A WORKFLOW MANAGEMENT SYSTEM TO FACILITATE REPRODUCIBILITY OF SCIENTIFIC COMPUTING APPLICATIONS

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Peter Ivie

Douglas Thain, Director

Graduate Program in Computer Science and Engineering

Notre Dame, Indiana

April 2018

A WORKFLOW MANAGEMENT SYSTEM TO FACILITATE REPRODUCIBILITY OF SCIENTIFIC COMPUTING APPLICATIONS

Abstract

by

Peter Ivie

Reproducibility is becoming an increasingly challenging requirement of the scientific process. Compared to more human intensive scientific procedures, it would seem that scientific applications executed on computers could easily produce identical results despite slight changes to hardware, software, or simply timing. However, implicit dependencies on data and execution environment, coupled with ambiguous definitions of identity and equivalence throughout the process, make reproducibility rarely possible. To address this problem, I created 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. With this information PRUNE can directly execute each task. 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. I show that with a minimal amount of overhead, these capabilities can be available for large scale and complex workflows, such as an analysis of high-energy physics data, a bio-informatics application, and processing of U.S. census data. PRUNE also minimizes the cost of collaborative development of computational science.

CONTENTS

FIGURES

TABLES

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.

I thank my advisor, Prof. Douglas Thain, for the time he took to review my work, provide guidance and expertise, and for the stronger grounding in computer science I have obtained through his insights and perspective along the way, and he has been an example of good leadership. And especially for helping me learn the best way to end a talk or presentation.

I am grateful to my dissertation committee, Prof. Kevin Lannon, Prof. Scott Emrich, and Prof. Greg Madey, for their willingness to take time out of their busy schedules to give advice and direction in the completion of this dissertation.

I would never have started this journey without the wise advice from my father Prof. Evan Ivie and the late Gordon B. Hinckley, who both encouraged me to get the most education I could, and to use it to bless the lives of others. I am also grateful to my mother Betty Jo Ivie for providing a safe environment for me, with the freedom to learn.

Victoria Goodrich and Leo McWilliams gave me the opportunity to teach and helped me improve from their examples and guidance. They also provided a much needed boost in my ability to make ends meet, for which I was extremely grateful while I pursued the work summarized in this dissertation. I am also grateful to Patrick Flynn for giving me the opportunity to teach some more this last year.

I thank the CRC for providing fantastic resources for me to perform my research on, and Joyce, Dian, and Ginny who always had answers to the important questions.

I thank Ben Tovar my considerate and kind office mate, and the Cooperative Computing lab, especially Nick Hazekamp, Patrick Donnelly, Charles Zheng, Haiyan Meng, Nate Kremer-Herman, Kyle Sweeney and Tim Shaffer for many laughs, a few meals together, and a log of enlightening discussion.

I am grateful to Shereen and my parents in law, Jo Ellen and Don Ziegler for helping out at home often so that I could focus more on getting this work completed.

I am grateful to my kids, Jessica, Bethany, Miriam, Beck, Thomas and Nora Ivie who made sure that there were always things for me to do at home to give me a daily break from academia.

But most of all, I thank my wife Nichole Ivie. She has always been an example of one who chooses the “better part” in her daily activities, which reminds me that people are more important than things (I forget that a lot). She has given comfort, advice and encouragement at many important times during this endeavor, and I owe her a huge debt for these years I have been working on this dissertation.

CHAPTER 1

INTRODUCTION

Growing concerns about the usefulness of some scientific publications have generated an increased focus on the concept of reproducible computational science [12]. Scientists may disagree on the use of the term reproducibility [63, 82, 54], but few seem to dispute it's importance. I consider computational science to be reproducible if results can be achieved which are equivalent to the original results when an experiment is re-executed where conditions differ only in ways that are not expected to be significant. Ideally, reproducible experiments would contain sufficient details to be effectively incorporated into the research of other scientists, with observations that are correct and can be sufficiently validated. Scientific discovery is often stumbled upon, but must still be separated from transient circumstances.

Knowing that some results can be obtained on demand by re-executing a workflow[1] is an important step towards enabling a skeptic to perform an equivalent experiment and thereby confirm or refute the results. Ideally, the main responsibility to ensure the correctness of publications would rest on the author, with the peer reviewing providing secondary confirmation. However, even without considering the cost of re-executing scientific workflows in an age of widespread big data, the difficulty of even getting set up to re-execute the workflow is too daunting of a task for most peer reviewers to undertake.

[1]A scientific workflow is the collection of data and tasks that are used to execute computational scientific research.

In recent years, serious questions have been raised about the reproducibility of scientific work in general, and scientific computing more specifically. While there appears to be a general sense that most scientific computing is not as easily reproduced as it could be, there is no general agreement on what, precisely, reproducibility entails, and what mechanisms are needed to achieve it.

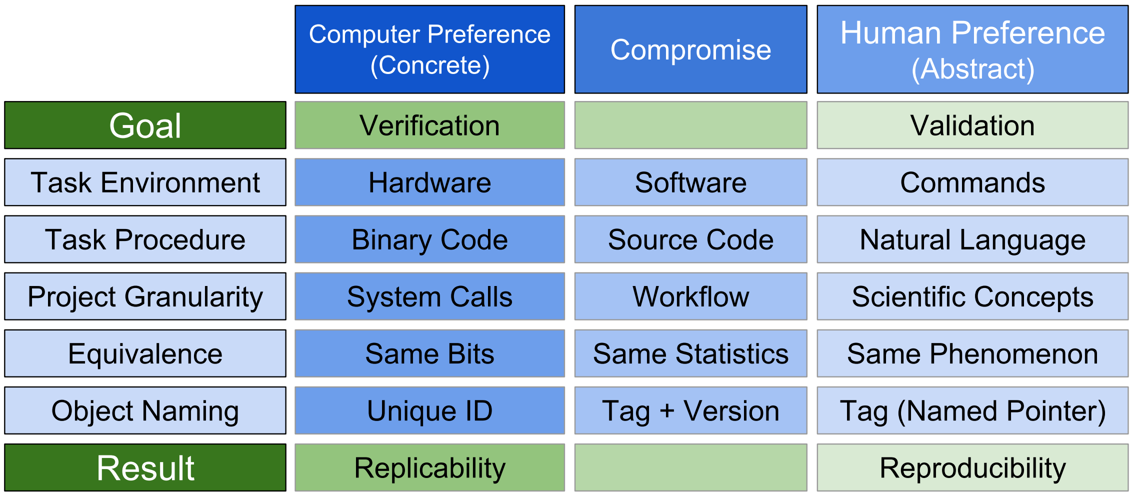
Some argue that this situation has reached crisis proportions:

In the biotech industry, Amgen [17] attempted to confirm the findings in 53 "landmark" articles in cancer research. These attempts were not merely computational, but also involved working in the original labs under the direction of the original authors in attempts to resolve discrepant findings. They only succeeded with 10% of them. In pharmaceuticals, Bayer [156] had slightly better results and were able to verify 21% of 67 different projects. These efforts involved scientific research where the computational resources required were generally low.

In principle, computational experiments should be easy to reproduce, when compared with physical experiments. Assuming that a computer is a deterministic machine, then simply applying the same program to the same inputs on an equivalent architecture should yield equivalent results. Many design principles and recommendations surrounding scientific computing have been encouraged [23] for years. But in practice, the complexity of today's software and hardware makes it surprisingly difficult to even accurately describe the inputs, construct a deterministic program, or identify equivalent hardware. Many of the difficulties stem from a need to simultaneously satisfy the needs of both the computer and a human as summarized in figure 1.1, rather than being able to focus exclusively on one or the other.

1.1 Scope and contributions

In high-performance computing, the overhead of starting and stopping processes is often minimized by allowing long running processes to send intermediate results



	Computer Preference (Concrete)	Compromise	Human Preference (Abstract)
Goal	Verification		Validation
Task Environment	Hardware	Software	Commands
Task Procedure	Binary Code	Source Code	Natural Language
Project Granularity	System Calls	Workflow	Scientific Concepts
Equivalence	Same Bits	Same Statistics	Same Phenomenon
Object Naming	Unique ID	Tag + Version	Tag (Named Pointer)
Result	Replicability		Reproducibility

Figure 1.1. Overall Perspective

Overall Perspective for the Reproducibility of Computational Science

to other processes during run-time. As the duration and scale of such workflows grow, there is an increasing risk that at least one process in the system will fail. This typically forces the entire workflow to fail unless checkpoints of the entire state of the workflow are stored on a regular basis. In the case of a system failure with checkpoints, the workflow can resume from the last successful checkpoint. There can be a significant amount of redundancy between checkpoints, and each checkpoint can slow down the workflow. So a tradeoff has to be made between the overhead of creating more frequent checkpoints as compared to the overhead of stopping all processes when any failures occurs and re-executing from the last checkpoint.

In a high-throughput workflow, the checkpointing is often minimized by breaking the workflow into tasks that do not pass messages during run-time, but provide final results when the task is completed that can be used by other tasks. In essence, checkpointing is done at the task level rather than the workflow level. In these cases the overhead for system failures is limited to the duration of the tasks that failed, and only the tasks need to be restarted, not the entire workflow. However, the overhead of starting and stopping each task is a significant factor.

For workflows where the intermediate data (in passed messages) is relatively small compared to the processing requirements, high-performance computing can be a more efficient choice. However, in the presence of opportunistic or heterogeneous computing resources, the failure rates are often higher, so the cost of restarting the entire workflow is high. And when a workflow has both a lot of data that would need to be check-pointed and uses opportunistic and/or heterogeneous computing resources, the high-performance approach is poorly suited as a solution. In such cases, intermediate data can be stored and transferred as files in any format. Tasks need to fit the compute resources upon which they execute, or they must be able to modify the resource to satisfy their needs. Upon failure, a task can easily be moved to an alternative compute resource without interrupting other running processes.

The computational science we have encountered is more suited to a task-based workflow with no message passing. This eliminates the need to consider race conditions or how message passing affects parallelism at the workflow level. An individual task may utilize parallelism in this model, but it may not communicate with other tasks in real-time.

In terms of reproducibility, heterogenous computing resources come with an inherent chance for results to be affected by the configuration of the underlying resource. With such resources, a scientist should work in collaboration with a system administrator to ensure that resources are appropriate for the desired task, or can be modified in real-time to become appropriate. In addition, the scientist should assume the responsibility to either personally or programmatically verify that the results are correct. Part of the scientific process is to perform an experiment more than once to ensure that the results can be trusted. Computational science should not be absolved of this responsibility based on an assumption that any computing resources, heterogenous or not, are more predictable than a more human dependent experiment (because they might not be).

To further limit the scope of this dissertation, I focus primarily on technical problems in scientific computing, and refer the reader to other publications that address the broader questions of publication habits [166], the role of funding agencies [122, 172], fraud [118, 43], legal issues [173], similar questions [145], and related fields such as computer aided engineering [59]. Elements of scientific discovery that do not involve computers are not the focus of this dissertation.

Computer aided engineering (CAE) is a similar field where computer software is used to help perform engineering analysis tasks, rather than scientific workflows. While not applied to the same domain, some of the concepts used can have applicability to scientific computing, such as executing simulations of a model can be used to evaluate the validity of the model in both CAE [120] and scientific computing [147].

The work presented in this dissertation is intended to expose techniques and approaches that lead to more effective reproducibility. This is done by identifying important properties relating to reproducibility that can be included in a system used for computational science. It is not intended as a new utility that in and of itself will solve the challenges encountered when trying to make scientific results reproducible. As such, most of the evaluations and comparisons will focus on the presence vs. absence of various characteristics. While there may be some comparisons to other workflow management systems, I recognize that some components needed for an effective workflow management system were left out because they were not needed for reproducibility. Specific contributions include:

First, this work identifies and summarizes barriers to reproducibility, a wide range of existing solutions to those problems, and tradeoffs that need to be considered in particular cases.

Second, it asserts the importance of a Preserve First approach to workflow execution, where tasks are preserved and then executed by the system to help ensure that implicit dependencies do not exist. This approach also makes it so that intermediate data can be treated like a cache, since the data can be re-executed if needed.

Third, it introduces a way to name workflow objects based on the derivation tree. This enables the system to uniquely and consistently identify objects shared as a part of a collaboration and to distinguish between them and new objects even in the presence of non-determinism.

Fourth, it demonstrates that in large workflows the overhead of these added efforts can be computationally minimal in comparison to the workflow execution.

Fifth, it quantifies the benefits of the Preserve First and derivation based ID combination as applied to a collaborative arrangement, and estimates the cost of various options for transferring workflow evolutions from one collaborator to another.

1.2 Publications

The following publications contributed to this dissertation as I, and the others involved, wrestled with behaviors that made reproducibility challenging, but also emphasized the need for more reproducible computational science.

DeltaDB: A Scalable Database Design for Time-Varying Schema-Free Data. In IEEE International Congress on Big Data (BigData), 2014. [99] DeltaDB is a log based database with an algebra for performing operations on the log to summarize the records for reports. It reduced what would have been stored in 5TB of snapshots down to 11GB.

Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster. In IEEE Conference on Cluster Computing, 2015 [194] Lobster is a system designed to perform high energy physics on compute resources outside of the dedicated clusters designed for such activities. Shown to work with over 10,000 concurrent cores, it produces throughput comparable with the largest dedicated clusters a part of the LHC infrastructure.

Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness? at the 2015 International Conference on Digital Preservation (iPres) [180] An overview of what challenges exist in attempting to preserve scientific software research for reproducibility. Preserving the mess and encouraging cleanliness are two generic ways to make this happen.

A Case Study in Preserving a High Energy Physics Application with Parrot in the Journal of Physics: Conference Series (JPCS) [137] Preserving a complex high energy physics application such as *Tau-Roast* is challenging. The *Parrot Packaging Tool* can be used to capture a minimum execution environment package for a scientific application. Various technologies can be used to instantiate this package, making it more likely to be reproducible in the future.

An Analysis of Reproducibility and Non- Determinism in HEP Software and ROOT Data. In International Conference on Computing in High Energy and Nuclear Physics, 2016. [105] Generically comparing high energy physics ROOT files generated with the same parameters can lead to an evaluation on the equivalence level of the results. Another approach attempted is to capture all system calls and provide predictable responses to those calls so that application doesn't know anything is different. This achieved deterministic results in some cases.

PRUNE: A Preserving Run Environment for Reproducible Computing. IEEE Conference on e-Science, 2016. [102] In PRUNE, tasks are 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. This tree can be used for reproducibility storage management and collaboration.

1.3 Dissertation overview

So far, this chapter has painted a high level view of what reproducibility is, why it is important and how effective current strategies are at satisfying the need for reproducibility. Then the scope and contributions of this work were presented and papers that led to the completion of this work were summarized.

CHAPTER 2: THE REPRODUCIBILITY PROBLEM DEFINED includes detailed definitions not only for reproducibility, but also for terms surrounding reproducibility. It also includes explanations for why it is difficult to actualize reproducibility for single commands not only because of implicit dependencies on the environment, but also due to a mismatch between the needs of the computer and the needs of the user. And then additional challenges related to workflows are discussed, such as the tradeoffs between finely and coarsely granulated preservation and difficulties in the execution of the workflow.

CHAPTER 3: RELATED WORK addresses the topic of object naming and describes categories of existing choices for executing workflows each with example systems as related work. Object naming is both challenging and important because it is what computers and humans both use to uniquely identify objects in the workflow so that changes to a workflow can be distinguished from previously included objects. The categories of existing solutions range from tracing all system calls initiated during the execution of a workflow to forcing the use of dedicated clusters in a walled garden to ensure no implicit dependencies exist.

CHAPTER 4: PRUNE OVERVIEW introduces a few ideas and practices designed to encourage the use and development of reproducibility focused workflow management systems. The ideology of Preserve First eliminates the chance of forgetting to preserve the workflow or of disparities between provenance and practice. It also helps avoid hidden implicit dependencies on the environment. The components of PRUNE are identified with the idea that the components make up an ever growing tree of immutable tasks that describe every evolution of the workflow without having to store all intermediate files. The reasoning behind combining a context based identifier with a derivation based identifier for each object is then explained.

CHAPTER 5: PRUNE SINGLE USER EVALUATION lists specific technologies that are chosen for the various components of Prune. A simple merge sort workflow is shown with the equivalent operations described as a Prune workflow. Examples of the underlying immutable objects are included. An workflow using U.S. Census data is used as a proof of concept and reproducibility relevant metrics are recorded during execution of the workflow. Storage savings for re-executing the workflow after changes a various stages compared to creating a new folder are presented. Overhead is shown to be negligible compared to the execution of workflow tasks, and a quota system is put into action to keep storage consumption within specific bounds. Prune is then applied to both bioinformatics and high energy physics workflows and the results are used to confirm the overhead measurements. However, a few problems that arose are described with solutions to those problems.

CHAPTER 6: PRUNE COLLABORATION shows how the data stored in Prune is used to estimate the financial and temporal cost of attempting to minimize re-execution of tasks on a collaborators system vs. attempting to minimize the transfer of intermediate files over a network. Prune to obtain and transfer either the intermediate results or task information on a task by task basis. That coupled with the ability to detect matching object using the content and derivation based identifiers allows evolutions to a workflow to be transferred to collaborators in much more efficient modes. These modes can also be chosen after the fact, which is helpful because the optimal mode can change depending on the circumstances. The method for estimating network and compute costs are detailed and applied to all modes for the bioinformatics and high energy physics workflows to show the benefits of the newly available modes.

CHAPTER 7: CONCLUSION summarizes the dissertation and reflects on the successes and failures that were encountered. Future directions that could further advance the convenience and viability of making scientific results reproducible are then illustrated. Followed by a section on how to reproduce the results of the workflows used in this paper. I will note upfront that ssues were encountered with satisfying data dependencies with each workflow both from technical and legal perspectives. Obtaining permission to publicly share these workflows in their entirety turned out to be less successful than addressing the technical challenges. I share all files and data that I was permitted to. A link to this data is found in section 7.5.

CHAPTER 2

THE REPRODUCIBILITY PROBLEM DEFINED

2.1 Perspectives on reproducibility

It is commonly expressed that reproducibility of computational science is a desirable quality, and that there is a need to move the beyond the printed paper as a means of communicating results. [151, 170]

> An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures. [29]

This type of information can get very complex and detailed very quickly, but from a user perspective it can be easier than it sounds with the proper tools and mindset.

> It is a big chore for one researcher to reproduce the analysis and computational results of another [...] I discovered that this problem has a simple technological solution: illustrations (figures) in a technical document are made by programs and command scripts that along with required data should be linked to the document itself [...] This is hardly any extra work for the author, but it makes the document much more valuable to readers who possess the document in electronic form because they are able to track down the computations that lead to the illustrations. [39]

Researchers at the University of Arizona [157] considered the repeatability of 402 ACM papers published in computer systems conferences. In this work, minimal repeatability was defined simply as the ability to download and build the source code within a reasonable amount of time. They were able to build 32.3% of them within 30 minutes. 15.9% more took over 30 minutes and 5.7% more with additional but reasonable effort. The code failed to build in 2.2% of the cases, and the authors declined to provide code in 7.5% of the cases. 36.3% of the authors never responded to requests for the code. The subjects of the study were invited to post corrections or addenda to the material, and the responses resulted in a wide variety of strong opinions about the procedure.

All these numbers are definitely dismal, but is it really a crisis? Future computational science is likely to become even more complex [161] and resource intensive, making reproducibility even more challenging. To understand why this is such a problem, I will expound upon some of the reasons for, and benefits that come from making research reproducible.

2.1.1 How is reproducibility defined?

A wide variety of authors have defined reproducibility and related terms in somewhat different ways. [106, 81, 190, 1, 195, 132, 171, 147] Although complete consensus has not been achieved on these terms, I will use them in the following way:

To **replicate** an experiment [54] is to carry out exactly the same task as the original researcher, with the expectation that the result will be the same. In scientific computing, exact replication would constitute building the same program with the same compiler running on the same hardware and the same operating system as the original. Obviously, it may be difficult or impossible to replicate every last detail. Seemingly innocuous details (like the system time [104]) may affect the final result.

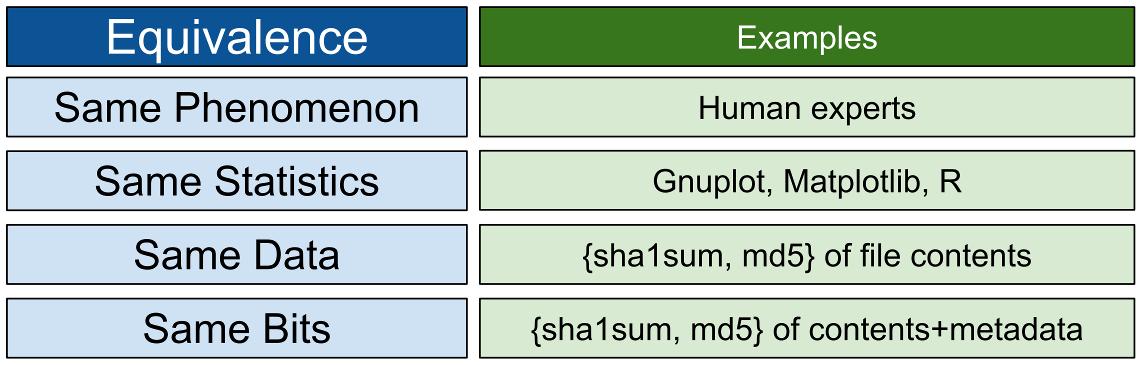
To **reproduce** an experiment [195] is to carry out tasks that are equivalent in substance to the original, but may differ in ways that are not expected to be significant to the final result. In scientific computing, these differences could range from minor to sweeping. One attempt to reproduce might run the same version of the software on a new version of an operating system, while another attempt to reproduce might involve writing a new piece of software that implements the same algorithm.

The terms **verify** and **validate** are often used interchangeably. In the broader literature [21, 147], they are used to indicate technical correctness and fitness for purpose, respectively. In the context of scientific computing reproducibility, I define **verification** as the task of replicating an experiment to see if it produces the claimed output, while **validation** is the task of evaluating a result to see if the author's conclusions are warranted. One experiment **corroborates** another when they reach the same overall conclusions.

When the components underlying an experiment are easily named and shared, it becomes possible to make use of them in other contexts. This is known as **variation** or **reuse** or **extension**.

The term **provenance** is used broadly to describe retrospectively the many potential sources of input or variation to a program. For example, when a program is run in a distributed system, it may be desirable to record the incidental details of the machine on which it ran (architecture, operating system, system time) in case those details are later found to be significant. Or, if a program B consumes input data X that was the output of a previous program A, then it may be fruitful to record that $A \rightarrow X \rightarrow B$ to note that the output of B originally depended on the output of A. [112, 78, 31, 175, 60, 162, 119, 167]

A **deterministic** program always produces the same result when run with the same input in the same computing environment. Some programs are non-deterministic by design: for example, a Monte Carlo simulation uses a random number generator to evaluate a function with randomly chosen inputs. Other programs are non-deterministic by accident: concurrency, operating system services, or the vagaries of floating point math may all introduce differences where they are not desired [58]. For example, even a Monte Carlo simulation with a fixed seed can produce non-deterministic results [103]. There may be ways to avoid some sources of non-determinism, but it is difficult (if not impossible) to avoid all of them. Efforts can



Equivalence	Examples
Same Phenomenon	Human experts
Same Statistics	Gnuplot, Matplotlib, R
Same Data	{sha1sum, md5} of file contents
Same Bits	{sha1sum, md5} of contents+metadata

Figure 2.1. Equivalence levels and Examples

Equivalence can be gauged with different methods depending on the desired focus. Discussed in detail in section 2.1.1.1

be made to detect [16], or mitigate such behavior, even at large scales [34], but there is still a need to support better reproducibility at the system level.

2.1.1.1 Equivalence

I must also be careful to define what constitutes the "same result" when comparing two experiments (see figure 2.1):

- Two experiments could produce the **exact same bits**.
- Two experiments could produce the **same data** in the sense that they encode the same numeric contents, but differ in some irrelevant detail. For example, an output file might incidentally contain the system time and the name of the user who ran the program.
- Two experiments could produce **statistically equivalent results**, in that the numeric values are different, but they both conform to the same statistical distribution, modulo some error tolerance.
- Two experiments could observe the **same phenomenon** but not the same data.

These distinctions have an important bearing on whether a result can be verified automatically. If equivalence is defined by the same bits or the same data, then simple technical tools can be used to perform the comparisons. Evaluating statistically equivalent results requires a domain-specific tool, while comparing phenomena requires a human with domain-specific knowledge. Consequently, it is desirable to achieve reproducibility at the lowest level at which it is feasible, so that verification can be performed automatically.

A part of reproducibility is communicating methods and intent [54], in addition to communicating how to obtain identical results. When tools for logical equivalence are not available, the burden of comparison rests on the scientist.

When the computer itself is the object of study, then performance or resource consumption may be the primary result. In other cases, issues of performance are relevant in terms of cost and/or convenience, but are not the focus of their research. Various systems exist which are both appropriate for computational science and have a focus on maximizing, measuring, or repeating performance goals. [109, 107, 33, 116, 169, 53] *Repeatability* can refer to the ability to get the same performance [1] in the presence of changing conditions in the underlying system. I will assume that for the general case a focus more on reproducibility has benefits that outweigh the advantages of a focus more on performance.

Topics discussed in other works, but not addressed in the paper include best practices beyond the technical aspects [171] and roles of not only the scientist, but the funding agency and the journal editor [195].

Several authors [152, 132] have presented these reproducibility concepts as a spectrum starting with replicability by a single researcher as the minimum level of scientific integrity, and increasing through verification, reproduction, validation, extension, and reuse by many researchers. Each stage requires a greater amount of work but has increasing value to the community at large.

2.1.2 Why should computing be reproducible?

There are a variety of reasons underlying the need for reproducibility:

To verify (or disprove) other's results. [140] argues that the basic function of a paper is to both announce some result and convince the reader that the result is correct. However, [98] claims that the majority of published research findings are false, due to small sample sizes, statistical noise, confirmation bias, and publication bias. In most fields, peer-reviews serve to evaluate whether the work, as described, is sound, significant, and interesting. With rare exceptions, reviewers do not have the time, inclination, or skills to perform and verify the work described in a paper, particularly if it requires access to unusual or expensive methods and facilities.

However, scientific computing has the unique advantage that any computational activity is potentially reproducible, given the same code and input data and execution on a compatible machine.

This has given rise to the concept of "reproducible research" [38] or an "executable paper" [24, 32] in which the source code and data used to reach a conclusion are coupled and distributed with the paper itself. In principle, this should allow the reviewer and the reader to carry out the same action and evaluate the conclusions.

This concept has been offered as part of a number of special issues and efforts, but has not been accepted broadly by research communities as of this writing. This may be due to the fact that peer review considers more broadly the novelty, significance and correctness of a work. For example, [121] notes that merely re-running the same code does not guarantee that the research results are correct. There are also many cases where accessibility of the code and data is not sufficient: the results may require access to specialized or high performance hardware, and may still require a large amount of time or other resources to complete.

To verify one's own results. In practice, I have encountered relatively few researchers who wish to actively develop an adversarial relationship with others by disproving their work. However, some have argued that a healthy distrust of one's own work should drive reproducibility:

> We do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them. Only by such repetitions can we convince ourselves that we are not dealing with a mere isolated coincidence, but with events which, on account of their regularity and reproducibility, are in principle intersubjectively testable. [155]

To improve one's own productivity. Some researchers perceive that efforts to make their research reproducible will result in decreased productivity [13] as effort is shifted towards technologies instead of their primary work. Others have argued the opposite. For example, Jon Claerbout (who coined the term "reproducible research") made the following statement after many years working towards that end:

> It takes some effort to organize your research to be reproducible. We found that although the effort seems to be directed to helping other people stand up on your shoulders, the principal beneficiary is generally the author herself. This is because time turns each one of us into another person, and by making effort to communicate with strangers, we help ourselves to communicate with our future selves. [37]

To enable extension by others. Frequently, one researcher may wish to build upon another's work positively by augmenting it or evaluating it against a new dataset or situation. This is easier said than done. Even when two researchers working contemporaneously can share notes and advice, moving a code from one institution to another can take months before the same setup is "working" in the new context. [79] It becomes even harder when the original researcher is no longer available: they may have graduated, have taken a new job, or have died. If the goal is to enable reproducibility on the scale of 10-20 years [29], significant care is needed to record all the necessary details. A focus on the human side of scientific computing [90] can also make it so that others can understand and incorporate published research into their future efforts.

To survive technology evolution. Many research codes depend on a large number of sub-components like libraries, compilers, runtime systems that must be independently installed, configured, and tested on a given operating system. A large computing site must occasionally go through an upgrade cycle to activate new hardware, change system facilities, or upgrade the operating system. These are frequently not backwards-compatible changes, and so all the supporting components must be re-built to accommodate the new environment. The unsuspecting user may face an enormous amount of work to reconstruct all these components. Reproducibility techniques can assist in recreating the dependency tree (and testing it) after a major upgrade, hopefully for years to come.

To enable community maintenance and support. A code developed by a single researcher typically has a short productive lifetime. Keeping the code working on multiple platforms and relevant to current research trends takes time, and eventually the researcher moves on to other activities, leaving "orphan" code behind. However, if reproducibility techniques make it easy to execute a code in many different contexts, responsibility for the code can be held by a larger community. When multiple stakeholders are familiar with the code, technical problems are more easily solved. Even automated techniques can be employed to perform maintenance on an experiment when the research is adequately reproducible. [66] By publishing reproducible research, ownership of maintenance is effectively transferred to the community [69] level. This allows the publishing scientist(s) to focus more on future work than previous work.

In summary, computational scientists are often encouraged to make their research reproducible so that that other scientists can **verify**, **reproduce**, and **extend** their computational experiments, but there may be personal benefits also.

2.2 Technical barriers to replicating a single command

Let us begin by considering the technical challenges of reproducing just a single command. Suppose that an end user connects to a university computing facility and enters the following command:

```
do_science.sh lab.dat model.csv 8 plot.jpg > stdout.csv
```

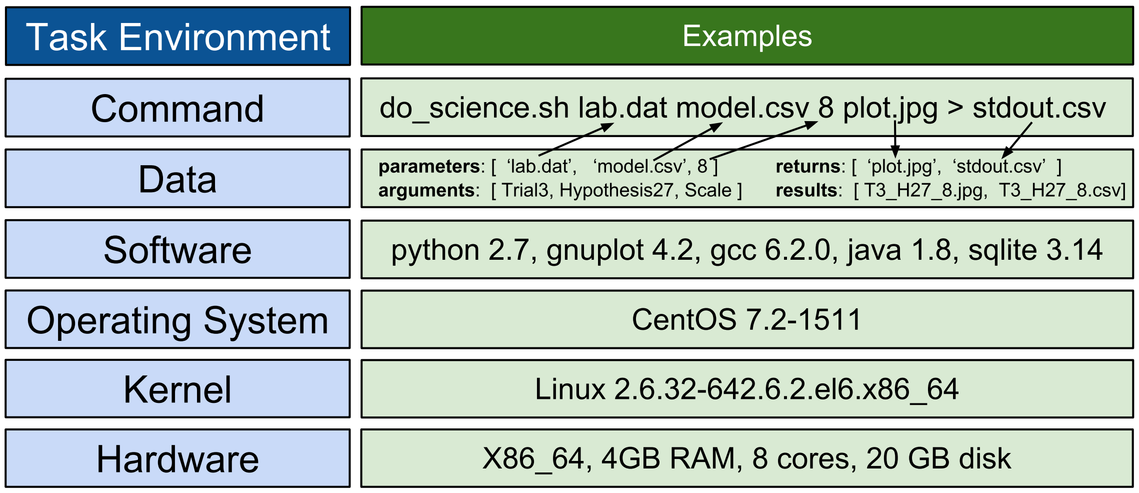
From the user's perspective, the command string is the only visible evidence of the program. The command itself provides the most superficial form of replicability: by entering the exact same command into the same terminal later that day, there is a good chance that exactly the same outputs will be produced.

However, there is no guarantee that the same command applied by a different user on the same machine, much less a different user on a different machine, will succeed at all, much less produce the same output. This is because the command string replies on a large number of dependencies in the form of hardware, software, and data, as shown in Figure 2.2 Some of these dependencies are explicitly mentioned on the command line (like the file `model.csv`) while others are implicitly provided by the system.

The following sections cover such environmental challenges, in addition to challenges connected with abstractions, run-time anomalies, verification of results, and a discussion about whether source or binary code should be the target of preservation.

2.2.1 Environment

I define the *environment* as both the system resources and the domain methods used to perform the computational side of scientific research. The scope at which systems preserve or describe the environment varies widely. Research is more likely to be reproducible when all levels of the environment are preserved or at least identified.



Task Environment	Examples
Command	do_science.sh lab.dat model.csv 8 plot.jpg > stdout.csv
Data	**parameters**: ['lab.dat', 'model.csv', 8] **returns**: ['plot.jpg', 'stdout.csv'] **arguments**: [Trial3, Hypothesis27, Scale] **results**: [T3_H27_8.jpg, T3_H27_8.csv]
Software	python 2.7, gnuplot 4.2, gcc 6.2.0, java 1.8, sqlite 3.14
Operating System	CentOS 7.2-1511
Kernel	Linux 2.6.32-642.6.2.el6.x86_64
Hardware	X86_64, 4GB RAM, 8 cores, 20 GB disk

Figure 2.2. Task Environment Levels and Examples

Examples of environmental components at various levels. More details in sections 2.2.1-2.2.1.6.

2.2.1.1 Command scope

A 'do_science.sh' script can contain all information needed to replicate the experiment. However, this approach can mask valuable information from the user, making it difficult for another scientist to extend the research in order to explore or build on the experiment. Requiring additional parameters that are handled by the script may seem to overcomplicate an experiment, but doing so communicates those decisions made by the original researcher which are deemed most relevant. *Parameterization* can be an important tool for extension. If crafted carefully, parameters can give both the original researcher and collaborators the ability to easily explore the parameter space to gain confidence in the validity of the research. Figure 2.2 starts with an example of a parameterized command at the top. Parameters can be numbers or strings in the command scope, but in the data scope (section 2.2.1.2) the parameters can refer to files.

2.2.1.2 Data scope

Most scientific research involves some kind of input or starting data in addition to the final generated results. For reproducibility purposes I will mostly consider this data to be in the form of files, but it could come in the form of Unix standard output, literal parameters, etc.

This data could involve network dependencies which can make reproducibility more challenging. For replicability, these dependencies might be satisfied by recording the data retrieved over the network and then simulating the network in subsequent runs of the experiment. However, for other changing factors (especially to the domain methods and original data) this approach becomes less likely to capture the network resource adequately. It might be necessary to capture and transfer the entire database from behind the network resource to ensure that the workflow will still be reproducible in the presence of changes.

Authorization issues are another common challenge with data, either from a security or privacy perspective. Authorization keys are sometimes kept in pre-determined file or location, and certain data files may include private information. Such information should normally be excluded from a publication, but without it, the research is not reproducible. Accepting this information as an input parameter can identify a need for the information without publishing the sensitive data, enhancing reproducibility. This could be done in the form of a template.

Templates are programs that expect input parameters for dynamically specifying the data the program is supposed to operate on. They can be helpful for the original researcher when the input data is updated incrementally or to compare different datasets. This form of parameterization is also useful, for extensible reproducibility, when other researchers have different original data and want to use that to evaluate the domain methods with.

Using a single command with lots of parameters can get confusing for large experiments. In the interest of extensible reproducibility, and to simplify things for the original researcher, it is advisable to break down the experiment into smaller parts and organize them in a *workflow* [47, 185, 124]. Different levels at which a workflow can be composed are discussed in the section 2.3.1.

Descriptions that involve both the command and data scopes are similar to *functions* (consider figure 2.2), where an external name needs to be given to all *arguments* (for the inputs) and the *results* (for the outputs). The internal names used for that data inside the function are considered *parameters* (for the inputs) and *returns* (for the outputs). This separation between internal and external names can be important for workflows because sometimes legacy software expects input from fixed filenames and uses fixed locations for generated files. In these cases, and when a template is used multiple times as part of a workflow, the naming of data files can become complicated from a workflow perspective. More on the issue of naming can be found in section 3.1.

2.2.1.3 Software scope

Part of the challenge with software dependencies is that different versions of a particular software program or library are mostly compatible, but always include some changes. Even newer versions of software that claim to be backwards compatible may have unintended differences that can affect reproducibility. Software names can be ambiguously used without version numbers for ease of use by the scientists, but for reproducibility, all necessary software should be uniquely identified to ensure consistent behavior (more in section 3.1). *Packages* can be used to make this process easier and can include any combination of elements from the command, data, and software scopes, as shown in figure 2.2.

Also, scientists generally prefer to focus on their science and care less about lower level resource management [88] handled by system administrators. They are more interested in the destination than the journey, in part due to the exploratory nature of scientific research. Scientific research is more often marked by a desire for *infrastructure independence* than with an emphasis on *benchmarks* and how quickly a system can execute a workflow. Readers interested in infrastructure and performance can find more information about system-centric workflow systems and distributed test beds in [28].

If a given research experiment is infrastructure independent, it will be more easily reproducible. However, scientists sometimes decide they need a little more control over the system resources. At this point, the boundary between system administrator and domain scientist comes into question, which in turn makes the responsibility for reproducibility more ambiguous. What software should be provided by system administrators versus how much control should scientists have in setting up their own domain specific software on generic computing resources?

From a reproducibility perspective, the scientist is generally unaware of modifications made by the system administrators. Even the system administrator may not put a priority on tracking all aspects of the system's configuration. The line between domain methods and system resources also varies between domains, making it difficult to come up with a reproducibility solution appropriate for all domains.

2.2.1.4 Operating System scope

Occasionally the scientist will want a different version of the operating system than is available on provided resources, but this is often beyond their control. *Containers* such as Docker [138], Kubernetes [27], and Mesos [91] have emerged to address the need for users to have easier access to specific versions of operating systems. Container popularity is evidence that there is a need for this level of specificity in a description of how computational science is performed.

Containers also depend on a specific kernel in order to work. This means that by specifying an appropriate container for computation research, the kernel is also specified. But it also means that a container depending on one kernel cannot be initiated on a computer running a different kernel.

2.2.1.5 Kernel scope

Virtual machine images can satisfy the need for all system software in addition to providing virtual support for some hardware requirements of computational research. However, the overhead of instantiating virtual machines can be prohibitively high, especially for short running tasks if a virtual machine is instantiated for each task. This can cause domain scientists to gear the workflow towards performance, with larger tasks, rather than extensible reproducibility, with logically sized tasks. These larger tasks are likely to be more obscure to other scientists than those designed with logical domain science granularity (see section 2.3.1).

2.2.1.6 Hardware scope

For workflows executed on a single machine with no network dependencies, the computer itself could be preserved in a museum or library as a part of reproducibility [144], but this is clearly not feasible especially for large data sets analyzed on distributed systems.

Another aspect of the hardware scope not normally handled at the other scopes, is the concept of finite resources. A scientist focused on domain specific issues can neglect to preserve the memory, cpu, disk, and perhaps network resources needed for a workflow and it's parts. This information is also difficult for system administrators to track because additional resources are required to monitor the resources used by a workflow. Both groups are disincentivized to preserve information this detailed, but it may be difficult for another scientist to reproduce the research without this information, and will almost certainly make reproducing the results less efficient.

2.2.2 Source code or binary code?

When attempting to preserve a workflow for reproducibility, decisions must be made about when to preserve source code and when to preserve the binary code that was generated by compiling the source code.

Considering reproducibility, source code seems like the obvious choice. But the compiler then becomes an important part of the workflow and measures need to be taken to ensure that the compiler is available and can execute on future compute resources. Some time in the future it might be easier to find a modern replacement for a compiler than it would be to find a modern way to execute the the binary code. At some point in this recursive problem, assumptions may need to be made about what will be available in the future. So just preserving the compiler doesn't completely ensure reproducibility in the long term.

Source code more easily communicates to colleagues what each task is doing and lends itself more easily to modification by those other scientists, making it a better choice for reproducibility. In fact, important information about the science is embedded in the source code, whether through comments, structure or naming. This information is ignored by the computer as irrelevant, but could be considered a collection of facts that help support the claims made in the published research. Those facts can add knowledge about a workflow in general and also software components individually, especially when the components are novel in the scientific domain.

However, in order to use the source code, it must be converted to binary code by a compiler which can take a significant amount of time. If the compilation is done on each compute node in a distributed system, the compile time can add significant costs to executing the workflow. This makes preserving the binary code a better choice if replicability is all that is needed, not reproducibility.

However, there is no need for a rule that states one option must be chosen at the expense of the other. If both forms are preserved, the preferred option can be chosen

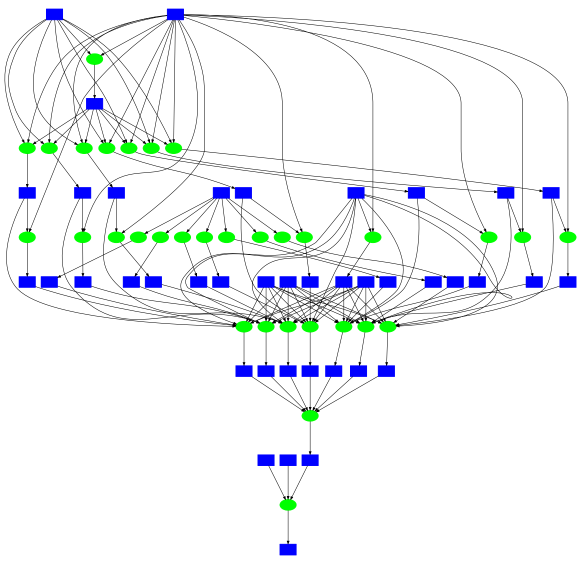


Figure 2.3. Visualization of a Genome Analysis Workflow

Data or information (indicated by squares) and actions or processes (indicated by circles) are connected by a derivation tree, where the root is source data, and any subset of the remaining generated data components could be considered the results.

later on, when the needs are more clear. For data-intensive workflows, preserving the source code, compiler and the binary code is unlikely to make an unreasonable addition to the total storage or communication costs.

Having both options also makes it more likely that one or the other will provide the level of replicability/reproducibility needed at some future date on some future compute resources. Indeed, having more than 2 levels of abstraction where the highest level describes the task or workflow in very broad terms might allow for a task or workflow to be re-created in a situation when neither the source code nor the binary code can be executed.

2.3 Technical barriers to reproducing a workflow

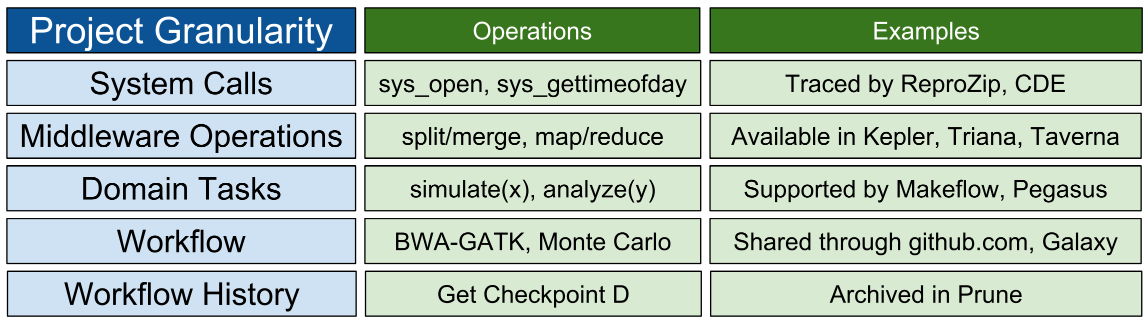
The Workflow Management Coalition [93] defines a workflow as the computerized facilitation or automation of a business process, in whole or part. For a scientific workflow, the business process is an experiment with a focus on scientific discovery, innovation, and/or invention. A set of procedural rules describe the processing and generation of documents or information. I also focus specifically on solutions for scientific workflows which are data-intensive [167]. While there may be some scientific computing efforts that would not be considered workflows, there is value [168] in applying workflow concepts wherever computers are used as a part of the scientific research process.

A visualization of a workflow for Genome Analysis is shown in figure 2.3. Data or information (indicated by squares) and actions or processes (indicated by circles) are connected by a derivation tree, where the data at the root is source data, and any subset of the remaining generated data components can be considered the results.

> The workflow programming paradigm is seen as a means of managing the complexity in defining the analysis, executing the necessary computations on distributed resources, collecting information about the analysis results, and providing means to record and reproduce the scientific analysis. [177]

> The workflow programming paradigm is seen as a means of managing the complexity in defining the analysis, executing the necessary computations on distributed resources, collecting information about the analysis results, and providing means to record and reproduce the scientific analysis. [177]

A workflow management system [185] provides a bridge between the work people do and the work the computers do. They are important [130] in making computational science more convenient for scientists, but they can also help improve reproducibility. Unfortunately, there are many workflow management systems, and there is no common or accepted format or procedure by which a workflow should be recorded or shared.



Project Granularity	Operations	Examples
System Calls	sys_open, sys_gettimeofday	Traced by ReproZip, CDE
Middleware Operations	split/merge, map/reduce	Available in Kepler, Triana, Taverna
Domain Tasks	simulate(x), analyze(y)	Supported by Makeflow, Pegasus
Workflow	BWA-GATK, Monte Carlo	Shared through github.com, Galaxy
Workflow History	Get Checkpoint D	Archived in Prune

Figure 2.4. Project Granularity Levels and Examples

Computations can and are preserved at various granularities. Each option has certain benefits and shortcomings (see sections 2.3.1.1-2.3.1.4).

2.3.1 Project granularity

The size of each step in a scientific workflow can as small as a system call, or as large as a single command that performs a complex system of hidden computations. For extensible reproducibility the granularity should be domain specific and chosen by a scientist to reflect the granularity of the scientific concepts involved. Many systems impose restrictions on the granularity, making it more difficult to use the workflow as a way to communicate the details of the research between scientists. However, those restrictions can also make it easier to use and more effective for a specific class of user. [44] Each of the levels of granularity shown in figure 2.4 have advantages, but also disadvantages which can be a barrier to reproducibility. In addition, the existence of so many options can be a barrier, as a scientist accustomed to using one level, may have difficult adapting to another.

2.3.1.1 Granularity: system calls

One simple solution for preserving a workflow is to trace [35, 153, 154] and log all system calls (such as **sys_open, sys_stat, sys_gettimeofday, sys_getuid**, etc.) during the execution of a workflow. This system call log can be used to identify which files were actually used for the workflow. The remaining files can be excluded from a package or image that contains the environment the workflow is to be executed in.

While this approach can provide replication for deterministic workflows, it may not work if the workflow is modified or if part of the workflow is non-deterministic, since different library files might be needed in a subsequent execution.

And even after eliminating excess files for a given package, duplicates will exist across packages that are only slightly different from each. This becomes a storage problem as a workflow evolves through progressive iterations.

Sharing a workflow at this level can definitely provide replicability, but it is difficult for a colleague to understand what the workflow does. The log itself can be valuable for a very experienced user, but for a domain scientist, it is probably only useful as a last resort when other more coarsely organized workflow descriptions fail.

2.3.1.2 Granularity: middleware operations

Another solution is to allow a middleware designer to choose which logical operations can be applied to data. More complex operations must be created by the scientist composing new operations using a combination of provided logical operations, such as **merge/split** operations, or **map/reduce** operations.

Kepler [4] is an extensible system for the design and execution of scientific workflows with a focus on GUI presentation. *Directors* are execution models with plug-ins which manage *actors* or tasks (sources, sinks, transformers, analytical steps, compute steps). The Triana [176] workflow environment is designed for managing distributed applications (P2P, Grid, middleware toolkits). It works at a web services level (GUI for connecting tasks), but more complex services can be built on the ones provided by the system. Taverna [97] is a tool for building and running workflows which is also based on web services.

While better than working with system calls, these low level operations are typically less abstract than a domain scientist would prefer to deal with and make it difficult to understand the science without an abstraction at higher levels.

2.3.1.3 Granularity: domain tasks

Alternatively, a scientist can choose what happens in a task. As an example, one task could be designed to **simulate** events, while another **analyzes** them. Parameters on the task could include things like the numberx of events, their type, and a seed. Using abstractions at a domain level [46] makes it easier for other users to understand the workflow when it is shared with them. This flexibility makes domain tasks a good granularity for extensible reproducibility.

Problem Solving Environments or PSEs provide all the computational facilities necessary to solve a target class of problem. Users can use the language of the target class of problems, while the PSE fills in the details with appropriate hardware and software. The user does not need not have specialized knowledge of the underlying hardware or software. [111, 76, 75]

In effect, they separate problems and solutions from the hardware and software that carries out the solution. This helps the user focus on their domain [23], and could also support the use of advancements in hardware and software without additional effort from the user.

Such systems are particularly well suited for education [179, 143] because they allow the focus to be on concepts, not programming. PSEs can also be used in distributed computing, allowing a researcher to access more computing power with less effort. [74]

A related topic called Computer Assisted Engineering (CAE) [2] is applied specifically to engineering, but can have some applicability to scientific computing.

2.3.1.4 Granularity: workflow and history

A scientist can group tasks together into a specific workflow. This level of abstraction can identify the results that are used for a publication, but is more effective for reproducibility when it includes components at lower levels of granularity. There is also significant information to be found in the evolution of workflows (see section 2.3.1.7). Comparing workflows after small changes or comparing between researchers, can uncover portions that are similar [101] or identical. This enhances reproducibility by allowing scientists to focus on differences rather than comparing the identical portions.

However, a scientist can't keep everything forever. The archiving of knowledge is generally done by libraries who keep records through books. Their influence is expanding into the digital realm, but their exact role is still unsure. I generally assume that once a decision has been made to keep something, it is easy to keep it forever. However, libraries constantly have to make decisions about what to keep and what to discard, and deal with such issues as copyright and access. Another publication [160] provides more information about such issues and how they apply to research data.

2.3.1.5 Abstract vs. concrete

The chosen method for describing a workflow can fall along a spectrum between abstract and concrete, or can incorporate both abstract and concrete elements which are connected together. Abstractions can allow the scientist to work with high level concepts which can be later compiled into more concrete components [128]. The more abstract the workflow description is, the easier it is for scientists, making it more likely to be extensibly reproducible. High level visualizations can be used as a guiding tool for solving specific problems [22].

But at the same time, an abstraction can leave room for unexpected behavior, putting even replicability at risk. For example, a graphical user interface can be bad for reproducibility because unexpected behaviors may go unnoticed.

At the same time, a more abstract workflow might be chosen because it can be more adaptive to changes in the runtime environment. Especially with the exploratory nature of scientific computing, the exact number of computations needed in a particular step may be unknown in advance. In such situations, the importance of verifying results becomes even more significant.

2.3.1.6 Data management

Data set sizes are increasing in all fields involving computational science. Maybe not on the order of petabytes such as with high energy physics, but usually large enough to merit distributed systems for processing and sometimes even simply storage. Most version control systems for managing source code are designed to fit on a single machine. In addition, with a focus on managing lines of code, data is typically treated as an inscrutable blob of bytes.

A workflow is of no use without the data it depends on, but with big data, the data must often be kept separate from information on the workflow with such version control systems. The connection between the two can be the first component to breakdown when attempting to reproduce the workflow. With a little bit of personification [83], some have come to accept that data needs more public attention.

In addition to large input datasets, the data generated by the workflow might be too large to share with others practically or efficiently. However, without the final datasets it is difficult for a collaborator to verify the results of an attempt to reproduce a workflow.

Data provenance refers to the derivation history of a data product starting from it's original sources. Derivation steps could include database queries, command line strings, executable files, or other similar actions eventually producing some data result. Specific examples of ways to preserve provenance for computational tasks are available [72]. But the exact metadata recorded varies widely depending both on the requirements of the system and the purpose for the provenance. In fact, a full survey [167] is dedicated to this topic. Data provenance is not always include sufficient detail to re-execute the history of operations.

2.3.1.7 Evolution

Designing a scientific workflow is an evolutionary process. Recording the evolution can be valuable in communicating the validity of research. If another scientist wants to try changing some parameter in the workflow, the evolution history might reveal that the path has already been tried. In addition, seeing the various workflow attempts can help to convince other scientists that sufficient attention has been given to the parameter space surrounding the final research. The absence of this data is a missed opportunity for more extensible reproducibility.

This evolutionary workflow data could also be useful in bi-directional research sharing. If multiple research groups are working in a similar vein, they can benefit from each others' efforts. However, this type of data could easily grow beyond the scientist's capacity to preserve the workflow evolution data without sorting through it to find the minimum data that must be recorded, so that derived data does not have to be stored indefinitely.

2.3.2 Workflow execution

Certain methods for executing the workflow can introduce problems with both reproducibility and in the validity of the scientific research itself. Research is vulnerable to *measurement bias* in many different forms [142] especially when the scientist can execute code manually.

Automating the execution of all parts of the workflow can resolve some of the measurement bias, while at the same time saving time and being more convenient for the researcher. However, any model used to automate a workflow can be restrictive to the researcher. This could be because of a concern for performance or, more likely, the automation language is prohibitively complex for the scientist. Whatever the reason, scientists are temptated to execute at least some of their workflow manually, making it hard to maintain a reproducible representation of the workflow.

Even an automated workflow can run into isolation issues where data is available on the original computing resources, but is not available in the workflow description. For example, either the scientist or system administrator may be unaware of the dependency on some resource. In this case, more isolation between the workflow and the user space would help with reproducibility because such problems would have to be resolved before the experiment could complete execution in the first place.

In other cases, the resource could also be intentionally unavailable based on proprietary or privacy restrictions in place. In this situation, the isolation between the workflow and the user space can actually get in the way of reproducibility.

Also, the size of a resource could make it impractical for inclusion in the workflow, or for performance reasons, the resource could have been made available on a site specific resource such as a shared or distributed filesystem. In such cases the resource should be identified in the workflow to satisfy isolation, but the actual run-time connection to the resource may need to be more flexible. Finding a balance for isolation which is appropriate for reproducibility is difficult.

There are different ways [197] to make sure data is getting to the right places for execution. In a *user-directed* approach, users must identify file locations in the specification and a method for obtaining them, if not already available. In a *centralized* approach, a central repository holds all the data and each execution node must transfer files to/from there. In a *mediated* approach, a central repository holds only meta data about files and their locations. The files themselves can be distributed across many nodes reducing the bottleneck on the central repository. In a *peer-to-peer* approach, no central repository exists, so each execution node has a list of neighbors that can be queried for the data.

The centralized approach is the most reproducible, assuming the central repository remains available. The peer-to-peer approach does not require a central repository, but a workflow might not be reproducible if even one of the input files can't be found. The mediated approach is also more risky since the node(s) containing the actual data could be unavailable even if the central repository is working. After long periods of time (such as decades), a user-directed approach might need to be a fallback for reproducibility purposes, in the event that resources needed for the other methods are no longer available.

Alternatively, the code can be moved to the data when the data is big and the code is small. This can be a big boon to performance in some cases, but might not be supported by a workflow management system designed for a data movement approach. It is also possible that this mode of execution is not available for another scientist using different resources, so relying on code movement would decrease the chances of a workflow being reproducible.

Fault tolerance is a phrase often used when describing distributed systems that can handle the failure of some nodes in the system. But various levels of fault tolerance can be appropriate, depending on the scientific domain. For example, in some stages of high energy physics workflows, not all tasks need to be completed in order to consider that stage complete. This type of behavior can be a challenge for reproducibility because the failed tasks might vary between scientists. It can be difficult to determine whether a workflow is indeed reproducible when different individual failures exist in two workflow executions, but the number of failures is acceptable for both executions.

If multiple users are a part of a single workflow, there can be a great deal of confusion when someone updates the workflow at an earlier stage. A scientist working on the later portions of the workflow will need to somehow merge that update into what they are working on. The update may or may not affect the final results of the workflow, but it can be complex and challenging to figure out an appropriate and convenient time at which to incorporate the update.

When multiple users are involved, the naming of objects becomes challenging because each user would like to have control over naming, but names also need to identify identical objects across collaborating versions of a workflow. So an overview of existing approaches to naming on computers is needed.

CHAPTER 3

RELATED WORK

Many approaches are available for managing scientific workflows. But considering collaboration between scientists introduces an additional problem with naming conflicts across scientist repositories. Naming solutions are well established and related to the challenges in collaborative scientific workflows, so I introduce various approaches in detail as related work.

3.1 Naming

Due to the exploratory nature of scientific discovery, attempting to introduce too much organization into the workflow too early can be wasteful. Scientists may prefer to wait on coming up with a name for certain components (such as 'analysis' or 'simulation') until they are sure their value in the workflow is proven. Or they might only give human-centric names to the most significant components. In the meantime, the computer still needs to distinguish between the remaining objects and must most likely also group or link them together.

In addition, the names used to identify certain objects at one point in time, can be repurposed to refer to new objects as the workflow evolves. A workflow could also evolve in two different directions (perhaps by multiple scientists, or by one scientist trying two different ideas). Given two workflows with similar origins, it is difficult to identify which components are the same, and which are different. If this comparison becomes too difficult, the effort needed to incorporate a colleague's research can be greater than the benefits that may come. In fact, a bad experience attempting such a comparison can lead to a perception that no benefits from a colleague's research can outweigh that cost.

3.1.1 Namespaces

Each scientist, working individually, has full control over their own namespace which allows them to ensure that names identify the appropriate entity or entities. The home directory for a particular user on a computer is an example of a namespace. A user generally has complete control over the file and folder names in their home directory (except for a few reserved names and characters such as the '.' and '..' folders and certain characters not permitted in file names). Users can also create new namespaces when they create sub-directories.

An entity named 'analysis' by one scientist could refer to a completely different entity by another scientist. When their namespaces are separate, this is not an issue, but when sharing research methods with other scientists, these collisions can be impossible for a computer to resolve automatically.

One solution for the merging of namespaces is to take an approach similar to how directories hierarchies work. Individual scientists are allowed to continue with their own namespace, and a parent organization (or a super-namespace) is created to distinguish between the individual namespaces when more than one namespace is involved. An identifier for each namespace (such as a path or folder name) can be prefixed before the rest of the name.

However, the prefix can become tedious for frequently used entities that happen to have been created in a different namespace. A scientist may want to give a new name for that entity in their own namespace which can also become confusing because now a single entity has multiple names. If colleagues communicate with names that are only appropriate in their own namespace, there can be a great of confusion about what is being referred to.

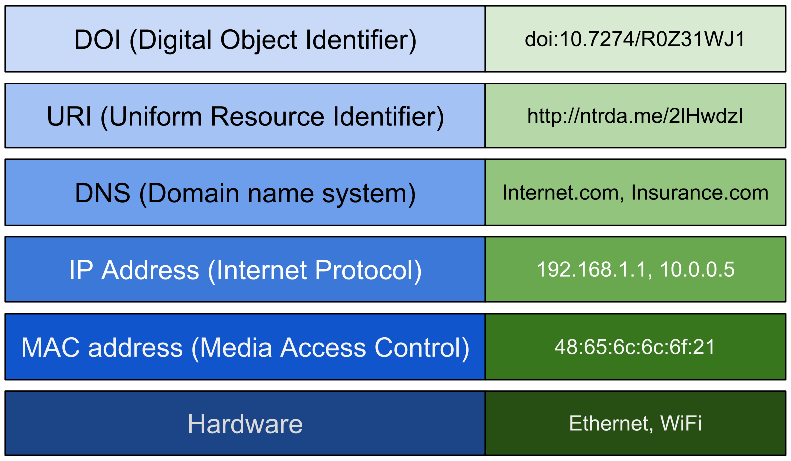

DOI (Digital Object Identifier)
doi:10.7274/R0Z31WJ1
URI (Uniform Resource Identifier)
http://ntrda.me/2lHwdzI
DNS (Domain name system)
Internet.com, Insurance.com
IP Address (Internet Protocol)
192.168.1.1, 10.0.0.5
MAC address (Media Access Control)
48:65:6c:6c:6f:21
Hardware
Ethernet, WiFi

Figure 3.1. Entity Naming

Examples of network and internet entity naming

3.1.2 Persistent identifiers

A scientist will occasionally need to move workflow files to new storage locations. If the naming of workflow components is tied too closely to their pathname, this hinders the ability to compare different evolutions of a workflow. On the internet, the ability to move and replace lower level network components is ensured by creating a hierarchy of names, each level of which can be modified without affecting the higher level names. Such higher level names (such as DOIs) are especially relevant when reproducibility is taken into consideration.

Each device on a network is an entity, identified at the lowest level by a *MAC address* (Media Access Control), which is essentially a name for an entity on a network (see bottom of figure 4.1.2.4). The MAC address is 6 byte number which is typically displayed to the user in hexadecimal notation (for example 48:65:6c:6c:6f:21). The purpose of the MAC address is to uniquely identify that specific network device globally worldwide.

If that device fails and is replaced, the new name must be propagated to everyone who used the old name. *IP addresses* were created to mitigate this problem. They are a new higher level entity or abstraction designed to identify a computer (rather than a device), and can be mapped to a specific device as needed behind the scenes.

In general, lower level entity names are typically more effective for replicability. They help to identify the exact set of operations needed to re-execute an experiment precisely. The MAC address is better for doing exactly what was done before, but an IP address makes it easier to account for later changes to the system. This makes IP addresses a better choice when extensible reproducibility is the goal, since the MAC addresses will not be appropriate on another set of computing resources.

But IP addresses are still difficult for people to remember, so domain names were created. *Domain names* are more human friendly names that identify a new entity called a site (rather than a computer), and can be mapped to IP addresses.

Each of these namespaces is managed by an organization which allocates sub-namespaces to other organizations. For example the first 3 bytes of a MAC address identifies the organization (such as a hardware manufacturer) which manages the last three bytes. And with domain names, top level domains (such as .com and .org) have control over the namespaces below that top level.

The identifiers for these namespaces are directly tied to a location, a domain name resolves down to an individual network device at any given point in time. But there is also a need for persistent identifiers that are separated from the location of the entity they refer to. And in addition to identifiers for physical entities, such as a network device, there is also a need for identifiers for any type of digital entity.

A URI (Uniform Resource Identifier) is designed to identify any resources, with a URL being the most common type of URI. A URL is connected to a location through the domain name embedded in it's identifier, but a URI does not have to include a location. The Handle System [174] is also part of the URI specification and is a namespace for global persistent and unique identifiers with a specific data type, but no changeable attributes such as location, ownership, permissions, or timestamps.

One implementation of the Handle System is the DOI (Digital Object Identifier) system. A DOI can be an identifier for any type of entity, and is associated with a URL, but the URL can change as needed.

A similar hierarchical approach could enhance extensible reproducibility by providing names that persist across scientific domains, but such a system is not yet in place. DOIs are often used for this purpose (to provide a global name for some entity) and are a great step towards reproducibility. However, their ability to change can also be a problem over time, because there is no guarantee that a given DOI will hold the same contents in the future as it did at the time a scientific paper was published.

3.1.3 Immutable identifiers

The ability to change the entity that a name refers to can make it difficult, from a historical perspective, to effectively communicate what entities were used to generate research results. For a given name, if an entity is replaced after or even while results are generated, referring to that name later on will likely prevent reproducibility if the system does not prevent this behavior.

In between a persistent identifier (which is designed for user convenience) and the location of an entity (which is all the computer needs) is an immutable identifier that allows multiple copies of the entity to exist in various locations, but prevents revisions or alterations to the entity which could prevent reproducibility.

If a central authority exists which manages unique entities, the authority can assign an immutable identifier to each entity using methods ranging from an incrementing number to UUIDs. In a distributed environment, a checksum of only the significant parts of an entity can provide an immutable identifier using an agreed upon algorithm. But a significant amount of forethought is needed since there are many checksum algorithms available, and it can be difficult to distinguish between significant and changeable attributes of an entity.

3.1.4 Overloading

Choosing an appropriate system for generating identifiers is further complicated by the challenges that come from having too many names for an entity or too many entities for a name.

Reproducibility can be more expensive when there is more than one appropriate name for an entity. In workflow descriptions, this *entity overloading* can cause extra network traffic, storage space, and computational resources because they might be treated as separate entities. These problems usually cause inefficiencies, but have no effect on reproducibility in the presence of sufficient time and resources. Filesystems that support file linking can deal with this problem by allowing a single file entity to appear to be in multiple folders and/or have multiple filenames.

On the other end of this spectrum, when namespaces are not clearly defined or managed *name overloading* can occur. This is a much more significant reproducibility problem than entity overloading. Imagine a folder on your computer with a single filename that points to two file entities. When attempting to access the filename the computer is unable to decide which actual file to open. Perhaps the filesystem could prompt the user to choose one over the other, but in a script, this is not possible.

The actual solution in most filesystems is that the new file entity replaces the old one. The user is often prompted to make sure this is the desired behavior, but once replaced, the old file entity with that name is no longer available. Even if the file system keeps all versions [164], it is difficult to resolve filenames when the version is ambiguous or without a desired timestamp, and the system is forced to make a guess.

This is a common occurrence in evolving workflows when there is a progression of the specific details (or entities) used to achieve some generic purpose. From the scientists perspective, each progressive version is an improvement on the last, so the most recent one should be used. However, even a small change in a single entity can drastically change the final results. So for reproducibility, it is important to uniquely identify entities in the workflow, so that the correct entities get used for a historical workflow. This means that a name alone may not be sufficient if previous versions need to still be available.

3.1.5 Versioning

A example of this problem that is a common source of reproducibility problems is the evolving versions of software libraries. The intent may be for all versions to be backwards compatible, but there can still be subtle changes that alter results. In addition, for the purposes of reproducibility, forward compatibility might be needed if the original scientist used a newer version of the library than a later scientist. A version hierarchy (ex. 1.2.10) is often used to distinguish between updates that are more or less likely to break a system that relies on the library. In order to identify relevant entities, this version "number" should be used in connection with the name of the library to uniquely identify an entity such that the name and version is an immutable snapshot of that library at a specific point in time. The version hierarchy has meaning to the user, but a computer typically sees it as no different than what could be achieved with a timestamp or a number that auto-increments with each new version. The combination of a name and a version provides both; an appropriate name for the user to understand the purpose of an entity, and an immutable entity that should be used by the computer.

3.1.6 Hashing

Another problem is that computers are unable to detect similarities between entities the same way humans can. However, they are very good at quickly identifying when two objects are in fact the same identical entity. A hash function can be used to map data of some arbitrary size to an identifier (or hash key) of some fixed size. In order to be useful, hash functions must always produce the same fixed identifier for a given data entity. When using a hash function to name a data entity, there is almost no possibility for entity overloading even without a central authority. A perfect hash function will always produce a unique identifier with no collisions (i.e. no name overloading). While a universally perfect hash function is unlikely, a wide range of hashing algorithms are available with various collision probabilities, many of which are appropriate for almost all situations.

Unfortunately, despite the advantages, humans find it difficult to work with hash keys because they appear to be random and need to be fairly large to provide sufficient assurance that collisions will be avoided.

3.1.7 Distributed hashing

To avoid a central authority, a hash function can be agreed upon as a way to uniquely identify relevant entities. For example, git [125] uses hashes of file and directory content to generate a hash that uniquely identifies each commit. A description of the commit is highly encouraged, but is not intended to be unique or to serve as a key for finding the commit. The description is solely for the benefit of the user.

A central authority is avoided because all participating computers agree upon a programmatic division of entities. Each computer maintains the data agreed upon by the groups hashing function, and the system relies on that function to ensure entities can be found using appropriate names.

3.1.8 Tags

Tags aren't much different than names, but the word is used to describe objects with no attempt at or expectation of uniqueness. If fact, they are used more to group objects than distinguish between them. In a way they are the opposite of versioning. Versions are a computer friendly addition to human friendly names, while tags are a human friendly addition to computer friendly unique identifiers (using hashes or an authority). Tags are also typically only relevant within some localized namespace.

3.1.9 Lineage

Another approach to naming that is particularly attractive for reproducible research is to embed the lineage or history of a particular entity in the name itself. For example, in a Merkle Tree [139], the leaves are ordinary data entities without names, and every other node has a hash key as it's name. The hash key is generated using the hash of the sum of the hashes from the child nodes. Other technologies that incorporate similar techniques include Git [125] and CVMFS [20].

Attempts have been made to create a universal identifier [79] for computational results. But there are still many different approaches being taken and it seems like there is more divergence in methods occurring than there is convergence. The conflicting goals of naming versus entity identification make it difficult for both humans and computers to find and distinguish between computational entities.

It should be clear at this point that there are many barriers to overcome, each of which can distract a scientist from their domain of expertise. All combined together the barriers seem to form a wall that effectively prevents most computational science from being reproducible.

3.2 Techniques for achieving reproducibility

Scientists should carefully choose a reproducibility technique that aligns with their goals for replicability and/or reproducibility. Some techniques are convenient and provide replicability, but are too complex to be effective for reproducibility. Others offer limited flexibility, but provide a high level of reproducibility. The following reproducibility techniques draw from related ideals in computer science or have a focus on satisfying domain specific needs.

3.2.1 Track all operations

One simple solution for preserving a workflow is to trace [35, 153, 154, 95] and log all system calls during the execution of a workflow. As mentioned before, the system call log can be used to reduce the size of a package or image enabling the execution environment. This approach works for replicability, but is not always resilient enough for extensible reproducibility. However, there are other considerations with can make this approach desirable.

Transparent Result Caching (TREC) [184] is one way to automatically track the dependencies that are explicitly stated in a Makefile, so the user can just use ordinary shell scripts to execute workflows. If the method needed to generate results is recorded, the results themselves can be treated as a cache. When an input changes, cascading results can be prompted for or automatically regenerated. Certain dangers exist with this approach, such as when there are non-deterministic operations, network communications, and/or failed tasks.

Nectar [86] is a more modern system designed for data centers. It can detect duplicate execution requests before they get sent to execution nodes and instead return the cached results. Since VisTrails [15] is designed to generate images interactively, caching the results can be a large benefit so that all historically explored images don't have to be stored indefinitely. [126] also addresses saving and reusing previous computations to reduce the amount of traffic that has to be aggregated. They can also use that information to detect effective file equivalence even when a checksum comparison says the files are different.

3.2.2 Track all actions within a walled garden

Environment dependencies can also be resolved by requiring all execution [57] to occur on a shared, public testbed. In such a system, a workflow can be tracked at a very high level. For example, an interactive text editor [114] could track lines of a script as they are entered by the user and execute them in the background on behalf of the user.

In order to prevent conflicts between user and system control of the environment, such executions are more reproducible if performed in a clean sandbox. In other words, execution should occur in an environment which is both separate from user space and loaded without implicit dependencies.

This type of system can even support multiple users [80] with the possibility of some shared state between them. This virtually ensures reproducibility and extension of workflows, in the short term. However, flexibility is very limited and scalability is often out of the hands of the researchers, and eventually the shared system will need to be updated, requiring new consideration of all archived workflows.

3.2.3 Track all actions from an achievable initial state

Some methods assume an implied initial state, but it is a good idea to make the initial state more explicit. Recording changes in a way to support undo [198] can provide a way to achieve a consistent initial state that can be shared with others. If the ability to replay the changes is included, then there may also be the ability to revise the replay instructions to support extensions to research rather than just replication. This replay ability also provides good efficiency during execution with the ability to look at a particular operation in more detail later [48], without having to store all details on the fly.

3.2.4 Execute a detailed specification

Rather than let the user perform or request operations on the fly, the user can be expected to get organized and plan their workflow in advance, evolving the workflow as needed to handle the exploratory nature of computational research. Such a plan should start with a clear recipe for a repeatable environment, and then domain

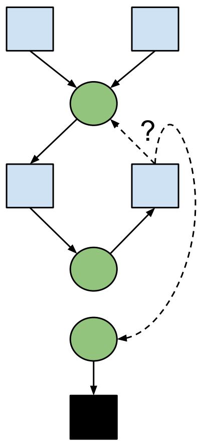


Figure 3.2. Conditional Loop

Directional graph with conditional loop.

appropriately sized building blocks can be used to advance the state of the workflow using that environment.

The relationship between all tasks, in a workflow, and their dependencies can often be fully described using a DAG (directed acyclic graph). Some tasks may need to be run in *series* (a specific order) if they depend on the results of other tasks in order to be executed. These series tasks contribute to the height of a DAG that describes the workflow. Tasks without such dependencies on other components can be run in *parallel*, and contribute to the width of a DAG.

Some workflow systems [3] only support tasks that can fit into a DAG structure. Such systems can optimize the use of resources during execution because all required tasks are known in advance. The DAG could be extended to allow for additional functionality, such as with conditional DAGs [146], but adding additional components to a DAG may reduce or negate optimizations that a DAG based system can do.

Tasks whose execution is *conditional* or tasks within *loops* (see figure 3.2) are beyond the scope of a DAG. However, a higher level abstraction can be used to describe conditional or looped tasks without losing DAG optimizations as long as the conditions and loops can be decided before execution begins. For example 2 tasks inside a loop that iterates exactly 10 times where one of the tasks only executes on odd iterations can be easily translated into 15 tasks in a DAG, even if each iteration of the loop depends on the results of the previous iteration.

However, any loop or condition that relies on run-time results cannot be directly translated into a DAG before execution. Such workflows must be handled on a more on-demand basis. Prediction or estimation may be possible [133, 129], but the full benefits of a DAG are not available. If other elements beyond a DAG are needed, the description must be more abstract.

A language based workflow description can support large and/or complex workflows which can be directly shared with collaborators. Also a person with programming experience may be able to easily write a program (perhaps in their preferred scripting language) which generates the desired workflow description in the target language. This allows a user to create more complex (and more abstract) workflows than are available in the workflow language itself.

Some systems created and use custom languages designed specifically for their workflow system [73, 77, 148, 159]. Others focus more on adapting the workflow into a standard language (XML for example) [25, 6, 9, 196, 192]. Somewhere in the middle there are standardized languages created for use in multiple workflow systems [7, 187]. A language can even be interactive with the ability to run complex workflows programmatically, while at the same time preserving the workflow in various convenient languages even before the code is executed [158].

There is great value in designing the language to approximate the scientists' natural language [94]. This is more convenient for the original scientist, but is also easier for collaborators to understand. Taken to the extreme, such systems [111] might remove the need for scientists to deal with any kind of programming, since the language provides all scientific needs, and the low-level details are handled by individuals outside the scientific domain.

However, users often prefer a graph based system (rather than language based) for workflows (especially new users) because learning a new language can be difficult. A graph based approach [89] also typically abstracts lower level details away from the user, forcing/allowing them to focus on higher level concepts which are more applicable to their scientific research.

Graph based modeling is sometimes implemented in the form of Petri nets [186, 92]. FlowManager [11] is one example. More recently, UML (Unified Modeling Language) [163, 14, 55] has also been used to address issues with Petri nets.

The best of both graph and language based approaches can be available by exposing both options to the user. In addition, allowing multiple representations of the workflow [128] can offer new insights and capabilities. Grid-Flow [85] includes a Petri-net based interface and a programmable Grid-Flow Description Language. XRL/Flower [191] also uses Petri-nets, but uses XML to support standard parsing and validation of the language.

For large/complex workflows, a graph based approach can become unwieldy due to the vast details available. In order to help graphically modeled systems support larger and more complex workflows, low level details [149] should be abstracted away from the user. A system of templates [92] can be used to specify sub-portions of a workflow in a hierarchy of abstractions. Triana [176] supplies a graphical user interface which allows users to drag and drop tools and connect them to inputs and outputs. Tools can then be "grouped" together to both simplify the visualization of the workflow and to support an abstraction hierarchy. Kepler [129] supports "abstract components" which collapse the details of a subworkflow to tame complexity. By using WSFL (web services flow language) to compose workflow elements, each composition can be used hierarchically as a web service for higher level compositions. [123] Another interesting approach [78] is to attempt to automatically generate higher level abstractions on top of the low level provenance.

Ideally a user could describe a workflow in very abstract terms and some domain specific system could flesh out the details automatically. Pegasus [46] combined with Chimera [70] support this type of approach. Given a somewhat abstract description of inputs and the desired outputs, Chimera can automatically lookup operations in a database and those operations can then be used to accomplish the desired behavior. Pegasus then marshals resources to execute the workflow, generating the results.

For all of the techniques, but especially for a detailed specification, it is helpful to ensure that a history of the evolution of the workflow is somehow recorded. In software development, this is done with a version control system where changes to the software are periodically recorded. This is so that the state at important points in time can be obtained even after future changes have been made. Such checkpoints can be automatic (so the user doesn't forget), or the user must develop a habit of choosing appropriate times to record the state. More frequent checkpoints make it easier to isolate and undo bad changes, but they are more work to create and sort through. Websites such as http://github.com and http://bitbucket.org have made it convenient to share this information with collaborators, and are often used for scientific collaboration. However, such systems can break down when large amounts of data are involved as they are designed more for source code than for data.

Even with reproducible research there is no guarantee that the research results will be correct. [121]

3.2.5 Verify and/or validate the final state

The above mentioned Pegasus+Chimera method of workflow abstraction is also an example of another broad technique for reproducibility. This technique is to pay very close attention to the final result and less to the steps along the way.

Take for example, the difference between Puppet [127] and Chef [178]. Both are systems widely used for system configuration. Chef takes an imperative if-then approach where the system administrator designs a sequence of statements that specify exactly what actions should be performed to get the computer in the desired state. On the other hand, Puppet allows the system administrator to declaratively say what packages should be on a computer, and Puppet has some freedom in determining the best way to install those packages.

A declarative evaluation of scientific results is usually merited, with or without an imperative list of steps required to get there. The responsibility to ensure that scientific research results pass declarative needs generally rests completely on scientists. With tools that can assist or automate some of this process, scientists can be responsible to ensure those tools are applicable and appropriate. Sometimes verification and validation are as simple as comparing newly generated results to results which have already been verified or validated.

If it is possible to automatically determine that results from a replication attempt are equivalent to the results from the original research, then the exact methods used to achieve those results may need less scrutiny. Verifying and/or validating the final state could be the only technique needed to reproduce very simple workflows, but for large workflows, this technique would be more effective when used in conjunction with another technique focused on the steps along the way.

Ideally, a scientific workflow will be fully deterministic and the generated results will always be bitwise identical. However, in practice, there are many sources of non-determinism [104] which contribute to results being different after a workflow replication attempt. The degree to which small sources of non-determinism affect an entire experiment [110, 41, 50] is important to consider.

Some tools help with a comparison, but still require a user to make a final decision on equivalence. For example, a comparison tool called *sfvplotdiff* [66] is used in the Madagascar project to compare a plot generated by an established version of a workflow with a plot generated by an extended evolution of that workflow. Special care is given to tolerate precision differences [58] between the computers executing the workflow, so that a scientist can observe only significant differences between the plots. Then the scientist can more easily decide whether the differences are justified depending on how the workflow was extended.

3.2.6 Require formal dependencies

Relying on conventions is more convenient (for most users) than creating elaborately configured frameworks. However, preferring configuration over convention is helpful for reproducibility.

At the programming language level, dependencies on libraries can be specified using commands like import, include and require. However, it is unlikely that the programming language will locate and retrieve those dependencies if they have not already been installed. A container image [138] or virtual machine image can be coupled with the code, to satisfy those dependencies.

The dependencies can also be specified in a functional manner [52] if the dependencies need to be more granular than a single image. Alternatively, a set of commonly used dependencies can be bundled together [136] so that a single reference to the bundle can indirectly include all of the dependencies from the hardware to the command level.

A system that can embed a full environment specification into each component of a workflow [49, 100] enables users to ensure that all required dependencies are included. If the system only executes the workflow using the fully specified environments, then any generated results have a high likelihood of being at least replicable. In addition to providing the ability to replicate component execution on similar hardware, this form of encapsulation [94] could include subcomponents such as a compiler, to make the environment specification more portable to hardware systems that are less similar to the original ones used. This also allows for more separation between domain science and computer science [111], reducing the need for domain scientists to be involved in programming in favor of focusing on their domain, and also increasing the Reproducibility.

3.2.7 Validate continuously

In software engineering, continuous integration [71] is when developers flag working updates to a shared codebase as soon as they are ready, and then several times a day, updates from all developers are merged, tested, and put into production. Various websites [36, 181, 108] can be used to support continuous integration. Comprehensive testing (or validation) is typically automated so that unexpected problems can be quickly identified before going live.

This concept can be applied (in part) to scientific workflows for collaboration. While the merging is less likely to be valuable several times a day (compared to internet applications), the measures taken to make that possible can simplify the collaboration process enough to make it more effective for reproducibility. For this to work, a few practices need to be adopted.

There is a big difference between a scientist manually reporting the command they executed and the scientist requesting a command to be executed with the system automatically reporting the command used. When the scientist manually reports a command, there is always a possibility that something was left out, or that something was changed after the report was made. When the reporting is automatic, there may still be room for implicit dependencies built into the system, but those can be easier to track and resolve than transient changes based on what a user types into the command line.

Automation is a good first step to ensure that a workflow, and any sub-components, are correctly reported in connection with some results. [45] By some definitions [93], automation is the purpose of creating a workflow in the first place.

One of the earliest and most prevalent ways to automate a workflow is the Unix *make* [135] utility. Early attempts [165] to encourage reproducible research using *make* resulted in some success, but over 38% of the files (figures) were not reproducible. In addition, the system was fairly complex including scripts in various languages and LaTeX macros, the combination of which made it difficult for other scientists to reproduce. [68]

More recent attempts [67] using the more modern automation utility *SCons* [115] (based on Python) were eventually more successful [66], but that success isn't necessary tied to those utilities alone. There has been more pressure to make research reproducible in recent years, and some of the successes could be attributed to that momentum.

In many cases, the benefits of automation may actually contribute to the overall "ease of experimenting" [28] while at the same time making great strides toward reproducibility.

In software engineering, test-driven development is where the programmer creates a test for desired behavior before the behavior is even implemented. [61] This ensures that even if changes are made to the software, the desired behavior is preserved. Scientific research is generally too exploratory to be able to define the desired workflow fully in advance. However, once some research has been published reproducibly, it can be treated as a test in the sense that it can be used as a basis for comparison as other researchers attempt to replicate or extend the research. Tests are vital for performing continuous integration [56].

Various systems [84, 141, 8, 113] designed for automatic deployment and task execution could be used to assist in efforts to validate continuously.

The Madagascar project [66] applies this concept to reproducible research by running tests whenever someone submits a modification to preserved research. If automatic validation fails and no one in the community steps in to correct the error, the project is removed from the set of maintained research workflows.

Some workflow systems are specifically geared towards automating and preserving the generation of graphs and figures to be shown in a publication. One approach combines Git And Org-Mode [169] to handle the execution of a workflow that is fully documented as it is created. The final result is similar to a lab notebook and can be published with all details on how to reproduce it. Similarly, Paper mâché [24] manages a workflow and a $\LaTeX$ or .doc file, directly inserting images generated by the workflow into the document for publication. Vistrails [32] is designed to interactively generate images so that the viewer can explore visualization with custom arguments to available parameters. These are tools that provide some automation, but final verification/validation is performed by a user.

3.2.8 Make the environment explicit

The computing environment used by most scientists is provided by system administrators who attempt to balance the needs of many users when making decisions about how to provision hardware. When scientists need system resources that are not already provided, they may ask the system administrator to install or procure those new resources for them. They might go as far as choosing a specific version of an Operating System or even bleeding edge hardware for their research. If those resources are not automatically provided, some scientists will even take on the some of the role of system administrator to obtain those resources themselves. Virtual machines and containers are relatively new technologies which give scientists more flexibility. Some of these new technologies improve reproducibility at the same time.

3.2.8.1 Hardware provisioning

The traditional way to provide compute resources to scientists is to consistently provision dedicated hardware for each research group. This can include a detailed list of imperative operations [178] which describe how to install the environment directly on hardware. Alternatively, a declarative specification [127, 136] lists all the components that are needed in the environment, but does not dictate how they are to be installed.

It is also important to consider the community, reliability and usability [150] of hardware provisioning specifications as they can get very complicated.

3.2.8.2 Virtual machines

A more recent technique involves provisioning the hardware with a generic system that allows virtual machines to dynamically be instantiated as needed. This provides the scientist the most flexibility. But more importantly, it automatically provides an easy to preserve a copy of the full environment used in the original workflow execution. This virtual machine image can then be easily shared [96] with colleagues using cloud services.

This approach is quite effective when the research can reasonably fit in a single virtual machine. However, a networked system can break down with large datasets or with research that requires distributed execution to complete in a reasonable timeframe. Vagrant [87] goes one step further by including complex network configurations in addition to managing software within the virtual environment. However, it is likely that many unneeded files will get included in the virtual machine image, making the image excessively large and more difficult to curate and share.

3.2.8.3 Containers

A container image [138] serves many of the same purposes as a virtual machine image. But while the virtual machine image is a single file with everything needed, a container image is a progression of new packages added to previous ones. As such, a container can have less overhead (especially when booting) because it may be able to rely on already running parts of an Operating System. Except for the fixed Operating System kernel, a container can provide what appears to be a completely independent system. This allows more efficient use of RAM and faster startup times, in addition to layered filesystems and common files that make disk usage more efficient.

3.2.8.4 Package management

A package management system mostly handles changes to installed software libraries. However, most package management systems require root access which means that access to such systems must be restricted [182] for security reasons. This contributes to a barrier between system administrators and scientists leading to confusion when attempting to identify and share the environment.

However a few non-root package managers [52, 42] are emerging which take a functional approach to setting up appropriate environments. Giving the scientist such control can help ensure that upgrades don't sneak in leading to unexpected results, and can help even with heterogeneous [188] devices.

3.2.8.5 Functions

Organizing a workflow into functions is one way to parameterize tasks into abstract components that perform domain specific operations and are organized in a way logical to scientists. A simple command can be treated as a function if the executable is designed to work that way (see figures 2.2 and 3.3).

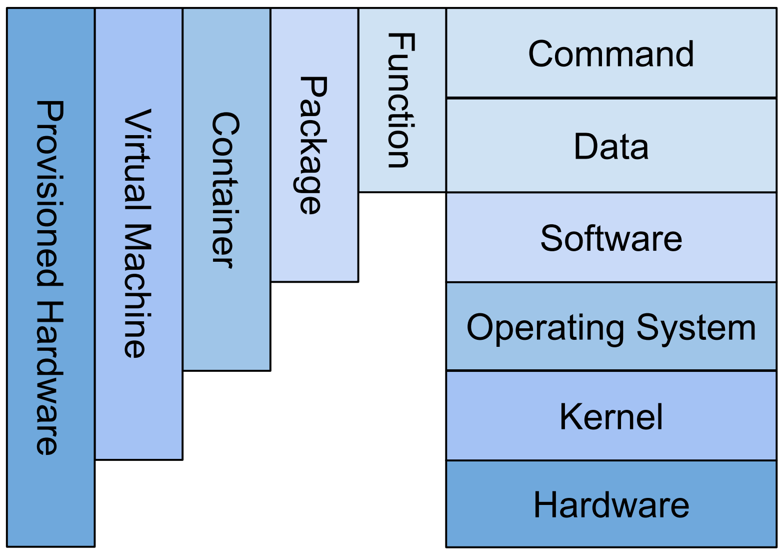


Figure 3.3. Environment Scopes

The environment can describe anything from a command involving some data, to the full computational stack down to the hardware specifications.

3.2.8.6 Distributed systems

For workflow that don't fit in a single node, a connecting structure needs to exist between instances of an environment. To execute complex multi-cloud and multi-VM applications reproducibly, cloudinit.d [26] can launch, configure, monitor, and repair a set of interdependent virtual machines over multiple IaaS clouds. A launch plan describes a series of run levels, each of which contain tasks that can be run in parallel. A service handles the launching, configuring, and status of VMs starting with package management tools like Yum or configuration management tools like Puppet. In the case where failure is detected and repair actions are needed, cloudinit.d only restarts the affected sections of a launch plan.

Whatever the method for defining and creating environments, doing so not only helps with reproducibility, but also allows the scientist to delegate the responsibility of providing reproducible system resources.

With many options for representing and defining environments, I proceed with an overview of PRUNE which I developed to record a user's intended environment and instantiate it to execute each task in a workflow. Each task must be executed in a sandbox so that when the scientist gets results, there is high confidence that no implicit dependencies on the environment were missed.

CHAPTER 4

PRUNE OVERVIEW

This chapter introduces a tool used to explore possible solutions to some of the reproducibility challenges. It relies on some of the related work in previous chapters, and identifies desirable qualities in a workflow management system designed for reproducibility. Environments are specified independently from the tasks in the workflow, but each task is coupled with a defined environment. Workflow evolutions are preserved before tasks are executed, rather than at a user defined time (as with git), so that changes that led to results are never forgotten and the granularity is consistently at the task and files levels.

The components and operations used to identify a workflow and it's parts are described. Considerations in object naming are explored, including the naming of objects stored outside of the workflow on the internet. Desirable features, such as the ability to automatically manage file storage, account for non-determinism, and collaborate with other scientists are explained.

4.1 A preserving run environment for reproducible scientific computing

To address problems with current tools for reproducible scientific computing, Prune [102], the Preserving Run Environment, was designed and implemented. 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 avoid the possibility of implicit dependencies. 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.

The simplicity of PRUNE is first demonstrated with a sample merge sort implementation. A workflow involving the U.S. Censuses demonstrates PRUNE's ability to scale to large amounts of data and make efficient use of available storage. Then computational science workflows from bioinformatics and high energy physics (HEP) were executed with PRUNE. Initial attempts to efficiently handle the data needed for dependencies and creating appropriate execution environments caused a cascade of failures, and it was discovered that linked files are not always equivalent to copied files. Adjustments to reduce and/or eliminate these problems were effective, but there is still room for improvement.

4.1.1 "Preserve First" strategy

Preservation is often perceived as an activity undertaken after research has been completed [24]. 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 replicability [157], and more work is needed to fill in gaps in the preserved form of the research [18]. This process is shown in Figure 4.1a.

In contrast, I advocate a **preserve-first strategy** for reproducible computational research as shown in Figure 4.1b. I argue that researchers should first (before executing any code in the workflow) preserve (at least locally) the components they wish to use. Automated execution based on the preserved components can then

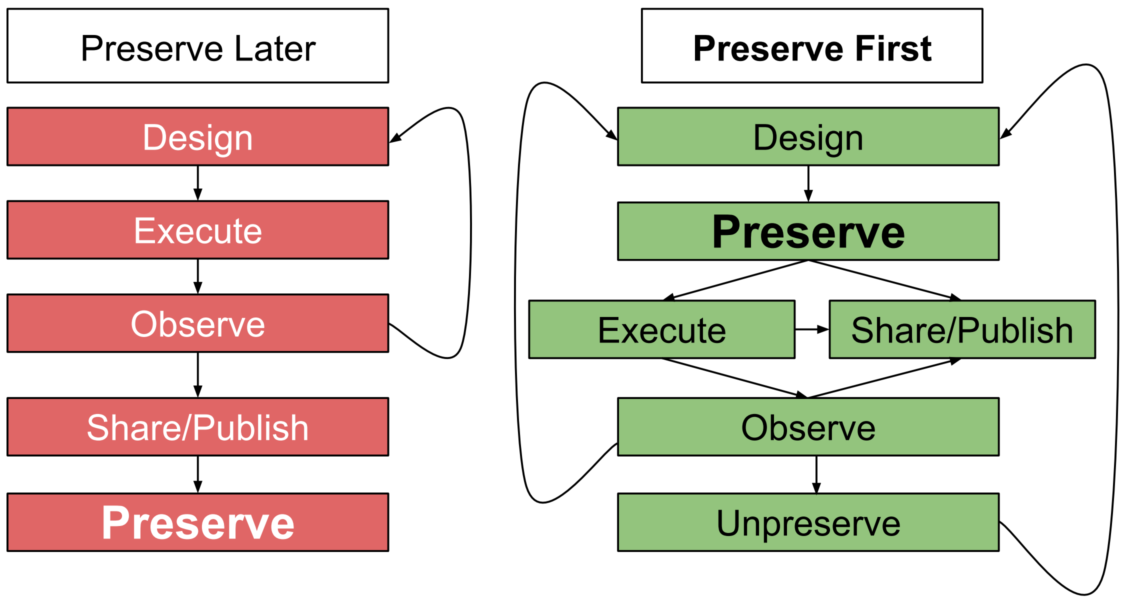


Figure 4.1. Preserve First

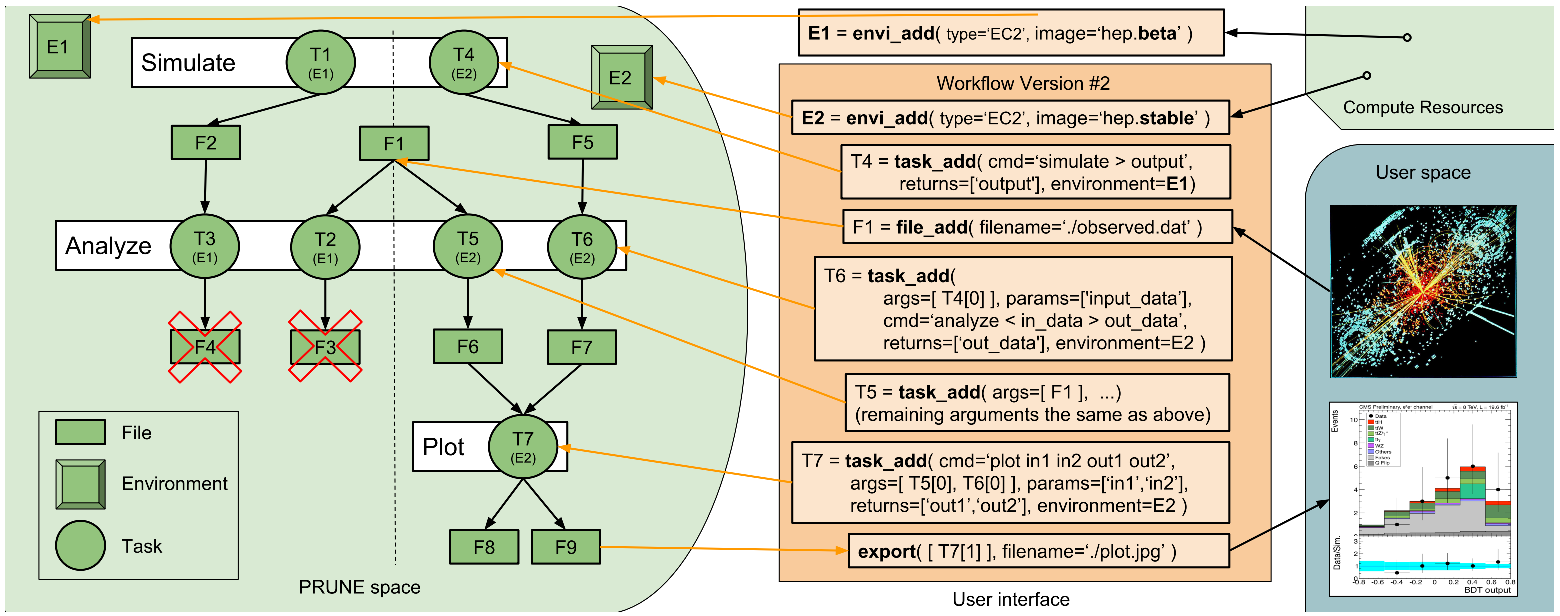
Digital items should be preserved first (before use) to avoid ambiguity about results.

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 I believe with this approach, PRUNE moves towards greater structure and oversight such as with the adoption of: block-structured programming [51]; graph-structured Make files [64]; and rigorous version control [125].

4.1.2 PRUNE overview

The following sections describe how a user would interact with PRUNE, what architectural components there are, the interface by which operations are performed on those components, and how namespaces are handled in PRUNE.



```
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' )
```

Figure 4.2. Prune Overview

Prune automatically manages storage and execution of workflows.

4.1.2.1 User's perspective

An end user begins using PRUNE 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, PRUNE can 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.

4.1.2.2 Workflow components

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. An overview of how the different elements are connected is shown in figure 4.2.

A **File** is an immutable string of bytes, identified by a hash of the content of the File. Any data 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, with 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. It is generally designed to be appropriate for a range of Tasks, rather than having a unique Environment for each Task. An Environment can take many forms, but in concept is distinct from the means used to deliver that Environment for a specific Task on a specific compute resource.

For example, an Environment could be as simple as a tarball with software to be added on top of an assumed operating system. Various methods can be used to deliver that environment. If a compute resource already runs that operating system, the tarball merely needs to be unpacked in the proper location, then the compute resource is ready to execute tasks in that Environment. Otherwise a virtual machine may need to be started up to supply the operating system, before the tarball can be unpacked. On the other hand, the Environment could be a virtual machine image which includes the operating system. Any number of virtual machine managers might be able to load the image and make it available for Tasks.

I 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.

If there are few assumptions about what resources will be available in the future, an Environment should be reproducible for many years to come. For example, using a virtual machine image in Amazon EC2 or a container image in a public Docker Hub, both make assumptions about relatively new technologies with a tendency to evolve quickly. However, a reference to Amazon EC2 might be more convenient in the short term.

By separating the Environment from Tasks and reusing a given Environment for multiple Tasks there should be more opportunities for caching (in addition to being a convenient way to preserve the environment for reproducibility). When creating an instance of an Environment from the input file(s), the results of intermediate steps (such as unzipping) could also be cached to reduce the resources needed to provide instances of appropriate Environments.

A workflow system generally considers entire files for inclusion or eviction from a cache. When multiple files are needed for an environment, it might be more useful to consider the Environment as a single element in the cache rather than a collection of files which can be individually evicted. This could be done more simply by bundling those files into a single file, such as a zip file, rather than by creating a more sophisticated caching system.

A tool called Umbrella [136] can observe the current resources and compare them against an Environment specification. If the current compute resources matches the specification, the Task can be immediately executed. If any software or a different operating system is needed, Umbrella can add the software or even start up a virtual machine, if necessary, to satisfy the specification. This can be much more efficient than always starting virtual machines, while still supporting heterogenous resources.

However, this means that it is not known until runtime what data resources are required. Ideally, even for the cases where a virtual machine image needs to be transferred to a compute resource, that image should only need to be transferred once. But without the knowledge of what Environment dependencies will be needed on a resource until runtime, a workflow scheduler might have a difficult time choosing the right resources for each Task.

4.1.2.3 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.

4.1.2.4 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. [10] PRUNE uses two types of identifiers for objects: content-based identifiers and derivation-based identifiers.

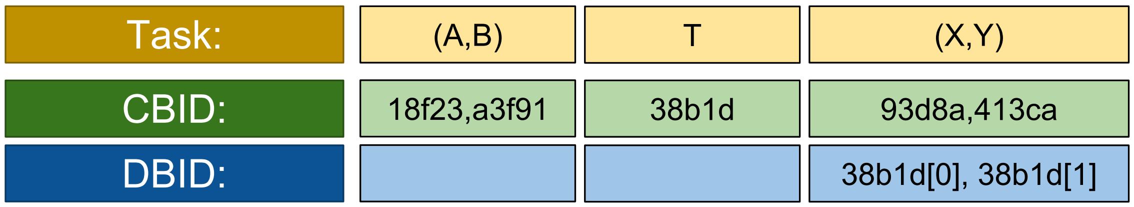
A content based identifier (CBID) is the fundamental name for all Files, Tasks, and Environments. It is generated by computing a hash function of either the content of the object, which is the binary data of a File, or the JSON document representing a Task or an Environment. Care must be taken to ensure 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, as shown in figure 4.3.

In addition to overhead and wall time measurements, Result objects record the



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

Figure 4.3. CBID vs. DBID

An example of the relationship between a CBID and a DBID. The output files X and Y each have both a CBID and a DBID

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 4.1.2.1. 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.

4.1.2.5 Sandboxes

One thing that can help avoid naming conflicts is to isolate tasks from each other. Before a Task is executed in an Environment it will be placed in a temporary sandbox. This helps prevent accidental interference with other Tasks, and can also provide a debugging snapshot when there is an error, while still allowing other Tasks to continue.

For example, in figure 2.2 a Task refers to a few input and output files. The 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. Then the entire sandbox can be discarded.

4.1.2.6 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. This can be an issue 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 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 if the user desires to.

4.1.3 Storage management

One of the challenges with preserving a workflow is the amount of storage space required. I 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). I propose treating derived Files as a disposable portion of a workflow as detailed in section 4.1.3.1. I assume that the second largest portion of the storage requirement is typically **root** Files (external input data directly imported into a Prune repository). I discuss ways to address this challenge in section 4.1.3.2. 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 4.1.3.3.

4.1.3.1 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.

4.1.3.2 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 [134], IRIS [189], the LHC [131], etc., but any changes to the data by the managing organization must be detectable and/or avoidable in the interest of ensuring reproducibility.

4.1.3.3 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.

I 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 such cases, it can be useful to record the workflow identifier in addition to a CBID and DBID.

This chapter proposed some of the goals for a reproducible workflow management system, but intentionally left room for the evaluation of different choices that could be used to satisfy those needs. The architecture was divided into File, Task, Result, and Environment components operated on by file_add, task_add, envi_add, execute, export and import operations to support the proposed goals. The need for both content based identifiers and derivation based identifiers is expressed to handle for the possibility of non-deterministic tasks. With such a framework prepared, the next chapter identifies a specific choice made or options to be compared in order to implement these ideals. With a single user implementation of PRUNE I then evaluate our choices with workflows involving high energy physics, bioinformatics, and the U.S. censuses.

CHAPTER 5

SINGLE USER EVALUATION

With a framework, in the last chapter, for what high level capabilities are needed in Prune the specific choices are identified in this chapter, and the implementation is applied to various workflows from a single user perspective. Many of the implementation details chosen here have other largely equivalent alternatives available, but alternatives were unlikely to improve the implementation in any significant way. For example, SHA1 hashes are chosen for the content based identifiers. A larger hash would be better if this hash was being used for security, but because security is not the goal, SHA1 should be adequate for gauging uniqueness between objects in a trusted environment. Since the multi-user aspect of Prune is designed at a higher level than the database level, SQLite is sufficient and a bit more convenient than MySQL. The high energy physics, bioinformatics, and U.S. censuses workflows used to validate Prune, show that overhead is minimal, the workflows are still scalable, and Prune can adequately manage storage consumption automatically based on a provided quota.

5.1 Implementation details

Prune 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 client library to expose Prune 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 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, 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 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, but the CBID can be derived.

If there are two Tasks which are identical except for the specified environment, Prune still preserves them 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 creates a single file with all relevant objects embedded. File content is treated as binary blocks, with the rest of the Files and other objects as JSON text. In addition to handling the depth of the lineage and/or progeny extracted from the repository graph, a 'files' argument allows the scope of the export to be more specific. For example, this allows the user to select whether or not intermediate Files should be included in the exported files. This file can be shared with other users of Prune 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 or fail, as appropriate.

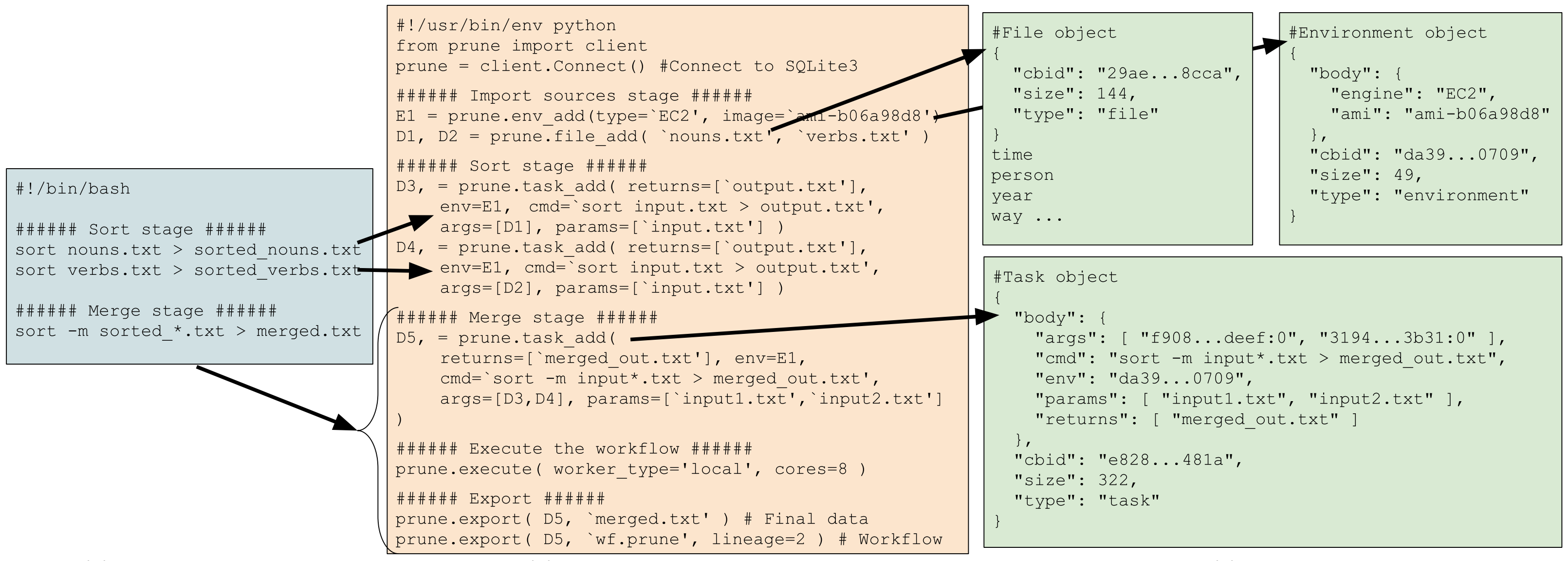
5.2 Compute resources

Prune can either spawn **local** worker processes to execute Tasks, or start a Work Queue[1] master [30] to coordinate Task execution on **remote** workers. In local mode, input Files are linked into Task sandboxes, with the assumption that Tasks 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 considered. In remote mode, Files are transmitted over the network, making it more appropriate for computationally intensive Tasks with small inputs.

PRUNE 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 currently allows Tasks to run without a specified environment (meaning the default available environment should be used), with a Wrap environment, or with a local Umbrella [136] 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.

[1]Work Queue is a framework for building large master-worker applications that span thousands of machines drawn from clusters, clouds, and grids. Tasks are executed by a standard worker process that can run on any available machine. Each worker calls home to the master process, arranges for data transfer, and executes the tasks. The system handles a wide variety of failures, allowing for dynamically scalable and robust applications.



```
#!/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

Figure 5.1. Example Workflow

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

5.3 Example workflow

Consider the shell script shown in figure 5.1a designed to take two input files and efficiently produce a new file with all lines merged and sorted.

The Python script in figure 5.1b will preserve and execute a workflow equivalent to figure 5.1a. The last line exports the minimum objects needed to reproduce the workflow, and saves these objects in the "merge_sort.prune" file.

The PRUNE client library converts the script at figure 5.1b into the PRUNE (slightly abbreviated) objects at figure 5.1c which are not exposed directly to the user. These objects are what is stored in the PRUNE repository.

This may seem verbose compared to the original workflow. But I 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.

5.4 Evaluation (using U.S. censuses)

In order to evaluate the storage management abilities, computational overhead, and scalability of PRUNE, it was used to manage workflows doing some analyses on U.S. Census records. The U.S. Census [62] for years 1850 to 1940 consume 23 GB using 7-Zip compression. Due to spelling, transcription, and other errors, it is difficult to find individuals in the census records. In a "Census Name Comparison" workflow a list of the most frequent surnames in all censuses is created and compared against the list of all surnames to obtain lists of possible alternate spellings. These alternate spellings can be used to do a "fuzzy" search for individuals in the censuses. This workflow is broken down into the following 8 stages:

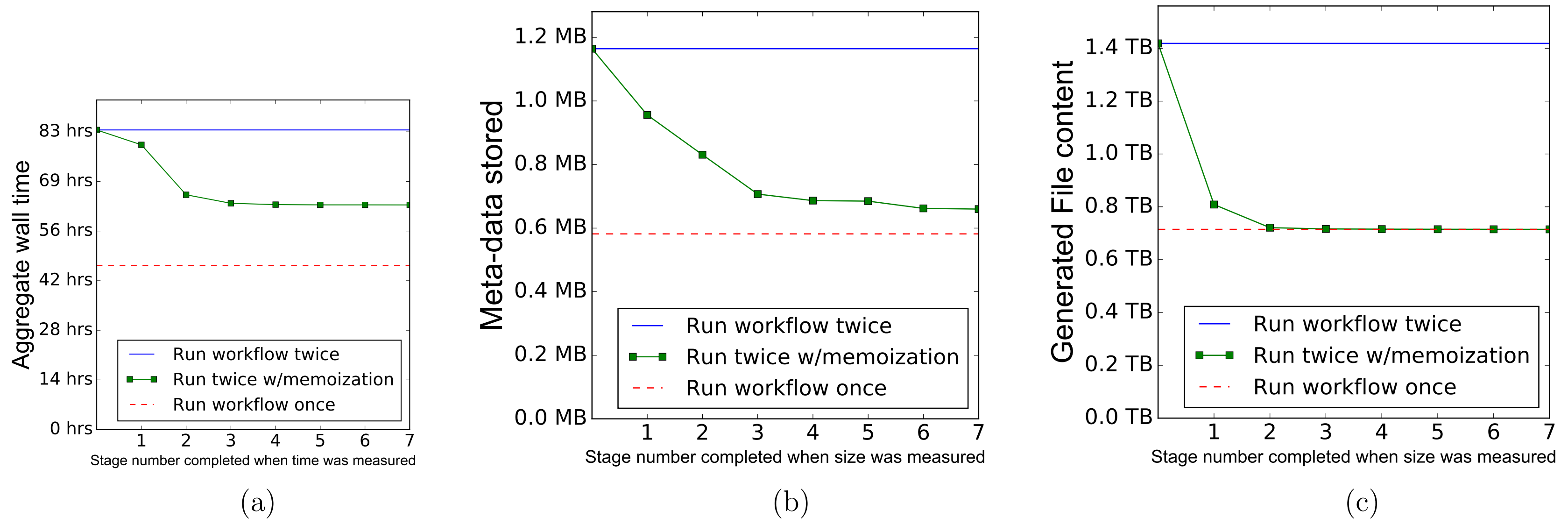


Figure 5.2. PRUNE Behavior

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) caches extra disk space used for generated Files.*

Census Name Comparison workflow stages

1 Decompress
(7-Zip unpacking)

2 Normalize
(Standardize field inclusion, names, and order)

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)

Importing original files into PRUNE takes a significant amount of time. But since that is more a side effect of preserving the original files than a part of the workflow, I consider this Stage 0.

5.4.1 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 5.2a, b, and c comparisons are made between this situation where two versions of the workflow are in separate folders (upper line) compared to a situation where only one version of the workflow exists (lower line).

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

TABLE 5.1

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**

NOTE: The overhead of checksumming the files is a significant factor in the first stages, but minimal overall.

In figure 5.2a, 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 [40] is computationally expensive, so even changes to only that final stage still require a significant amount of work.

The measurements in figures 5.2a, b, and c 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 5.2b 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 is most effective when evolutionary changes to a workflow are made at the leaves of the workflow rather than at the roots.

Figure 5.2c 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 database.

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 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 5.2c is a candidate for eviction. In extreme cases, intermediate files could be deleted as soon as they are consumed by later tasks.

5.4.2 Overhead

To measure the overhead of PRUNE, the 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 5.1.

Stage 0 (the "import sources" stage) is included here as 100% overhead, since PRUNE 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 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.

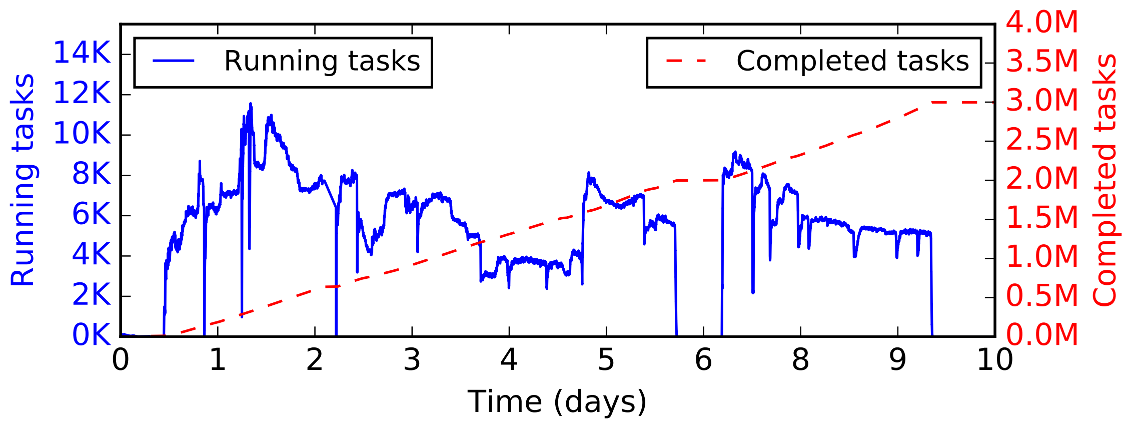


Figure 5.3. Prune Scalability

Prune can handle preservation and execution of large scale workflows. 3 million Tasks were executed within 10 days with a concurrency of O(10k).

5.4.3 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 5.3 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.

5.4.4 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 was given a quota of 30TB. Prune v1 appropriately removed Files from the repository cache whenever it observed that generated Files caused the repository to go over quota.

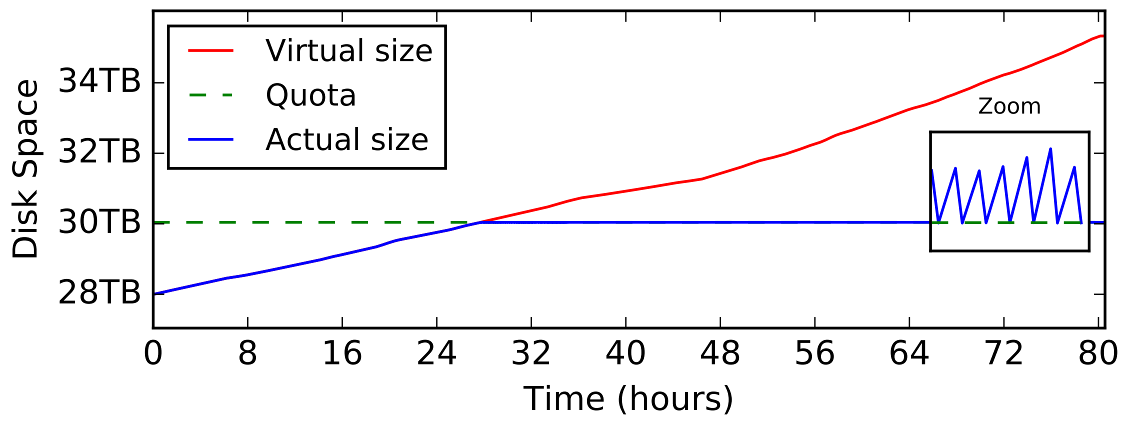


Figure 5.4. Prune Quota Management

Prune can keep disk consumption within a quota during workflow execution. In this case the quota was 30TB.

Figure 5.4 shows that PRUNE stayed within 700MB of the quota after reaching the quota. This was done in the background to avoid interference with the workers.

5.4.5 Collaboration

PRUNE can be used to facilitate evolutionary changes by multiple users concurrently. In this other workflow, one user might find an interesting match and wants to share those results with another user. Examples of commands for exporting a workflow are shown here:

```
1:  export( [id], ‘result.txt’ )
2:  export( [id], ‘result.prune’, lineage=INF )
3:  export( [id], ‘result.prune’, lineage=INF,
                                files=[‘root’,‘leaf’] )
4:  export( [id], ‘result.prune’, lineage=INF,
              files=[‘root’,‘intermediate’,‘leaf’] )
```

The first command only exports a single file named result.txt which exists in the database with the CBID or DBID in the variable 'id'. The second command exports a zipped folder called result.prune which includes the minimum files needed to execute the workflow or in other words the root Files and all Tasks. The third command export adds the final generated (leaf) Files also to the minimum Files and Tasks. The fourth command includes all Tasks and all Files used to generate the file with a CBID or DBID in the variable 'id'.

Table 5.2 lists properties of the exported file with these 4 approaches. A 5th export approach is used to demonstrate the full scope of the Matching workflow without limiting it to just the most relevant portion for a specific result.

In addition to the Census Name Comparison workflow just described, I used a separate workflow that just does matching on exact census records (with no "fuzziness"). The exported package with all tasks and root and intermediate files resulted in a 1.5TB file and took 1 hour and 25 minutes to generate. However, 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. Even better, In 4 seconds, another 2.6GB package was created with the Tasks, root Files, and the interesting File. The interesting match 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 because PRUNE 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. The imported/exported file might be larger than it needs to be, but PRUNE handles the import appropriately.

TABLE 5.2

PARTIAL EXPORT/IMPORT COSTS

Scenario	**Export time**	# of Files	# of Tasks	File size (MB)	**Import time**	**Re-execution**
1: Leaf File for 1 match	˜0	1	0	0.000232	˜0	-
2: Root Files + Tasks for 1 match	0:03	22	56	2,604	0:03	5:30
3: Root, Leaf Files + Tasks for 1 match	0:04	23	56	2,604	0:03	**(5:38)**
4: All Files + Tasks for 1 match	2:10	78	56	74,899	2:18	**(2:55)**
5: All Files + Tasks	1:25:51	58,884	16,118	1,551,581	2:22:05	0:02

NOTE: PRUNE enables collaboration by only exporting/importing desired objects from a workflow.

TABLE 5.3

WORKFLOW WALL TIMES

	Time spent on		
Stage	Environment	Workflow	PRUNE
BWA-GATK: Stage 0	0:00:01	0:35:09	0:00:00
BWA-GATK: Stage 1	0:00:01	0:04:38	0:00:00
BWA-GATK: Stage 2	0:00:00	0:02:03	0:00:00
BWA-GATK: Stage 3	0:00:03	0:05:46	0:00:00
BWA-GATK: Stage 4	0:00:02	0:11:41	0:00:01
BWA-GATK: Stage 5	0:00:00	0:11:32	0:00:02
BWA-GATK: Stage 6	0:00:01	0:06:56	0:00:01
BWA-GATK: Stage 7	0:00:00	0:00:13	0:00:00
BWA-GATK: Stage 8	0:00:08	6:29:19	0:00:05
BWA-GATK: Total	0:00:16	7:47:17	0:00:09
Monte-Carlo: Stage 0	1 day, 17:33:26	14 days, 13:00:20	0:00:23
Monte-Carlo: Stage 1	13:57:01	8 days, 20:55:08	0:00:24
Monte-Carlo: Stage 2	18:04:45	46 days, 9:49:46	0:00:28
Monte-Carlo: Stage 3	13:07:48	1 day, 17:00:26	0:00:25
Monte-Carlo: Total	3 days, 14:43:00	71 days, 12:45:40	0:01:40

NOTE: This shows the distribution of execution time spent on the environment vs. Prune vs. the actual workflow.

TABLE 5.4

WORKFLOW DATA FOOTPRINT

	Unique Input		Aggregate Tasks		Environment		Generated Output	
Stage	files	data	files	data	tasks	data	files	data
BWA-GATK: Stage 0	17	7,084 MB	80	8,063 MB	10	~0 MB	10	636 MB
BWA-GATK: Stage 1	27	7,721 MB	90	8,700 MB	10	~0 MB	10	8,951 MB
BWA-GATK: Stage 2	12	8,989 MB	30	9,326 MB	10	~0 MB	100	6,883 MB
BWA-GATK: Stage 3	101	6,885 MB	200	7,018 MB	100	~0 MB	100	1,846 MB
BWA-GATK: Stage 4	101	1,847 MB	200	1,937 MB	100	~0 MB	100	1,380 MB
BWA-GATK: Stage 5	101	1,380 MB	200	1,470 MB	100	~0 MB	100	1,385 MB
BWA-GATK: Stage 6	101	1,386 MB	200	1,476 MB	100	~0 MB	100	159 MB
BWA-GATK: Stage 7	12	39 MB	40	98 MB	20	~0 MB	20	62 MB
BWA-GATK: Stage 8	232	1,691 MB	700	7,182 MB	100	~0 MB	100	171 MB
BWA-GATK: Total	704	37,022 MB	1,740	45,270 MB	550	~0 MB	640	21,473 MB
Monte-Carlo: Stage 0	2	0 MB	2,000	3 MB	1,000	~633,848 MB	2,000	6,719 MB
Monte-Carlo: Stage 1	1,002	6,543 MB	3,000	6,551 MB	1,000	~633,848 MB	2,000	20,552 MB
Monte-Carlo: Stage 2	1,003	20,352 MB	4,000	22,011 MB	1,000	~633,848 MB	5,000	58,675 MB
Monte-Carlo: Stage 3	3,002	58,662 MB	5,000	58,669 MB	1,000	~633,848 MB	2,000	3,352 MB
Monte-Carlo: Total	5,009	85,557 MB	14,000	87,234 MB	4,000	~2,535,392 MB	11,000	89,298 MB

NOTE: The data needs and behaviors of the BWA-GATK and MCProduction workflow are summarized.

5.5 Evaluation (Bioinformatics and HEP)

Prune was designed for computational science, so this section evaluates how effectively it can be applied to a bioinformatics workflow called "BWA-GATK" and a High Energy Physics workflow I call "MC Production" (short for Monte Carlo production). The execution and data footprint for these workflows can be found in tables 5.3 and 5.4.

5.5.0.1 BWA-GATK

BWA-GATK is a bioinformatics workflow conceptually divided up into two parts; BWA is executed first, followed by GATK. BWA is a light-weight alignment tool for performing queries on genomes. It supports paired-end mapping, gapped alignment, various file formats, and employs the Burrows Wheeler Transform algorithm to align the genome queries. GATK takes the SAM (Sequence Alignment Map) output from BWA and applies a sophisticated Bayesian algorithm to compare aligned sequences with the reference. The final output expresses how closely alignments match with additional information about the analysis as a whole.

This workflow was chosen because there are many stages, each of which could be an opportunity for changes in an evolving workflow.

5.5.0.2 MCProduction

The MCProduction (Monte Carlo Production) workflow consists of 4 steps that make up a chain of tasks used to simulate possible collision events. This is done using models based on the real events observed through detectors in the Large Hadron Collider. This workflow was chosen because it has a highly complex environment and is non-deterministic [104] by design, sometimes unavoidably. Each of the 4 steps is described below, and the output generated from earlier steps is used as the input for later steps.

Physics Simulation (step #1 - LHE): This is a simulation of the first part of the physics involved in the collision. There is no attempt to account for the detector at this stage. The acronym LHE stands for Les Houches Event [5].

Detector Simulation (step #2 - GEN-SIM): For very technical reasons, there is a second part of simulating the physics of the collision that happens in this step. After this, the effects of the detector are simulated, but the data format read out is not the same as what the detector readout produces.

Reconstruction (step #3 - DIGI-RECO): The next step, is actually broken into two separate sub-steps that are run sequentially: The DIGI step takes the simulation file output and changes it into a format that is identical to what the detector produces. After this step, no distinction needs to be made in the software between running on simulated and real data. The RECO step is the same reconstruction that's applied to real data which takes detector signals and figures out which particles would have made those signals in the detector.

Data Reduction (step #4 - MiniAOD): This last step takes the output of the RECO step (which is in a format called AOD for Analysis Object Data), and simplifies it into a reduced data format that contains the information that almost all scientists use to do their research. Some small fraction of analyses actually need the level of detail in AOD and can't use MiniAOD, but most researchers use the MiniAOD data.

5.5.1 Caching

When all computations for a workflow can be performed on a single machine and the relevant data is on a disk on that machine, the computer automatically manages when to move data from the disk into the hierarchy of caches closer to the CPU for processing. Carefully designed algorithms determine when to replace data in these caches in the hope of avoiding data transfer when the same data might be used more than once. When many computers are working together, connected by a network, data needs to be moved to each computer that is doing some computations. As with the data in the CPU caches, there is a chance that a given datum could be useful more than once on a multi-processor computer. Ideally any given file would only get transferred to a specific physical computer on a network a maximum of one time.

In Work Queue, the user must specify whether or not each input/output file should be cached. With the delays involved in network transfer being so much more expensive and less predictable than transfers from disk into a CPU cache, the added user burden of making such choices is merited. In fact, as disks are generally much more capacious than RAM, making good choices in this area could allow the computers in a network to avoid transferring any file more than once.

In addition, Work Queue keeps track of the cache contents for workers so that new Tasks can be assigned to workers that already have needed files if such workers are available.

In general, the workflows had few (and/or small) dependencies shared between Tasks in a given stage. In table 5.4 on BWA-GATK: Stage 8, there is only 1.7GB of unique input data, but for full concurrency without a per machine cache and multiple Tasks per machine, 7.2GB of data would need to be transferred to satisfy those dependencies.

Based on the input data PRUNE is aware of, this appears to be the only stage that would benefit significantly from caching files on each machine, and only if there are multiple Tasks running on that machine. However, in the environment column of that table are approximations of how much data is depended on by the Umbrella Environment. Environment dependencies were treated somewhat differently than the data dependencies, and more on this is in the next section.

5.5.2 Environment dependencies

When the compute resource Umbrella is running on does not even closely match what is needed in the Environment specification, large virtual machine images are needed to create an instance. In a worst case scenario, where each compute resource requires a virtual machine, the data transfer requirements are significant, as show in table 5.4 under Monte-Carlo and Environment. The data requirements for providing environments can far supersede what is needed for the workflow itself.

Initially PRUNE configured Umbrella to use the same pathname for the cache. Rather than copy files from the cache into each sandbox for executing Tasks, Umbrella links to the cached files. This seemed more efficient and still acceptable because Umbrella is in charge of that folder and expects the Tasks to 'behave' and stay within their own namespace.

However, there appeared to be cache integrity issues with multiple concurrent instances of Umbrella working from a single folder on a given computer. In a very naïve attempt to eliminate stale cached information, PRUNE was modified to delete the Umbrella cache folder each time it attempted to execute a Task using Umbrella.

Unfortunately, this clearing of the cache usually occurred while an existing instance of the Environment was still running and linked into that folder. Tasks quickly started failing as their Environments were deleted, and the 'worst case scenario' of transferring large virtual machine images to every Task became orders of magnitude worse as the progress of every computer with multiple such Tasks ground to a halt. Tasks restarted over and over again as they failed, overloading the system providing the virtual machine images, which prevented even machines with only one Task per computer from progressing. After too many failures each machine was added to a blacklist, and many of the machines with only one Task received more Tasks and joined the cycle of failures then blacklisting.

In the end, the 'worst case scenario' was better than this new situation and caching was effectively turned off by using a unique cache folder for every Task. A more appropriate long term solutions are proposed in the next section.

5.5.3 Solutions

The bottleneck encountered was with Tasks at the same hierarchical level in the workflow concurrently depending on the same files. This is in contrast to sequential dependencies in the workflow from one Task to the next. Sequential dependencies are the obvious job of a workflow manager, and often come in the form of recently finished workers having generated the data needed as an input for the next Task. In an ideal environment, static concurrent dependencies can be easily satisfied by keeping a copy of that data on all computers used in the workflow. However, in a heterogeneous (and perhaps opportunistic) system such assumptions cannot be made, placing a higher realtime burden on the workflow management system. However, in addition to being a bottleneck, in the case of multiple Tasks running on a single machine, satisfying these concurrent dependencies can cause conflicts and failures.

5.5.3.1 Locking/concurrency in the Environment cache

One obvious solution is to design the system that creates instances of the environment to prevent conflicts, and in so doing prevent this type of failures. However, sometimes the necessary Environment provider is not designed to handle such a case, and it is not always feasible to request or implement this as a new feature.

5.5.3.2 Hoisting dependencies

Another solution is to make the Environment dependency a part of the workflow dependencies. This allows the workflow system to manage getting the file to the right places, which will work to avoid conflicts, as long as the workflow system is designed to handle concurrent Tasks on a single machine.

However, it might be not known in advance what dependencies need to be satisfied to instantiate the Environment. In the case of Umbrella, there are a few different methods that are acceptable for creating the environment. If Umbrella finds that the virtual machine image is not required on a particular machine, then bandwidth was wasted transferring the image to the machine.

5.5.3.3 Only prepared workers

Another option is to only use computers for executing the workflow which have been prepared for the necessary environment. This doesn't mean that all machines must match the requirements, but certain resources must be available on each machine before it is considered a part of the compute resources available for the workflow. This could just mean that a virtual machine image is placed in a known location on the machine in advance. Or it could mean that the virtual machine is instantiated before making itself available to the workflow.

Notre Dame physicists are using Singularity[2] with the later method [117]. Singularity is used to instantiate a given virtual machine image, and then the instance announces itself to the workflow manager as available for execution.

5.5.4 More linking problems

Another problem with symbolic file linking occurred with the BWA-GATK workflow. One of the stages was consistently failing when using Work Queue, but it would work fine running locally with the same Tasks. I put a check into Prune to verify that the contents of the files existed just before executing a Task in Work Queue and the files existed, but it would still fail. I eventually figured out that the Task was unable to handle linked files. The problem was resolved by modifying the Task to make a

[2]Singularity is a container technology designed to enable the user to have full control of their environment. It can be used to package entire scientific workflows, software and libraries, and even data.

copy of the file before executing the command. This was not a problem because the file was small (unlike the Umbrella Environment files above).

However this situation is a great example of the unexpected challenges that often appear when reproducibility is attempted. The code had an implicit requirement that the file be real and not a symbolic link. Even seemingly trivial changes to the data or environment can break a workflow despite attempts to make it reproducible.

This chapter included implementation details used to implement the PRUNE ideals and applied this implementation to 3 complex and/or large workflows. A few challenges were uncovered which were not anticipated, but in all, the tool was successful at managing storage space automatically without incurring excessive overhead. It was also still scalable to tens of thousands of CPU cores. The export of a workflow components after the execution of the workflow was touched on in this chapter, but more exploration is needed into the options for collaboration. The next chapter describes 5 collaboration modes that can be used make working on partial workflow sharing easier. The estimated temporal and financial cost of each mode is then applied to the MCProduction, and BWA-GATK workflows.

CHAPTER 6

COLLABORATION EFFICIENCY

The previous chapter on the implementation and evaluation of PRUNE focuses on all the requirements and features for a single user using PRUNE to automatically track evolutions to their workflow so that they can be shared with other scientists. However, the quantitative benefits of being able to selectively include or exclude generated files when exporting a partial workflow were only lightly addressed. This chapter defines 5 collaboration modes that can be considered and chosen at any time (not just in advance). Most workflow management systems only include 2 of these modes, or require all workflow execution to be performed on dedicated cloud resources managed by the workflow management system maintainers. The advantages and disadvantages of each mode are explained and quantified for the MCProduction and BWA-GATK workflows. Then the temporal and financial costs of each mode are estimated for each workflow. While the results do not indicate that one mode is always better than another, it is clear that certain modes can have dramatic reduction in the more quantitative aspects of collaboration.

6.1 Typical collaboration

A common approach to preserving scientific workflows is to make a copy of the working directory containing the final results and all files and code used in it's generation. Any refinements can then be made to the original workflow as new ideas are explored, without a concern that the workflow connected to the publication will no longer be available. The copied folder might be archived for up to a year after a paper is published just in case something is needed. For example Notre Dame has a shared drive called /scratch365 where such data can be stored, but files are automatically deleted when they reach 365 days old. This resource is specific to the University of Notre Dame, but similar services are common at most HPC centers.

This archived folder could be sent to collaborators if desired, but if changes are made to both copies of the workflow, it is typically very difficult to reconcile the changes and share them between collaborators again. In addition to the inconvenience of manually identifying changes in the workflow and choosing which version to use in the case of a conflict, there is also a cost to transferring and/or re-executing portions of the workflow that have already be executed.

Prune is designed to simplify the user effort involved in reconciling changes to multiple related evolving workflows, but quantifying user effort is difficult. In addition to the user benefits, there are some quantifiable benefits to knowing what portions of a workflow already exist on a collaborator's system. A balance between re-execution and transfer of execution results can be achieved, saving network bandwidth and computational effort.

However, in order to ensure that the system is meeting the needs of the collaborating users while this optimization is being performed, I propose three basic requirements that should exist in any workflow management system designed for collaboration:

1. Collaborators must get all code, environments and input files needed to create the final results.
2. Collaborators must re-generate or at least transfer those final results onto their own system.
3. Collaborators must not lose any changes they have made to the workflow in order to obtain someone else's results.

TABLE 6.1

TRANSFER MODE PROPERTIES

Changed		Pr1	Pr2	**Pr3**	Pr4	Pr5
Transfer new workflow or changed input	Transfer files	A	A	A,Z	*	*
	Transfer tasks	*	*	*	*	*
	Execute tasks	*	*	-	-	-
Transfer changes on stage D	Transfer files	-	-	Z	D+	*
	Transfer tasks	*	D+	D+	D+	*
	Execute tasks	*	D+	-	-	-
		Most CPU		Least CPU		

Key

Prune (various modes)	Pr#
Input files for the workflow	A
Final results for the workflow	Z
Items at change and beyond	D+
All items	*
No items	-

NOTE: These 5 different options for dealing with collaboration needs have an impact on the network transfer and CPU execution connected to the collaboration.

6.2 Collaboration modes

With those requirements in mind there are a few modes of workflow operation that are available within these constraints. Figure 6.1 summarizes these modes, and they are described in more detail as follows.

6.2.1 Pr1 mode

In this mode all generated results are ignored so that only the minimum amount of data needs to be transferred to a collaborator. For a brand new workflow being sent from one computer to another, only the input files ('A' in figure 6.1 next to Transfer files) and the code used to execute the entire workflow ('*' in figure 6.1 next to Transfer tasks) need to be transferred. These numbers could be reduced if some of the workflow (perhaps input data) happens to already exists on the collaborators system, but the assumption is made that an entirely new workflow is what is being transferred. However, the cost for this minimization of network traffic is that in order to satisfy requirement #2, all the code must be re-executed on the destination system ('*' in figure 6.1 next to Execute tasks).

It is not unusual for a workflow system to have the option to delete all generated files so they can be regenerated after changes to the workflow have been made. In one system called Makeflow [3] [1] all generated files are deleted when 'makeflow –clean' is executed. Immediately after this operation the workflow could be transferred minimally to a collaborator. However, in addition to re-executing on the destination system, it would likely need to be re-executed on the source system or a full copy of the workflow would need to be made prior to executing 'makeflow –clean'.

The data in PRUNE's immutable tree makes this 'Pr1' mode possible as shown in figure 6.1. The initial export of the workflow would be performed with a PRUNE command like this:

```
export( <list of CBID/DBIDs for final result Files>, ‘result.prune’,
              lineage=float(‘inf’), files=[‘root’] )
```

In 'Pr1' mode, after changes have been made on both sides of the collaboration,

[1]Makeflow is a workflow system for executing large complex workflows on clusters, clouds, and grids. It is designed to have syntax similar to the familiar Makefile approach and is used for large scale workflow execution on a large number of different systems.

any sharing in a typical workflow management system would require a fully separate copy of the shared workflow in order to satisfy requirement #3 and not lose the most recent evolution on either side. However, it could be possible to share the input files in this mode (see the '-' in figure 6.1 next to Transfer files). This could be performed in PRUNE like this:

```
export( <list of CBID/DBIDs for final result Files>, 'result.prune',
               lineage=float('inf'), files=[] )
```

While 'Pr1' mode requires the most execution, it also provides an opportunity to test the reproducibility by executing the workflow on the collaborator's system.

6.2.2 Pr5 mode

On the opposite end of the spectrum, execution is conserved by copying all data including intermediate files and the final results. This conservation of execution comes at the cost of additional network transfer bandwidth and the associated delays. Nothing out of the ordinary needs to be done with Makeflow to operate in the mode. Collaboration is done by copying the entire workflow as is. PRUNE could also operate in this manner with the following command.

```
export( <list of CBID/DBIDs for final result Files>, 'result.prune',
               lineage=float('inf'), files='all' )
```

In order to satisfy requirement #3, the exact command above would be used both when a new workflow is shared and when an existing workflow is modified.

While 'Pr5' mode is likely to take the longest to transfer, it would be provide the most opportunity to identify and resolve reproducibility problems introduced by switching to the collaborator's system.

6.2.3 Pr2 mode

In this mode, new workflow transfers are performed identically as in Pr1, by copying the input files and all executable code for the workflow. However, using the data available in the PRUNE database, namely the content-based identifiers and derivation-based identifiers, tasks which already exist in the databases can easily be identified. So any data before a change in the workflow need not be transferred. Continuing with the goal of minimizing network traffic, no intermediate files or final results are transferred, so in order to satisfy requirement #2 all tasks from the change in the workflow and beyond must be re-executed on the destination system.

This mode is not possible with Makeflow or with typical workflow management systems. For a new workflow the PRUNE command is the same as with Pr1 mode, but with a modified workflow, only the changes need to be exported with the following command [2] :

```
export( <list of CBID/DBIDs for final result Files>, ‘result.prune’,
      lineage=<number of stages from change to results>, files=[] )
```

6.2.4 Pr4 mode

In Pr4 mode, new workflows are transferred as in Pr5 mode, all files and tasks, so no re-execution is necessary. After a change in the workflow, all tasks and their generated files are transferred from the point of change to the end of the workflow. In the end, the PRUNE database holds the original workflow, the new local workflow, and the new remote workflow, all at the same time. No re-execution is necessary.

For a new workflow the PRUNE command is the same as with Pr5 mode, but with a modified workflow, only the changes and changed files need to be exported with

[2] However, the number of stages from the stage where the origin and target systems diverge and the final results must be tracked and derived through manual discussions between the collaborators.

the following command [3] :

```
export( <list of CBID/DBIDs for final result Files>, ‘result.prune’,
      lineage=<number of stages from change to results>, files=‘all’ )
```

6.2.5 Pr3 mode

This mode starts with the input files and all workflow tasks, as with Pr1 and Pr2. But by adding the final results without any intermediate files, requirement #2 can be satisfied without the need for any re-execution. When the workflow is changed, all changed tasks must be transferred, and transferring the new final results again keeps the network traffic low without the need for re-execution. All 3 requirements are satisfied. This is likely to be the best option available, but depending on the costs of execution and network traffic, one of the other modes might be less costly.

Here is the command for a new workflow using Prune:

```
export( <list of CBID/DBIDs for final result Files>,
      lineage=<number of stages from change to results>,
      ‘result.prune’, files=[‘root’,‘leaf’] )
```

Here is the command for a modified workflow:

```
export( <list of CBID/DBIDs for final result Files>,
      lineage=<number of stages from change to results>,
      ‘result.prune’, files=[‘leaf’] )
```

However, assuming the number of stages parameter is chosen properly, the 2 commands above should behave identically.

While this appears to be the most appealing choice, there is some value in choosing the least appealing approach of doing both ‘Pr1’ and ‘Pr5’. This least appealing

[3] Again, the number of stages from the stage where the origin and target systems diverge and the final results must be tracked and derived through manual discussions between the collaborators.

case provides the best chances both to test reproducibility and identify and resolve reproducibility issues.

6.3 Case studies

In Makeflow, switching from Pr5 to Pr1 mode is permanent because all generated files are deleted, and the original workflow must be re-executed to make the final results available again. This is not desirable for large workflows and magnifies the CPU execution cost proportional to the frequency of workflow evolutions amongst all the collaborators.

Currently PRUNE is able to operate in all 5 of these modes using the methods described in 5.4.5 and all modes are available concurrently depending on which one is best in a given situation. However, negotiation of the best choice based on what might already be available to the destination system has not been implemented. The users would currently have to have a detailed conversation about what is already available and what needs to be transferred in order to fully operate in these modes. The best mode to use in a given situation depends both on the inherent properties of the workflow, the nature of the evolution between collaborators in the workflow the history of collaboration on those evolutions. Each workflow evolution could be individually optimized to meet the collaborator's preferences. However, PRUNE could be extended such that the two systems could automatically negotiate the best transfer solution based on a user's priority in avoiding network traffic or re-execution. Also, the work required to create a package to perform these operations ignored below, but was measured and reported on in 5.4.5.

Given the same two use cases used in the implementation chapter, the behavior of evolutions under each collaboration mode and at each stage is shown in table 6.2 and table 6.3. For all workflows, modes Pr3-Pr5 require no re-execution on the collaborator's system.

TABLE 6.2

MCPRODUCTION WORKFLOW MB+DAYS

Changed		Pr1	Pr2	**Pr3**	Pr4	Pr5
All	Files (MB)	1	1	**20,554**	89,299	89,299
	Tasks (MB)	2	2	**2**	2	2
	Exec (days)	75.15	75.15	**0.00**	0.00	0.00
Stage 0	Files (MB)	<1	<1	**20,552**	85,559	89,299
	Tasks (MB)	2	2	**2**	2	2
	Exec (days)	75.15	75.15	**0.00**	0.00	0.00
Stage 1	Files (MB)	<1	<1	**20,552**	85,559	89,299
	Tasks (MB)	2	2	**2**	2	2
	Exec (days)	75.15	58.87	**0.00**	0.00	0.00
Stage 2	Files (MB)	<1	<1	**20,552**	79,015	89,299
	Tasks (MB)	2	1	**1**	1	2
	Exec (days)	75.15	49.42	**0.00**	0.00	0.00
Stage 3	Files (MB)	<1	<1	**20,552**	58,662	89,299
	Tasks (MB)	2	<1	**<1**	<1	2
	Exec (days)	75.15	2.26	**0.00**	0.00	0.00
		Most CPU		Least CPU		

NOTE: Estimated costs of doing collaboration on an MCProduction workflow when the workflow evolves at different stages and different Prune modes are used.

6.3.1 MCProduction

In table 6.2, since modes Pr3-Pr5 all avoid any re-execution, the only metrics to consider are the network transfers. Because MCProduction is a simulation, there is very little input data. In Pr3 mode all stages only need to transfer the final results (plus negligible task transfers), which are 23% of the full data required in Pr5 mode. Pr4 mode reduces the data transfer down to about 66% of Pr5 in the best case, with no reduction at all in the worst case where the input files change. The network transfer for Pr4 is reduced to 89% of Pr5 on average.

Modes Pr1 and Pr2 have negligible differences in network transfers, but Pr2 can reduce the re-execution to as little as 3% of Pr1 in the best scenario. The average re-execution reduction is 69% of the maximum.

Based only on the 3 collaboration requirements, the best choices for the collaborator are likely to be 1) elimination of re-execution and reduction of network transfer to 23% with Pr3 or 2) minimization of network transfer and reduction of re-execution to an average of 69% with Pr2.

6.3.2 BWA-GATK

In table 6.3, modes Pr3-Pr5 again avoid all re-execution. However, in the BWA-GATK workflow, the original input files are much larger than the final results. In Pr3 mode all stages only need to transfer the final results (plus negligible task transfers), which are only 6% of the full data required in Pr5 mode. Pr4 mode reduces the data transfer down to less than 8% of Pr5 in the best case, with again no reduction at all in the worst case where the input files change. The network transfer for Pr4 is reduced to about 73% of Pr5 on average.

Modes Pr1 and Pr2 have negligible differences in network transfers, and Pr2 can reduce the re-execution only to 84% of Pr1 even in the best scenario. The average re-execution reduction is negligible.

TABLE 6.3

BWA-GATK WORKFLOW MB+DAYS

Changed		Pr1	Pr2	**Pr3**	Pr4	Pr5
All	Files (MB)	7,173	7,173	**8,553**	21,478	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.32	**0.00**	0.00	0.00
Stage 0	Files (MB)	<1	<1	**1,380**	37,026	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.32	**0.00**	0.00	0.00
Stage 1	Files (MB)	<1	<1	**1,380**	29,942	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.30	**0.00**	0.00	0.00
Stage 2	Files (MB)	<1	<1	**1,380**	22,221	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.30	**0.00**	0.00	0.00
Stage 3	Files (MB)	<1	<1	**1,380**	13,232	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.30	**0.00**	0.00	0.00
Stage 4	Files (MB)	<1	<1	**1,380**	6,346	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.29	**0.00**	0.00	0.00
Stage 5	Files (MB)	<1	<1	**1,380**	4,499	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.28	**0.00**	0.00	0.00
Stage 6	Files (MB)	<1	<1	**1,380**	3,118	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.28	**0.00**	0.00	0.00
Stage 7	Files (MB)	<1	<1	**1,380**	1,731	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.27	**0.00**	0.00	0.00
Stage 8	Files (MB)	<1	<1	**1,380**	1,691	21,478
	Tasks (MB)	<1	<1	**<1**	<1	<1
	Exec (days)	0.32	0.27	**0.00**	0.00	0.00
		Most CPU		Least CPU		

NOTE: Estimated costs of doing collaboration on an BWA-GATK workflow when the workflow evolves at different stages and different Prune modes are used.

Based only on the 3 collaboration requirements, the best choices for the collaborator are 1) elimination of re-execution and reduction of network transfer to 6% with Pr3 or 2) minimization of network transfer and reduction of re-execution to an average of 84% with Pr2.

6.3.3 Partial workflow (Pr3) disadvantage

Transferring the final results and all tasks, but no intermediate files in Pr3 mode satisfies all 3 collaboration requirements. However, there is another desirable collaboration property that the Pr3 mode does not provide. If there is a chance that the collaborator will make a change to the evolved workflow at any stage, the collaborator's workflow reverts to Pr2 mode since the generated input file at that stage are not present. The generated files before that stage could be transferred, but otherwise the full workflow must be re-executed to obtain those files. Both Pr2 and Pr4 modes have this 4^{th} collaboration property and can be chosen when it is needed.

6.4 Estimated cost comparison

Collaboration would not require EC2, however, their pricing model can be an effective way to estimate costs for the purposes of this dissertation. The current financial cost per GB for typical network transfer out of Amazon EC2 is \$0.09 per GB. Transferring data into EC2 is free, but since the estimate is more important than the particulars of EC2, I will use \$0.09 per GB as the basis for estimating all network traffic. Proclaimed throughput varies from to 62-1,750 Mbps depending on the EC2 instances used. For the *temporal cost* (the amount of time required) for network traffic, a full 100 MBps will be assumed for estimating bandwidth. This is approximately what could be sustained on a 1 Gigabit internet connection after considering overhead.

Estimating the computing costs is a little more challenging. Amazon uses an

ECU metric to measure the compute capabilities of their EC2 resources. An ECU can be assumed to be approximately equivalent to a computing core available at Notre Dame's Center for Research Computing (CRC). So the cost of an ECU will be the basis for estimating computing costs, regardless of RAM, disk or any other features that might be available. Amazon charges $0.0096 per ECU hour based on an m5.large instance which is currently the cheapest instance that lists a fixed (not variable) ECU. However, startup/shutdown costs for server instances will be ignored for the sake of simplicity in these estimates. With these assumptions in mind, the temporal computing cost depends on the level of concurrency possible in the workflow and the concurrency available with compute resources. Maximum concurrency can be calculated from the PRUNE database. The temporal cost will be estimated without placing a limit on the number of compute resources concurrently available.

Table 6.4 estimes the temporal costs of network traffic in this theoretical situation with no limit on concurrent compute resources and no startup/shutdown cost needed in order to execute tasks in the workflow. The slowest task in the tables is simply one that took the longest to complete, in that stage, when the workflow was originally executed. The hours on the right are the time to complete that stage based on completion of the slowest task. It is assumed that all other tasks can complete more quickly than the slowest one since it is likely that a re-execution of the workflow will have the same performance properties. The times on the right side of table 6.4 are cumulative. In other words, if a change to the workflow occurred in Stage 0, all stages must be re-executed. The time listed in the table on the top-right side is how long it would take to perform a full re-execution of the BWA-GATK workflow in this ideal or best case scenario.

Table 6.5 shows the temporal cost (wait time) due to network traffic that might be experienced in each of the collaboration modes. Again, a 1 Gbps connection is assumed with actual available bandwidth of 100 MBps. The tables show that to some

TABLE 6.4

TEMPORAL COST OF EXECUTION

Stage changed	Slowest task (minutes)	Hours to final results
BWA-GATK 0	6 (10 cores)	0.28
BWA-GATK 1	<1 (10 cores)	0.18
BWA-GATK 2	<1 (10 cores)	0.17
BWA-GATK 3	<1 (100 cores)	0.16
BWA-GATK 4	<1 (100 cores)	0.16
BWA-GATK 5	<1 (100 cores)	0.15
BWA-GATK 6	<1 (100 cores)	0.14
BWA-GATK 7	<1 (20 cores)	0.14
BWA-GATK 8	8 (100 cores)	0.14
MCProduction 0	56 (1000 cores)	3.29
MCProduction 1	32 (1000 cores)	2.34
MCProduction 2	99 (1000 cores)	1.80
MCProduction 3	8 (1000 cores)	0.14
Census 0	25 (227 cores)	3.45
Census 1	5 (227 cores)	3.02
Census 2	6 (10 cores)	2.93
Census 3	<1 (10 cores)	2.82
Census 4	1 (1 cores)	2.82
Census 5	5 (1 cores)	2.79
Census 6	139 (681342 cores)	2.70
Census 7	22 (682 cores)	0.38

NOTE: Temporal cost of re-executing the workflow after evolutions in each stage in Pr1 mode (assuming the shown number of cores can be used concurrently, and that one stage must complete before the next one starts).

TABLE 6.5

TEMPORAL COST OF NETWORK

	Transfer time (hours:minutes:seconds)				
Changed	Pr1	Pr2	Pr3	Pr4	Pr5
BWA-GATK 0	00:01:11	00:01:11	00:01:18	00:04:46	00:04:46
BWA-GATK 1	00:01:11	<0:00:01	00:00:07	00:03:33	00:04:46
BWA-GATK 2	00:01:11	<0:00:01	00:00:07	00:03:33	00:04:46
BWA-GATK 3	00:01:11	<0:00:01	00:00:07	00:03:31	00:04:46
BWA-GATK 4	00:01:11	<0:00:01	00:00:07	00:03:17	00:04:46
BWA-GATK 5	00:01:11	<0:00:01	00:00:07	00:03:04	00:04:46
BWA-GATK 6	00:01:11	<0:00:01	00:00:07	00:02:45	00:04:46
BWA-GATK 7	00:01:11	<0:00:01	00:00:07	00:01:36	00:04:46
BWA-GATK 8	00:01:11	<0:00:01	00:00:06	00:00:06	00:04:46
MCProduction 0	<0:00:01	<0:00:01	00:01:07	00:14:53	00:14:53
MCProduction 1	<0:00:01	<0:00:01	00:01:07	00:14:19	00:14:53
MCProduction 2	<0:00:01	<0:00:01	00:01:07	00:04:32	00:14:53
MCProduction 3	<0:00:01	<0:00:01	00:01:07	00:01:07	00:14:53
Census 0	00:05:59	00:05:59	00:07:08	02:00:07	02:00:07
Census 1	00:05:59	00:00:13	00:01:21	01:53:50	02:00:07
Census 2	00:05:59	00:00:13	00:01:21	00:02:25	02:00:07
Census 3	00:05:59	00:00:13	00:01:21	00:02:23	02:00:07
Census 4	00:05:59	00:00:13	00:01:21	00:02:22	02:00:07
Census 5	00:05:59	00:00:13	00:01:21	00:02:16	02:00:07
Census 6	00:05:59	00:00:13	00:01:21	00:02:08	02:00:07
Census 7	00:05:59	00:00:06	00:01:14	00:01:14	02:00:07

NOTE: Temporal cost of network traffic after evolutions in each stage and in each of the collaboration modes.

degree the collaboration modes each have some advantages and disadvantages. To get a better picture the financial cost of each collaboration mode should be considered.

Figure 6.1 plots detailed behavior of the 3 workflows each with the 5 collaboration modes. The Census and BWA-GATK workflows are similar because they both have a lot of input data in Stage 0. The green triangles on the left images show how significant the input data is compared to all the data in the entire workflow. Both Pr2 and Pr3 come with a significant decrease in network traffic for almost all stages. Pr1 mode (again the green triangles) has a decent reduction in network traffic, but at the high execution cost for all workflows. The axes in the middle show the financial cost of using the resources indicated on the outer axes. The costs are very low in BWA-GATK with the execution cost never getting above a dollar. That scale may not merit even asking the question of which collaboration mode to use. However, looking at BWA-GATK and MCProduction can provide some insight about which mode might be the best choice if those workflows were to scale up proportionally.

A larger scale workflow can also give a better picture, and there just so happens to be one in the Census example. In that figure the execution cost estimates get over $5,000. In addition, it appears that most of that cost is in the latter stages, because Pr2 mode doesn't really see much benefit until the final stage. Pr3-Pr5 are the clear winners in terms of financial cost for the Census workflow, but it is not so clear for the other workflows. The temporal cost to transfer the Census data is over 2 hours for Pr5 (and the worst case for Pr4) in table 6.5). The temporal cost to re-execute the Census workflow is about 3.45 hours in table 6.4. Unless the makeup of the census workflow changes drastically, it should clearly use Pr1-Pr3, as Pr4,Pr5 are significantly more expensive both temporally and financially.

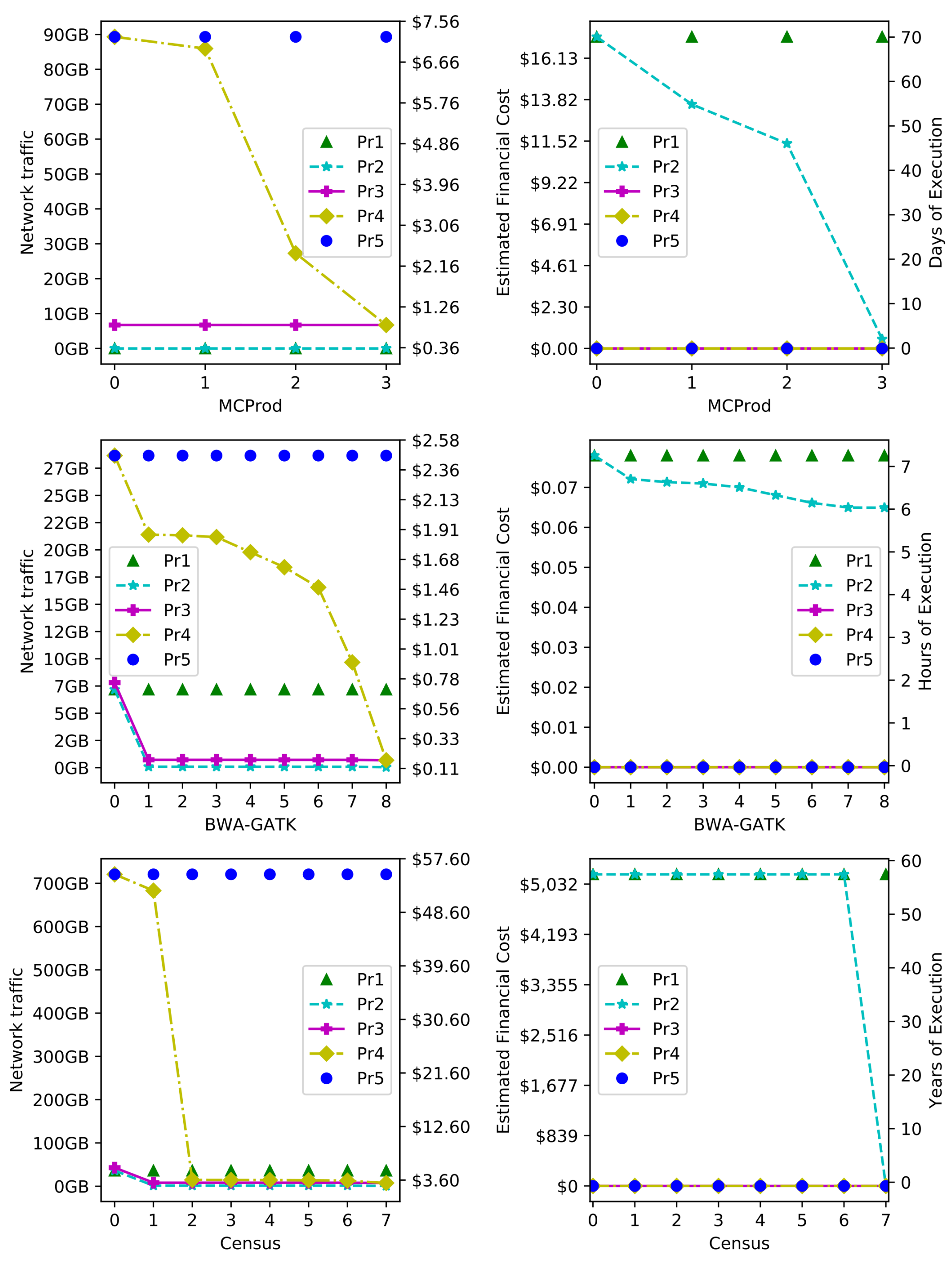


Figure 6.1. Financial Cost of Network/Execution

Plots of network and execution requirements of workflows in each of the collaboration modes. Financial costs are incorporated in the middle.

Once familiar with these plots, even just looking at them can help inform the right decision. Again in the census plot, it is clear that Pr2 is only better than Pr1 if changes happen at the last stage of the workflow. On the left side Pr4 provides significant reductions all the way down to Stage 2, when it ceases to be one of the best options. The cutoff is not so clear in the other workflows, as Pr2 and Pr4 demonstrate much more gradual usefulness in figure 6.1.

CHAPTER 7

CONCLUSION

7.1 Summary

In summary, a shift in intent and a greater focus on reproducibility needs to be adopted by researching scientists. Many existing efforts [94] to provide frameworks, middleware, and environments to support computational science are available. However, in general, reproducible research needs to be perceived by all involved as a more valuable contribution to science than non-reproducible research, rather than an inconvenient and somewhat unachievable ideal. There are differing opinions on the definition of reproducibility and many related terms. But the main goal is to encapsulate a scientific experiment executed by computers into a form that allows other collaborating scientists to re-execute part or all of that experiment.

However, just preserving and sharing the bits needed to execute the workflow is insufficient. Binary code and data can communicate low level operations between computers, but a higher level representation of the operations needs to be available which is designed for humans to understand. This higher level representation can be more useful to scientists especially when it can be modified to explore the parameters, operations can be replaced as desired, and other operations can be incorporated into a collaborator's work.

If the scientist can choose the granularity of these operations, they will be more effective as a collaborative communication tool than if the operations are based on generic system provided actions such as system calls.

Science can advance more quickly by building on the work of others than by competing for the greatest accomplishments and then allowing them to be obscured for others. However, just making something reproducible in all of these ways does not absolve scientists from an obligation to make sure the conclusions they draw from an experiment are appropriate and correct.

This encapsulation is difficult because there must necessarily be assumptions on what capabilities are required to interpret the encapsulation by scientists and by a computer. There may be an implicit assumption that a certain version of a library will be available, or a certain operating system. Or there may even an assumption that certain hardware will be available to other scientists, such as GPUs, x86, or ARM architectures. This work, in part, asserts that separating these assumptions (the environment) from the operations in a workflow is useful because the scientist can become accustomed to the assumptions and focus on their domain. If all assumptions are made explicit, system administrators can then be separately tasked with appropriately satisfying them.

Computers often use obscure names to guarantee uniqueness when referring to objects. Scientists want names to convey the purpose or place for those objects, and often redirect the name to something new when their workflow evolves. Attaching a version or timestamp to the scientist defined names helps to identify when something has changed, similar to the incrementing version numbers associated with evolution of software libraries. However, this can become even more difficult in a collaborative setting where multiple people use the same name for different objects concurrently. Usually the solution is to either force users to adopt a common namespace or give each their own namespace and affix the username to the name for an object. In this dissertation an approach is proposed where each user has their own namespace, but content and derivation based identifiers are also affixed to objects to provide the ability to recognize identical objects between workflow.

The derivation based identifiers are especially important in light of a Preserve First mentality which is needed in computational science. Even before tasks are executed derivation based identifiers are available for their generated results. These identifiers can immediately be shared with collaborators and both users could potentially execute a workflow concurrently while retaining the ability to identify those objects as the same from a global system perspective.

The additional use of content based identifiers can enable the collaborators to confirm that results are identical in the case of a deterministic workflow, but with a non-deterministic workflow, collaborators must rely on the derivation based workflows and an assumption that both systems are behaving equivalently. Unfortunately it is not always possible to ensure non-deterministic behavior in a workflow.

Also, the Preserve First ideal also helps avoid implicit assumptions when the tasks are executed in a sandbox. Finally, with the Preserve First approach enables intermediate files to be treated like a cache which can be flushed if storage resources become overloaded, because the system can re-execute the task to retrieve it's results if they are needed later on.

These additional features come at a cost, but in the experiments used the cost was negligible compared to the workflow tasks themselves. And the workflows were still able to scale in spite of the additional actions.

But perhaps the most significant benefit of applying these capabilities to a workflow system is the ability to minimize collaboration overhead. Significant reductions to re-execution and network transfer are possible while still achieving the most valuable requirements for a collaborative system. In addition the choice of collaboration modes does not need to be made in advance. The optimal solution in specific circumstances can be calculated based on the relative cost of execution compared to network traffic.

7.2 Successes

The overhead of obtaining content based identifiers for all Files was around 1% of a large workflow, but made it so that identical files across collaborating systems could be easily identified and transferring those files could be avoided.

Forcing the user to specify desired tasks and letting them be executed by the system rather than executing them directly enabled the ability to successfully put a quota on the disk usage of the workflow without losing any information about the historical evolution of the workflow.

PRUNE enables new collaboration modes which allow the user to choose between two optimized options. 1) Eliminate re-execution for the collaborator and reduce network traffic to 6%-23% of the full workflow amount on average, or 2) Minimize network transfer to negligible amounts, and reduce re-execution down to 69%-84% of the full workflow execution time on average.

7.3 Limitations

As shown in figure 5.1 the specification of a workflow is much more verbose in PRUNE than in a simple script. It is likely that the syntax could be simplified, but is still likely to be more complicated than a workflow specified without PRUNE. This intellectual overhead is a significant drawback, and can only be offset by separate benefits such as a disk quota and easier collaboration which can help to reduce the intellectual overhead.

There could be more external motivation to ensure scientific computing publications are reproducible, that go beyond what technologies are likely to do. Perhaps a metric needs to be created to measure reproducibility so that a sizable prize [19] could be offered to the most reproducible scientific computing publication. In the absence of a funding source, maybe publishers could simply start offering a Most Reproducible Paper Award similar to the Best Paper Awards commonly given. A little notoriety could go a long way in encouraging scientists to strive for reproducibility.

Scientists could also benefit from more exposure to any good software development practices [193], that are not included in their education or training. There is no guarantee that even the most reproducible techniques available today will be reproducible at any given date in the future. Therefore, a community of experts is needed who are willing to maintain a collection of relevant research. This community would secure funds, decide what research is no longer relevant, what research needs to be updated to accommodate technological advances, and develop additional tools to encourage new commitments to the reproducibility of computational scientific research.

Even research that was fully reproducible at the time of publication may cease to be usable in the ensuing year as a result of unexpected hardware or software evolution. The Madagascar project [69] and observations of it's use after a couple of years [66] make a strong arguments for making research preservation a community effort, rather than placing the burden entirely on the original researcher.

This is not an easy task and generic open source software techniques [65] are not always applicable. Strides have been made and lessons learned in very specific situations [183], but more needs to be done for scientific workflows as a whole.

7.4 Future work

Scientists rightly feel some ownership over their discoveries. They deserve credit or acclaim for their work, and they should have some control over the **distribution** of their efforts. They should be able to manage the **integration** of other published research into their own work, without having to re-implement everything themselves. However, currently, these abilities often come in the form of decreased **convenience** and/or **performance**, and are too often sacrificed in fear of publication delays. In such cases, scientists may prefer to work completely on their own, planning for reproducibility later on.

7.4.1 Distribution

Access requests for objects that are restricted for any reason could be automated and alternatives could be provided when access is not possible. Also, a scientist might want to grant access to any portion of their workflow on a case-by-case basis. For other portions of the workflow, it might be acceptable to grant access automatically under certain conditions. Appropriate conditions could be membership in some sort of group or organization, or agreement to attribute credit for borrowed portions in future publications. Content-based IDs for shared objects may help identify provenance in cases where attribution is not retained for whatever reason.

7.4.2 Integration

Ideally computational science could be a very collaborative effort with improvements frequently being published at all levels of a workflow. Even with highly sensitive information, if a 'scrubbed' but statistically equivalent version of that information is available, individuals without special access could potentially contribute to improvements in the workflow. Scientists could configure when notifications about new versions of a shared object are available and how to deal with incorporating those changes into their own research.

7.4.3 Convenience

A balance between user convenience and computer requirements can be difficult to achieve. A mapping between the two is often used, such as when source code (for users) is translated into machine code (for computers). Object naming in a collaborative workflow is even more difficult because the preferences of multiple users should be accommodated. Another problem is managing storage space not just in a single repository (as in my previous research), but also across a collaborative workflow. These seem to be the most significant pain points, but other situations might be bigger problems in other sciences. Users might tolerate extra effort for reproducibility if even more effort is eliminated in other challenging areas. More research is needed to discover the worst pain points for users in scientific computing in general.

With that in mind and with a focus on usability, additional tools are needed to reduce the intellectual load on scientists so that they and their collaborators can focus on their scientific domain instead of on the computer science. In general, the less work the scientist has to do to execute their research workflows and evolutions of their workflows, the more likely it is that their collaborators will be able to accept and benefit from that research. In this vein, efforts in various other areas can be continued, including; validation, infrastructure independent and performant execution, recording, sharing and synchronizing of workflows.

7.4.4 Performance

Scientists are concerned with how quickly results can be achieved without excessive financial costs or delays. If reproducibility increases either of those barriers, scientists will be less likely to collaborate. Cloud resources, such as Amazon EC2 or GCE can greatly reduce delays, and shared resources between collaborators could reduce the costs. Either one of these could outweigh the barrier to reproducibility compared to existing solutions without such resources. Better tools for estimating the (financial or temporal) cost of computing need to be created in order derive mutual benefits from shared resources.

7.5 Reproducibility of this paper

As I mentioned previously, issues were encountered with satisfying data dependencies with each workflow both from technical and legal perspectives. Obtaining permission to publicly share these workflows in their entirety turned out to be less successful than addressing the technical challenges. I share all files and data that I was permitted to.

The shareable result of using PRUNE comes in the form of an exported file that includes the Files, Tasks, and Environments used to execute the workflow. These final files are available on gitlab and include all the data we are permitted to share:

https://gitlab.com/pivie/ccpe-prune/
doi:10.7274/R0TD9VDQ
http://bit.ly/2hNNqD5

These files can be loaded into a PRUNE repository by executing the included Python scripts. For more information the PRUNE User's Manual can be found at:

http://ccl.cse.nd.edu/software/prune/

BIBLIOGRAPHY

1. A. Abedi, A. Heard, and T. Brecht. Conducting repeatable experiments and fair comparisons using 802.11 n mimo networks. *ACM SIGOPS Operating Systems Review*, 49(1):41–50, 2015.

2. E. Abraham, H. Kress-Gazit, L. Natale, and A. Tacchella. Computer-assisted engineering for robotics and autonomous systems (dagstuhl seminar 17071). In *Dagstuhl Reports*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

3. M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

4. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

5. J. Alwall, A. Ballestrero, P. Bartalini, S. Belov, E. Boos, A. Buckley, J. M. Butterworth, L. Dudko, S. Frixione, L. Garren, et al. A standard format for les houches event files. *Computer Physics Communications*, 176(4):300–304, 2007.

6. K. Amin, G. Von Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. Gridant: A client-controllable grid workflow system. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.

7. P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes, and L. Stojanovic. Common Workflow Language, v1.0. 7 2016. doi: 10.6084/m9.figshare.3115156.v2. URL `https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156`.

8. P. Anderson and E. Smith. Configuration tools: Working together. In *LISA*, pages 31–37, 2005.

9. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, 2003.

10. A. Asserson, K. G. Jeffery, and A. Lopatenko. *CERIF: past, present and future: an overview.* euroCRIS, 2002.

11. L. Aversano, A. Cimitile, P. Gallucci, and M. L. Villani. Flowmanager: a workflow management system based on petri nets. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 1054–1059. IEEE, 2002.

12. D. H. Bailey, J. M. Borwein, R. P. Brent, and M. Reisi. Reproducibility in computational science: a case study: randomness of the digits of pi. *Experimental Mathematics*, pages 1–8, 2016.

13. L. A. Barba. The hard road to reproducibility. *Science*, 354(6308):142–142, 2016.

14. R. M. Bastos and D. D. A. Ruiz. Extending uml activity diagram for workflow modeling in production systems. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 3786–3795. IEEE, 2002.

15. L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *VIS 05. IEEE Visualization, 2005.*, pages 135–142. IEEE, 2005.

16. O. Beaumont, J. Erhel, and B. Philippe. Aquarels: A problem-solving environment for validating scientific software. In *Enabling Technologies for Computational Science*, pages 351–362. Springer, 2000.

17. C. G. Begley and L. M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

18. 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*, pages 97–104. IEEE, 2011.

19. R. Bell, J. Bennett, Y. Koren, and C. Volinsky. The million dollar programming prize. *IEEE Spectrum*, 46(5), 2009.

20. J. Blomer, P. Buncic, and T. Fuhrmann. Cernvm-fs: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56. ACM, 2011.

21. B. Boehm. Software risk management. In *European Software Engineering Conference*, pages 1–19. Springer, 1989.

22. C. BOONMEE and S. KAWATA. Computer-assisted simulation environment for partial-differential-equation problem. *Transactions of the Japan Society for Computational Engineering and Science*, 1998:19980002–19980002, 1998.

23. R. Bramley, B. Char, D. Gannon, T. T. Hewett, C. Johnson, and J. R. Rice. Workshop on scientific knowledge, information and computing (sidekic 98). *Enabling Technologies for Computational Science: Frameworks, Middleware and Environments*, 548:19, 2000.

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

25. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0, 2008.

26. J. Bresnahan, T. Freeman, D. LaBissoniere, and K. Keahey. Managing appliance launches in infrastructure clouds. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, page 12. ACM, 2011.

27. E. A. Brewer. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 167–167. ACM, 2015.

28. T. Buchert, C. Ruiz, L. Nussbaum, and O. Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45:1–12, 2015.

29. J. B. Buckheit and D. L. Donoho. *Wavelab and reproducible research.* Springer, 1995.

30. 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.

31. P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *Database Theory - ICDT 2001*, pages 316–330. Springer, 2001.

32. S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

33. F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.

34. D. Chapp, T. Johnston, and M. Taufer. On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 166–175. IEEE, 2015.

35. 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.

36. CircleCI. Continuous integration and delivery - circleci. `https://circleci.com/`, 2017. (Accessed on 08/02/2017).

37. J. Claerbout. Making scientific contributions reproducible. `http://sepwww.stanford.edu/oldsep/matt/join/redoc/web/iris.html`, 2011. (Accessed on 07/11/2016).

38. J. Claerbout and M. Karrenbach. Electronic documents give reproducible research a new meaning. In *Proc. 62nd Ann. Int. Meeting of the Soc. of Exploration Geophysics*, pages 601–604, 1992.

39. J. F. Claerbout. Electronic document preface. Technical Report SEP-72, Stanford Exploration Project, 1991. URL `http://sepwww.stanford.edu/public/docs/sep72/jon3/paper_html/node4.html`.

40. 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*, volume 3, pages 73–78, 2003.

41. N. R. Council et al. *Assessing the reliability of complex models: mathematical and statistical foundations of verification, validation, and uncertainty quantification.* National Academies Press, 2012.

42. L. Courtès and R. Wurmus. Reproducible and user-controlled software environments in hpc with guix. In *European Conference on Parallel Processing*, pages 579–591. Springer, 2015.

43. J. Crocker and M. L. Cooper. Addressing scientific fraud. *Science*, 334(6060): 1182–1182, 2011.

44. D. Dabdub, K. M. Chandy, and T. T. Hewett. Managing specificity and generality: tailoring general archetypal pses to specific users. In *Enabling Technologies for Computational Science*, pages 65–77. Springer, 2000.

45. A. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, 2012.

46. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

47. E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

48. D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

49. P. Di Tommaso, E. Floden, M. Chatzou, and C. Notredame. Using the nextflow framework for reproducible in-silico omics analyses across clusters and clouds. *PeerJ Preprints*, 5:e2796v1, 2017.

50. A. Dienstfrey and R. Boisvert. *Uncertainty Quantification in Scientific Computing: 10th IFIP WG 2.5 Working Conference, WoCoUQ 2011, Boulder, CO, USA, August 1-4, 2011, Revised Selected Papers*, volume 377. Springer, 2012.

51. E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

52. E. Dolstra and A. Löh. Nixos: A purely functional linux distribution. In *ACM Sigplan Notices*, volume 43, pages 367–378. ACM, 2008.

53. C. Dominik. *The Org Mode 7 Reference Manual-Organize your life with GNU Emacs*. Network Theory Ltd., 2010.

54. C. Drummond. Replicability is not reproducibility: nor is it good science. 2009.

55. M. Dumas and A. H. Ter Hofstede. Uml activity diagrams as a workflow specification language. In *International Conference on the Unified Modeling Language*, pages 76–90. Springer, 2001.

56. P. M. Duvall. *Continuous Integration*. Pearson Education India, 2007.

57. S. Edwards, X. Liu, and N. Riga. Creating repeatable computer science and networking experiments on shared, public testbeds. *ACM SIGOPS Operating Systems Review*, 49(1):90–99, 2015.

58. B. Einarsson. *Accuracy and reliability in scientific computing*. SIAM, 2005.

59. J. I. Elkind, S. K. Card, and J. Hochberg. *Human performance models for computer-aided engineering*. Academic Press, 2014.

60. J. Emeras, B. Bzeznik, O. Richard, Y. Georgiou, and C. Ruiz. Reconstructing the software environment of an experiment with kameleon. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent & scalable system technologies*, page 16. ACM, 2012.

61. H. Erdogmus. On the effectiveness of test-first approach to programming. 2005.

62. FamilySearch.org. "United States Census, 1850-1940." Database. URL `http://FamilySearch.org/`. Citing NARA microfilm publication T626. Washington, D.C.: National Archives and Records Administration, 2002.

63. D. G. Feitelson. From repeatability to reproducibility and corroboration. *ACM SIGOPS Operating Systems Review*, 49(1):3–11, 2015.

64. S. I. Feldman. Make – A program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

65. K. Fogel. *Producing open source software: How to run a successful free software project.* " O'Reilly Media, Inc.", 2005.

66. S. Fomel. Reproducible research as a community effort: Lessons from the madagascar project. *Computing in Science & Engineering*, 17(1):20–26, 2015.

67. S. Fomel and G. Hennenfent. Reproducible computational experiments using scons. In *ICASSP (4)*, pages 1257–1260, 2007.

68. S. Fomel, M. Schwab, and J. Schroeder. Empowering sep?s documents. *SEP-94: Stanford Exploration Project*, pages 339–361, 1997.

69. S. Fomel, P. Sava, I. Vlad, Y. Liu, and V. Bashkardin. Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments. *Journal of Open Research Software*, 1(1), 2013.

70. I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 37–46. IEEE, 2002.

71. M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122, 2006.

72. J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3), 2008.

73. J. Frey. Condor dagman: Handling inter-job dependencies. *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

74. H. Fuju, S. Kawata, H. Sugiura, Y. Saitoh, Y. Hayase, H. Usami, M. Yamada, Y. Miyahara, H. Kanazawa, and T. Kikuchi. Scientific simulation execution support on a closed distributed computer environment. In *e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on*, pages 109–109. IEEE, 2006.

75. E. Gallopoulos, E. Houstis, and J. R. Rice. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Computational Science and Engineering*, 1(2):11–23, 1994.

76. S. Gallopoulos, E. N. Houstis, and J. R. Rice. Future research directions in problem solving environments for computational science. 1992.

77. R. Garcia and M. T. Valente. Nextflow: Business process meets mapping frameworks. [Online; accessed 9-Mar-2017].

78. D. Garijo, O. Corcho, and Y. Gil. Detecting common scientific workflow fragments using templates and execution provenance. In *Proceedings of the seventh international conference on Knowledge capture*, pages 33–40. ACM, 2013.

79. M. Gavish and D. Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4:637–647, 2011.

80. 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*, 15(10):1451–1455, 2005.

81. C. Goble. Results may vary. reproducibility, open science, and all that jazz, 7 2013. URL `http://www.slideshare.net/carolegoble/ismb2013-keynotecleangoble/17`. Keynote given by Carole Goble on 23rd July 2013 at ISMB/ECCB 2013 [Accessed: 2016 11 09].

82. O. Gómez, N. Juristo, and S. Vegas. Replication, reproduction and re-analysis: Three ways for verifying experimental findings. In *Proceedings of the 1st international workshop on replication in empirical software engineering research (RESER 2010), Cape Town, South Africa*, 2010.

83. A. Goodman, A. Pepe, A. W. Blocker, C. L. Borgman, K. Cranmer, M. Crosas, R. Di Stefano, Y. Gil, P. Groth, M. Hedstrom, et al. Ten simple rules for the care and feeding of scientific data. *PLoS computational biology*, 10(4):e1003542, 2014.

84. P. Groot, A. Serebrenik, and M. v. Eekelen. Proceedings of verification and validation of software systems (vvss2007). pages 65–77, 2007.

85. Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, and Y. Liu. Grid-flow: a grid-enabled scientific workflow system with a petri-net-based interface. *Concurrency and Computation: Practice and Experience*, 18 (10):1115–1140, 2006.

86. P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

87. M. Hashimoto. *Vagrant: Up and Running.* " O'Reilly Media, Inc.", 2013.

88. L. Hatton and G. Warr. Full computational reproducibility in biological science: Methods, software and a case study in protein biology. *arXiv preprint arXiv:1608.06897*, 2016.

89. F. Hernández, P. Bangalore, J. Gray, and K. Reilly. A graphical modeling environment for the generation of workflows for the globus toolkit. In *Component Models and Systems for Grid Applications*, pages 79–96. Springer, 2005.

90. T. T. Hewett and J. L. DePaul. Toward a human centered scientific problem solving environment. *KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, pages 79–90, 2000.

91. B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

92. A. Hoheisel. User tools and languages for graph-based grid workflows. *Concurrency and Computation: Practice and Experience*, 18(10):1101–1113, 2006.

93. D. Hollingsworth and U. Hampshire. Workflow management coalition: The workflow reference model. *Document Number TC00-1003*, 19, 1995.

94. E. N. Houstis, J. R. Rice, E. Gallopoulos, and R. Bramley. *Enabling Technologies for Computational Science: Frameworks, Middleware and Environments*, volume 548. Springer Science & Business Media, 2012.

95. B. Howe. Cde: A tool for creating portable experimental software packages. *Computing in Science & Engineering*, 14(4):32–35, 2012.

96. B. Howe. Virtual appliances, cloud computing, and reproducible research. *Computing in Science & Engineering*, 14(4):36–41, 2012.

97. 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*, 34(suppl 2):W729–W732, 2006.

98. J. P. Ioannidis. Why most published research findings are false. *PLoS Med*, 2 (8):e124, 2005.

99. P. Ivie and D. Thain. DeltaDB: A Scalable Database Design for Time-Varying Schema-Free Data. In *IEEE International Congress on Big Data (BigData 2014)*, pages 104–111, 2014.

100. P. Ivie and D. Thain. Prune: A preserving run environment for reproducible scientific computing. *IEEE Conference on e-Science*, 2016.

101. P. Ivie and D. Thain. PRUNE: A Preserving Run Environment for Reproducible Computing. In *IEEE Conference on e-Science*, 2016.

102. P. Ivie and D. Thain. PRUNE: A Preserving Run Environment for Reproducible Computing. IEEE Conference on e-Science, 2016.

103. P. Ivie, C. Zheng, and D. Thain. A first look at reproducibility and non-determinism in cms software and root data. 2016.

104. P. Ivie, C. Zheng, and D. Thain. An analysis of reproducibility and non-determinism in hep software and root data. In *Journal of Physics: Conference Series*. IOP Publishing, 2016.

105. P. Ivie, C. C. Zheng, and D. Thain. An Analysis of Reproducibility and Non-Determinism in HEP Software and ROOT Data. In *International Conference on Computing in High Energy and Nuclear Physics*, 2016.

106. B. R. Jasny, G. Chin, L. Chong, and S. Vignieri. Again, and again, and again? *Science*, 334(6060):1225–1225, 2011.

107. E. Jeanvoine, L. Sarzyniec, and L. Nussbaum. Kadeploy3: Efficient and scalable operating system provisioning for clusters. *USENIX; login:*, 38(1):38–44, 2013.

108. Jenkins. Jenkins. `https://jenkins.io/`, 2017. (Accessed on 08/02/2017).

109. I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1561–1570. IEEE, 2017.

110. C. Johnson. Top scientific visualization research problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, 2004.

111. S. Kawata. Computer assisted problem solving environment (pse). In *Encyclopedia of Information Science and Technology, Third Edition*, pages 1251–1260. IGI Global, 2015.

112. J. Kim, E. Deelman, Y. Gil, G. Mehta, and V. Ratnakar. Provenance trails in the wings/pegasus system. *Concurrency and Computation: Practice and Experience*, 20(5):587–597, 2008.

113. J. Klinginsmith, M. Mahoui, and Y. M. Wu. Towards reproducible escience in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 582–586. IEEE, 2011.

114. T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, et al. Jupyter notebooks?a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, page 87, 2016.

115. S. Knight. Building software with scons. *Computing in Science and Engineering*, 7(1):79–88, 2005.

116. I. Krsul, A. Ganguly, J. Zhang, J. A. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 7–7. IEEE, 2004.

117. G. M. Kurtzer. Singularity 2.1.2 - Linux application and environment containers for science, Aug. 2016. URL `https://doi.org/10.5281/zenodo.60736`.

118. C. Laine, S. N. Goodman, M. E. Griswold, and H. C. Sox. Reproducible research: moving toward research the public can really trust. *Annals of Internal Medicine*, 146(6):450–453, 2007.

119. D. T. Larsen, J. Blomer, P. Buncic, I. Charalampidis, and A. Haratyunyan. Long-term preservation of analysis software environment. In *Journal of Physics: Conference Series*, volume 396, page 032064. IOP Publishing, 2012.

120. Y.-L. Lee, M. E. Barkey, and H.-T. Kang. *Metal fatigue analysis handbook: practical problem-solving techniques for computer-aided engineering.* Elsevier, 2011.

121. J. T. Leek and R. D. Peng. Opinion: Reproducible research can still be wrong: Adopting a prevention approach. *Proceedings of the National Academy of Sciences*, 112(6):1645–1646, 2015.

122. R. J. LeVeque, I. M. Mitchell, and V. Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science and Engineering*, 14(4):13, 2012.

123. F. Leymann et al. Web services flow language (wsfl 1.0), 2001.

124. J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015.

125. J. Loeliger. Collaborating with git. *Linux Magazine, June*, 2006.

126. D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62. ACM, 2010.

127. J. Loope. *Managing Infrastructure with Puppet.* " O'Reilly Media, Inc.", 2011.

128. B. Ludascher, I. Altintas, and A. Gupta. Compiling abstract scientific workflows into web service workflows. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 251–254. IEEE, 2003.

129. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

130. C. Lueninghoener. Getting started with configuration management, 2011.

131. C. Lynch. Big data: How do your data grow? *Nature*, 455(7209):28–29, 2008.

132. B. Marwick. Computational reproducibility in archaeological research: Basic principles and a case study of their implementation. *Journal of Archaeological Method and Theory*, pages 1–27, 2016.

133. A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. Iceni dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, volume 634, page 627, 2003.

134. 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. International Society for Optics and Photonics, SPIE Astronomical Telescopes+ Instrumentation, 2016.

135. R. Mecklenburg. *Managing projects with GNU make.* " O'Reilly Media, Inc.", 2004.

136. 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, 2015. ACM.

137. 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.

138. D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

139. R. C. Merkle. Method of providing digital signatures, Jan. 5 1982. US Patent 4,309,569.

140. J. P. Mesirov. Accessible reproducible research. *Science*, 327(5964):415–416, 2010.

141. S. Meyer, P. Healy, T. Lynn, and J. Morrison. Quality assurance for open source software configuration management. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 454–461. IEEE, 2013.

142. R. E. Millsap and H. T. Everson. Methodology review: Statistical approaches for assessing measurement bias. *Applied psychological measurement*, 17(4):297–334, 1993.

143. G. Molnár and B. Csapó. Exploration and learning strategies in an interactive problem-solving environment at the beginning of higher education studies. 2017.

144. K. Murrell. The harwell dekatron computer. In *Making the History of Computing Relevant*, pages 309–313. Springer, 2013.

145. 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*, pages 485–494. IEEE, 2015.

146. C. J. Oates, J. Q. Smith, and S. Mukherjee. Estimating causal structure using conditional dag models. *Journal of Machine Learning Research*, 17(54):1–23, 2016.

147. W. L. Oberkampf and C. J. Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

148. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

149. T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

150. S. Pandey. Investigating community, reliability and usability of cfengine, chef and puppet. 2012.

151. R. Peng. The reproducibility crisis in science: A statistical counterattack. *Significance*, 12(3):30–32, 2015.

152. R. D. Peng. Reproducible research in computational science. *Science*, 334(6060): 1226–1227, 2011.

153. 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.

154. 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.

155. K. Popper. *The logic of scientific discovery.* Routledge, 2005.

156. F. Prinz, T. Schlange, and K. Asadullah. Believe it or not: how much can we rely on published data on potential drug targets? *Nature reviews Drug discovery*, 10(9):712–712, 2011.

157. T. Proebsting and A. M. Warren. Repeatability and benefaction in computer systems research. 2015.

158. M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonier. The jupyter/ipython architecture: a unified view of computational research, from interactive exploration to communication and publication. In *AGU Fall Meeting Abstracts*, volume 1, page 07, 2014.

159. A. Rajasekar, R. Moore, C.-y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, et al. irods primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–143, 2010.

160. J. M. Ray. *Research data management: Practical strategies for information professionals.* Purdue University Press, 2014.

161. J. R. Rice. Future challenges for scientific simulation. In *Enabling Technologies for Computational Science*, pages 7–17. Springer, 2000.

162. C. Ruiz, O. Richard, and J. Emeras. Reproducible software appliances for experimentation. In *Testbeds and Research Infrastructure: Development of Networks and Communities*, pages 33–42. Springer, 2014.

163. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The.* Pearson Higher Education, 2004.

164. D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 110–123. ACM, 1999.

165. M. Schwab, M. Karrenbach, and J. Claerbout. Making scientific computations reproducible. *Computing in Science & Engineering*, 2(6):61–67, 2000.

166. B. Sierman. The scape policy framework, maturity levels and the need for realistic preservation policies. *IPRES 2014 proceedings*, page 259, 2014.

167. Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *ACM Sigmod Record*, 34(3):31–36, 2005.

168. M. P. Singh and M. A. Vouk. Scientific workflows: scientific computing meets transactional workflows. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, pages 28–34, 1996.

169. L. Stanisic, A. Legrand, and V. Danjean. An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):61–70, 2015.

170. V. Stodden. Trust your science? open your data and code. *Amstat News*, pages 21–22, 2011.

171. V. Stodden and S. Miguez. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Available at SSRN 2322276*, 2013.

172. V. Stodden, J. Borwein, and D. H. Bailey. Setting the default to reproducible. *computational science research. SIAM News*, 46:4–6, 2013.

173. V. Stodden, F. Leisch, and R. D. Peng. *Implementing reproducible research.* CRC Press, 2014.

174. S. Sun, L. Lannom, and B. Boesch. Handle system overview. Technical report, 2003.

175. M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *On the move to meaningful Internet systems 2003: CoopIS, DOA, and ODBASE*, pages 603–620. Springer, 2003.

176. I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*, pages 320–339. Springer, 2007.

177. I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: scientific workflows for grids.* Springer Publishing Company, Incorporated, 2014.

178. M. Taylor and S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation.* " O'Reilly Media, Inc.", 2014.

179. T. Teramoto, T. Okada, and S. Kawata. A distributed education-support pse system. In *e-Science and Grid Computing, IEEE International Conference on*, pages 516–520. IEEE, 2007.

180. 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.

181. TravisCI. Travis ci - test and deploy your code with confidence. `https://travis-ci.org/`, 2017. (Accessed on 08/02/2017).

182. C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th international conference on Software Engineering*, pages 178–188. IEEE Computer Society, 2007.

183. M. J. Turk. Scaling a code in the human dimension. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 69. ACM, 2013.

184. A. Vahdat and T. E. Anderson. Transparent result caching. In *USENIX Annual Technical Conference*, 1998.

185. W. Van Der Aalst and K. M. Van Hee. *Workflow management: models, methods, and systems*. MIT press, 2004.

186. W. M. Van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

187. W. M. Van Der Aalst and A. H. Ter Hofstede. Yawl: yet another workflow language. *Information systems*, 30(4):245–275, 2005.

188. S. Van Der Burg, M. de Jonge, E. Dolstra, and E. Visser. Software deployment in a dynamic cloud: From device to service orientation in a hospital environment. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 61–66. IEEE Computer Society, 2009.

189. 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. 18:8190, 2016.

190. M. Varia, B. Price, N. Hwang, A. Hamlin, J. Herzog, J. Poland, M. Reschly, S. Yakoubov, and R. K. Cunningham. Automated assessment of secure search systems. *ACM SIGOPS Operating Systems Review*, 49(1):22–30, 2015.

191. H. Verbeek, A. Hirnschall, and W. M. van der Aalst. Xrl/flower: Supporting inter-organizational workflows using xml/petri-net technology. In *International Workshop on Web Services, E-Business, and the Semantic Web*, pages 93–108. Springer, 2002.

192. G. Von Laszewski, M. Hategan, and D. Kodeboyina. Java cog kit workflow. In *Workflows for e-Science*, pages 340–356. Springer, 2007.

193. G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.

194. A. Woodard, M. Wolf, C. Mueller, N. Valls, B. Tovar, P. Donnelly, P. Ivie, K. H. Anampa, P. Brenner, D. Thain, K. Lannon, and M. Hildreth. Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster. In *IEEE Conference on Cluster Computing*, 2015.

195. R. P. Yale. Reproducible research. *Computing in Science & Engineering*, 12(5): 8–13, 2010.

196. J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 119–128. IEEE, 2004.

197. J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

198. X. Zhao, E. R. Boose, Y. Brun, B. S. Lerner, and L. J. Osterweil. Supporting undo and redo in scientific data analysis. In *TaPP*, 2013.

This document was prepared & typeset with pdfLaTeX, *and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.2017.2[2017/05/09])*