# PRUNE: A Preserving Run Environment for Reproducible Scientific Computing

Peter Ivie
University of Notre Dame
pivie@nd.edu

Douglas Thain
University of Notre Dame
dthain@nd.edu

***Abstract*—Computing as a whole suffers from a crisis of reproducibility. Programs executed in one context are astonishingly hard to reproduce in another context, resulting in wasted effort by people and general distrust of results produced by computer. The root of the problem lies in the fact that every program has implicit dependencies on data and execution environment which are rarely understood by the end user. To address this problem, we present PRUNE, the Preserving Run Environment. In PRUNE, every task to be executed is wrapped in a functional interface and coupled with a strictly defined environment. The task is then executed by PRUNE rather than the user to ensure reproducibility. As a scientific workflow evolves in PRUNE, a growing but immutable tree of derived data is created. The provenance of every item in the system can be precisely described, facilitating sharing and modification between collaborating researchers, along with efficient management of limited storage space. We present the user interface and the initial prototype of PRUNE, and demonstrate its application in matching records and comparing surnames in U.S. Censuses.**

## I. INTRODUCTION

Reproducibility of results has become a major concern in scientific computing [1], [2], [3]. The challenge lies largely in the complexity of software and environments. A scientist's *productivity* can be reduced due to time spent getting shared software working, and the *integrity* of science becomes suspect due to the difficulty in verifying claims. A large part of the problem is **implicit dependencies**, such as configuration files, supporting libraries, scripting languages, an OS kernel, and suitable hardware [4]. The standard command line interface is not designed to encourage reproducibility.

One common solution is to use virtual machines (or containers) to store the full context available to an application. This can result in excessive overhead, so another common technique is to trace only dependencies that are actually used, such as with CDE [5], PTU [6], ReproZip [7], and Parrot [4]. The next step is to capture the evolution of the application over time, or the data flowing between multiple containers for large scale applications.

PRUNE is a computing system designed to support scientific reproducibility from the ground up.[1] PRUNE expects all data, software, and environment dependencies to be explicitly registered with the system and directly computes the tasks that make up an application, rather than recording the system calls or merely the end results of user executed commands. As various execution paths are explored, the user builds up a large graph of data, software, and environments used.

PRUNE bears some similarity to a distributed version control system like Git [9], except that it records a tree of continuous derivation information consisting of immutable files and computations, rather than a tree of periodic commits consisting of line changes in a directory of mainly textual files. This allows the end user to understand the precise provenance of any generated results. PRUNE is also similar to the Nectar [10] datacenter management system, except that it supports distributed de-duplication of storage and computation, rather than relying on a centralized system. This distributed provenance of computation facilitates flexible collaboration using export/import operations directly between individual scientists.

We have implemented a prototype of PRUNE (version 1) that exploits distributed execution, allowing it to scale to a large number of tasks. As a case study, we show how PRUNE is used to manage a data analysis workflow based on US censuses, developing a repository that preserves 34.5 terabytes of workflow data representing 3.9 million executions, 3.9 million files, and 2 different execution environments. To characterize the performance of this prototype, we measure the overhead of system operations in the context of this large repository. We show the wall clock time consumed is 101% compared to native execution (or 1% overhead).

## II. A PRESERVE-FIRST STRATEGY

Preservation is often perceived as an activity undertaken after research has been completed. [11] But, by the time the results based on a scientific workflow are accepted for publication, the authors have moved on to other work, students may have graduated, or the environment in which the work was done has been changed, upgraded, or destroyed. The funding that supported the research may have expired, and so it is hard to justify any post-facto effort in preservation. Even when such an effort is made, the focus is often only on repeatability [12], and more work is needed to fill in gaps in the preserved form of the research [13]. This process is shown in Figure 1a.

In contrast, we advocate a **preserve-first strategy** for reproducible computational research as shown in Figure 1b. We argue that researchers should first (before any computation) preserve (at least locally) the components they wish to use. Automated execution based on the preserved components can

[1] PRUNE was briefly introduced (along with a few other utilities) in a paper at iPres 2015 [8]. This is the first detailed presentation of the architecture, performance, and applications of the system.

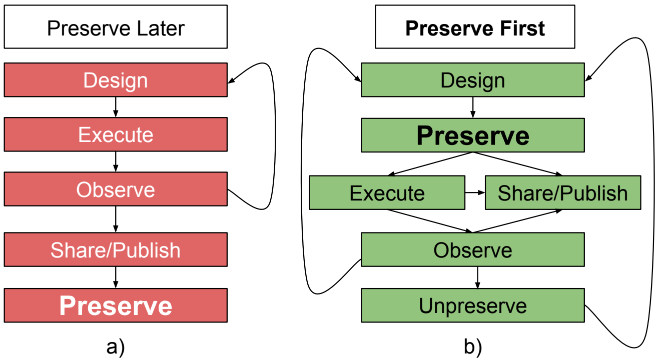


Fig. 1: Preserve-First or Preserve-Later?
*We propose a preserve-first strategy, in which digital items are preserved before use, so there is no ambiguity about results.*

then ensure all necessary dependencies are included, otherwise the execution fails. Once the desired research results are obtained, it is then trivial to publish them with full provenance in a public repository. Then others can build upon the same work with a high probability of success.

Adopting this strategy requires additional user and computer overhead. But we believe with this approach, PRUNE moves towards greater structure and oversight such as with the adoption of: block-structured programming [14]; graph-structured Make files [15]; and rigorous version control [9].

## III. Overview of PRUNE

### A. User's Perspective

An end user begins by creating their own private PRUNE repository, which may simply exist on their own laptop. The user describes a workflow which explicitly adds (into the repository) any input data and tasks that should be performed to derive some result. When the user submits this description, PRUNE detects portions of the workflow that are already in the repository, and records and then adds the remainder. Observing the results, the user may submit a revised workflow, expanding the graph in the repository. If space consumption becomes a problem, PRUNE will automatically delete derived results, because it retains the ability to re-create them on demand.

Other users or organizations may operate their own repositories. When a user has a result of interest to be shared, he/she asks PRUNE to export the appropriate meta-data into a portable package. The package can contain all the meta-data necessary to describe how the result was obtained, so that a receiving user can examine, re-execute, or build upon that result within their own repository. The most interesting results can be widely disseminated through a public repository.

### B. Repository Structure

A PRUNE **repository** contains a graph of immutable objects describing the data and computational elements needed to execute a workflow. The following 4 basic objects constitute the nodes of the graph: Files, Tasks, Results, and Environments. Once a workflow has been described in terms of these objects, the objects can be shared with collaborators or published as a complete and reproducible description of the workflow.

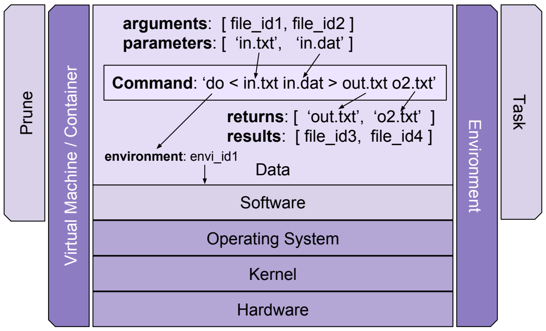

Prune
Virtual Machine / Container
arguments: [ file_id1, file_id2 ]
parameters: [ 'in.txt', 'in.dat' ]
Command: 'do < in.txt in.dat > out.txt o2.txt'
returns: [ 'out.txt', 'o2.txt' ]
results: [ file_id3, file_id4 ]
environment: envi_id1
Data
Software
Operating System
Kernel
Hardware
Environment
Task

Fig. 2: Tasks and Environments
*Data and software can be handled in a Task or Environment. The Environment describes down to the hardware layer.*

A **File** is an immutable string of bytes, identified by a hash of the content of the File. Any data that the user wishes to use must first exist as a File within a repository.

A **Task** is a program to be executed, represented as a brief JSON document that describes a command line, the input Files, and the Environment in which the Task should run.

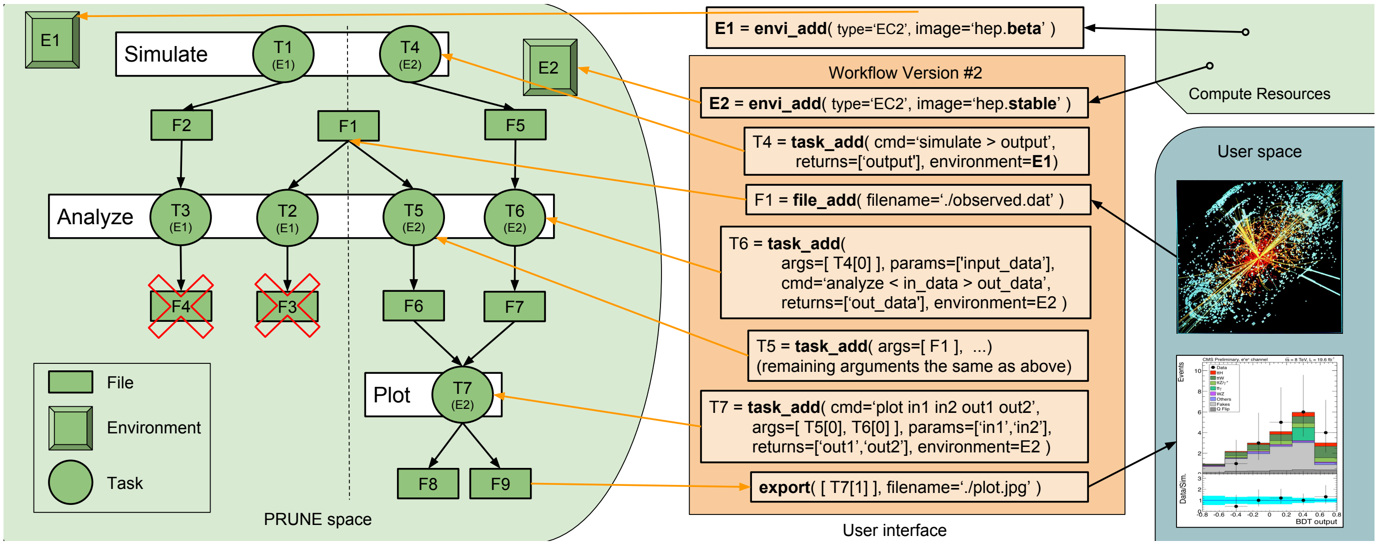
A **Result** object contains information about the completed execution of a Task, including identifiers for the output files (which were not known until the Task completed) along with the time and resources consumed during execution.

An **Environment** is an explicit statement of the hardware and software needed to execute a Task. An Environment can take many forms, depending on the technology used and the priority placed on reproducibility compared to convenience.

For example, an Environment could be a virtual machine image with instructions for creating and using a virtual machine for executing Tasks. Such an Environment may be more likely to be reproduced in 10 years than an Environment which is a reference to a virtual machine image in Amazon EC2 or a container image in a public Docker Hub. However, a reference to Amazon EC2 might be more convenient in the short term. An Umbrella [16] Environment can choose an efficient mode when available, but include backup modes if needed. However, even a tarball of software needed on top of an assumed operating system can be considered an Environment.

We assume that an Environment is something created infrequently by working closely with a system administrator, in the same way that a physical machine's operating system is infrequently changed and constantly re-used.

Figure 2 illustrates how a Task relates to an Environment. To execute a Task, the Environment is used to create a temporary sandbox. Any input File arguments are mapped to local pathnames within the sandbox `["in.txt","in.dat"]` where they can be accessed via the running command. After the command is executed, the output files are retrieved from their expected location `["out.txt","o2.txt"]` where they can be extracted and stored within the PRUNE repository as Files and a Result. The remainder of the sandbox is discarded.



```
E1 = envi_add( type='EC2', image='hep.beta' )
E2 = envi_add( type='EC2', image='hep.stable' )
T4 = task_add( cmd='simulate > output',
       returns=['output'], environment=E1)
F1 = file_add( filename='./observed.dat' )
T6 = task_add(
       args=[ T4[0] ], params=['input_data'],
       cmd='analyze < in_data > out_data',
       returns=['out_data'], environment=E2 )
T5 = task_add( args=[ F1 ],  ...)
(remaining arguments the same as above)
T7 = task_add( cmd='plot in1 in2 out1 out2',
     args=[ T5[0], T6[0] ], params=['in1','in2'],
     returns=['out1','out2'], environment=E2 )
export( [ T7[1] ], filename='./plot.jpg' )
```

Fig. 3: Overview of a PRUNE repository

*Files, Tasks and Environments are used to explicitly describe an evolving workflow from the beginning to the current state.*

### C. Interface

Prune has six fundamental operations:

```
id = file_add( filename );
id = task_add( task-description );
id = envi_add( type, image );
execute( available_resources );
export( id-list, filename, options );
import( filename );
```

Three operations add to the repository: `file_add` adds a file to the repository from the local filesystem, and returns an identifier for it's File object. `task_add` adds a Task to be executed to the repository and then immediately returns an identifier. The Task is queued for execution and the results will become available when time and resources permit. `envi_add` adds a new Environment to the repository, specifying the type of the environment (VMWare, Amazon, Docker, TGZ, etc) and the name of the image.

The `execute` command specifies what resources can be used to execute Tasks, and when they are to be used. The `export` operation creates a package which includes a subgraph of the repository. It expects of a `query anchor` (a list of ids as a starting point) and optionst that describe which direction(s) to follow derivation lines and which object types to include in the package. The `import` operation adds new objects into the repository from such a package. Because `task_add` returns an identifier before executing the Task, it is possible that an export will request File objects that do not yet exist. It is a matter of preference whether such a request will block or require the user to poll until objects are available.

### D. Example

A snapshot of a workflow is illustrated in Figure 3. Here, we show a common workflow pattern in high energy physics. A researcher runs several simulations that mimic the behavior of a device like the Large Hadron Collider. The behavior of simulations and observations are analyzed separately. The analyses are then plotted together to produce a publishable graphic. Each step of this process may be repeated many times with different parameters, and then adjusted and retried as the user refines the workflow.

The left side of Figure 3 shows a graph stored in a PRUNE repository, while the middle of the figure shows some operations used to add new items to the repository. The dotted line divides the objects into separate versions of the workflow. F1 (on the dotted line) is the observed data and is used in both revisions of the workflow. Initially, the user ran simulation T1 in Environment E1 to produce synthetic data F2, which was analyzed by T3, producing F4. Then, some actual data was imported as F1 and analyzed by T2, producing F3.

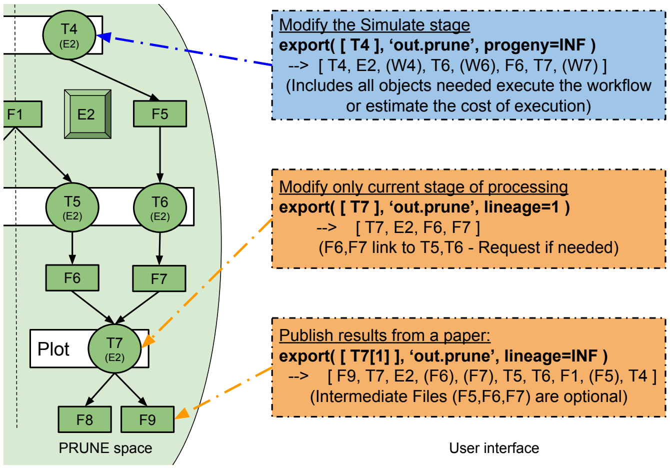
Suppose that the user realizes that the (crossed out) Files F3 and F4 are invalid results, due to some bug in the supporting libraries found in Environment E1. To remedy this, the user prepares a second Environment E2 with new libraries, and then runs T4, T5 and T6 to simulate and analyze the same data again. Finally, the simulated and real data are combined into a single plot (T7) that produces files suitable for graphing.

Figure 4 shows some examples of export being used for collaboration. Export can be used to retrieve how a result was generated (lineage) or what objects were derived from an anchor object (progeny).

### E. Naming

The issue of naming in computing has long been a challenge and various approaches have been proposed to resolve the disconnect between computer and human naming. [17] PRUNE uses two types of identifiers for objects: content-based identifiers and derivation-based identifiers.

A content based identifier (CBID) is the fundamental name for all Files, Tasks, and Environments. It is generated by



Modify the Simulate stage
**export( [ T4 ], 'out.prune', progeny=INF )**
--> [ T4, E2, (W4), T6, (W6), F6, T7, (W7) ]
(Includes all objects needed execute the workflow or estimate the cost of execution)

Modify only current stage of processing
**export( [ T7 ], 'out.prune', lineage=1 )**
--> [ T7, E2, F6, F7 ]
(F6,F7 link to T5,T6 - Request if needed)

Publish results from a paper:
**export( [ T7[1] ], 'out.prune', lineage=INF )**
--> [ F9, T7, E2, (F6), (F7), T5, T6, F1, (F5), T4 ]
(Intermediate Files (F5,F6,F7) are optional)

Fig. 4: Export Example
*Different options determine which objects are shared.*

computing a hash function of either the content of the object, which is the binary data of a File, or the JSON document representing a Task or an Environment. Care must be taken to ensure the ordering of JSON elements (alphanumeric or fixed order keys) so that a CBID does not change as the item is shared among repositories.

PRUNE also stores some auxiliary meta-data about each object type, such as owner, creation time, resources consumed, etc. This meta-data is excluded from the checksum so that the CBID can be used to detect if an object is logically unique.

A derivation-based identifier (DBID) is used to identify files that have not yet been generated. It consists of the CBID of a Task, followed by a subscript that selects one of the results of the Task. DBIDs can be used as arguments to later tasks, so that multiple Tasks can be chained together before the intermediate Files have even been generated.

For example, suppose that Task T consumes files A and B (which exist in the repository) and produces files X and Y. The CBIDs for Files A and B are used in the JSON document that describes Task T. The CBID for Task T is simply the checksum of its JSON document (`38b1d`). When Files X and Y are produced, they can be addressed using the CBIDs computed from their checksums. But they may also be addressed as `38b1d[0]` and `38b1d[1]`, which indicate they are the first and second output Files of Task T respectively.

| | | | |
|---|---|---|---|
| Task: | (A,B)→ | T | → (X,Y) |
| CBID: | 18f23, a3f91→ | 38b1d | → 93d8a, 413ca |
| DBID: | | | → 38b1d[0], 38b1d[1] |

Result objects record the mapping between DBIDs and output CBIDs (once the Task has been executed). Keeping this information separate from the Task allows the Task to remain immutable. Sometimes generated Files are deleted to make room for other Files as mentioned in section III-A. If those Files are needed again, the Task is re-executed, generating an additional Result object for the Task. If derived Files are deleted, the checksums in the Result can be used to validate re-generated output Files.

### F. Non-Determinism

If a Task is non-deterministic, multiple executions of the Task can generate Files that are bitwise different, but logically equivalent for a given scientific domain. PRUNE is unable to detect such logical equivalence. For example, this can happen with the Monte Carlo simulations used in high energy physics workflows. In these cases a single DBID can refer to multiple CBIDs. Since the input File identifiers are part of a Task's checksum, equivalent Tasks could end up with (any number of) different Task CBIDs.

In an effort mitigate this issue while still allowing the workflows to be fully specified before execution, PRUNE encourages, when possible, the use of DBIDs throughout. This enhances the ability to effectively collaborate and de-duplicate, which is discussed in later sections, but CBIDs can also be used where the user feels it is more appropriate.

## IV. Storage Management

One of the challenges with preserving a workflow is the amount of storage space required. We observe (and assume) that, in general, the largest portion of the storage requirement for a scientific workflow consists of Files generated during the execution of a workflow. These **derived** Files can be **leaf** Files (not used as an argument for any Task) or **intermediate** Files (used as an argument in one or more Tasks). We propose treating derived Files as a disposable portion of a workflow as detailed in section IV-A. We assume that the second largest portion of the storage requirement is typically **root** Files (external input data directly imported into a Prune repository). We discuss ways to address this challenge in section IV-B. The smallest portion of the storage requirement is the data describing the Tasks needed to get from the root Files to the leaf Files. Reducing the storage requirements in this category is covered in section IV-C.

### A. Derived File Cache

Derived Files can be deleted to save disk space without limiting reproducibility, since all the information needed to recreate them is found in the Tasks, root Files, and Environments. In a sense, these derived Files can be treated as a temporary cache. The Result objects remain in the database for consumed resource statistics and checksum validation.

The priority used to determine which derived Files to evict first could be as simple as evicting the oldest derived File. However, more advanced algorithms could be based on File sizes and their position in the repository graph. The same algorithms used to follow lineage and progeny in the export operation could also be useful in deciding which derived Files are the least likely to be used. The cost (financial or otherwise) of reproducing a File should also be considered.

### *B. External Objects*

Since root Files cannot be re-generated, they must be set apart from the derived Files to prevent the system from disposing of them. An advanced implementation of PRUNE could extend Tasks to allow input files specified as URLs rather than restricting them to Files only. In such a case, additional rules (based on the bandwidth, reliability and longevity of the external resource) would be needed to determine whether the results of such Tasks could be generated again in the future.

For very large workflows, a smaller repository could treat derived Files from another repository as rooted files, but also include a Task that refers to the full repository for additional lineage. This permits flexibility in constructing repositories appropriate for a given researcher, while still ensuring full preservability (spanning multiple repositories) back to the root Files. In some cases there should be overlap between repositories for added replication and availability, but for others it would be sufficient to simply have a well defined line between repositories.

This is in line with large central data approaches like IVOA [18], IRIS [19], the LHC [20], etc., but any changes to the data by the managing organization must be detectable and/or avoidable in the interest of ensuring reproducibility.

### *C. Workflow merging*

Recording each workflow DAG individually in a PRUNE repository satisfies the need for preservation. However, this can cause unnecessary duplication of Task objects and their executions. Even with the assumption that Task objects are small compared to File objects, eliminating duplication at this level can result in more efficient use of both storage and execution resources.

We observe that as a researcher creates a workflow, there is generally a gradual evolution of that workflow while adjustments are made. Only a portion of the PRUNE objects describing the workflow will change with each evolution. Especially for changes made closer to the leaf Files, or by extending from leaf Files, only a small portion of the objects will differ from a previous version of the workflow. To merge a new workflow into a repository, PRUNE identifies the duplicates and effectively grafts the new objects onto a merged repository graph.

The expanded graph after de-duplication describes both the old and the new workflows simultaneously with shared objects defining the earlier portions of the workflow. As the workflow continues to evolve the graph continues to expand. This expanded graph approach makes up a more efficient PRUNE repository. The ability to detect duplicate Tasks coupled with the ability to treat their generated results as a cache enables memoization. This optimization technique reduces the time it takes to execute a workflow which already includes generated Files in the repository.

In order to support queries (such as those for the export operation) on a merged repository graph, tracing the lineage of the query anchor forward can be enabled by attaching a workflow identifier to each new object added to the graph. However, since any existing duplicate objects are immutable, they cannot be updated with a list of workflows they were used in. When tracing the progeny of the query anchor backwards, there may be multiple paths that could be traversed. This could happen, for example, if two Tasks achieve identical results, but reached those results using a different approach. In order to ensure that the progeny of a result matches how the result was achieved, an identifier based on the workflow needs to be used in addition to a CBID and DBID.

## V. PROTOTYPE

PRUNE v1 is written in Python and uses SQLite3 to keep track of all workflows submitted to it. The user creates a Python script which uses a PRUNE v1 client library to expose PRUNE v1 operations inside of the Python script. The client library translates API commands into SQLite3 queries to preserve new workflow objects and ignore duplicate objects when detected. The client library can also export or import entire workflows or portions of workflows.

A PRUNE v1 repository is a database of workflow objects recorded over time. It is divided into 3 parts; **persistence**, **cache**, and **status**. Both the cache and status portions can be re-created by PRUNE v1, but the persistence portion contains objects that contain irreplaceable information. The cache portion stores generated Files. The persistence portion stores the remaining objects. The status portion tracks the progress of Tasks that still need to be executed and which of those are ready to execute immediately as compared to those that depend on Files which are not yet available in the cache.

SHA1 checksums are computed on object content to create the CBIDs. When the content is in JSON format (Tasks, Environments, and Results), the keys are sorted alphanumerically to keep the CBIDs consistent.

DBIDs use a ':' character after the Task CBID, followed by an index number to distinguish between outputs of a given Task. To encourage meaningful variable names in Python `task_add` returns the list of DBIDs instead of the CBID for the Task. The CBID prefix is still available if needed.

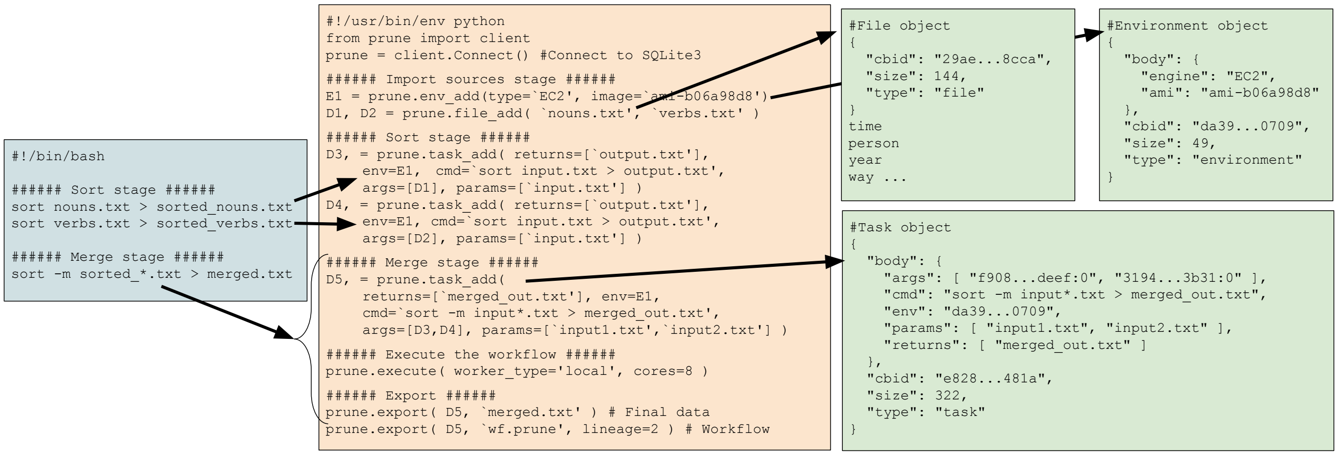
PRUNE v1 treats 2 Tasks which are identical except the Environment, as separate Tasks in the database. Each Task must be executed, and each Result stored, but if the generated Files are identical, they are only stored once (using the CBID and first DBID).

An export in PRUNE v1 creates a single file with all relevant objects embedded. This file can be shared with other users of PRUNE v1 either directly or via the internet.

If any Files requested in the export command have not been generated or were evicted from the cache, the user receives a message indicating that Files are not yet available. The user may then repeat the request until the results are available.

### *A. Compute Resources*

Prune can either spawn **local** worker processes to execute Tasks, or start a Work Queue [21] master to coordinate Task execution on **remote** workers. In local mode, input Files are linked into Task sandboxes, with the assumption that Tasks



```
#!/bin/bash

###### Sort stage ######
sort nouns.txt > sorted_nouns.txt
sort verbs.txt > sorted_verbs.txt

###### Merge stage ######
sort -m sorted_*.txt > merged.txt
```

(a) Original Workflow script

```
#!/usr/bin/env python
from prune import client
prune = client.Connect() #Connect to SQLite3

###### Import sources stage ######
E1 = prune.env_add(type=`EC2', image=`ami-b06a98d8')
D1, D2 = prune.file_add( `nouns.txt', `verbs.txt' )

###### Sort stage ######
D3, = prune.task_add( returns=[`output.txt'],
     env=E1,  cmd=`sort input.txt > output.txt',
     args=[D1], params=[`input.txt'] )
D4, = prune.task_add( returns=[`output.txt'],
     env=E1, cmd=`sort input.txt > output.txt',
     args=[D2], params=[`input.txt'] )

###### Merge stage ######
D5, = prune.task_add(
     returns=[`merged_out.txt'], env=E1,
     cmd=`sort -m input*.txt > merged_out.txt',
     args=[D3,D4], params=[`input1.txt',`input2.txt'] )

###### Execute the workflow ######
prune.execute( worker_type='local', cores=8 )

###### Export ######
prune.export( D5, `merged.txt' ) # Final data
prune.export( D5, `wf.prune', lineage=2 ) # Workflow
```

(b) Prune workflow (with Python client library)

```
#File object
{
  "cbid": "29ae...8cca",
  "size": 144,
  "type": "file"
}
time
person
year
way ...
```

```
#Environment object
{
  "body": {
    "engine": "EC2",
    "ami": "ami-b06a98d8"
  },
  "cbid": "da39...0709",
  "size": 49,
  "type": "environment"
}
```

```
#Task object
{
  "body": {
    "args": [ "f908...deef:0", "3194...3b31:0" ],
    "cmd": "sort -m input*.txt > merged_out.txt",
    "env": "da39...0709",
    "params": [ "input1.txt", "input2.txt" ],
    "returns": [ "merged_out.txt" ]
  },
  "cbid": "e828...481a",
  "size": 322,
  "type": "task"
}
```

(c) Prune objects

Fig. 5: Example Workflow

*An example workflow (a) is shown using* PRUNE *commands (b), with a few of the individual objects that are recorded (c).*

will "play nice" and not modify those files. This is how files are treated when executing commands outside of PRUNE, and is appropriate for the high energy physics and census workflows we considered. In remote mode, Files must be transmitted over the network, making it more appropriate for computationally intensive Tasks with small inputs.

PRUNE v1 puts all submitted Tasks (which don't have their output files in the cache) into the status portion of the database. These Tasks are eagerly evaluated whenever a prune_worker is running. When the command for a locally run Task returns an error code, the sandbox is left in tact so the user can see what modifications would be needed to submit a corrected Task.

PRUNE v1 currently allows Tasks to run without a specified environment (meaning that the default available environment should be used), with a Wrap environment, or with a local Umbrella [16] environment. A Wrap environment runs an *open* command to prepare the environment for command execution (then an optional *close* command). A Wrap environment was used to extract a tarball with software needed for the workflows used in evaluating PRUNE v1.

### B. Example Workflow

Consider the shell script shown in figure 5a designed to take two input files and efficiently produce a new file with all lines merged and sorted.

The Python script in figure 5b will preserve and execute a workflow equivalent to figure 5a. The last line exports the minimum objects needed to reproduce the workflow, and saves these objects in the "merge_sort.prune" file.

The PRUNE v1 client library converts the script at figure 5b into the PRUNE v1 (slightly abbreviated) objects at figure 5c which are not exposed directly to the user. These objects are what is stored in the PRUNE v1 repository.

This may seem verbose compared to the original workflow. But we claim that the benefits of adopting a preservation-first strategy (beyond just the preservation benefits) can outweigh the added complexity. The following section evaluates some of those benefits.

## VI. EVALUATION

In order to evaluate the storage management abilities, computational overhead, and scalability of PRUNE v1, it was used to manage workflows doing some analyses on U.S. Census records. The U.S. Census [22] for years 1850 to 1940 consume 23 GB using 7-Zip compression. In a "Matching" workflow, the censuses are searched for instances where identical attributes occur exactly once in a pair of censuses, indicating a high probability both records refer to the same person in real life. Due to spelling, transcription, and other errors, a workflow with exact matching achieves very few matches.

To improve the matching results, a "Comparison" workflow creates a list of the most frequent surnames in all censuses and compares it against the list of all surnames to obtain lists of possible alternate spellings. The goal is to use these alternate spellings to feed fuzzy matching (rather than exact matching) into the Matching workflow. The Matching workflow is broken down into the following 7 stages:

| **Matching workflow stages** | |
|---|---|
| 1 | Decompress<br>(7-Zip unpacking) |
| 2 | Normalize<br>(Standardize field inclusion, names, and order) |
| 3 | Map key split<br>(Split into blocks that will include matches) |
| 4 | Summarize year<br>(Merge into 1 file per year per block) |
| 5 | Merge pairs<br>(Merge each pair of years together) |
| 6 | Group by key<br>(Make groups based on block key) |
| 7 | Find 1-1 matches<br>(Find unique matches - exactly 1 entry per year) |

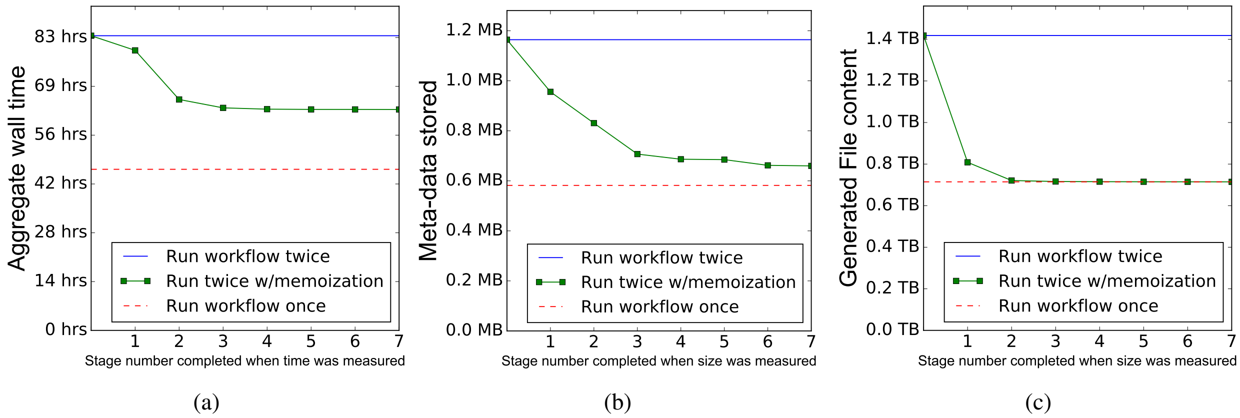


Fig. 6: *When changes to a workflow occur in later stages,* PRUNE *(a) avoids duplicate execution, (b) avoids extra disk space used to specify the workflow, (c) avoids extra disk space used for generated Files.*

Importing original files into PRUNE v1 is more a part of PRUNE v1 behavior than that of the workflow, so we consider this Stage 0. Stages 0-2 are identical between the two workflows. The "Comparison" workflow has 6 unique stages:

**Comparison workflow stages**

3 Count attributes
(Count appearances of field-attribute pairs)
4 Summarize year
(One file per year summarizing pairs in that year)
5 Summarize all
(A single file for summarizing pairs across all years)
6 Filter by field
(A separate file for each field type)
7 Sort by frequency
(Most frequently occurring attribute on top)
8 Similar attributes
(Score similar alternates for most frequent surnames)

### *A. Collaboration*

PRUNE v1 can be used to facilitate evolutionary changes by multiple users concurrently. Take for example a situation where one user finds an interesting match and wants to share those results with another user. One of the result files in the full comparison workflow was chosen as an export query anchor. The exported package with all tasks and root and intermediate files resulted in a 1.5TB file and took 1 hour and 25 minutes. But it only took 3 seconds to create a 2.6GB package with only the root Files and the Tasks, and it took 5 minutes and 30 seconds to read the package and recreate the query anchor File on a separate machine. In 4 seconds, another 2.6GB package was created with the Tasks, root Files, and the anchor File. The anchor didn't need to be generated on a separate machine, but all information was available to reproduce the File if desired.

Re-importing any of these exports back into the original repository has no effect as PRUNE v1 detects duplicates and ignores them. However, consider a situation where slight changes are made to the workflow by the collaborator. Importing a new export received from the collaborator, would still result in the detection and ignoring of duplicate objects, and then any new portions of the workflow would be added to the repository.

### *B. Conservation*

A common approach to preservation is to create a separate folder for each snapshot of all scripts and files each time a paper is published or some other milestone. In figures 6a, 6b, and 6c comparisons are made between this situation where two versions of the Comparison workflow are in separate folders (upper line) compared to a situation where only one version of the workflow exists (lower line).

The line in the middle shows the resources consumed by storing both workflow versions in PRUNE v1, after making a change to the workflow stage number indicated on the x-axis.

In figure 6a, the wall time improvements due to memoization are modest in the first stage since it is not very CPU intensive. The normalization stage is more significant computationally. The final stage is the next most significant one in terms of computation. Doing an all-pairs match on surnames using the Jaro-Winkler algorithm [23] is computationally expensive, so even changes to only that final stage still require a significant amount of work.

The measurements in figures 6a, 6b, and 6c were taken after doing comparisons on only 100 of the 11,400,952 unique surnames in the censuses. Executing more comparisons is covered in the following sections.

In figure 6b File content (but not metadata) is ignored. A workflow change in the first 3 stages results in a larger database because of the large number of files generated by those stages. The later stages have a more negligible affect on the database size. This indicates PRUNE v1 might be most effective when evolutionary changes to a workflow are made at the leaves of the workflow rather than at the roots.

Figure 6c shows the intermediate File space. The decompress stage creates large files with duplicate and extraneous (in this context) fields. This data is included in the graph even though it is only stored once in the PRUNE v1 database.

TABLE I: Wall clock time overhead

| | Preserve workflow | Prepare Execution | Execute Tasks | Checksum results | Preserve executions | **Total Time** | **Wall clock time overhead** | **Total # of Tasks/Files** | **Space (MB)** |
|---|---|---|---|---|---|---|---|---|---|
| Import sources | 1:21 | - | - | - | - | **1:21** | **100%** | 168 | 24.37 |
| Decompress | ˜0 | 0:10 | 1:41:33 | **5:19:36** | 0:25 | 7:01:43 | **315%** | 168 | 609,984 |
| Normalize | ˜0 | 0:12 | 10:16:11 | 52:30 | 1:48 | 11:10:42 | 9% | 167 | 86,234 |
| Count attributes | ˜0 | 0:01 | 5:41:12 | 0:18 | ˜0 | 5:41:33 | ˜0% | 167 | 4,799 |
| Summarize year | ˜0 | ˜0 | 22:05 | 0:03 | ˜0 | 22:08 | ˜0% | 10 | 819 |
| Summarize all | ˜0 | ˜0 | 4:22 | 0:01 | ˜0 | 4:24 | ˜0% | 1 | 407 |
| Filter by field | ˜0 | ˜0 | 0:07 | 0:01 | ˜0 | 0:09 | 24% | 16 | 407 |
| Sort by frequency | ˜0 | ˜0 | 2:02 | 0:02 | ˜0 | 2:04 | 1% | 16 | 407 |
| Similar attributes | 0:25 | 2:52 | 544:18:38 | 8:26 | 3:00 | 544:37:39 | ˜0% | 10,000 | 102,689 |
| **Total** | **1:47** | **7:16** | **562:26:11** | **6:20:57** | **5:13** | **569:01:43** | **1%** | **10,713** | **830,114** |

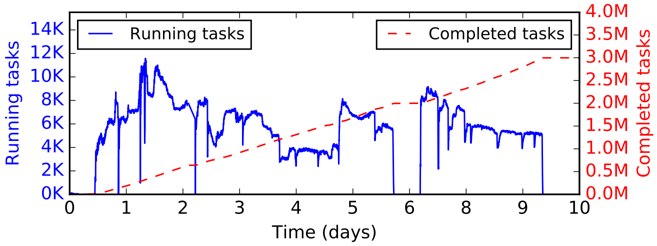


Fig. 7: Tasks running over time
*3 million Tasks were executed within 10 days with a concurrency of O(10k).*

The normalize stage then strips much of that out and produces smaller files. All other stages have comparatively small intermediate Files. This is great for PRUNE v1 because the unpacked data becomes a better candidate for eviction from the cache since the normalized data will be used more often than the raw unpacked data.

However, all the data depicted in figure 6c is a candidate for eviction. In extreme cases, intermediate files could be deleted as soon as they are consumed by later tasks.

### C. Overhead

To measure the overhead of PRUNE, the Comparison workflow was executed to produce a list of similar surnames for each of the 10,000 most frequent surnames (Stage 8). This workflow was executed using only local workers because the files were large compared to the compute resources needed to process them for these stages. The execution time, wall clock time overhead and data storage requirements for each stage is shown in table I.

Stage 0 (the "import sources" stage) is included here as 100% overhead, since PRUNE v1 must make a copy of all the original data, whereas in preserve later system, the files in user space would be used directly. It is interesting to note that checksumming the files after the decompression stage is more than 3 times more computationally expensive than just decompressing the files. Two options are available to address this issue. Option 1) Skipping a checksum of Files altogether (perhaps when Files are large) would result is less computational time, but the system might have to transfer and store duplicate copies of the data. This might not be bad since this data is intermediate and can be evicted from the cache anyway. Option 2) Checksumming in the background could both avoid the immediate delay and the duplicate storage. However, when Tasks are executed remotely (see the Scaling section below), the data still has to be transferred twice.

However, while this overhead seems significant when looking at that one stage, the overall overhead is only around 1%. The low overhead in the CPU intensive final stage (with a relatively small input file) makes the overhead in the decompress stage much less significant.

PRUNE v1 chooses to always do duplicate elimination as in some cases this can also lead to avoiding the re-execution of later stages if the duplicate is caught early on. Also, this overhead is likely to only occur for the first evolution of the workflow. Only a change in the environment for stage 1 or a change to the files in stage 0 would result in having to perform these checksums again.

### D. Scaling

For the scaling evaluation, the earlier stages of the workflow are mostly disk intensive, so they were performed using 16 local processes on the server to avoid network transfer congestion and delays. The final stage is more CPU intensive, so a Work Queue master in Prune with O(10k) remote workers was used to bring the total number of surname comparisons to 3 million. Figure 7 shows the concurrency of Tasks running for about 9 days. The total storage space for the entire workflow after these 3 million+ Tasks was about 28TB.

### E. Storage Quota

In any storage-constrained system, it is important to keep the intermediate data within those constraints. While executing an additional ˜864k Tasks of the workflow, PRUNE v1 was given a quota of 30TB. Prune v1 appropriately removed Files from the repository cache whenever it observed that new generated Files caused the repository to go over quota.

Figure 8 shows that Prune stayed within about 700MB of the quota after reaching the quota. This was done in the background to avoid interference with the remote workers.

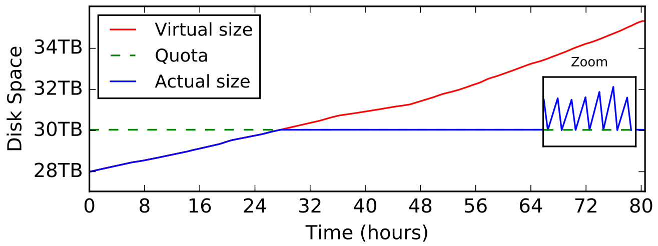


Fig. 8: Virtual vs. actual storage with quota
*A storage quota system held disk consumption close to 30TB during an additional ˜864k tasks.*

### F. Reproducibility

We do not have permission to share the root Files for the census workflows, so the full workflows cannot be shared in their entirety. However, we simulated some census data so that it could be used as root files for Stage 3 and beyond (no need for unzipping or normalizing the simulated data). PRUNE and example workflows (including the simulated census and a high energy physics workflow using Umbrella [16]) can be downloaded and executed by following the instructions at: http://ccl.cse.nd.edu/software/prune/

## VII. Related Work

PRUNE builds on, and derives from, many existing technologies. It combines capabilities from revision control systems, workflow management systems, and tools for hardware/software reproducibility. Other systems that combine such capabilities were also considered and drawn from.

Revision control systems like Git[9], CVS[24], and Subversion[25] track changes a user makes to files in a directory tree rather than a tracking a graph of task provenance. “Git and Org-Mode”[26] augments Git to be more applicable for workflow preservation. However, there is also normally an underlying assumption that a repository is small enough that making a full copy of both the current state and state history is not a problem. These revision control systems expect the user to decide when something should be committed. This is more efficient and leads to cleaner histories, but runs the risk of the user forgetting to commit important states.

A user might attempt to include portions of the needed environment, such as a compiler, with a repository, but more often than not, the necessary environments are merely implied. In a ”preserve-first” source code version control system, every time a user wanted to check if a change to source code is working, the changes would first be committed (just in case the changes are “good”), and the system would automatically compile or test the code, and then present the results (or errors) to the user, with the assumption that “bad” commits can later be removed, if needed.

Management systems such as Pegasus [27], Swift [28], Galaxy [29], and Nextflow [30] are specifically geared towards workflows. They vary widely on their support of reproducibility, collaboration, resource constraints, and preservation. Pegasus focuses on automating compute and storage resource consumption while facilitating recovery and debugging of failures. Swift uses implicit parallelism in it’s programming model to automatically scale workflows while requiring minimal programming expertise. Galaxy is designed specifically for biomedical research and uses a web interface to facilitate collaboration. Nextflow uses streams (based on Unix pipes) to automate concurrency for any programming language.

A few systems track workflows over time. VisTrails [31] tracks workflows that generate images. An Eidetic System [32] tracks and records the evolution of the file system on a single machine and can replay previous events. Nectar [10] centralizes computations performed in a datacenter in connection with their results and supports treating result data as a cache similar to the data reuse feature in Pegasus. [27]

CDE [5], PTU [6], ReproZip [7], and Parrot [33] are ways of capturing an Environment designed to execute a specific Task (but maybe not a more generic one). Transparent result caching [34] was designed to continuously and automatically record all dependencies over time on a single machine.

Preserving a full Environment (designed for a generic category of Tasks) is made possible with Docker [35], Umbrella [36], and virtual machine images. They can create an instance of an entire software stack for Tasks which have been modified from their original execution.

GridDB [37], Apt [38], Taverna [39], dataref versuchung [40], and Paper Mâché [11] are also related. With GridDB and Apt, the user directly executes operations in the workflow and the system assumes they will preserve how the execution was performed. GridDB includes detailed specifications for intermediate data allowing automatic parallelization between operations. Taverna is designed to assist bioinformaticians using web services to share sequence analysis methods online. Dataref versuchung and Paper Mâché encourage automation of the full publication life-cycle when it can be contained on a single machine.

## VIII. Conclusion

The PRUNE framework is designed to facilitate reproducible work, by capturing dependencies in a way that is easily shared, but also easily modified, extended, and understood. This seems in line with modern research directions [41] encouraging collaboration across traditional boundaries such as applied vs. basic research and engineers vs. scientists vs. designers.

With a proven concept, a few important qualities in Prune could be applied to other systems. An existing workflow system could adopt a preserve-first strategy by exporting all executed tasks (at any granularity) into a common repository which can then be queried as needed or pruned down to only the most relevant information, if necessary.

If the generated results are included in the exported data, those results can be saved and the workflow system can check the repository for cached results before executing anything. Such a system could use derivation or content based identifiers, or a combination of the two (as in Prune), or a full database implementation might have some advantages also.

Of course, technology is only one piece of the reproducibility puzzle. The problem of reproducibility encompasses publication practices, intellectual property, incentive systems, and many other issues [42]. But we hope that by reducing the technical burden of reproducibility, we can stimulate the erosion of other barriers to scientific progress.

## IX. Acknowledgments

This work was supported in part by the National Science Foundation under grants PHY-1247316 and OCI-1148330, and the Department of Education under grant P200A120206.

## References

[1] J. P. Ioannidis, "Why most published research findings are false," *PLoS Med*, vol. 2, no. 8, p. e124, 2005.

[2] L. P. Freedman, M. C. Gibson, S. P. Ethier, H. R. Soule, R. M. Neve, and Y. A. Reid, "Reproducibility: changing the policies and culture of cell line authentication," *Nature methods*, vol. 12, no. 6, pp. 493–497, 2015.

[3] B. Nosek, G. Alter, G. Banks, D. Borsboom, S. Bowman, S. Breckler, S. Buck, C. Chambers, G. Chin, G. Christensen, *et al.*, "Promoting an open research culture: Author guidelines for journals could help to promote transparency, openness, and reproducibility," *Science (New York, NY)*, vol. 348, no. 6242, p. 1422, 2015.

[4] H. Meng, M. Wolf, P. Ivie, A. Woodard, M. Hildreth, and D. Thain, "A Case Study in Preserving a High Energy Physics Application with Parrot," in *Journal of Physics: Conference Series (CHEP 2015)*, 2015.

[5] P. J. Guo and D. R. Engler, "Cde: Using system call interposition to automatically create portable software packages.," in *USENIX Annual Technical Conference*, 2011.

[6] Q. Pham, T. Malik, and I. Foster, "Using provenance for repeatability," in *Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[7] F. Chirigati, D. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility," in *Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance*, 2013.

[8] D. Thain, P. Ivie, and H. Meng, "Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness?," in *12th International Conference on Digital Preservation (iPres)*, 2015.

[9] J. Loeliger, "Collaborating with git," *Linux Magazine, June*, 2006.

[10] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters.," in *OSDI*, vol. 10, pp. 1–8, 2010.

[11] G. R. Brammer, R. W. Crosby, S. J. Matthews, and T. L. Williams, "Paper mâché: Creating dynamic reproducible science," *Procedia Computer Science*, vol. 4, pp. 658–667, 2011.

[12] T. Proebsting and A. M. Warren, "Repeatability and benefaction in computer systems research," 2015.

[13] K. Belhajjame, C. Goble, S. Soiland-Reyes, and D. De Roure, "Fostering scientific workflow preservation through discovery of substitute services," in *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pp. 97–104, IEEE, 2011.

[14] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[15] S. I. Feldman, "Make – A program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[16] H. Meng and D. Thain, "Umbrella: A portable environment creator for reproducible computing on clusters, clouds, and grids," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, (New York, NY, USA), ACM, 2015.

[17] A. Asserson, K. G. Jeffery, and A. Lopatenko, "Cerif: past, present and future: an overview," 2002.

[18] T. McGlynn, G. Fabbiano, A. Accomazzi, A. Smale, R. L. White, T. Donaldson, A. Aloisi, T. Dower, J. M. Mazzerella, R. Ebert, *et al.*, "Providing comprehensive and consistent access to astronomical observatory archive data: the nasa archive model," in *SPIE Astronomical Telescopes+ Instrumentation*, pp. 99100A–99100A, International Society for Optics and Photonics, 2016.

[19] M. van Driel, A. Hutko, L. Krischer, C. Trabant, S. Stähler, and T. Nissen-Meyer, "Syngine: On-demand synthetic seismograms from the iris dmc based on axisem & instaseis," in *EGU General Assembly Conference Abstracts*, vol. 18, p. 8190, 2016.

[20] C. Lynch, "Big data: How do your data grow?," *Nature*, vol. 455, no. 7209, pp. 28–29, 2008.

[21] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue+Python: A framework for scalable scientific ensemble applications," in *Workshop on Python for High Performance and Scientific Computing at SC11*, 2011.

[22] FamilySearch.org, ""United States Census, 1850-1940." Database." Citing NARA microfilm publication T626. Washington, D.C.: National Archives and Records Administration, 2002.

[23] W. Cohen, P. Ravikumar, and S. Fienberg, "A comparison of string metrics for matching names and records," in *Kdd workshop on data cleaning and object consolidation*, vol. 3, pp. 73–78, 2003.

[24] P. Cederqvist, R. Pesch, *et al.*, "Version management with cvs," 1992.

[25] B. Collins-Sussman, "The subversion project: buiding a better cvs," *Linux Journal*, vol. 2002, no. 94, p. 3, 2002.

[26] L. Stanisic, A. Legrand, and V. Danjean, "An effective git and org-mode based workflow for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 61–70, 2015.

[27] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[28] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[29] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, *et al.*, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[30] R. Garcia and M. T. Valente, "Nextflow: Business process meets mapping frameworks,"

[31] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Managing the evolution of dataflows with vistrails," in *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pp. 71–71, IEEE, 2006.

[32] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 525–540, 2014.

[33] D. Thain and M. Livny, "Parrot: An Application Environment for Data-Intensive Computing," *Scalable Computing: Practice and Experience*, vol. 6, no. 3, pp. 9–18, 2005.

[34] A. Vahdat and T. E. Anderson, "Transparent result caching," in *USENIX Annual Technical Conference*, 1998.

[35] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[36] H. Meng and D. Thain, "Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids," in *Workshop on Virtualization Technologies in Distributed Computing (VTDC) at HPDC*, 2015.

[37] D. T. Liu and M. J. Franklin, "Griddb: a data-centric overlay for scientific grids," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 600–611, VLDB Endowment, 2004.

[38] R. Ricci, G. Wong, L. Stoller, K. Webb, J. Duerig, K. Downie, and M. Hibler, "Apt: A platform for repeatable research in computer science," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 100–107, 2015.

[39] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic acids research*, vol. 34, no. suppl 2, pp. W729–W732, 2006.

[40] C. Dietrich and D. Lohmann, "The dataref versuchung: Saving time through better internal repeatability," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 51–60, 2015.

[41] B. Shneiderman, *The New ABCs of Research: Achieving Breakthrough Collaborations*. Oxford University Press, 2016.

[42] J. Myers, M. Hedstrom, D. Akmon, S. Payette, B. A. Plale, I. Kouper, S. McCaulay, R. McDonald, I. Suriarachchi, A. Varadharaju, *et al.*, "Towards sustainable curation and preservation: The sead project's data services approach," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pp. 485–494, IEEE, 2015.