Improving Data Scientist Efficiency with Provenance

Anonymous Author(s)

ABSTRACT

Data scientists frequently analyze data by writing scripts. We conducted a contextual inquiry of interdisciplinary researchers, which revealed that parameter tuning is a highly iterative process and that debugging is time-consuming. As analysis scripts evolve and become more complex, analysts have difficulty conceptualizing their workflow. In particular, after editing a script, it becomes difficult to determine precisely which code blocks depend on the edit. Consequently, scientists frequently re-run entire scripts instead of re-running only the necessary parts. We present ProvBuild, a tool that leverages language-level provenance to streamline the debugging process by reducing programmer cognitive load and decreasing subsequent runtimes, leading to an overall reduction in elapsed debugging time. ProvBuild uses provenance to track dependencies in a script. When an analyst debugs a script, ProvBuild generates a simplified script that contains only the information necessary to debug a particular problem. We demonstrate that debugging the simplified script lowers a programmer's cognitive load and permits faster re-execution when testing changes. The combination of reduced cognitive load and shorter runtime reduces the time necessary to debug a script. We quantitatively and qualitatively show that even though ProvBuild introduces overhead during a script's first execution, it is a more efficient way for users to debug and tune complex workflows. ProvBuild demonstrates a novel use of language-level provenance, in which it is used to proactively improve programmer productively rather than merely providing a way to retroactively gain insight into a body of code.

CCS CONCEPTS

• **Software and its engineering** → *Software development techniques.*

KEYWORDS

Provenance, incremental execution, dependency tracking, data analysis

ACM Reference Format:

Anonymous Author(s). 2020. Improving Data Scientist Efficiency with Provenance. In *ICSE '20: International Conference on Software Engineering, May 23–29, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

ICSE '20, May 23–29, 2020, Seoul, South Korea

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Researchers across a wide range of disciplines routinely parse, transform and process data by writing data analysis scripts. Scripts are a convenient and flexible way for data scientists to decompose a data processing procedure into steps, including preprocessing data input, training models, tuning arguments or parameters, adding new analysis functions, and propagating changes through to other data. By some estimates, millions of people write scripts to conduct data analysis tasks. However, only a small portion are professional software engineers [44].

It is standard for researchers to arrange their scripts and data into a pipeline [3], which typically consists of reading data from more than one input, analyzing and ingesting data with multiple processing steps and producing one or more outputs. We conducted a contextual inquiry with five research scientists at a large research university to understand common data processing procedures and the pain points of analysis pipeline development. While each participant struggled with a unique set of challenges, a few problems were common to all. The participants all used an iterative process based on editing, executing, and evaluating. In particular, researchers repeatedly changed parameters and reran scripts until they arrived at "good" parameter settings. Although in theory each edit required re-executing only dependent portions of the analysis pipeline, in practice, researchers defaulted to rerunning the entire scripts, because it was not obvious how to rerun only the necessary parts. This procedure was both time-consuming and cognitively loaded.

Rerunning a pipeline after modifying a script technically requires rerunning only the dependent (downstream) components. However, identifying these dependencies requires reasoning about the entire workflow, which is complex and inconvenient. Consequently, researchers usually rerun entire workflows after each change. As a result, each iteration takes more time than is strictly necessary. Thus, tuning parameters and debugging can take hours. A tool that optimizes this process has the potential to increase researcher efficiency.

To address this inefficiency, we developed ProvBuild, a data analysis environment that uses change impact analysis [5] to improve the iterative editing process in script-based workflow pipelines by harnessing a script's *provenance*. In particular, we use language-level provenance [12], which records information about every line of code executed by a script, including variable names, variable values, function definitions, function calls and their parameters, and the relationships among all these objects [34]. ProvBuild demonstrates a novel use of such provenance. Traditionally, provenance tools have been used for visualizing workflows (e.g., [7]) explaining the results of relational queries [9, 10], or recording system behavior [33]. In contrast, ProvBuild uses provenance to improve programmer productivity. ProvBuild obtains provenance using noWorkflow [40], a Python-based provenance capture tool. Using the provenance information, ProvBuild identifies dependencies between inputs, program statements, functions, variables, and outputs, allowing it to precisely identify the sections of a script that

must be re-executed to correctly update results after a modification. Provbuild then generates a customized script, the *ProvScript*, that contains only those sections affected by a modification. We hypothesize that this streamlined script allows users to reason more easily and quickly about the consequences of their edits incurring less cognitive load and allowing users to complete their job more quickly. We evaluate this hypothesis in multiple ways.

First, we evaluated ProvBuild in a controlled laboratory experiment. Twenty-one participants performed a series of debugging tasks and evaluated the utility of ProvBuild through an online survey. Next, we ran benchmarks to quantify how much time ProvBuild saves during script re-execution. Finally, we evaluated ProvBuild in another user study. In this real-world deployment study, we gave 12 participants access to ProvBuild for a week and used surveys to assess ProvBuild's utility. We asked participants how and when they chose to use ProvBuild in their daily work.

The contributions of this paper are:

- ProvBuild demonstrates a novel use of language-level provenance, in which it is used proactively to improve programmer productively rather than merely providing a way to retroactively gain insight into a body of code.
- A quantitative experiment demonstrating that ProvBuild shortens re-execution time using stored provenance.
- A controlled lab study demonstrating that users prefer programming with ProvBuild to programming without it, that they complete programming tasks more quickly, and that ProvBuild reduces their cognitive load.
- A real-world deployment study where users explained that ProvBuild saved them time, helped them understand their workflow, and provided more immediate results.

Section 2 describes our contextual inquiry that leads to the development of Provbuild. Section 3 presents ProvBuild's design and implementation. Sections 4-7 describe and report the results of our various evaluations. Discussion, related work and conclusions follow in Sections 8-10.

2 PROBLEM FORMATION

ProvBuild is the result of a contextual inquiry into how researchers interact with their data. Using the contextual inquiry method [6], we conducted a field study with five researchers at a large research university. The researchers' areas of expertise included applied mathematics, computer science, geography, applied physics, and clinical biology.

We began by interviewing each participant about the specifics of their data analysis task and the steps they take in performing that task. Then, two researchers observed participants with minimal interference as they demonstrated their normal analysis. We took notes on how they executed their analysis (e.g., by typing commands to an interpreter or running a script), what they did when they encountered a surprising or unexpected result, and how they evaluated changes they made. We also asked them explicitly to verbally express any frustrations with their process; afterward we asked them what sorts of tools might reduce that frustration.

After comparing the individual researchers' notes, we drew two main conclusions. First, users spent significant time re-executing

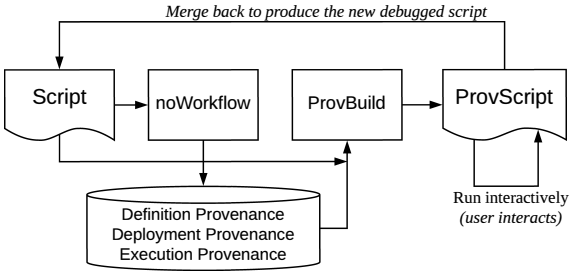

Figure 1: Architecture of ProvBuild.

scripts during the data analysis process. Three participants mentioned that they had to run scripts multiple times to identify appropriate patterns, engineer features, and train models. Some also stated that it required a great deal of effort to produce a desired output. They reported that tuning model parameters was time-consuming, because the process involved manually re-running whole scripts for each parameter combination. Although changes in the middle of an analysis pipeline do not require executing statements prior to the change, users report that it was difficult to identify which parts of a script were affected by a change; doing so required too much effort. The researchers tended to treat the data analysis pipeline as a discrete, indivisible unit. After editing a script, our subjects all simply reran their entire pipeline.

Second, researchers place a high premium on ease of adoption when considering new tools. During interviews, participants expressed interest in a tool that would reduce inefficiencies surrounding data processing. However, this interest was qualified by hesitations about the overhead of learning and adopting a new tool. We concluded that familiarity and usability must be first class considerations in addressing the re-execution inefficiency.

3 PROVBUILD: PIPELINE DEBUGGING USING PROVENANCE

We designed and developed ProvBuild to address the challenges we discovered in our contextual inquiry. ProvBuild leverages *language-level provenance* collected by noWorkflow [34], which uses program slicing [48] to record every action taken in a script and the dependencies between objects, such as variables, values, and function. Using these data dependencies, when a user modifies a script, ProvBuild identifies precisely which script statements must be re-executed to produce new results. ProvBuild constructs a shortened script, called a *ProvScript*, that reduces re-execution time and makes it easier to reason about the effect of a user's modification. ProvBuild consists of a backend engine (Fig.1, Sec.3.3) and a user interface (Fig.3, Sec.3.5).

3.1 The User View

When a user begins tuning or debugging a script, they upload their script to ProvBuild. Using the frontend GUI, described in Section 3.5, they select the function, output, or variable they are debugging/tuning. (For the rest of the section, we refer to this as the *target*.) ProvBuild then generates a *ProvScript*, which contains only the code necessary to produce the target. The example shown in Figure 2 illustrates this procedure. Imagine that a user wishes to

```
# original script                 # ProvScript
[1]  def foo(var):                [1] a = 10
[2]    return var                 [2] b = 20
[3]  def bar(var):                [3] d = 3
[4]    return var*3               [4] f = 204
[5]  def baz(var1,var2):          [5] def baz(var1,var2):
[6]    return var1+var2                          #####L5
[7]  a = foo(10)                  [6]   return var1+var2
[8]  b = foo(20)                            #####L6
[9]  c = baz(a,b)                 [7] c = baz(a,b) #####L9
[10] if a % 2 == 0:               [8] e = b + c #####L15
[11]   d = foo(3)
[12] else:
[13]   d = bar(2)
[14] e = b + c
[15] f = b * 10
[16] for i in range(1,5):
[17]   f += foo(1)
```

Figure 2: Comparison between an original script and the *ProvScript*.

debug the baz function. The baz function affects lines 9 and 15 only, since c contains the return value of baz (line 9), which is then used to compute e in line 15. ProvBuild treats variables a, b, d, and f as constants, since their values do not depend on baz and do not need to be recomputed. ProvBuild extracts the appropriate values for them from the provenance and assigns those values to the variables in the resulting *ProvScript*.

Without ProvBuild, users typically rerun the entire script after each edit; with ProvBuild, users run the automatically generated *ProvScript* after an edit. In the example, ProvBuild inserts four lines at the top of the *ProvScript* assigning the constant values of a, b, d, and f, inserts the definition of function baz, and includes the dependent lines (7 and 8). ProvBuild ignores the definition of functions foo and bar, because they are irrelevant and do not depend on bar. As such, the resulting *ProvScript* contains only the code necessary to test changes to baz. The shortened script minimizes re-execution time and isolates the code being debugged, making it easier to reason about how modifications affect their pipeline. Users need not modify their current debugging or tuning behavior, since ProvBuild hides the provenance driven optimization process behind the user interface and presents them with a simpler editing task.

3.2 Provenance Collection

Figure 1 depicts ProvBuild's high level architecture. ProvBuild captures provenance using noWorkflow [34, 40], an open source provenance collection tool for Python. noWorkflow uses a combination of static and dynamic analysis to capture three types of provenance [17]. *Definition provenance* is a record of all global variables and function definitions, calls, and arguments in a script. *Deployment provenance* includes the execution environment and library dependencies. *Execution provenance* accumulates while a script runs and can be either coarse-grain or fine-grain. Coarse-grain provenance includes information about every function invocation (the function, its arguments, and the return value), and file accesses. noWorkflow uses program slicing [48] to capture fine-grain provenance, such as control flows and variables and their dependencies.

Although ProvBuild currently works with provenance captured by noWorkflow, there is nothing in its design that precludes it from working with other language-level provenance capture systems, such as R's RDataTracker [31]. We leave development of a provenance-capture-agnostic version of ProvBuild to future work.

ProvBuild analyzes the fine-grain provenance to construct a dependency graph, which identifies the parts of the script on which a particular edit depends. It then produces the *ProvScript* by traversing the dependency graph, assigning variables concrete values where possible and computed values otherwise. We discuss this in more detail in the next section.

3.3 Dependency Exploration

ProvBuild's backend consists of two main parts: (1) dependency exploration and (2) script merging. We discuss dependency exploration here and script merging in the next section.

The key to ProvBuild lies in its ability to construct an accurate dependency graph. Some of the components of the dependency graph appear in the provenance, while others do not. We describe the different strategies employed in the following paragraphs, which address function definitions and control flow, and construct the provenance graph to explore dependency.

3.3.1 Function Definitions. There are two kinds of functions that the *ProvScript* might need: those appearing in the script (which are part of the definition provenance) and those that come from imported libraries. ProvBuild identifies the necessary subset of the the functions appearing in the script using the function invocation information in the provenance. Rather than identifying precisely which functions are needed from libraries, we retain all import statements from the original script in the *ProvScript*.

3.3.2 Control Flow. There are aspects of a script that ProvBuild needs to construct for the *ProvScript* that are not recorded in the provenance. For example, noWorkflow provenance does not fully capture conditional control flow, because provenance is an execution record, and any specific execution follows only one clause of an `if-elif-else` statement. However, ProvBuild needs to include all conditional clauses in a *ProvScript* to ensure that re-execution is correct. ProvBuild accomplishes this using static analysis of the original script. During this static analysis, it creates a pseudo function for the the entire conditional expression (i.e., including all conditions). When ProvBuild needs to include a conditional expression in the *ProvScript*, it includes the pseudo function instead of including just the single clause from the actual execution.

Consider the following. In the original script in Figure 2, a is even, so foo always executes on line 11 and bar never does. The execution provenance records the use of foo but not bar, which appears only in the definition provenance. Now, let's say that a user is interested in tuning the value of a (i.e., change the argument to foo from 10 to 9). When they re-execute, noWorkflow's execution provenance indicates that foo is the only function that depends on x. ProvBuild observes that a conditional appears in the *ProvScript* and includes the full conditional clause as shown in the following

code, so that re-execution produces the correct result, even when a is odd during a later execution.

```
# ProvScript
[1]  ... # variable b, c, e, f assignments
[2]  def foo(var): #####L1
[3]    return var #####L2
[4]  def bar(var): #####L3
[5]    return var*3 #####L4
[6]  a = foo(9) #####L7
[7]  if a % 2 == 0: #####L10
[8]    d = foo(3) #####L11
[9]  else: #####L12
[10]   d = bar(2) #####L13
```

Proactively including every possible function definition could lead to an overly complicated *ProvScript*. Instead, ProvBuild uses iterative exception handling to identify only those function definitions needed for a particular execution. ProvBuild initially relies on the execution provenance; in the example, that produces a *ProvScript* including only foo's definition. If the user makes the value of a odd and re-executes, the *ProvScript* throws a NameError exception when it encounters the invocation of bar. ProvBuild catches the exception, extracts the necessary function definition from the original script, and regenerates the *ProvScript* with the additional function. ProvBuild continues catching such exceptions, regenerating the *ProvScript* each time, until script execution completes without the NameError exception. The *ProvScript* is a superset of a program slice and our iterative dynamic trapping ensures that we are not missing necessary parts of the program.

3.3.3 Constructing the Provenance Graph. Given a target, we divide the execution provenance of a script into *upstream provenance*, everything on which the target depends, and *downstream provenance*, everything that depends on the target. If we view execution provenance as a graph, ProvBuild first collects upstream provenance by selecting the ancestors of the target and then selects the target's descendents to capture the downstream provenance. While most provenance systems avoid cycles in provenance graphs, noWorkflow allows them, and they require special attention. Consider the original script (line 15 - 17) in Figure 2. Assume that the user is interested in the derivation of the value of f. The provenance indicates that x depends on both lines 15 and 17. However, neither noWorkflow's static nor dynamic analysis identify that it also depends on the values of i that occur in line 16.

ProvBuild performs its own loop analysis to address this omission. It creates upstream provenance for variables dependent upon loop iterators. So, when ProvBuild constructs the *ProvScript* for the example above, it includes the iterator i in the upstream provenance of f.

ProvBuild also uses the dynamically collected upstream and downstream provenance to track implicit dependencies. Consider the original script (line 7 - 9). Function baz takes foo's return values a and b as parameters. This means that c depends on foo implicitly and baz explicitly. Thus, changing baz requires rerunning only line 9 and its downstream, while changing foo requires rerunning all three lines and their downstream. ProvBuild constructs the transitive closure of the dependency graph to include implicit function and variable relationships, ensuring that the resulting *ProvScript* is correct.

3.4 Script Merging

Users can edit the simpler *ProvScript* directly, and ProvBuild automatically merges those changes back into the original script. After merging, future *ProvScript*s are based on the new, merged version of the file.

To merge successfully, ProvBuild maintains a record of the differences between the original script and the *ProvScript*. We annotate each line in the *ProvScript* with comments that explicitly map the line to its corresponding line in the original file. For example, in the following *ProvScript*, ProvBuild adds comments beginning with the special ##### token, since the definition of function baz and the assignments of variable z in lines 5, 7, 8 and 9 are inherited from the original script. The assignments in lines 1 - 4 are inserted by ProvBuild, not the user, so they are not marked.

```
# ProvScript (After user edits)
[1]  a = 10
[2]  b = 20
[3]  d = 3
[4]  f = 204
[5]  def baz(var1,var2): #####L5
[6]      var1 = var1 + 1
[7]      return var1+var2 #####L6
[8]  c = baz(a,b) #####L9
[9]  #####L15
```

New lines in the *ProvScript* (e.g., line 6), contain no added markers, while deleted lines (e.g., line 9) appear as a line with the marker of the deleted line, but no code. The information provided by the markers enables ProvBuild to correctly merge changes into the original script.

These comment characters are incidental to our prototype. We used them to avoid making changes to the underlying editor. A real deployment would implement tracking in the editor, making it invisible to the user. There are standard techniques for such tracking [28, 32].

3.5 Interface Design

Our interface allows users to debug functions or variables on a simplified version of an original script and seamlessly merge those modifications back into the original script. To facilitate evaluation, the ProvBuild prototype interface supports both conventional editing (i.e., editing on the entire script) and the ProvBuild provenance-driven editing of a *ProvScript*. In either case, the user begins by selecting the mode of interaction (conventional or ProvBuild) and identifying the script with which they are working. In ProvBuild mode, the interface activates the provenance tracking backend.

ProvBuild is designed to abstract the provenance-driven incremental build process away from the user. Users interact with their scripts through the three main modules shown in Figure 3. Each module is explained in more detail below.

- **Search**: The user inputs the name of the function or variable to edit (see (1) in Figure 3). ProvBuild extracts the object's dependencies based on the stored provenance information

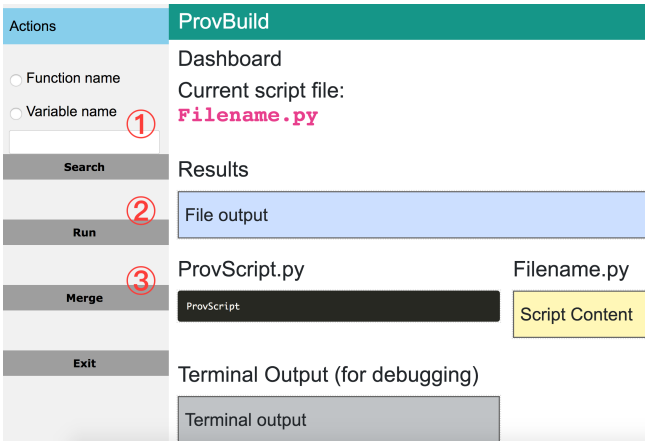

Figure 3: The ProvBuild interface features four boxes. We assume that all user scripts write to an output file. The blue box displays the contents of this output file. The yellow box displays the original script and the black box displays the *ProvScript*. The grey box displays terminal output, such as errors and print statements. The sidebar on the left provides easy access to the user commands. In ProvBuild mode, the user specifies the target, either a function or variable, to change/debbug. After editing the *ProvScript*, the user runs it, and examines the output. When the user is satisfied with the output, the user merges the changes into the original script. At this point, the yellow box updates to contain the revised script; the user can then select another object to edit.

and generates a *ProvScript* containing only code pertaining to the chosen object.

- **Execute**: Instead of running the original script, ProvBuild executes the shortened *ProvScript* for the user (see (2) in Figure 3). This reduces run time.
- **Merge**: ProvBuild allows users to easily merge edits from the *ProvScript* into the original file (see (3) in Figure 3).

When used in conventional mode, the interface is similar to common modern text editors. Users simply edit their scripts and re-execute them in their entirety.

4 RESEARCH QUESTIONS

We conducted three studies to evaluate ProvBuild's performance, effectiveness, and usability. Our goal is to answer the following four research questions (RQ):

- **RQ1:** How well can ProvBuild improve debugging efficiency in basic programming tasks?
- **RQ2:** How much overhead does noWorkflow introduce during initial script execution?
- **RQ3:** How much speedup does ProvBuild produce when re-executing a script after a modification?
- **RQ4:** In real-world settings, how do data scientists use ProvBuild in their daily work, and what benefits and challenges do they experience?

We address RQ1 using the results of a controlled experiment involving a lab setting with scripts constructed specifically for the experiment. We address RQ2 and RQ3 using quantitative measures of ProvBuild runtime and overhead. We conducted another user study involving data scientists using ProvBuild on their own analysis scripts in the wild to answer RQ4.

5 STUDY 1: CONTROLLED LABORATORY EXPERIMENT

To answer RQ1, we conducted a controlled lab study to quantify ProvBuild's ability to reduce the time to complete a task and to obtain empirical insights into real-world challenges. We asked participants to perform a series of debugging tasks with and without ProvBuild and evaluated their behavior both qualitatively and quantitatively. The study design was driven by the following hypotheses:

- **Hypothesis 1: ProvBuild will decrease task completion time.** Since the *ProvScript* is shorter than the original file and its re-execution time is shorter, users will complete the tasks more quickly.
- **Hypothesis 2: ProvBuild will decrease users' cognitive load.** Cognitive load is the total amount of mental effort being used in working memory. Because ProvBuild removes irrelevant code, reasoning about code in *ProvScript* will be less taxing for users.
- **Hypothesis 3: Participants show more positive subjective responses when programming with ProvBuild.** Participants will report that their experience is better when using ProvBuild than when using a standard code editor.

5.1 Participants and Apparatus

We used snowball sampling [19] to recruit participants from multiple fields including computer science, electrical engineering, applied physics, and applied mathematics. 21 volunteers participated in the study (14 men, 7 women; 21 – 28 years old, M = 24.3, 2 undergraduate students, 11 graduate students, 8 professional data scientists). All had some experience programming in Python and had worked on at least one data analysis project.

We conducted all trials in the same room using a Macbook Pro laptop running macOS. The ProvBuild interface ran in the Google Chrome browser.

5.2 Procedure

Each experiment started with a basic demographic and technical background survey. Next, participants were given instructions on how to edit in each mode (ProvBuild and Conventional mode). They then engaged in one practice round with each tool. The practice tasks were similar to the tasks given during the main study and let the participants familiarize themselves with the interface.

Finally, each participant completed a series of four debugging tasks. These tasks were modeled after common data analysis procedures. For each task, participants were asked to minimally modify a script to a get specified desired output. The task conditions consisted of programming modes (ProvBuild and Conventional) and difficulty levels. We designed two difficulty levels of tasks and validated their difficulties (Q9) in the user study (Section 5.4.1). Easy tasks contained fewer than 100 lines of code about math calculation and matrix transformation, while hard tasks contained nearly 300 lines of code about model training and parameter tuning. The task

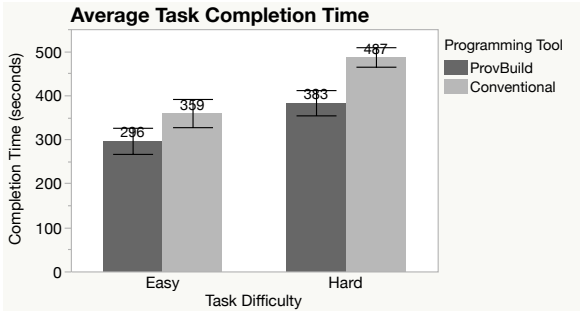

Figure 4: Average task completion time for easy and hard tasks sorted by tool.

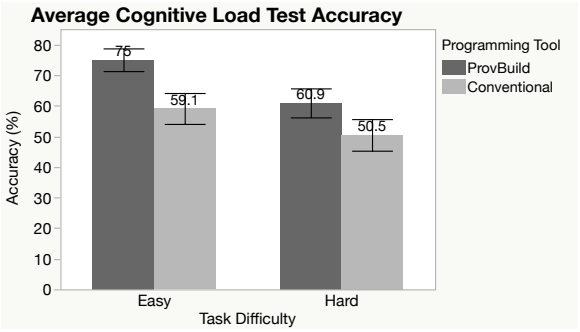

Figure 5: Average cognitive load test accuracy for easy and hard tasks sorted by tool.

orderings were counterbalanced both in the order of programming mode and difficulty level.

Participants were given ten minutes to complete each task, with instructions to complete each task as quickly and accurately as possible. We timed each task from the moment the participant began reading the script to the moment she verbally expressed completion. Most participants completed each task comfortably within the time limit. If the participant was unable to complete the task within the 10 minute cap, we recorded the result as unfinished.

In the ProvBuild programming mode, participants had the option to identify functions or variables they wanted to modify thereby generating a *ProvScript* as shown in Figure 3. After generating a *ProvScript*, participants could modify and execute that *ProvScript* instead of working with the original script. After participants obtained the desired output, they triggered the merge module to integrate changes from the *ProvScript* back into the original file. In the Conventional programming mode participants edited the original script using a text editor and checked results by executing the whole script.

To evaluate cognitive load, we examined their digital memorization behaviors [13, 47] in each task: we asked participants to memorize ten random numbers in one minute before each task. Upon task completion, participants were asked to recall the numbers. If participants were able to recall more numbers, this was indicative of lighter cognitive load during task completion. After completing each task, the participants were given a final questionnaire asking them to respond on a 7-point Likert scale. The first six questions were the NASA-TLX standard questions, which evaluate perceived workload [23, 24] and provide subjective ratings along six subscales: Mental Demands, Physical Demands, Temporal Demands, Own Performance, Effort Level and Frustration Level [22]. The remaining three questions evaluated the user's perceived self-efficacy and subjective assessment of ease of use and effectiveness. At the end of the experiment, participants were asked whether they had any feedback concerning ProvBuild. Each participant spent approximately 60 minutes completing the experiment.

5.3 Data Analysis

This was a within-subjects study with two factors: task difficulty {easy, hard}, and programming tool {Conventional, ProvBuild} and the following measures:

- **Completion time.** We measured the completion time from the moment they started reading the script to the moment they declared that they were done. For participants who were unable to complete the task within the 10 minute time limit, we recorded their time as 10 minutes[1].
- **Accuracy on cognitive load test.** We asked participants to memorize ten random numbers and recall the numbers after each task. We interpreted recall accuracy as a measure of cognitive load.
- **Self-reported subjective measures.** After each task, we asked participants six questions relating to their perceived workload and three subjective assessment questions. We computed the sum of all nine subjective responses with the same response order (lower indicates better) as the self-reported subjective measure (two items were reverse coded for analysis such that lower number always indicated a more positive response).

To guard against Type I errors due to multiple hypotheses being tested, we applied the Holm's sequentially-rejective Bonferroni procedure [25, 42] to the analyses, which introduces fewer Type II errors than the more common simple Bonferroni correction. We report both raw and adjusted p-values.

During evaluation, we first summed all nine subjective responses to assess whether participants had an overall preference for either mode (two items were reverse coded for analysis). We then conducted a statistical analysis of each question separately.

5.4 Study Results

5.4.1 Main Analyses. The main results are summarized in Table 1.

Everyone who completed a task in the allotted 10 minutes did so correctly. However, 1 of the 21 participants did not finish an easy task with either programming tool. 3 of the 21 participants did not finish a hard task with the Conventional mode while only one did not finish with ProvBuild.

We validated task difficulty to ensure that the Provbuild and Conventional tasks were comparable (Q9): there is no significant difficulty difference between the two easy tasks (averaged difficulty rating 3.23 on a 7-point Likert scale, $F(1, 20) = 0.5333$, $p = 0.4737$) and two hard tasks (averaged difficulty rating 4.07 on a 7-point Likert scale, $F(1, 20) = 2.3343$, $p = 0.1422$), respectively.

As shown in Figure 4, participants had statistically significant shorter average completion time ($F(1, 20) = 66.64$, raw $p < 0.0001$,

[1]Because more participants failed to finish in time using Conventional mode rather than ProvBuild mode, trimming the completion time to 10 minutes for incomplete tasks did not unfairly advantage ProvBuild during analysis.

Table 1: Summary of the results. All nine subjective measures were reported on a 7-point Likert scale, while self-reported preference is reported as the sum of nine responses; lower indicating higher preference. We used the Holm's sequentially-rejective Bonferroni procedure since we tested multiple simultaneous hypotheses. We report both raw and adjusted p-values. Statistically significant results are marked with an asterisk.

Hypothesis	ProvBuild	Conventional	Raw p-values	Adjusted p-values	Sig.?
H1 Completion time	339.70s	423.18s	<0.0001	<0.0003	*
H2 Accuracy on cognitive load test	68.09%	54.79%	= 0.0007	= 0.0014	*
H3 Self-reported preference	25.98	30.05	= 0.0131	= 0.0131	*

Table 2: Detailed subjective results of the NASA-TLX standard questions and subjective assessment questions. The answer were rated on a 7-point Likert scale. $*$ means the main effect for tools is statistically significant ($p < 0.05$), while $**$ means the main effect is marginally significant ($p < 0.10$).

Questions	p-values	Sig.?
1. How much mental and perceptual activity was required? *(1 = low, 7 = high)*	0.0237	*
2. How much physical activity was required? *(1 = low, 7 = high)*	0.6849	
3. How much time pressure did you feel due to the pace at which the tasks or task elements occurred? *(1 = low, 7 = high)*	0.0482	*
4. How successful were you in performing the task? How satisfied were you with your performance? *(1 = low, 7 = high)*	0.0837	**
5. How hard did you have to work (mentally and physically) to accomplish your level of performance? *(1 = easy, 7 = hard)*	0.1049	
6. How irritated, stressed, and annoyed versus content, relaxed, and complacent did you feel during the task? *(1 = relaxed, 7 = stressed)*	0.0798	**
7. How confident were you about your answer(s)? *(1 = low, 7 = high)*	0.2308	
8. How hard (irritating) was it to use the tool? *(1 = easy, 7 = hard)*	0.0101	*
9. How hard was it for you to accomplish this task? *(1 = easy, 7 = hard)*	0.1003	

adjusted $p < 0.0003$) using ProvBuild ($M = 339.70$ seconds) than using the Conventional mode ($M = 423.18$ seconds). Hypothesis 1 was supported. ProvBuild decreased average task completion time and significantly improved programming efficiency in both task difficulty levels.

Participants had greater number recall accuracy after programming with ProvBuild ($M = 68.09\%$) than after using the Conventional mode ($M = 54.79\%$) as shown in Figure 5. This main effect was statistically significant ($F(1, 20) = 16.00$, raw $p = 0.0007$, adjusted $p = 0.0014$). These results support Hypothesis 2 and indicate that ProvBuild was able to lighten cognitive load. In other words, ProvBuild is less taxing on the user's mental resources.

Participants also reported being more satisfied overall after programming with ProvBuild ($M = 25.97$, lower is better) than with the Conventional mode ($M = 30.05$). The difference was statistically significant ($F(1, 20) = 7.42$, raw $p = 0.0131$, adjusted $p = 0.0131$). Hypothesis 3 was supported by our subjective data analysis: participants overall preferred ProvBuild over conventional processing methods.

Also, those who preferred Conventional mode commonly stated their preference was due to the fact that they were more familiar with the Conventional programming method than with ProvBuild. 5 of the 21 participants indicated that they felt it was more convenient to program with ProvBuild in the latter tasks as they became more familiar with the interface.

5.4.2 Additional Analyses. Table 2 presents the detailed subjective results. These self-reported results demonstrate that the participants experienced less mental effort (Q1, $F(1, 20) = 3.3727$, $p = 0.0237$) and felt less time pressure (Q3, $F(1, 20) = 4.4268$, $p = 0.0482$). Participants also felt that the tasks were significantly easier and less irritating to complete with ProvBuild (Q8, $F(1, 20) = 8.0769$, $p = 0.0101$). These results were statistically significant.

The measure also indicates that ProvBuild users experience relatively lower levels of irritation regarding their code (Q6, $F(1, 20) = 3.4069$, $p = 0.0798$) and higher levels of success in task completion (Q4, $F(1, 20) = 3.3140$, $p = 0.0837$). These differences are marginally significant.

6 STUDY 2: PERFORMANCE EVALUATION

To answer our performance-related research questions (RQ2 and RQ3), we ran benchmarks on a Ubuntu 18.04 LTS computer with Core i5 3.5GHz CPUs 16GB of memory, running with Python 2.7 and SQLite 3.15 (noWorkflow's storage engine). noWorkflow introduces three sources of overhead. It's possible that a different provenance capture mechanism would produce lower overheads, but analysis of different capture mechanisms is outside the scope of this study. First, noWorkflow initializes its data storage prior to script execution in a preprocessing step. Second, during initial script execution for which we collect provenance, noWorkflow's dynamic analysis introduces overhead. Third, noWorkflow writes its provenance data to a SQLite database during execution; SQLite is not a terribly performant database. We report these overheads in Table 3. Note that we ran an unmodified version of noWorkflow in this study. As ProvBuild does not depend on all the functionality of noWorkflow, it's likely that a streamlined implementation of noWorkflow could reduce the overhead.

Table 3: Comparison of running times and lines of code of data analysis script executed without ProvBuild and with it, averaged over 5 runs (variances negligible)

Script #	LOC	Original Exec time	Runtimes w/ noWorkflow			Class A		Class B		Class C	
			Pre-processing	Exec time	Storage time	LOC	Re-exec time (Speedup)	LOC	Re-exec time (Speedup)	LOC	Re-exec time (Speedup)
1	137	0.19	11.12	1.57	1.06	13	0.03 (6.33x)	123	0.12 (1.58x)	90	0.07 (2.71x)
2	121	2.68	6.92	4.18	3.69	33	1.08 (2.48x)	110	2.10 (1.28x)	68	1.32 (2.03x)
3	57	4.42	15.14	124.03	6.24	12	1.05 (4.24x)	20	4.01 (1.10x)	16	2.00 (2.21x)
4	61	6.29	8.72	97.26	326.85	10	0.16 (39.31x)	29	5.81 (1.08x)	28	3.65 (1.72x)
5	543	7.25	14.06	14.64	5.88	72	1.43 (5.07x)	486	5.29 (1.37x)	182	2.15 (3.37x)
6	51	20.15	2.87	337.97	1520.11	26	11.30 (1.78x)	40	17.73 (1.14x)	37	13.21 (1.53x)
7	230	69.61	11.32	1540.81	2023.65	40	22.91 (3.04x)	178	60.47 (1.15x)	106	42.30 (1.64x)
8	854	78.73	63.47	298.88	39.45	76	14.24 (5.53x)	783	63.99 (1.23x)	576	33.40 (2.36x)
9	222	81.00	22.98	1917.23	3815.01	26	6.41 (12.64x)	184	67.30 (1.20x)	163	13.95 (5.81x)
10	102	129.12	1.89	4017.26	2914.30	92	56.61 (2.28x)	97	103.46 (1.25x)	94	82.04 (1.57x)
11	175	140.71	2.22	2835.34	2109.23	41	13.10 (10.74x)	127	118.35 (1.19x)	92	68.24 (2.06x)

6.1 Initialization Slowdown

To quantify ProvBuild's overhead (RQ1), we collected Python scripts from published work and compared script length and running times with and without ProvBuild. Dataverse is a platform for publishing data sets used in research publications [26]. We used its programmatic API [35] to obtain real-world Python scripts. We queried Harvard University's public Dataverse instance [36] for every archived data set containing Python scripts. We then downloaded the 92 published data sets including Python code. Unfortunately, many of the archived data sets were missing key files. Only 54 of the 92 contained both scripts and the accompanying data. Of those 54 scripts, only eleven ran to completion as published.

Table 3 shows the breakdown in running time. Execution time increases dramatically, in the best case, by only 56%, but in the worst case by around a factor of 30. The majority of this overhead is due to noWorkflow's dynamic provenance tracking. Writing provenance to the database also adds significant overhead (column Storage Time). Other provenance tracking solutions [31] keep provenance in main memory and write it persistently after execution; this approach seems attractive. As we will see in Section 7, users did voice concern over the initial run time, but not enough to prevent them from using ProvBuild.

6.2 Debugging Speedup

To demonstrate how ProvBuild can ultimately increase development efficiency (RQ2), we measured ProvBuild's performance after making three types of changes to each of the eleven scripts from Table 3. A *Class A* change directly alters script output, e.g., changing the format of the output. A *Class B* change alters an input file or input variable. A *Class C* change modifies the parameter of a function in the script, e.g., changing the value of a model parameter. For those eleven scripts, we randomly selected one of each type of change and measured how long it took to execute the *ProvScript* produced by ProvBuild.

The speedup inherently depends on the length of the code path following the edit. As *Class A* changes affect only the output stage of analysis, ProvBuild often generates significantly shorter scripts and produces significant speedup. For our eleven scripts, these speedups ranged from a factor of 1.78 to 39.31 (i.e., the *ProvScript* ran almost 40 times faster than the full script). Table 3 shows that the *ProvScript*s generated from *Class A* changes had 74% fewer lines of code, on average. *Class B* and *Class C* changes induce smaller speedups. The resulting *ProvScript*s retained, on average, 77% and 58% of the lines of the original script, respectively, while the speed-ups averaged 1.23X and 2.46X, respectively. ProvBuild explores the dependencies downstream of the edits. Speed-ups are smaller, because ProvBuild must retain all the downstream dependencies; the earlier in the script a modification is made, the more of the script must be retained for re-execution, producing longer re-execution times. Even in the worst case (*Class B*), we attain some speedup. In all three classes, ProvBuild is able to use stored provenance to shorten run time.

7 STUDY 3: DEPLOYMENT IN THE WILD

We conducted another user study to evaluate ProvBuild's usefulness (RQ4) and efficacy for data scientists from different domains. The study was a real-world deployment of the system in which participants could choose when and how to use the tool. We ran this field study to see if participants would choose ProvBuild in real scenarios in place of other tools available to them. We used surveys to obtain feedback from participants. Participants received no incentive to use ProvBuild; feedback was solicited only after they used it. We approached the 21 participants from the prior study, and 12 of them (8 men, 4 women) agreed to participate in this study. The details of the study were revealed only once a participant agreed to participate.

7.1 Procedure and Data Analysis

We gave participants access to ProvBuild for one week. This allowed them to explore and use the tool for Python debugging in their daily work. We intentionally gave participants complete latitude about when they used ProvBuild. At the end of each day, we asked the following survey questions to understand if and how participants chose to use ProvBuild: (1) Did you use ProvBuild today? If so, what were you trying to accomplished by using it? (2) When you used ProvBuild, what did you like about it? (3) Did you have any problems using ProvBuild? If so, please describe them. (4) Are you inclined to use ProvBuild again? Why? (5) Would you recommend ProvBuild to

others, why? These five questions were all free-response questions with no character limit requirement. We asked them daily and only required them to answer on the days that they had chosen to use ProvBuild.

7.2 Study Results

In this study, we collected 18 surveys from 12 participants. All participants chose to use ProvBuild at least once, while four participants used it more than once in a one-week period. 11 out of 12 participants indicated that they would use ProvBuild in the future (Q4). Participants reported a large number of programming scenarios in which they used ProvBuild in their daily work. One common situation was script debugging and parameter tuning. Participants used ProvBuild to debug *"simple Python scripts" (P12)* and multiple scripts with *"complicated dependencies" (P1)*, or *"finely tune parameters in code" (P9)*. Several participants focused more on writing scripts for model training, while some were working on math calculations in Python.

To analyze the collected data, we utilized qualitative data analysis methods [20]. Two researchers independently clustered the data into themes. Themes included commonly mentioned benefits and common challenges. The researchers then collaboratively reviewed the data and developed an agreed upon set of themes. The primary researcher qualitatively coded the data using these themes to assess the prevalence of the themes within the data set.

7.2.1 ProvBuild Benefits. Participants mentioned ten different benefits of using ProvBuild. The most frequent benefits, mentioned by nine participants, were that ProvBuild **saved programming time** and allowed users to **find code dependencies more easily**: *"It helps find all the dependent code pieces when you target a specific problem, which greatly saves time and reduces errors." (P1) "It explicitly tells me the dependence of certain functions and data. It runs really fast." (P3) "It speeds up calculations by storing file data within the code." (P11)* Another respondents reported *"it really saves programming time (both execution time and thinking time)" (P8).*

ProvBuild is also considered particularly beneficial for **understanding project workflow**, mentioned by six participants: *"I like that it cut down on the amount of code I have to learn and understand." (P9) "It can provide the part that I want to rerun, so I don't need to do the whole preprocessing every time." (P2) "It would help with complex programming workflow." (P7)*

ProvBuild **reduces the need for memorization** of the details in a workflow by providing shortened scripts, which makes programming and debugging easier. Five participants mentioned this benefit, which also supports Hypothesis 2 from Section 5, in real-world studies: *"I love this because it can keep the records of the old files and I don't need to remember the workflow of all my programs. The only thing I need to decide is which part of the program needs to be changed." (P6)*

Another common benefit mentioned by seven participants is its **usefulness**. It *"simplifies" (P4)* the debugging process and *"shortens" (P2)* program scripts: *"It's really easy to make changes in old scripts in order to match new models." (P6) "I think it's useful to debug programs with complicated steps." (P4)*

Three participants explicitly mentioned that ProvBuild **provides intermediate results** to facilitate programming process: *"When I do math in Python, I often need to print all variables to find the problem and I forget to do that sometimes. With ProvBuild, I can check those constants in every step and it really helps" (P8) "I like the idea of considering the intermediate results as constants. Most of the time, I only care about the follow-up calculation after those values." (P4)*

Overall, participants expressed a preference for using ProvBuild and mentioned that ProvBuild improves the debugging process mainly by reducing programming time, allowing users to find dependencies and understand their workflow more easily, reducing the need for memorization. Participants do not report any significant barriers to independent use.

7.2.2 ProvBuild Challenges. Participants also raised several concerns after their use with ProvBuild. The predominant concern stated by five participants was the **slow initial run time** relative to running without ProvBuild: *"The initial execution with ProvBuild takes longer than I expected, especially when the training files are large. However, once I finish the first run, it ultimately saves time. I would like to continue to use ProvBuild for future debugging." (P2) "I found that the initialization of ProvBuild takes a bit longer. I guess it needs to trace everything and keep the records." (P7)*

While ProvBuild was able to generate a new simplified script, two participants were concerned about its *"accuracy" (P2)* and completeness: *"ProvBuild may have limitation in tracking the provenance of program execution, which may return an incomplete relevant code segment." (P12)*

Two participants explicitly mentioned that ProvBuild should support **multiple types of search targets**: *"I couldn't search for a variable that was a function argument." (P10) "It seems like ProvBuild can only search for global variables." (P2)*

Further concerns about ProvBuild are its **scalability** *"to process a large project." (P5)* and *"memory usage"(P6).*

Finally, since the participants have their own programming styles, one of them asked *"is it possible to show some visualization from ProvBuild?" (P3)*, while another wondered that *"it would be better if it works with Jupyter." (P4)*

Overall, we found these criticisms encouraging in that they addressed issues we knew about (e.g., initial run time) or that could be easily addressed (e.g., integration with Jupyter). Convincing users that ProvBuild produces correct results is an interesting challenge to address in future work.

8 DISCUSSION

ProvBuild allows data scientists to perform basic data analysis routine with lower completion time and less cognitive load, thereby increasing their programming efficiency. Although its provenance capture system increases initial runtime, participants found the cost-benefit trade-off worthwhile, demonstrating that its utility compensated for the increased initial runtime.

Threats to Validity. Regarding internal validity, user performance on the controlled experiments might depend on a user's ability to comprehend unfamiliar code. To mitigate this threat, we did within-subject experiments testing each participant under all conditions; the independent evaluation reduces errors associated with individual differences. A remaining challenge is to design a study that is rigorously controlled, but allows users to work on code with which they are already familiar.

Participants of the deployment study may have been inclined to answer favorably, since we were asking about their experience using ProvBuild. To reduce such bias, we used impersonal surveys, rather than face-to-face interviews.

The greatest external threat to validity is our assumption that noWorkflow captures provenance correctly. Its use in reproduction studies suggests that the community believes it to be appropriate for reproducing computation, which is effectively how we use it [16, 34]. It is also possible that we have not identified all instances in which the provenance does not capture all the information necessary to produce a complete and correct *ProvScript*. Should such situations arise, our experience suggests that the tools we've developed make it possible to easily collect additional information.

Limitation and Future Work. ProvBuild's initial runtime overhead might pose an obstacle to adoption, so our immediate plans including changing or improving the provenance collection strategy, disabling parts of the provenance capture that are unnecessary for this application, and tuning the remaining parts of it. Integration with a widely-used IDE, such as Eclipse, or other interactive computational environments, such as Jupyter [27], will also facilitate adoption. While ProvBuild is currently Python-specific, it should be straight forward to adapt it to other languages that have provenance-tracking support, e.g., R [31]. We also believe it is possible for these language-specific capture tools to produce provenance in a language-agnostic form, which would make it possible to develop a language-agnostic ProvBuild. In the longer term, we are looking for other opportunities to leverage provenance to help programmers streamline their development process.

Data Availability. ProvBuild and anonymized data from our user study is available on the open-access repository *Zenodo* [4]. Results for which public disclosure is not allowed by our IRB are not provided.

9 RELATED WORK

`Make` [14] is a build automation tool that uses static analysis to execute only those steps of a build process that depend upon modifications. Rather than using provenance, `make` uses `Makefiles` to explicitly keep track of file targets, inter-file dependencies, and command sequences. Users build `Makefiles` manually or use additional tools, e.g., `autoconf` [15, 46], to produce them. `make` is similar to ProvBuild in its ability to reduce re-execution time. However, `make` does not address the cognitive load issue nor does it help users identify problems more efficiently.

StarFlow is a `make`-like tool that tracks data dependencies in Python at function-call granularity [3]. The dependency tracking procedure for StarFlow uses static analysis, dynamic analysis, and optional user annotations for specifying function inputs and outputs. ProvBuild extends this work by removing the need for annotations to track dependencies. Like `make`, StarFlow does nothing to reduce cognitive load.

IncPy is a custom, open-source Python interpreter that performs automatic memoization and persistent dependency management at the function call level. IncPy automatically sends function calls, inputs, and outputs to a persistent on-disk cache immediately before the target program is about to return from a function call [21]. By automatically caching and reusing the results of function calls, IncPy enables programmers to iterate more quickly on their scripts, in a manner similar to ProvBuild. We wanted to compare IncPy re-execution times with those of ProvBuild; unfortunately IncPy has not been maintained, and we were unable to get it to run on modern Python scripts.

Similar to IncPy, Joblib is a dynamic analysis tool that transparently caches arbitrary Python objects to avoid unnecessary computation in Python [45]. IncPy and Joblib are similar to ProvBuild in the re-execution part, but they do not track dependencies other than functional ones and also do not address cognitive load.

ProvBuild uses both static and dynamic change impact analysis (CIA) [5]. Generally, CIA identifies the potential consequences of a change and estimates how to propagate the ramifications of that change. Static CIA analyzes the syntax, semantics, and change histories of a program without directly executing it [39, 43]. Dynamic change impact analysis captures information by executing programs on a real or virtual processor and utilizes dynamic information about program behavior to determine the potential effects of a change [29, 30, 37, 38]. CIA has been used in large and evolving industrial software systems [2] to evaluate test suites when a software system changes [41], compare large programs with different version to highlight changes [8], and analyze change propagation in large software systems and architectures [18]. Professional software engineers leverage CIA to estimate large software project changes. To the best of our knowledge, ProvBuild is a novel application of CIA, using it to improve runtime and cognitive load during data analysis development tasks.

Incremental compilation and self-adjusting computation are techniques that attempt to save time by recomputing only those outputs that depend on changed data [1, 11]. Most of these techniques rely on dependency graphs that record a computation's data and control dependencies so the change-propagation algorithm can identify sections that are affected by a user modification and rebuild only these relevant portions. The major result of change propagation is to incrementally build the script. It is similar to our work, while ProvBuild also directly improves the user programming experience.

10 CONCLUSION

ProvBuild is a novel use of language-level provenance that streamlines the iterative development process by allowing a developer to focus only on the code that the programmer is debugging. We use provenance to construct a dependency graph and generate a simplified script containing only those code blocks pertaining to a user-specified function or variable. This accomplishes three things. First, it helps users avoid unnecessary changes to their script (which frequently introduce new bugs [49]). Second, it makes it easier to identify and reason about code modifications that are necessary to correctly achieve a goal. Third, it reduces execution time and users' cognitive load, because only a portion of the original script is run at each iteration. Together, these reduce the time and effort it takes to debug data analysis pipelines. The results of the quantitative evaluation and the user feedback show that ProvBuild can be an easy, effective, and efficient tool for data scientists who use scripts to process and analyze data. To the best of our knowledge, this is the first time language-level provenance has been used to address programmer efficiency.

REFERENCES

[1] Umut A. Acar. 2009. Self-adjusting Computation: (an Overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '09)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/1480945.1480946

[2] Mithun Acharya and Brian Robinson. 2011. Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 746–755. https://doi.org/10.1145/1985793.1985898

[3] Elaine Angelino, Daniel Yamins, and Margo Seltzer. 2010. StarFlow: A script-centric data analysis environment. In *Proceedings of the 3rd International Provenance and Annotation Workshop (IPAW 2010). Lecture Notes in Computer Science*, Vol. 6378. Springer Berlin Heidelberg, Troy, NY, USA, 236–250.

[4] Anonymous. 2019. Supporting code and data for ProvBuild. https://doi.org/10.5281/zenodo.2653945

[5] Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[6] Hugh Beyer and Karen Holtzblatt. 1997. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Michelle A Borkin, Chelsea S Yeh, Madelaine Boyd, Peter Macko, KZ Gajos, M Seltzer, and H Pfister. 2013. Evaluation of filesystem provenance visualization tools. *IEEE transactions on visualization and computer graphics* 19, 12 (2013), 2476–2485.

[8] L. C. Briand, Y. Labiche, and L. O'Sullivan. 2003. Impact Analysis and Change Management of UML Models. In *Proceedings of the International Conference on Software Maintenance (ICSM '03)*. IEEE Computer Society, Washington, DC, USA, 256–.

[9] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2000. Data provenance: Some basic issues. In *FSTTCS 2000: International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, Springer Berlin Heidelberg, Berlin, Heidelberg, 87–93.

[10] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, Berlin, Heidelberg, 316–330.

[11] Magnus Carlsson. 2002. Monads for Incremental Computing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/581478.581482

[12] James Cheney, Amal Ahmed, and Umut a. Acar. 2011. Provenance As Dependency Analysis. *Mathematical. Structures in Comp. Sci.* 21, 6 (Dec. 2011), 1301–1337. https://doi.org/10.1017/S0960129511000211

[13] Cary Deck and Salar Jahedi. 2015. The effect of cognitive load on economic decision making: A survey and new experiments. *European Economic Review* 78 (2015), 97–119. https://doi.org/10.1016/j.euroecorev.2015.05.004

[14] Stuart I Feldman. 1979. Make - A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.

[15] Inc. Free Software Foundation. 2016. Autoconf. https://www.gnu.org/software/autoconf/. Accessed Aug 16, 2019.

[16] Juliana Freire and Fernando Chirigati. 2018. Provenance and the Different Flavors of Reproducibility. *IEEE Data Eng. Bull.* 41, 1 (2018), 15–26.

[17] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. 2008. Provenance for computational tasks: A survey. *Computing in Science & Engineering* 10, 3 (2008), 11–21. https://doi.org/10.1109/MCSE.2008.79

[18] Arda Goknil, Ivan Kurtev, and Klaas van den Berg. 2016. A Rule-Based Change Impact Analysis Approach in Software Architecture for Requirements Changes. *CoRR* abs/1608.02757 (2016). arXiv:1608.02757

[19] Leo A Goodman. 1961. Snowball sampling. *The Annals of Mathematical Statistics* 32, 1 (1961), 148–170. https://doi.org/10.1214/aoms/1177705148

[20] Jaber F Gubrium, James A Holstein, Amir B Marvasti, and Karyn D McKinney. 2012. *The SAGE Handbook of Interview Research: The Complexity of the Craft*. SAGE Publications, Thousand Oaks, CA.

[21] Philip J. Guo and Dawson Engler. 2011. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 287–297. https://doi.org/10.1145/2001420.2001455

[22] Eija Haapalainen, SeungJun Kim, Jodi F. Forlizzi, and Anind K. Dey. 2010. Psychophysiological Measures for Assessing Cognitive Load. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp '10)*. ACM, New York, NY, USA, 301–310. https://doi.org/10.1145/1864349.1864395

[23] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (2006), 904–908. https://doi.org/10.1177/154193120605000909

[24] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. *Human mental workload* 1, 3 (1988), 139–183.

[25] S. Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 6, 65-70 (1979), 1979.

[26] Gary King. 2007. An Introduction to the Dataverse Network as an Infrastructure for Data Sharing. *Sociological Methods and Research* 36 (2007). https://doi.org/10.1177/0049124107306660

[27] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, and et al. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing*. IOS Press, Göttingen, Germany, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[28] BW Lampson. 1976. Bravo Manual in the Alto User's Handbook. *Xerox Palo Alto Research Center* (1976).

[29] James Law and Gregg Rothermel. 2003. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE Computer Society, Denver, CO, USA, 430–441. https://doi.org/10.1109/ISSRE.2003.1251064

[30] James Law and Gregg Rothermel. 2003. Whole Program Path-Based Dynamic Impact Analysis. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 308–318.

[31] Barbara Lerner, Emery Boose, and Luis Perez. 2018. Using Introspection to Collect Provenance in R. *Informatics* 5, 1 (2018), 12. https://doi.org/10.3390/informatics5010012

[32] Ian A. Macleod. 1977. Design and implementation of a display oriented text editor. *Software: Practice and Experience* 7, 6 (1977), 771–778. https://doi.org/10.1002/spe.4380070611

[33] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware Storage Systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference (ATEC '06)*. USENIX Association, Berkeley, CA, USA, 4–4.

[34] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2015. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *Provenance and Annotation of Data and Processes*. Springer International Publishing, Cham, 71–83.

[35] The President & Fellows of Harvard College. 2015. API Guide. http://guides.dataverse.org/en/4.2/api/. Accessed Aug 7, 2018.

[36] The President & Fellows of Harvard College. 2017. Harvard Dataverse. https://dataverse.harvard.edu/. Accessed Aug 7, 2018.

[37] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. 2003. Leveraging Field Data for Impact Analysis and Regression Testing. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*. ACM, New York, NY, USA, 128–137. https://doi.org/10.1145/940071.940089

[38] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. 2004. An Empirical Comparison of Dynamic Impact Analysis Algorithms. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 491–500.

[39] Maksym Petrenko and Václav Rajlich. 2009. Variable granularity for improving precision of impact analysis. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*. IEEE Computer Society, Vancouver, BC, Canada, 10–19. https://doi.org/10.1109/ICPC.2009.5090023

[40] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: A Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *Proc. VLDB Endow.* 10, 12 (2017), 1841–1844. https://doi.org/10.14778/3137765.3137789

[41] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 432–448. https://doi.org/10.1145/1028976.1029012

[42] Juliet P. Shaffer. 1995. Multiple Hypothesis-Testing. *Annual Review of Psychology* 46 (1995), 561–584.

[43] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change Impact Analysis Based on a Taxonomy of Change Types. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference (COMPSAC '10)*. IEEE Computer Society, Washington, DC, USA, 373–382. https://doi.org/10.1109/COMPSAC.2010.45

[44] Wil MP Van der Aalst. 2014. Data Scientist: The Engineer of the Future. In *Enterprise Interoperability VI*. Springer International Publishing, Cham, 13–26.

[45] G Varoquaux and O Grisel. 2009. Joblib: running Python function as pipeline jobs. packages. python. org/joblib.

[46] Gary V. Vaughan and Thomas Tromey. 2000. *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, USA.

[47] Adrian Ward, Kristen Duke, Ayelet Gneezy, and Maarte Bos. 2017. Brain Drain: The Mere Presence of One's Own Smartphone Reduces Available Cognitive Capacity. *Journal of the Association for Consumer Research* 2, 2 (2017), 140–154. https://doi.org/10.1086/691462

[48] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Computer Society, Piscataway, NJ, USA, 439–449.

[49] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/2025113.2025121