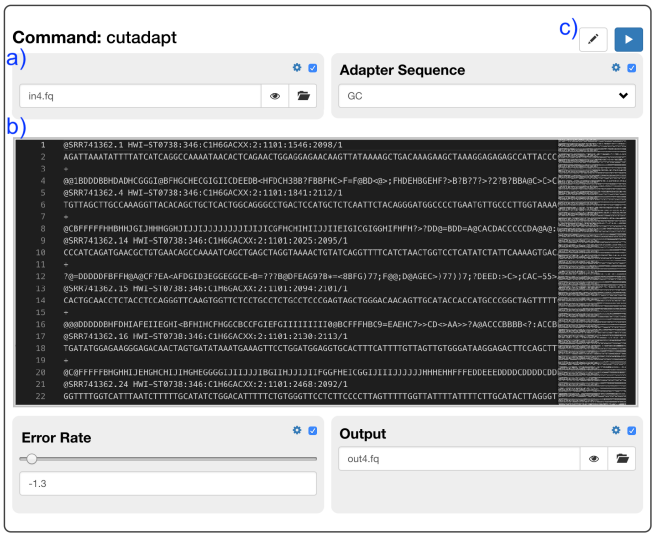

Figure 4: The GUI that Bespoke synthesizes for our `cutadapt` example, showing a) input elements, b) inline text file editor, c) edit button to provide additional command-line examples to refine the synthesis.

Some commands read and write files that are *not* explicitly passed in as arguments. For instance, running `make` without any arguments reads a `Makefile` in the current directory, `docker` reads a `Dockerfile`, and `npm` reads `package.json`. To detect these *implicit* arguments, Bespoke uses a heuristic of comparing access and modification times of files in the current directory and sub-directories both before and after each command invocation. It lists those accessed/modified files in the cell so users can view/edit them.

Figure 4 shows the final synthesized GUI for our `cutadapt` example: the input and output file choices are exposed as file selector widgets, "Adapter Sequence" is a text box with suggested dropdown values, and "Error Rate" is a numerical slider. The user can also edit input files directly in the cell (Figure 4b) and provide additional command-line invocations as examples (Figure 4c) to refine the synthesis.

Supporting Multi-Stage Workflows

So far we have described how Bespoke synthesizes a GUI for a single command within one cell. If that command includes a Unix pipe, then it is split into multiple sub-commands, which each render as sub-cells in that cell. However, more complex user workflows involve multiple independent commands. For instance, a user may `ssh` into a remote server and then run a series of commands on there. Bespoke allows users to create multiple independent cells (e.g., Figure 5 and Figure 6).

In addition, Bespoke detects if the same argument values are being passed into commands in different cells. For example, the same filename, numerical thresholds, or username may be passed into multiple commands, even if their keyword argument names differ. For example, in one command the user might specify their username as `-u alice` and in another, `--username=alice`, but both get detected as the same value `alice`. In those cases, Bespoke asks the user whether they want to *link* or *lift* those common argument values:

- **Link**: When the user enters a linked argument value in the GUI of the first cell where it occurs, its value propagates to the corresponding GUI input elements in all other cells. The user can override those values in each individual cell.
- **Lift**: Bespoke lifts up all common arguments that the user selects into a special top-level cell at the top of the UI. This optimization can make the synthesized GUIs much more compact since the same arguments are not repeated across different cells. For instance, if a workflow has ten command cells that each requires its own username input, lifting that up to the top will generate a GUI with only one username input text box instead of ten.

Each cell has its own Run button. Just like computational notebooks, there is a Run All button that runs all cells in a series, waiting for each one to finish before starting the next one. Finally, to foster reproducibility, the user can export the current state of Bespoke as a shell script containing all command argument values that they have currently set in the GUI.

DISCUSSION: SYSTEM SCOPE AND LIMITATIONS

The main benefit of Bespoke's GUIs is that they expose only the subset of options required for a particular team's workflow. In contrast, any official GUI made by an app's manufacturer would need to include far more options, since they must accommodate a wider range of use cases. For instance, web-based GUIs for managing cloud computing services such as Microsoft Azure or Amazon AWS are notoriously hard to navigate, with dozens of hierarchically nested pages each with complex option selectors [6, 16, 42]. (One of our study participants used Bespoke to create a minimal Azure GUI.)

Bespoke GUIs are similar to shell scripts in that they connect command-line apps together in ad-hoc, personalized ways. However, unlike scripting, Bespoke exposes arguments in a GUI and allows users to view input/output files inline to facilitate interactive exploration. Each cell in Bespoke runs independently and asynchronously as users tinker with options, unlike scripts which are text-based and non-interactive. Also, once the user is happy with argument values they have tuned in a Bespoke GUI, they can export its state as a Bash script.

Bespoke's programming-by-demonstration approach is simple, which has both benefits and drawbacks. The main benefit of simplicity is *interpretability*: Each time the user runs a command, it gets added to the cell history and its synthesized GUI updates accordingly. Users can manually remove history entries to see their effects on synthesis. They can also override any GUI elements to change types or value ranges. Since Bespoke does not infer conditionals, loops, or advanced programming constructs, there is usually a simple one-to-one mapping between user demonstrations and synthesis results.

The main drawback of simplicity is limited expressiveness: Bespoke-generated GUIs are fairly low-level, corresponding one-to-one with underlying command-line arguments. It does not perform higher-level inference of user intent such as grouping sets of related arguments together into a single higher-level option. It also does not perform value inferences within strings, so it misses out on DSLs implemented within string arguments of commands such as `sed`, `awk`, or `ffmpeg`. Finally, it does not infer subcommand hierarchy: e.g., if the user runs both `git commit` and `git diff` in the same cell with various arguments, the synthesized GUI will contain arguments for *both* commit and diff, which looks confusing. We suggest users create separate cells for, say `git commit` and `git diff` to make separate GUIs for each.

Relatedly, as Figure 3 shows, Bespoke GUIs consist of basic input elements such as text boxes, range sliders, and date pickers. It cannot generate more advanced input elements involving custom widgets or multi-page interactions. Its expressive range is limited to single-page, form-based GUIs.

Another limitation of Bespoke is that it does not handle software environment differences between users' computers. A Bespoke-generated GUI is simply a graphical wrapper that calls the underlying command-line apps on the host computer. Thus, it is the user's responsibility to ensure that they have all of the requisite commands installed, or else their GUI will not work as intended. Even if those commands are installed properly, if their versions differ too much from the creator's version, then some options may not work as originally intended. Pairing Bespoke with software package managers, Docker containers [8], or VMs can alleviate these issues.

EXPLORATORY FIRST-USE STUDY OF BESPOKE

What kinds of GUIs might first-time users create with Bespoke? How would these bespoke GUIs potentially benefit their colleagues? What are some shortcomings and limitations of Bespoke, in their view? To explore these questions, we recruited six graduate students who worked extensively with command-line applications in their jobs as engineering interns, scientific researchers, and teaching assistants.

Procedure: We began each individual session with a 15-minute tutorial of Bespoke by walking the participant through its features. Then we asked them to brainstorm what possible uses Bespoke could have in their workplace. We encouraged them to think about workflows that involved multiple stages, interactive exploration, and manual inspection of outputs. Otherwise they could just write scripts to automate those tasks, so Bespoke would not be as necessary. Once

Participant	User population	# stages	# arguments used	# arguments available	lift?
P1: Azure cloud computing	system administrators	5	8	~50	yes
P2: machine learning	data scientists	5	5	~50	no
P3: lecture video to searchable PDF	students	3	4	hundreds	no
P4: integrated circuit design	electrical engineers	4	6	~100	yes
P5: FTP-style code deployment	computing students	6	3	~10	yes
P6: Minecraft gaming server	game organizers	10	15	~50	yes

Table 1: GUIs created by our user study participants, shown with the number of workflow stages in each GUI, total number of arguments used across all stages, number of available arguments in underlying command-line apps, and whether it lifted common arguments across stages into a top-level cell.

they were satisfied with their workflow idea, they used Bespoke to run the appropriate commands on their machines to synthesize and customize their GUIs. Along the way, we encouraged them to think aloud about the perceived benefits and limitations of Bespoke. Each session lasted around 1.5 hours.

Study Design Limitations: We designed this study as an open-ended exploration of the range of potential use cases for Bespoke. Thus, we gave each participant the freedom to develop a GUI that was personally meaningful to them. We did not rigorously assess user performance on a controlled set of tasks, nor did we compare Bespoke against alternative methods such as manually coding up a GUI. Even though we wanted participants to create GUIs of their own design, it was still done in a lab setting with Bespoke's creator present to guide their brainstorming; a more realistic setting would be people voluntarily using it on their own without us being present. Finally, we did not test the GUIs synthesized by Bespoke on end users to assess their benefits and limitations, especially for novices without much command-line experience. We relied only on our participants' anecdotes about what they perceived to be benefits and drawbacks of using these GUIs.

Results: Table 1 summarizes the GUIs that our participants created using Bespoke. These workflows spanned domains such as cloud server management, electronic circuit design, and machine learning. Most had around half a dozen stages and exposed only a small fraction of the dozens of available arguments in their underlying command-line apps. Note that a comprehensive GUI would show dozens or even hundreds of total available arguments, which would overwhelm users. Bespoke's strength is in allowing creators to easily expose a narrow slice of these commands' interfaces to suit their own needs. We now describe each participants' experiences:

P1: Managing Azure Cloud Computing Resources

P1 used Bespoke to create a GUI for managing cloud resources such as virtual machines (VMs) on the Microsoft Azure platform [16]. They regularly used Azure to manage VMs for .NET software development and configured it using the default web-based GUI. They mentioned that this GUI was far too complex for their use cases: They needed to navigate through many panes to find the few options they wanted to adjust and simply left the vast majority of settings at their defaults. They decided to make their own GUI by running the Azure command-line app (called `az`) within Bespoke.

They created a five-stage workflow with each stage in its own cell. In the first cell they ran the `az` command to create a resource group with a group name and datacenter location:

```
az group create --name myGroup --location EastUS
```

In the next cell they created a VM in this group (`myGroup`):

```
az vm create --resource-group myGroup --name myVM \
  --image UbuntuLTS [more options omitted for space]
```

Next they configured this VM (`myVM`) to open port 80 to accept HTTP connections so that it could become a web server:

```
az vm open-port --port 80 -g myGroup -n myVM
```

They added two more cells for attaching virtual hard disks, network cards, and other peripherals. Bespoke synthesized an initial GUI with those argument values as defaults; but as they run those same commands in the future with different values, the GUI input elements will generalize accordingly. Figure 2 shows the top portion of this synthesized GUI.

Benefits: Bespoke synthesized a compact GUI and lifted up arguments that were common across multiple stages. Here it recognized that `myGroup` and `myVM` were entered in multiple stages, so it lifted them up to a top-level cell where users can enter their values once. P1 added Markdown documentation to explain each cell and added links to Azure reference webpages. They mentioned that such a streamlined GUI could serve as a quick-start tutorial for novices getting started with Azure and would be less intimidating than using either the command-line interface or the default web GUI.

Limitations: P1 observed that Bespoke still produced *outputs* in plain text on the terminal since it runs the underlying command-line apps. They wished Bespoke could parse those text outputs to display them in a visual way. They also wanted to use parts of those outputs as GUI input elements of downstream workflow stages. For instance, if a command listed a set of available datacenter locations, they wanted to use those as selectable options for the datacenter input field.

P2: Prototyping Machine Learning with a Bespoke GUI

In early 2019 Uber released Ludwig [15], a command-line app that allows users to train, test, and visualize a variety of machine learning (ML) models without writing any code. To use it, they pass in training/test data as .csv files and specify the type of model and parameter settings using a configuration file. For example, to train a model, they can run:

```
ludwig train --data_csv train.csv \
  --model_definition_file model.yaml
```

There is currently no official GUI for Ludwig, so the only way that people can use it is via this command-line interface, with has over 50 arguments, each with many possible values. P2 used Bespoke to synthesize a multi-stage GUI that uses

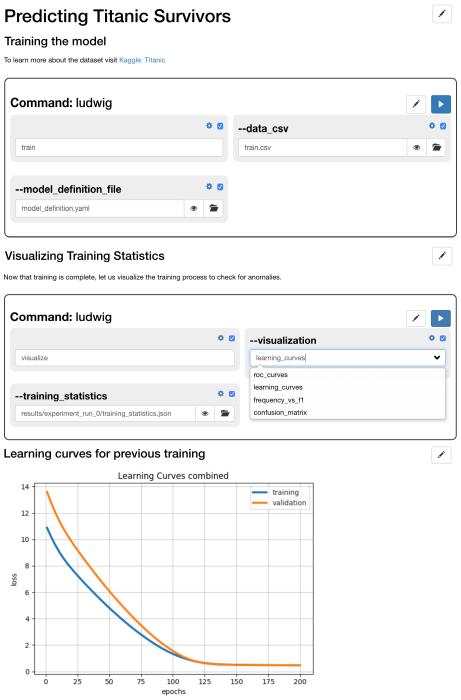

Figure 5: P2's GUI for training a machine learning model to predict Titanic survivors [25] and showing debugging visualizations in real time.

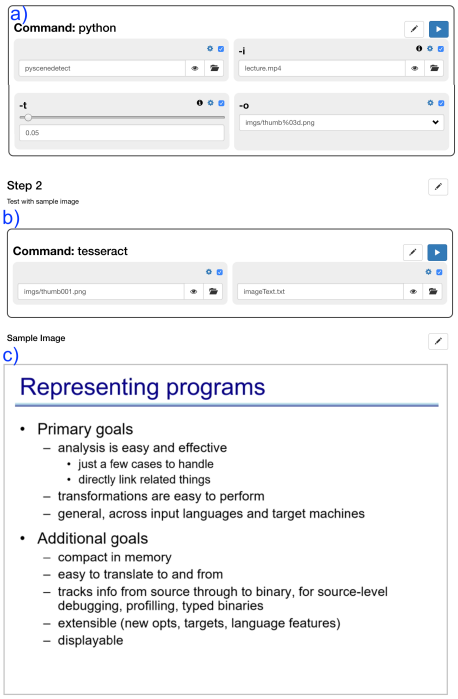

Figure 6: Excerpt from P3's GUI for processing lecture videos, showing a) scene detection, b) OCR, and c) visual previews of extracted frames.

Ludwig commands to: 1) train a binary classifier model on an input data set, 2) test the model on new data, and 3) create debugging visualizations such as learning and loss curves.

Benefits: Bespoke allowed P2 to quickly explore hyperparameter settings as they tuned their model. They can do so both by adjusting GUI input elements and by editing the configuration file (e.g., `model.yaml`) inline within each cell. As they tune their model, running the GUI synthesized for the `ludwig visualize` command lets them see the resulting debugging visualizations directly in that cell since Ludwig produces image files as output (Figure 5). Since Ludwig can work with many different types of models, P2 mentioned that a power user could make a specialized GUI for each specific model type and distribute it to novice users to run on their own input data without having to learn command-line tools.

Limitations: Bespoke's GUI input elements are fairly generic (Figure 3). P2 would have liked the ability to create custom input widgets for their ML workflows, such as one for interactively labeling input image data via direct manipulation.

P3: Converting Lecture Videos to Searchable PDFs

P3 often watches lecture videos on YouTube, which consist of lecture slides with instructor narration. They wanted to search for specific text within those video. Thus, they used Bespoke to create a three-stage GUI that lets them extract images of text-heavy slides from lecture videos, run OCR (optical character recognition) on those slide images to extract their text, and produce a PDF containing the slide images and searchable text. This workflow demonstrates chaining three different command-line apps into a single GUI: 1) PySceneDetect [24] for detecting scene changes in videos and splitting clips based on scenes (e.g., discrete slide transitions in these lecture videos), 2) FFmpeg for postprocessing videos and extracting still frames from them, 3) Tesseract for performing OCR and producing the final searchable PDF. The resulting GUI exposed only four input arguments out of the hundreds available for those complex command-line apps.

Benefits: The main benefit of a Bespoke-generated GUI over writing a command-line script for this kind of workflow is that scene detection and OCR both involve a lot of parameter tuning followed by manual inspection of outputs to determine accuracy. Each lecture video will likely require different settings to extract its text optimally. Bespoke allows users to tune parameters and see the split video clips directly in the GUI itself. It also allows them to tune OCR settings and see the recognized text inline in the cell. Figure 6 shows a portion of the GUI that renders extracted still frames from the video.

Limitations: Some commands take strings with syntactical structure, such as `-vf "select=gt(scene,0.05)"` for requesting scene detection with an app-specific threshold value of >0.05. Bespoke cannot parse those strings, so it renders them in the GUI as text input fields. A more advanced inference system could infer grammars for string arguments to present them as more structured input elements.

P4: Designing Integrated Circuits

P4 is an electrical engineering graduate student who works in integrated circuit design for SoC (system on chip) devices. They collaborate with industry partners to use command-line tools such as Synopsys PrimeTime [22] to perform layout prototyping, testing, and performance analysis of such circuits. These tools are complex, with instruction manuals up to thousands of pages long, and tend not to come with GUIs.

The GUI that P4 created with Bespoke used the Synopsys PrimeTime tool suite to perform static timing analysis, with cells for users to add buffers to sets of circuit pins and then test for different types of timing violations. Depending on the results of these tests, they may have to run additional diagnostics to assess failure severity and then perform different tests. These tasks cannot be fully automated by a script since each requires manual inspection of outputs and iterative tuning of parameters. With a Bespoke-produced GUI, engineers can see the output images and text files in the cell as they are tuning parameters. These tasks also sometimes need to be run in different orders, which is better supported by a GUI than by fixed command-line scripts.

Benefits: P4 noted that Bespoke would be a good match for this domain since electrical engineers usually do not have the software development skills to manually code up such GUIs. Each individual team also has idiosyncratic needs and navigates those tools in their own ways. Thus, a senior engineer could use Bespoke to create a GUI for their team's workflows and distribute it to junior team members.

Limitations: Users can manually override any of Bespoke's suggested values in the GUI. However, P4 wanted to be able to specify hard limits on allowed argument value ranges, since running PrimeTime with out-of-range values can risk damaging the chip hardware.

P5: Deploying Student Code with a Bespoke FTP Client

P5 is a teaching assistant for an introductory operating systems course where students need to run their code on school servers with a specialized set of tools installed on them. However, many students were not yet comfortable with Linux command-line environments, so they found it cumbersome to ssh into the server, edit code in terminal-based editors such as Vim or Emacs, and run that code from the command line. They much rather preferred coding with an IDE on their own computers. To facilitate this workflow, P5 used Bespoke to create a GUI that lets students 1) select files to copy to school servers using `scp`, 2) log into a selected server with `ssh`, 3) run commands on that server to compile and test their code, 4) run `scp` again to copy the output files of their tests back to their own computer for inspection. In essence, they created a bespoke FTP client for students in this course.

Benefits: This GUI demonstrated how, even though Bespoke is a local desktop app, it can be used to control remote servers via a graphical wrapper around ssh. It gives students the convenience of working in their own local IDE but running code remotely, without needing to alter existing IDEs. Bespoke also lifted up common string arguments such as the student's username, password, and home directory into a top-level cell to avoid repetition. (P5 did not mention any limitations.)

P6: Administering a Minecraft Game Server

P6 is a graduate student who uses their own server to run private instances of the popular Minecraft [17] multiplayer game for their friends. They currently administer this server using the free command-line tool MSM [19] (Minecraft Server Manager). Commercial server administration GUIs do exist, but they are complex and costly (over $100 per year [18]). Thus, P6 decided to use Bespoke to create a server administration interface, with one cell for each type of task such as creating game instances, adding worlds, players, and other entities to them, adjusting permissions, and deleting games.

Benefits: This GUI is different from the others in our study since it is not a step-by-step workflow. Each cell is independent and can be run any time with the requested settings whenever the user wants to perform a specific action, such as adding a world to the game. P6 noted that such an interface could be useful for hobbyists who do not need the full power of an expensive production-grade server administration GUI.

Limitations: P6 was the only one to use *link* but mentioned that it was confusing since it caused values to change without their intent. They also said that a benefit of professional Minecraft administration tools is that they have real-time dashboard visualizations of server state, which Bespoke does not provide. Also, since Bespoke cells do not know about one another, sometimes option values go stale as the user changes the state of the server. For instance, if the user deletes a game from one cell, then its settings might still show up as selectable option values in other cells. A more advanced system would link inter-cell values in more sophisticated ways than what Bespoke currently does with exact string matches.

CONCLUSION

Bespoke is an interactive system that synthesizes form-based workflow GUIs by observing user demonstrations of command-line apps. It aims to bring some long-standing benefits of GUIs (discoverability, visibility, recognition over recall) to the millions of command-line apps that now pervade many technical domains. Most of these apps are in the long tail where GUIs will never get created for them, and the few official GUIs that do exist often turn into overly-complex monoliths since they must serve a broad user base. The key insight that inspired Bespoke is that each individual or organization has idiosyncratic technical needs, which they can easily demonstrate by running selected command-line apps.

Zooming out, we learned three sets of lessons from building Bespoke that generalize to future systems: 1) *Demonstration bootstraps declarative*: Users may not know upfront what they want to write in declarative specs (e.g., for constructing GUIs), so it can be easier for them to first imperfectly demonstrate what they want. Once they see an initial sketch, that bootstraps the process of fine-tuning it using more precise declarative means. 2) *Multiple linked representations empower user choice*: Our study participants often flipped back-and-forth between command-line and GUI versions of cells while developing their notebooks. This helped them better understand and debug how the synthesis process worked. They ended up preferring command-line for simpler actions and GUIs for more nuanced ones, especially those involving parameter tuning or visual outputs. 3) *Narrative makes tools feel personal*: P2, P4, and P5 quickly generated several GUI variants for their command-line tools of choice, each one telling a specialized narrative that they preferred over making one monolithic general-purpose GUI. Letting users easily make niche tools by writing personal narratives can potentially make those tools feel more like their own.

ACKNOWLEDGMENTS

Thanks to Imran Haque for helping with the computational biology motivating example and Xiong Zhang for feedback and BibTeX wizardry.

REFERENCES

[1] 2018. Beyond Interactive: Notebook Innovation at Netflix. https://medium.com/netflix-techblog/notebook-innovation-591ee3221233. (2018). Accessed: 2019-07-14.

[2] 2018. file command. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/file.html. (2018). The Open Group Base Specifications Issue 7, 2018 edition.

[3] 2018. getopt command. http://pubs.opengroup.org/onlinepubs/9699919799/functions/getopt.html. (2018). The Open Group Base Specifications Issue 7, 2018 edition.

[4] 2018. POSIX Utility Conventions. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html. (2018). The Open Group Base Specifications Issue 7, 2018 edition.

[5] 2019. argparse — Parser for command-line options, arguments and sub-commands. https://docs.python.org/3.7/library/argparse.html. (2019). Accessed: 2019-04-02.

[6] 2019. AWS Management Console. https://aws.amazon.com/console/. (2019). Accessed: 2019-04-02.

[7] 2019. click: Python package for creating beautiful command line interfaces. https://click.palletsprojects.com/en/7.x/. (2019). Accessed: 2019-04-02.

[8] 2019a. Docker: Shape Your Digital Future. https://www.docker.com/. (2019). Accessed: 2019-04-02.

[9] 2019b. docopt: Command-line interface description language. http://docopt.org/. (2019). Accessed: 2019-04-02.

[10] 2019. ELECTRON: Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/. (2019). Accessed: 2019-04-02.

[11] 2019. ffmpeg - replace audio in video. https://superuser.com/questions/1137612/ffmpeg-replace-audio-in-video. (2019). Accessed: 2019-04-02.

[12] 2019. FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/. (2019). Accessed: 2019-04-02.

[13] 2019. git –local-branching-on-the-cheap. https://git-scm.com/. (2019). Accessed: 2019-04-02.

[14] 2019. ImageMagick. https://www.imagemagick.org/. (2019). Accessed: 2019-04-02.

[15] 2019. Introducing Ludwig, a Code-Free Deep Learning Toolbox. https://eng.uber.com/introducing-ludwig/. (2019). Accessed: 2019-04-02.

[16] 2019. Microsoft Azure portal. https://azure.microsoft.com/en-us/features/azure-portal/. (2019). Accessed: 2019-04-02.

[17] 2019. Minecraft. https://www.minecraft.net. (2019). Accessed: 2019-04-02.

[18] 2019. Minecraft Server Hosting Pricing. https://apexminecrafthosting.com/pricing/. (2019). Accessed: 2019-04-02.

[19] 2019. Minecraft Server Manager. http://msmhq.com/. (2019). Accessed: 2019-04-02.

[20] 2019. Moment.js: Parse, validate, manipulate, and display dates and times in JavaScript. https://momentjs.com/. (2019). Accessed: 2019-04-02.

[21] 2019. NPM. https://npmjs.com/. (2019). Accessed: 2019-04-02.

[22] 2019. PrimeTime Static Timing Analysis. https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html. (2019). Accessed: 2019-04-02.

[23] 2019. Project Jupyter. http://jupyter.org/. (2019). Accessed: 2019-04-02.

[24] 2019. PySceneDetect. https://pyscenedetect.readthedocs.io/en/latest/. (2019). Accessed: 2019-04-02.

[25] 2019. Titanic: Machine Learning from Disaster. https://www.kaggle.com/c/titanic/. (2019). Accessed: 2019-04-02.

[26] 2019. webpack. https://webpack.js.org/. (2019). Accessed: 2019-04-02.

[27] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3 (1990), 403 – 410. DOI: http://dx.doi.org/https://doi.org/10.1016/S0022-2836(05)80360-2

[28] Alan F Blackwell. 2001. SWYN: A visual representation for regular expressions. In *Your wish is my command*. Elsevier, 245–270.

[29] Raluca Budiu and Nielsen Norman Group. 2019. Memory Recognition and Recall in User Interfaces. https://www.nngroup.com/articles/recognition-and-recall/. (2019). Accessed: 2019-04-02.

[30] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 963–975. DOI: **http://dx.doi.org/10.1145/3242587.3242661**

[31] Mendel Cooper. 2014. *Advanced Bash-Scripting Guide*.

[32] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky (Eds.). 1993. *Watch What I Do: Programming by Demonstration*. The MIT Press.

[33] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 582–592. DOI: **http://dx.doi.org/10.1145/3106237.3106241**

[34] Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: Automatically Generating User Interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces (IUI '04)*. ACM, New York, NY, USA, 93–100. DOI: **http://dx.doi.org/10.1145/964442.964461**

[35] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. ACM, New York, NY, USA, 231–240. DOI: **http://dx.doi.org/10.1145/1294211.1294253**

[36] John Gruber. 2019. Markdown. **https://daringfireball.net/projects/markdown/**. (2019). Accessed: 2019-04-02.

[37] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. DOI: **http://dx.doi.org/10.1145/1926385.1926423**

[38] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. 2011. Proactive Wrangling: Mixed-initiative End-user Programming of Data Transformation Scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 65–74. DOI: **http://dx.doi.org/10.1145/2047196.2047205**

[39] Daniel Conrad Halbert. 1984. *Programming by example*. Ph.D. Dissertation. University of California, Berkeley.

[40] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 317–328. DOI: **http://dx.doi.org/10.1145/1993498.1993536**

[41] Andrew Head, Codanda Appachu, Marti A. Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 3–12. DOI: **http://dx.doi.org/10.1109/VLHCC.2015.7356972**

[42] Jeremy Howard. 2019. fastec2: AWS computer management for regular folks. **https://www.fast.ai/2019/02/15/fastec2/**. (2019). Accessed: 2019-04-02.

[43] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*. ACM, New York, NY, USA, 3363–3372. DOI: **http://dx.doi.org/10.1145/1978942.1979444**

[44] Donald E. Knuth. 1984. Literate programming. *Comput. J.* 27 (1984), 97–111.

[45] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. DOI: **http://dx.doi.org/10.1145/1922649.1922658**

[46] David Kurlander and Steven Feiner. 1992. A History-based Macro by Example System. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology (UIST '92)*. ACM, New York, NY, USA, 99–106. DOI: **http://dx.doi.org/10.1145/142621.142633**

[47] David Joshua Kurlander. 1993. *Graphical Editing by Example*. Ph.D. Dissertation. New York, NY, USA. Advisor(s) Feiner, Steven. UMI Order No. GAX94-12791.

[48] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (01 Oct 2003), 111–156. DOI: **http://dx.doi.org/10.1023/A:1025671410623**

[49] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration.. In *ICML*. 527–534.

[50] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553. DOI:http://dx.doi.org/10.1145/2594291.2594333

[51] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 1719–1728. DOI: http://dx.doi.org/10.1145/1357054.1357323

[52] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 723–732. DOI: http://dx.doi.org/10.1145/1753326.1753432

[53] Henry Lieberman. 1994. A User Interface for Knowledge Acquisition from Video. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1) (AAAI '94)*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 527–534. http://dl.acm.org/citation.cfm?id=199288.199319

[54] Henry Lieberman (Ed.). 2000. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann.

[55] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces (IUI '09)*. ACM, New York, NY, USA, 97–106. DOI: http://dx.doi.org/10.1145/1502650.1502667

[56] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. *Program synthesis from natural language using recurrent neural networks*. Technical Report UW-CSE-17-03-01. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.

[57] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. Miyazaki, Japan.

[58] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. ACM, New York, NY, USA, 943–946. DOI:http://dx.doi.org/10.1145/1240624.1240767

[59] Marcel Martin. 2011. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal* 17, 1 (2011), 10–12. DOI: http://dx.doi.org/10.14806/ej.17.1.200

[60] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. 1989. Metamouse: Specifying Graphical Procedures by Example. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '89)*. ACM, New York, NY, USA, 127–136. DOI: http://dx.doi.org/10.1145/74333.74346

[61] Robert C. Miller. 2002. *Lightweight structure in text*. Ph.D. Dissertation.

[62] Robert C. Miller, Victoria H. Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and mc schraefel. 2008. Inky: A Sloppy Command Line for the Web with Rich Visual Feedback. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology (UIST '08)*. ACM, New York, NY, USA, 131–140. DOI: http://dx.doi.org/10.1145/1449715.1449737

[63] Robert C. Miller and Brad A. Myers. 2000. Integrating a Command Shell into a Web Browser. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267724.1267739

[64] Dan H. Mo and Ian H. Witten. 1992. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology* 11, 1 (1992), 32–45. DOI: http://dx.doi.org/10.1080/01449299208924317

[65] Francesmary Modugno and Brad A. Myers. 1993. Graphical Representation and Feedback in a PBD System. In *Watch What I Do*. The MIT Press, Chapter 20.

[66] Francesmary Modugno and Brad A. Myers. 1994. *Pursuit: Visual Programming in a Visual Domain*. Technical Report. Pittsburgh, PA, USA.

[67] Brad A. Myers. 1990. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Trans. Program. Lang. Syst.* 12, 2 (April 1990), 143–177. DOI: http://dx.doi.org/10.1145/78942.78943

[68] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. 1990. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer* 23, 11 (Nov 1990), 71–85. DOI:http://dx.doi.org/10.1109/2.60882

[69] Jeffrey Nichols and Tessa Lau. 2008. Mobilization by Demonstration: Using Traces to Re-author Existing Web Sites. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08)*. ACM, New York, NY, USA, 149–158. DOI: **http://dx.doi.org/10.1145/1378773.1378793**

[70] Jeffrey Nichols, Brad A. Myers, and Brandon Rothrock. 2006a. UNIFORM: Automatically Generating Consistent Remote Control User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. ACM, New York, NY, USA, 611–620. DOI: **http://dx.doi.org/10.1145/1124772.1124865**

[71] Jeffrey Nichols, Brandon Rothrock, Duen Horng Chau, and Brad A. Myers. 2006b. Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 279–288. DOI: **http://dx.doi.org/10.1145/1166253.1166298**

[72] Dan Poirier. 2019. Dan's Cheat Sheets. **https://cheat.readthedocs.io/en/latest/**. (2019). Accessed: 2019-04-02.

[73] Atsushi Sugiura and Yoshiyuki Koseki. 1998. Internet Scrapbook: Automating Web Browsing Tasks by Demonstration. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '98)*. ACM, New York, NY, USA, 9–18. DOI: **http://dx.doi.org/10.1145/288392.288395**

[74] Andrew J. Werth and Brad A. Myers. 1993. Tourmaline (Abstract): Macrostyles by Example. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. ACM, New York, NY, USA, 532–. DOI: **http://dx.doi.org/10.1145/169059.169532**

[75] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A Colorful Approach to Text Processing by Example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 495–504. DOI: **http://dx.doi.org/10.1145/2501988.2502040**