SOFTWARE TOOLS TO FACILITATE RESEARCH PROGRAMMING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Philip Jia Guo

May 2012

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dawson Engler, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jeffrey Heer

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Margo Seltzer

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

Research programming is a type of programming activity where the goal is to write computer programs to obtain insights from data. Millions of professionals in fields ranging from science, engineering, business, finance, public policy, and journalism, as well as numerous students and computer hobbyists, all perform research programming on a daily basis.

My thesis is that by understanding the unique challenges faced during research programming, it becomes possible to apply techniques from dynamic program analysis, mixed-initiative recommendation systems, and OS-level tracing to make research programmers more productive.

This dissertation characterizes the research programming process, describes typical challenges faced by research programmers, and presents five software tools that I have developed to address some key challenges. 1.) Proactive Wrangler is an interactive graphical tool that helps research programmers reformat and clean data prior to analysis. 2.) IncPy is a Python interpreter that speeds up the data analysis scripting cycle and helps programmers manage code and data dependencies. 3.) SlopPy is a Python interpreter that automatically makes existing scripts error-tolerant, thereby also speeding up the data analysis scripting cycle. 4.) Burrito is a Linux-based system that helps programmers organize, annotate, and recall past insights about their experiments. 5.) CDE is a software packaging tool that makes it easy to deploy, archive, and share research code. Taken together, these five tools enable research programmers to iterate and potentially discover insights faster by offloading the burdens of data management and provenance to the computer.

Acknowledgments

For the past fifteen years, I have maintained a personal website where I write about a wide range of technical and personal topics. Here is one of the most amusing emails that I have ever received from a website reader:

> Hi, Mr. Guo:
>
> I found your website when I search some education issues for the Asians. I saw you are in the 6th year of the PHD and a little bit confused:
>
> Is it normal taking so long and still not graduated or you just like to stay in school? I thought only the Medical program will take that long, but for computer science??
>
> I heard some horror stories before that the reason why taking so long is that some boss just want to use the students until bone dry and try to delay the graduation as late as possible. Hope that is not the case here.

I am happy to report that I was not sucked "bone dry"! I had one of the most enjoyable, enriching, and fulfilling Ph.D. experiences that anybody could hope for.

Here is a passage that I wrote in my research notebook in the summer of 2006, just a few months before entering the Stanford Computer Science Ph.D. program:

> [I'd like to pursue] research into software development tools for non-software engineers, but rather for scientists, engineers, and researchers who need to program for their jobs—they're not gonna care about specs., model checking, etc...—they just want pragmatic, lightweight, and conceptually simple tools that they can pick up quickly and use all the time.

I feel so fortunate and grateful that I have been given the opportunity over the past six years to execute on this initial vision to turn it into my Ph.D. dissertation. This dream would not have been possible without the help of many people, including my advisor, dissertation reading committee, colleagues, friends, and family.

First, I want to mention one memorable trait about each member of my dissertation reading committee that I hope to cultivate as I begin my own career: Dawson Engler, my Ph.D. advisor, is my inspiration for the value of seeking simple solutions and focusing intensely on pursuing them; Jeffrey Heer embodies a high level of professional effectiveness that I would be lucky to partially attain in the future; and Margo Seltzer is my role model for how to be a wonderful mentor to younger colleagues.

I would also like to thank the colleagues who directly helped me on the research projects that comprise this dissertation. It goes without saying that they are all smart, hardworking, and talented, but what I cherish most is their generosity. Here is everyone listed in alphabetical order: Elaine Angelino, Christian Bird, Peter Boonstoppel, Joel Brandt, Suhabe Bugrara, Yaroslav Bulatov, Ewen Cheslack-Postava, Isil Dillig, Kathleen Fisher, Imran Haque, Peter Hawkins, Kurtis Heimerl, Joe Hellerstein, Paul Heymann, Bill Howe, Robert Ikeda, Sean Kandel, Eunsuk Kang, Seungbeom Kim, Scott Klemmer, Greg Little, Adam Marcus, Cory McLean, Leo Meyerovich, Rob Miller, Adam Oliner, Fernando Perez, David Ramos, Martin Rinard, Marc Schaub, Rahul Sharma, Rishabh Singh, Jean Yang, and Tom Zimmermann.

I am deeply appreciative of the NSF and NDSEG graduate research fellowships for funding the first five years of my Ph.D. studies. Other sources of funding include the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024, NSF grant CCF-0964173, the Boeing Company, and Greenplum/EMC.

Finally, nothing would have meaning if I did not have the support of my friends and family, especially my wife, Lisa Mai, and parents, Sam Nan Guo and Min Zhou.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

People across a diverse range of science, engineering, and business-related disciplines spend their workdays writing, executing, debugging, and interpreting the outputs of computer programs. Throughout this dissertation, I will use the term ***research programming*** to refer to a type of programming activity where the goal is to *obtain insights from data*. Here are some examples of research programming:

- **Science**: Computational scientists in fields ranging from bioinformatics to neuroscience write programs to analyze data sets and make scientific discoveries.
- **Engineering**: Engineers perform experiments to optimize the efficacy of machine learning algorithms by testing on data sets, adjusting their code, tuning execution parameters, and graphing the resulting performance characteristics.
- **Business**: Web marketing analysts write programs to analyze clickstream data to decide how to improve sales and marketing strategies.
- **Finance**: Algorithmic traders write programs to prototype and simulate experimental trading strategies on financial data.
- **Public policy**: Analysts write programs to mine U.S. Census and labor statistics data to predict the merits of proposed government policies.
- **Data-driven journalism**: Journalists write programs to analyze economic data and make information visualizations to publish alongside their news stories.

Research programming is ubiquitous: By some estimates, the number of people who perform research programming exceeds the number of professional software engineers by over a factor of three [142], and this disparity is likely to grow as more science, engineering, and business-related fields rely on computational techniques. Specifically, an analysis of U.S. Bureau of Labor Statistics data estimates that by the year 2012, 13 million Americans will do programming in their jobs, but only 3 million will be professional software engineers [142]. Thus, the other 10 million are likely engaging in research programming, according to my definition. In addition, even professional software engineers engage in research programming when they are prototyping new features, running performance experiments, or tuning algorithms.

Research programming is crucial for making advances in modern academic research: Surveys at universities indicate that people across dozens of departments ranging from the natural sciences (e.g., astrophysics, geosciences, molecular biology), engineering (e.g., aerospace and mechanical engineering, operations research), and social sciences (e.g., economics, political science, sociology) engage in research programming as part of their daily work [83, 133].

Finally, despite what its name might imply, research programming has broad implications beyond academic research: Professionals in fields ranging from science, engineering, business, finance, public policy, and journalism, as well as students and computer hobbyists, all perform research programming to obtain insights from data.

My thesis is that by understanding the unique challenges faced during research programming, it becomes possible to apply techniques from dynamic program analysis, mixed-initiative recommendation systems, and OS-level tracing to make research programmers more productive.

This dissertation characterizes the research programming process, describes typical challenges faced by research programmers, and presents five software tools that I have developed to address some key challenges. These tools allow research programmers to iterate and potentially discover insights faster by offloading the burdens of data management and provenance to the computer.

The rest of this dissertation is structured as follows:

- Chapter 2 introduces a typical research programming workflow and the challenges that people face while performing research programming.

- Chapter 3 discusses how software tools can potentially address these research programming challenges, surveys the landscape of related work on existing tools, and provides motivation for the five new tools that I built for this dissertation.

- Chapter 4 presents an interactive graphical tool called Proactive Wrangler [76], which provides semi-automated suggestions to assist research programmers in reformatting and cleaning data prior to analysis.

- Chapter 5 presents a Python interpreter called IncPy [75], which speeds up the data analysis scripting cycle and helps researchers manage code and data dependencies. To the best of my knowledge, IncPy is the first attempt to integrate automatic memoization and persistent dependency management into a general-purpose programming language.

- Chapter 6 presents a Python interpreter called SlopPy [71], which automatically makes existing scripts error-tolerant, thereby speeding up the data analysis scripting cycle. SlopPy supports fail-soft semantics, tracking provenance of code and data errors, and incremental re-processing of error-inducing records.

- Chapter 7 presents a Linux-based activity monitoring and in-context notetaking system called Burrito [77], which helps research programmers organize, annotate, and recall past insights about their experiments.

- Chapter 8 presents an automatic software packaging tool called CDE [70, 74], which helps researchers easily deploy, archive, and share their experiments. CDE eliminates the problems of "dependency hell" for a large class of Linux-based software that research programmers are likely to create and use.

- Chapter 9 shows how my five tools can be integrated together to improve the overall research programming experience, discusses what challenges still remain open, and introduces new directions for future research.

Chapter 2

Research Programming

This chapter introduces a typical research programming workflow and the challenges that people encounter throughout this process. I obtained these insights from a variety of sources:

- My own research programming experiences when working on empirical software engineering papers earlier in graduate school (2007–2010) [72, 78, 79, 156].
- An observational study of software prototyping that I helped to perform [39].
- My interviews and shadowing of computational scientists in fields such as computational biology, genetics, neuroscience, physics, and Computer Science.
- MacLean's shadowing of computational scientists at Harvard [115].
- Interviews of computational scientists at Los Alamos National Laboratory [148].
- Interviews of over 100 computational researchers across 20 academic disciplines at Princeton University [133].
- Case studies of programming habits in five U.S. government research projects [42].
- An online survey of computational scientists with 1,972 usable responses from 40 countries [83].

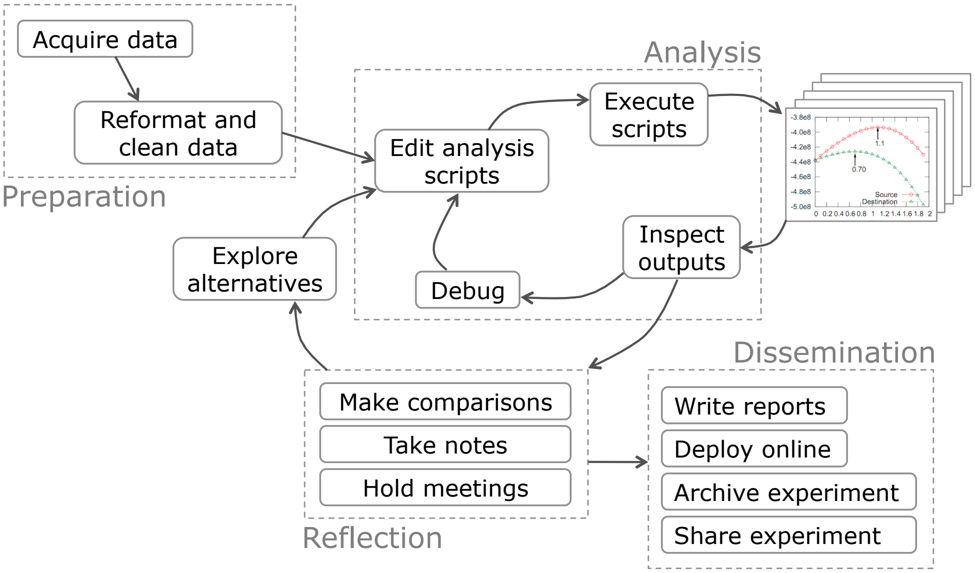


Figure 2.1: Overview of a typical research programming workflow.

Figure 2.1 shows the steps involved in a typical research programming workflow. There are four main phases, shown in the dotted-line boxes: *preparation* of the data, alternating between running the *analysis* and *reflection* to interpret the outputs, and finally *dissemination* of results in the form of written reports and/or executable code. I now describe the characteristics and challenges of each phase in detail.

2.1 Preparation Phase

Before any analysis can be done, the programmer must first acquire the data and then reformat it into a form that is amenable to computation.

Acquire data: The obvious first step in any research programming workflow is to acquire the data to analyze. Data can be acquired from a variety of sources. e.g.,:

- Data files can be downloaded from online repositories such as public websites (e.g., U.S. Census data sets).

- Data can be streamed on-demand from online sources via an API (e.g., the Bloomberg financial data stream).
- Data can be automatically generated by physical apparatus, such as scientific lab equipment attached to computers.
- Data can be generated by computer software, such as logs from a webserver or classifications produced by a machine learning algorithm.
- Data can be manually entered into a spreadsheet or text file by a human.

The main problem that programmers face in data acquisition is keeping track of *provenance*, i.e., where each piece of data comes from and whether it is still up-to-date [119]. It is important to accurately track provenance, since data often needs to be re-acquired in the future to run updated experiments. Re-acquisition can occur either when the original data sources get updated or when researchers want to test alternate hypotheses. Also, provenance can enable downstream analysis errors to be traced back to the original data sources.

Data management is a related problem: Programmers must assign names to data files that they create or download and then organize those files into directories. When they create or download new versions of those files, they must make sure to assign proper filenames to all versions and keep track of their differences. For instance, scientific lab equipment can generate hundreds or thousands of data files that scientists must name and organize before running computational analyses on them [124].

A secondary problem in data acquisition is storage: Sometimes there is so much data that it cannot fit on a single hard drive, so it must be stored on remote servers. However, anecdotes and empirical studies [42, 133] indicate that a significant amount of research programming is still done on desktop machines with data sets that fit on modern hard drives (i.e., less than a terabyte).

Reformat and clean data: Raw data is probably not in a convenient format for a programmer to run a particular analysis, often due to the simple reason that it was formatted by somebody else without that programmer's analysis in mind. A related problem is that raw data often contains semantic errors, missing entries, or inconsistent formatting, so it needs to be “cleaned” prior to analysis.

Programmers reformat and clean data either by writing scripts or by manually editing data in, say, a spreadsheet. Many of my interview subjects complained that these tasks are the most tedious and time-consuming parts of their workflow, since they are unavoidable chores that yield no new insights. A tangentially-related quantitative estimate is that data reformatting and cleaning accounts for up to 80% of the development time and cost in data warehousing projects [50].

However, the chore of data reformatting and cleaning can lend insights into what assumptions are safe to make about the data, what idiosyncrasies exist in the collection process, and what models and analyses are appropriate to apply [97].

Data integration is a related challenge in this phase. For example, Christian Bird, an empirical software engineering researcher that I interviewed at Microsoft Research, obtains raw data from a variety of `.csv` and XML files, queries to software version control systems and bug databases, and features parsed from an email corpus. He integrates all of these data sources together into a central MySQL relational database, which serves as the master data source for his analyses.

In closing, the following excerpt from the introduction of the book *Python Scripting for Computational Science* [107] summarizes the extent of data preparation chores:

> *Scientific Computing Is More Than Number Crunching*: Many computational scientists work with their own numerical software development and realize that much of the work is not only writing computationally intensive number-crunching loops. Very often programming is about shuffling data in and out of different tools, converting one data format to another, extracting numerical data from a text, and administering numerical experiments involving a large number of data files and directories. Such tasks are much faster to accomplish in a language like Python than in Fortran, C, C++, C#, or Java. [107]

In sum, data munging and organization are human productivity bottlenecks that must be overcome before actual substantive analysis can be done.

2.2 Analysis Phase

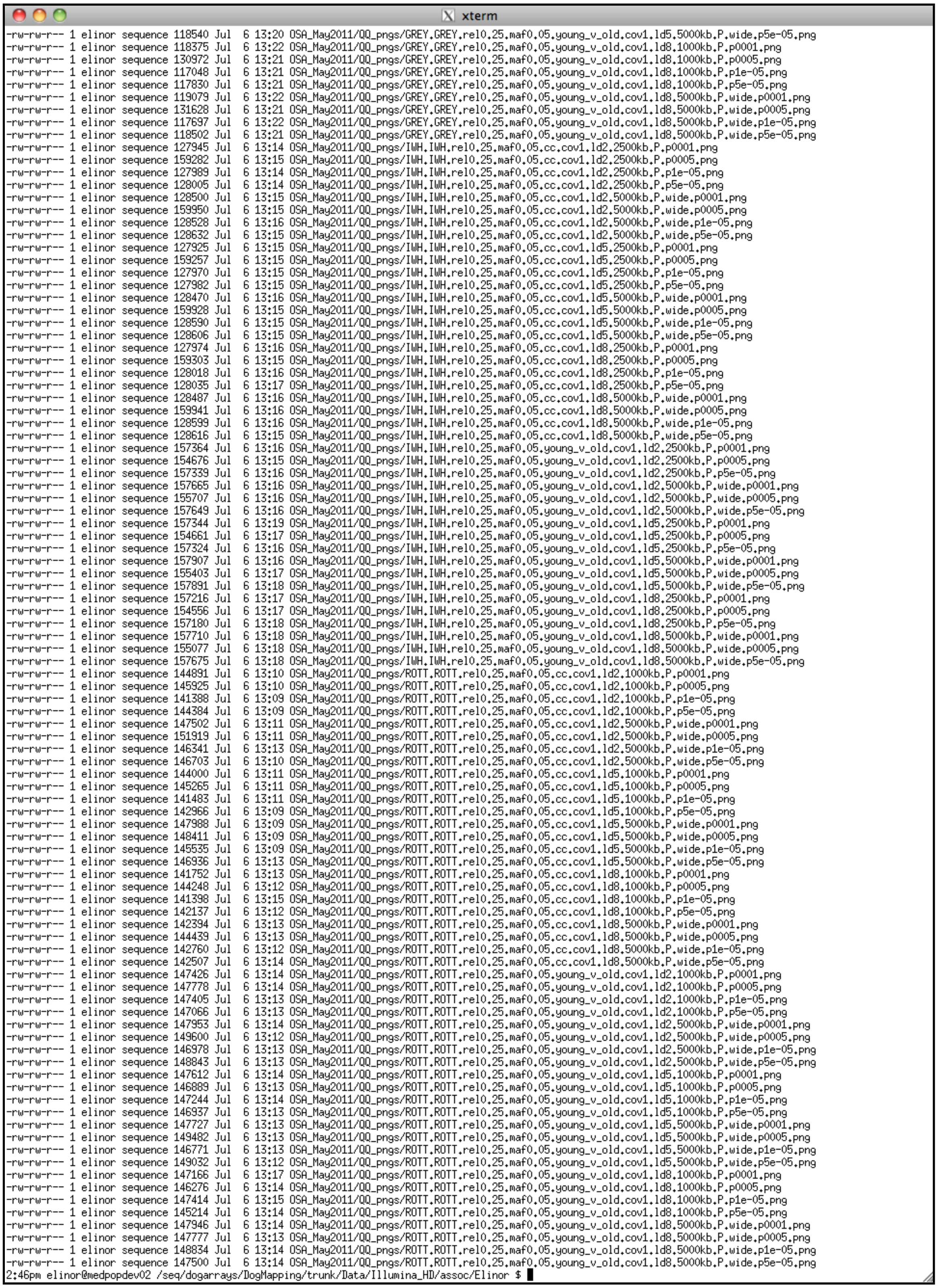
The core of research programming is the analysis phase: writing, executing, and refining computer programs to analyze and obtain insights from data. In this dissertation, I will refer to these kinds of programs as *data analysis scripts*, since research programmers often prefer to use interpreted "scripting" languages such as Python, Perl, R, and MATLAB [107, 133]. However, they also use compiled languages such as C, C++, and Fortran when appropriate.

Figure 2.1 shows that in the analysis phase, the programmer engages in a repeated *iteration cycle* of editing scripts, executing to produce output files, inspecting the output files to gain insights and discover mistakes, debugging, and re-editing. The faster the programmer can make it through each iteration, the more insights can potentially be obtained per unit time. There are three main sources of slowdowns:

- **Absolute running times**: Scripts might take a long time to terminate, either due to large amounts of data being processed or the algorithms being slow[1].
- **Incremental running times**: Scripts might take a long time to terminate after minor incremental code edits done while iterating on analyses, which wastes time re-computing almost the same results as previous runs.
- **Crashes from errors**: Scripts might crash prematurely due to errors in either the code or inconsistencies in data sets. Programmers often need to endure several rounds of debugging and fixing banal bugs such as data parsing errors before their scripts can terminate with useful results.

File and metadata management is another challenge in the analysis phase. Repeatedly editing and executing scripts while iterating on experiments causes the production of numerous output files, such as intermediate data, textual reports, tables, and graphical visualizations. For example, Figure 2.2 shows a directory listing from a computational biologist's machine that contains hundreds of PNG output image files, each with a long and cryptic filename. To track provenance, research programmers

[1] which could be due to asymptotic "Big-O" slowness and/or the implementations being slow



```
-rw-rw-r-- 1 elinor sequence 118540 Jul  6 13:20 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 118375 Jul  6 13:22 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 130972 Jul  6 13:21 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 117048 Jul  6 13:21 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 117830 Jul  6 13:21 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 119079 Jul  6 13:22 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 131628 Jul  6 13:21 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 117697 Jul  6 13:22 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 118502 Jul  6 13:21 OSA_May2011/QQ_pngs/GREY.GREY.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 127945 Jul  6 13:14 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 159282 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 127989 Jul  6 13:14 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.2500kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 128005 Jul  6 13:14 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 128500 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 159950 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 128528 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 128632 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 127925 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 159257 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 127970 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.2500kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 127982 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 128470 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 159928 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 128590 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 128606 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 127974 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 159303 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 128018 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.2500kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 128035 Jul  6 13:17 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 128487 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 159941 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 128599 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 128616 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157364 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 154676 Jul  6 13:15 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 157339 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157665 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 155707 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 157649 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157344 Jul  6 13:19 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 154661 Jul  6 13:17 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 157324 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157907 Jul  6 13:16 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 155403 Jul  6 13:17 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 157891 Jul  6 13:18 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157216 Jul  6 13:17 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.2500kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 154556 Jul  6 13:17 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.2500kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 157180 Jul  6 13:18 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.2500kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 157710 Jul  6 13:18 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 155077 Jul  6 13:18 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 157675 Jul  6 13:18 OSA_May2011/QQ_pngs/IWH.IWH.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 144891 Jul  6 13:10 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 145925 Jul  6 13:10 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 141388 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 144384 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147502 Jul  6 13:11 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 151919 Jul  6 13:11 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 146341 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 146703 Jul  6 13:10 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld2.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 144000 Jul  6 13:11 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 145265 Jul  6 13:11 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 141483 Jul  6 13:11 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 142966 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147988 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 148411 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 145535 Jul  6 13:09 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 146936 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld5.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 141752 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 144248 Jul  6 13:12 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 141398 Jul  6 13:15 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 142137 Jul  6 13:12 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 142394 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 144439 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 142760 Jul  6 13:12 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 142507 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.cc.cov1.ld8.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147426 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 147778 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 147405 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 147066 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147953 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 149600 Jul  6 13:12 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 146978 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 148843 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld2.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147612 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 146889 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 147244 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 146937 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147727 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 149482 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 146771 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 149032 Jul  6 13:12 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld5.5000kb.P.wide.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147166 Jul  6 13:17 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p0001.png
-rw-rw-r-- 1 elinor sequence 146276 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p0005.png
-rw-rw-r-- 1 elinor sequence 147414 Jul  6 13:15 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p1e-05.png
-rw-rw-r-- 1 elinor sequence 145214 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.1000kb.P.p5e-05.png
-rw-rw-r-- 1 elinor sequence 147946 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0001.png
-rw-rw-r-- 1 elinor sequence 147777 Jul  6 13:13 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p0005.png
-rw-rw-r-- 1 elinor sequence 148834 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p1e-05.png
-rw-rw-r-- 1 elinor sequence 147500 Jul  6 13:14 OSA_May2011/QQ_pngs/ROTT.ROTT.rel0.25.maf0.05.young_v_old.cov1.ld8.5000kb.P.wide.p5e-05.png
2:46pm elinor@medpopdev02 /seq/dogarrays/DogMapping/trunk/Data/Illumina_HD/assoc/Elinor $ █
```

Figure 2.2: A file listing from a computational biologist's experiment directory.

often encode metadata such as version numbers, script parameter values, and even short notes into their output filenames. This habit is prevalent since it is the easiest way to ensure that metadata stays attached to the file and remains highly visible. However, doing so leads to data management problems due to the abundance of files and the fact that programmers often later forget their own ad-hoc naming conventions. The following email snippet from a Ph.D. student in bioinformatics summarizes these sorts of data management woes:

> Often, you really don't know what'll work, so you try a program with a combination of parameters, and a combination of input files. And so you end up with a massive proliferation of output files. You have to remember to name the files differently, or write out the parameters every time. Plus, you're constantly tweaking the program, so the older runs may not even record the parameters that you put in later for greater control. Going back to something I did just three months ago, I often find out I have absolutely no idea what the output files mean, and end up having to repeat it to figure it out. [155]

Lastly, research programmers do not write code in a vacuum: As they iterate on their scripts, they often consult resources such as documentation websites, API usage examples, sample code snippets from online forums, PDF documents of related research papers, and relevant code obtained from colleagues. The computational scientists that I shadowed all keep several web browser tabs open and constantly switch back-and-forth between reading documentation and editing code. In addition, Brandt et al. found that web search, information foraging, and copying-and-pasting code are prevalent in programming tasks that resemble research programming [39].

2.3 Reflection Phase

Research programmers alternate between the *analysis* and *reflection* phases while they work, as denoted by the arrows between the two respective phases in Figure 2.1.

Whereas the analysis phase involves programming, the reflection phase involves thinking and communicating about the outputs of analyses. After inspecting a set of output files, a researcher might perform the following types of reflection:

Take notes: People take notes throughout their experiments in both physical and digital formats. Physical notes are usually written in a lab notebook, on sheets of looseleaf paper, or on a whiteboard. Digital notes are usually written in plain text files, "sticky notes" desktop widgets, Microsoft PowerPoint documents for multimedia content, or specialized electronic notetaking applications such as Evernote or Microsoft OneNote. Each format has its advantages: It is often easier to draw freehand sketches and equations on paper, while it is easier to copy-and-paste programming commands and digital images into electronic notes. Since notes are a form of data, the usual data management problems arise in notetaking, most notably how to organize notes and link them with the context in which they were originally written.

Hold meetings: People meet with colleagues to discuss results and to plan next steps in their analyses. For example, a computational science Ph.D. student might meet with her research advisor every week to show the latest graphs generated by her analysis scripts.

The inputs to meetings include printouts of data visualizations and status reports, which form the basis for discussion. The outputs of meetings are new to-do list items for meeting attendees. For example, during a summer internship at Microsoft Research working on a data-driven study of what factors cause software bugs to be fixed [78], I had daily meetings with my supervisor. Upon inspecting the charts and tables that my analyses generated each day, he often asked me to adjust my scripts or to fork my analyses to explore multiple alternative hypotheses (e.g., *"Please explore the effects of employee location on bug fix rates by re-running your analysis separately for each country."*).

Make comparisons and explore alternatives: The reflection activities that tie most closely with the analysis phase are making comparisons between output variants and then exploring alternatives by adjusting script code and/or execution parameters.

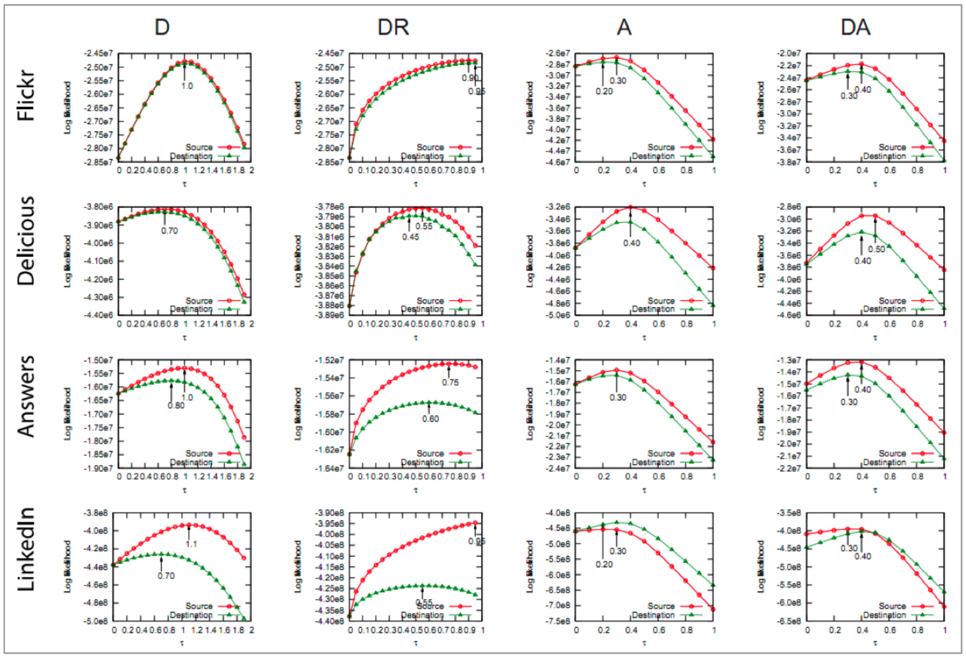


Figure 2.3: Image from Leskovec's Ph.D. dissertation [109] showing four variants of a social network model (one in each column) tested on four data sets (one in each row).

Researchers often open several output graph files (e.g., those in Figure 2.2) side-by-side on their monitors to visually compare and contrast their characteristics. MacLean observed the following behavior in her shadowing of scientists at Harvard, which is similar to my own interview and shadowing findings:

> Much of the analysis process is trial-and-error: a scientist will run tests, graph the output, rerun them, graph the output, etc. The scientists rely heavily on graphs — they graph the output and distributions of their tests, they graph the sequenced genomes next to other, existing sequences. [115]

Figure 2.3 shows an example set of graphs from social network analysis research, where four variants of a model algorithm are tested on four different input data sets. This example is the final result from a published paper and Ph.D. dissertation [109]; during the course of running analyses, many more of these types of graphs are produced by analysis scripts. Researchers must organize, manage, and compare these graphs to gain insights and ideas for what alternative hypotheses to explore.

2.4 Dissemination Phase

The final phase of research programming is disseminating results, most commonly in the form of written reports such as internal memos, slideshow presentations, business/policy white papers, or academic research publications. The main challenge here is how to consolidate all of the various notes, freehand sketches, emails, scripts, and output data files created throughout an experiment to aid in writing.

Beyond presenting results in written form, some researchers also want to distribute their software so that colleagues can reproduce their experiments or play with their prototype systems. For example, computer graphics and user interface researchers currently submit a video screencast demo of their prototype systems along with each paper submission, but it would be ideal if paper reviewers could actually execute their software to get a "feel" for the techniques being presented in each paper. In reality, it is difficult to distribute research code in a form that other people can easily execute on their own computers. Before colleagues can execute one's code (even on the same operating system), they must first obtain, install, and configure compatible versions of the appropriate software and their myriad of dependent libraries, which is often a frustrating and error-prone process. If even one portion of one dependency cannot be fulfilled, then the original code will not be re-executable.

Similarly, it is even difficult to reproduce the results of *one's own experiments* a few months or years in the future, since one's own operating system and software inevitably get upgraded in some incompatible manner such that the original code no longer runs. For instance, academic researchers need to be able to reproduce their own results in the future after submitting a paper for review, since reviewers inevitably suggest revisions that require experiments to be re-run. As an extreme example, my former officemate Cristian used to archive his experiments by removing the hard drive from his computer after submitting an important paper to ensure that he can re-insert the hard drive months later and reproduce his original results.

Lastly, researchers often collaborate with colleagues by sending them partial results to receive feedback and fresh ideas. A variety of logistical and communication challenges arise in research collaborations centered on code and data sharing [86].

In conclusion, Table 2.1 summarizes the major research programming challenges introduced throughout this chapter. The next chapter discusses how software tools can help address these challenges.

<table>
<tr><th></th><th>Preparation</th><th>Analysis</th><th>Reflection</th><th>Dissemination</th></tr>
<tr><td rowspan="2">Data Management</td><td colspan="2">Naming files and directories
File versioning
Storage of large data sets</td><td rowspan="2">Organize notes</td><td rowspan="2">Consolidation</td></tr>
<tr><td>Cleaning/reformatting
Data integration</td><td></td></tr>
<tr><td>Provenance</td><td>Track file origins
Raw/clean data relations</td><td>Track execution parameters
Track input/output relations</td><td>Link notes with context</td><td></td></tr>
<tr><td>Iteration Speed</td><td>Large data sets</td><td>Absolute running times
Incremental running times
Crashes from errors</td><td></td><td></td></tr>
<tr><td>Communication</td><td></td><td>Read documentation
Copy-and-paste code</td><td>Make comparisons
Hold meetings</td><td>Packaging
Reproducibility
Archiving
Collaboration</td></tr>
</table>

Table 2.1: Summary of research programming challenges introduced in this chapter.

Chapter 3

Addressing Research Programming Challenges

This chapter discusses how software tools can address the challenges of research programming identified in Chapter 2 (summarized in Table 2.1). I will first survey the state-of-the-art in both research prototypes and production-quality tools, discuss their shortcomings and areas for improvement, and conclude with motivations for the five tools that I built for this dissertation.

3.1 Software Engineering Tools

Decades of research and production of tools such as optimizing compilers, high-level programming languages, integrated development environments, debuggers, automated software bug finders, and testing frameworks have made the process of writing code far more pleasant and productive than it was in the early days of computing.

Since research programming is a form of programming, modern-day tools that help all programmers also help research programmers. However, research programmers often do not take full advantage of these tools since the target audience of these tools is people engaging in *software engineering*, an activity with vastly different characteristics than research programming. Here are the most salient differences:

- **Purpose**: The main goal of software engineering is to create robust, well-tested, maintainable pieces of production-quality software. In contrast, the main goal of research programming is to obtain insights about a topic; code is simply a means to this end [133].

- **Environment**: Software engineers usually work in a single unified software development environment. For example, iPhone developers work within the Apple Xcode IDE, and some enterprise software developers build .NET applications in Microsoft Visual Studio. In contrast, research programmers often work in a heterogeneous environment where they cobble together a patchwork of improvised ad-hoc scripts and "sloppy" prototype code written in multiple languages, interfacing with a mix of 3rd-party libraries and executables from disparate sources within a UNIX-like command-line environment [115, 133]. Heterogeneity is a fundamental property of research programming, since there is usually no "one-size-fits-all" solution for cutting-edge discovery-oriented programming tasks.

- **Specifications**: Software engineering involves a formal process of writing, revising, and maintaining specifications in addition to code. In contrast, the process of writing research code is exploratory and highly iterative, where specifications are ill-defined and constantly-changing in response to new findings.

- **Priorities**: The software engineering process prioritizes writing fast-running, robust, and maintainable code, whereas the research programming process prioritizes fast iteration to maximize the rate of discovery [39]. Thus, tools to facilitate research programming must be "lightweight" enough to be useful without causing delays in the programmer's workflow. Several of my interview subjects remarked that any time spent learning and dealing with tools is time not spent making discoveries.

- **Expertise**: Software engineering is mostly done by professional programmers. In contrast, research code is written by people of all levels of programming expertise, ranging from Computer Science veterans to scientists who learn barely enough about programming to be able to write basic scripts.

In summarizing their comprehensive interviews of over 100 research programmers at Princeton University, Prabhu et al. conclude: “Presently, programming tools are designed for professional programmers, and need to be carefully tailored to meet the needs of scientists” [133]. The contributions of this dissertation are identifying those needs (Chapter 2), showing how current tools address some of them (this chapter), and introducing five new tools that fulfill needs that are not sufficiently met by existing tools (Chapters 4–8).

3.2 Data Cleaning, Reformatting, and Integration

Research programmers typically perform data cleaning, reformatting, and integration by writing scripts or by manually editing the data within spreadsheets [76]. Computer Science researchers have proposed numerous strategies for simplifying these sorts of programming tasks, including domain-specific languages [116], visual programming [40], keyword programming [111], and programming-by-demonstration [49]. Myers et al. [122] provide a survey of end-user programming; Halevy, Rajaraman, and Ordille provide a survey of data integration techniques [81]; Kandel et al. provide a survey of issues related to data quality and uncertainty, and demonstrate how interactive visualizations can aid in the data cleaning process [95]. I will now review prior work in end-user programming tools for data transformation in more detail:

3.2.1 Menu-Driven Data Transformation

Both database and HCI researchers have devised graphical interfaces [65, 91, 134] for cleaning and reformatting data. Each of these tools share a similar design: a data table view and a set of menu-accessible transformation operators. Google Refine [91] combines menu-driven commands with faceted histograms for data profiling and filtering. Ajax [65] similarly provides an interface for transform specification with advanced facilities for entity resolution. Potter’s Wheel [134] is a graphical interface for authoring statements in a declarative transformation language and offers extensible facilities for data type inference. Each of these tools enable skilled practitioners to transform data more rapidly. However, each imposes a learning curve: Users must learn both the available operations and how to combine them to achieve common data transformation needs. Lowering these sorts of overheads could help such tools gain wider adoption amongst research programmers.

3.2.2 Programming-by-Demonstration Systems

Another class of targeted data manipulation tools rely on programming by demonstration (PBD) techniques [49]. Potluck [92] uses interactive simultaneous text editing features [118] to help users perform data integration. Karma [153] infers text extractors and transformations for web data from examples entered in a table. Vegemite [110] extends CoScripter [108] to integrate web data, automate the use of web services, and generate shareable scripts. Dontcheva et al. [55] use PBD techniques to enable extraction, integration, and templated search of web data. Gulwani [69] introduces an algorithm for learning expressive string processing programs from input-output examples. PADS [60] infers nuanced data parsers from a set of positive examples. Though well-suited for their intended tasks, these systems are insufficient for the more general demands of the kinds of data reformatting that research programmers often need to do. For example, while text extraction and mass editing are valuable for data transformation, the above tools lack other needed operations such as table reshaping, aggregation, and missing value imputation.

Harris and Gulwani created a system for learning table transformations from an example input-output pair [84]. Given example input and output tables, the system can infer a program that filters, duplicates, and/or reorganizes table cells to generate the output table. These resulting programs can express a variety of table reshaping operations. However, their approach treats table cells as atomic units, and thus has limited expressiveness. For example, it does not support transformations involving text editing or extraction. Second, their approach requires that a user specify both input and output examples, and thus know the details of the desired output in advance. In the case of large tables, creating the output example may be tedious. On the other hand, providing a small example from which the proper transformation can be inferred may require significant insight. Third, the actual mechanics of the resulting transformation are opaque, potentially limiting user skill acquisition.

3.3 File Naming and Versioning

Version control systems (a.k.a. revision control systems) enable different versions of a file to be stored under one visible filename. The modern line of version control systems started with RCS [152] in the 1980's, followed by CVS [56] and Subversion (SVN) [46]. In the past decade, decentralized version control systems such as Git [112] and Mercurial [129] have become more popular, especially in open-source software projects where developers work in multiple geographic locations. For example, Git is used to manage the Linux kernel codebase. While version control is widely used by professional software engineers, adoption amongst research programmers has been limited by a variety of reasons [83]: First, many research programmers are unaware of version control systems and lack the knowledge to install and configure them. Second, even those who are aware of these systems are reluctant to use them due to the overhead of deciding when to add and commit new versions of files, which is perceived as a distraction from their main goal of making discoveries. Instead, research programmers manually perform "versioning" by making multiple copies of their files and appending metadata to their filenames, as shown in Figure 2.2. Third, even researchers who are disciplined about using version control systems will need to materialize one or more old versions of selected files in order to do repeated analyses or to make comparisons; materializing multiple old versions can still lead to file naming and organization problems.

Versioning filesystems eliminate the aforementioned user overhead by automatically keeping track of all file versions. Research systems such as Elephant [140] and production systems such as NILFS [101] (part of Linux since June 2009) and Dropbox [6] track changes to all files and allow users to access old versions via read-only snapshots. Techniques such as causality-based versioning [120] can reduce storage and retrieval overheads by keeping only semantically-meaningful versions.

The main limitation of both version control systems and versioning filesystems is their lack of transparency. It is cumbersome to issue commands to check out (retrieve) multiple old versions of files to make comparisons, which is a key research programming activity (see Section 2.3). The ubiquitous "low-tech" habit of making multiple weirdly-named copies of files actually provides greater transparency, since it is easy to open all of those files side-by-side. Another limitation is lack of context. How does the user know which old versions are meaningful to retrieve? Research in context-aware personal file search [145] can help address this issue.

A more radical approach is to get rid of traditional filenames and directory hierarchies altogether. Research prototypes such as Lifestreams [58] and Time-Machine Computing [135] are GUI desktop environments where the user's files are organized in a chronological versioned stream rather than in directories, thus eliminating the need to provide filenames and directory names. However, these systems cannot interoperate with existing research programming tools that rely on a traditional filesystem.

3.4 Managing Large Data Sets

A large amount of research programming is being done on data sets less than a terabyte in size, which fit on one single modern desktop machine [133]. Raw data sets usually come in the form of flat files in either textual or binary formats. After data cleaning, reformatting, and integration, programmers might opt to store post-processed data in databases of either the relational (e.g., SQLite, MySQL, PostgreSQL) or NoSQL (e.g., Berkeley DB [127], MongoDB [15]) flavors. Most databases offer indexing and materialized view features that can speed up computations. NoSQL databases offer greater schema flexibility but sometimes at the expense of a lack of full ACID (atomicity, consistency, isolation, durability) guarantees.

In contrast, modern companies usually process petabytes or more of data as part of their technology or business needs. Analysts at those companies do research programming to gain insights from such data, but they need to use a different set of tools to manage data at those scales. In the past decade, technology companies have developed high-performance fault-tolerant key-value stores such as Bigtable [44] at Google, Dynamo [53] at Amazon, and Cassandra [105] at Facebook. These large-scale data management solutions are meant to be deployed across thousands of machines within a data center and are thus not applicable to desktop-scale research programming, which requires low latency and nearly-instant responsiveness. However, as more research programming is performed on larger-scale data in the future, new tools will need to be developed to facilitate both fast iteration times and extreme scalability.

3.5 Speeding Up Iteration Cycle

A variety of programming languages and run-time enhancements aim to speed up the iteration cycle during the analysis phase of research programming (see Section 2.2).

3.5.1 Speeding Up Absolute Running Times

Just-in-time compilers for dynamic languages (e.g., PyPy [38] for Python, TraceMonkey [64] for JavaScript) can speed up script execution times without requiring programmers to make any annotations (or to switch to a compiled language). However, JIT compilers only focus on micro-optimizations of CPU-bound code such as hot inner loops. No JIT compiler optimizations could speed up I/O or network-bound operations, which is what often consumes time in data analysis scripts [107].

Parallel execution of code can vastly speed up scientific data processing scripts, at the cost of increased difficulty in programming and debugging such scripts. In recent years, libraries such as Parallel Python [17], frameworks such as MapReduce [52] and Hadoop [1], and new high-level languages such as Pig [128] and Hive [151] have made it easier to write parallel code and deploy it to run on compute clusters. However, many researchers interviewed by me and others [133] felt that the learning curve for writing parallel code is still high, especially for scientists without much programming experience. It is much easier for people to think about algorithms sequentially, and even experts prefer to write single-threaded prototypes and then only parallelize later when necessary [144].

3.5.2 Speeding Up Incremental Running Times

Memoization is a classic optimization technique first introduced in a 1968 *Nature* paper [117]. It involves manually rewriting a function to save its inputs and outputs to a cache, so that subsequent calls with previously-seen inputs can be skipped, thus speeding up incremental running times. However, the burden is on the programmer to determine which functions are safe and worthwhile to memoize, write the memoization code, and invalidate cache entries at the appropriate times.

"make" is a ubiquitous UNIX tool that allows users to declaratively specify dependencies between commands and files, so that the minimum set of commands need to be re-run when dependent files are altered [57]. In the past few decades, `make` has spawned dozens of descendent tools that all operate on the same basic premise.

Vesta [88] is a software configuration management system that provides a pure functional domain-specific language for writing software build scripts. The Vesta interpreter performs automatic memoization and dependency tracking. However, since it is a domain-specific build scripting language, it has never been used for research programming tasks, to the best of my knowledge.

Self-adjusting computation [34, 82] is a technique that enables algorithms to run faster in response to small changes in input data, only re-computing outputs for portions of the input that have changed between runs. Self-adjusting computation tracks fine-grained dependencies between executed basic blocks and the objects that they mutate, which can provide large speed-ups but requires programmers to annotate exactly which objects to track. Its creator mentions, "Although self-adjusting computation can be applied without having to change existing code by tracking all data and all dependences between code and data, this is prohibitively expensive in practice" [34]. Even with annotations, there is at least a 500% slowdown on the initial (empty cache) run [82]. An ideal run-time incremental recomputation system should require no programmer intervention and have minimal slowdowns on initial runs.

3.5.3 Preventing Crashes From Errors

Silent errors in programming languages: Some programming languages, most notably Perl, are designed to silence errors as much as possible to avoid crashing scripts [33]. For example, Perl and PHP automatically convert between strings and integers rather than throwing a run-time type error, which seems convenient but can produce unexpected results that are difficult to debug. Failure-oblivious computing is a technique that silently hides memory access errors in C programs by ignoring out-of-bounds writes and returning fake (small integer) values for out-of-bounds reads [136]. Since failure-oblivious computing works on C code, it requires re-compiling the target program and incurs a slowdown due to memory bounds checking. An ideal run-time environment for data analysis should both prevent crashes and also flag errors to aid in debugging rather than silently hiding them.

Error tolerance in cluster data processing: Google's MapReduce [52] and the open-source Hadoop [1] frameworks both have a mode that skips over bad records (i.e., those that incite exceptions) when processing data on compute clusters. The Sawzall [132] domain-specific language (built on top of MapReduce) can skip over portions of bad records, but it lacks the flexibility of a general-purpose language.

3.6 Domain-Specific Programming Environments

Researchers have developed a variety of domain-specific environments catered towards specific types of research programming tasks.

Gestalt [130] is an integrated development environment for prototyping and evaluating machine learning algorithms. It allows programmers to visually compare the outputs of different algorithm variants and to quickly test alternatives.

d.note [85] is a visual programming tool where UI designers can tightly integrate annotations, automatic versioning, and visual comparison of alternatives when prototyping mobile device GUI applications.

Scientific workflow systems such as Kepler [114], Taverna [125], and VisTrails [143] are graphical development environments for designing and executing scientific computations. Scientists create workflows by using a GUI to visually connect together blocks of pre-made data processing functionality into a data-flow graph. In these domain-specific visual languages, each block is a pure function whose run-time parameters and other provenance are automatically recorded and whose return values are memoized. Due to their specialized nature, though, these systems are not nearly as popular for research programming as general-purpose languages such as Python or Perl. Even the creators of VisTrails admit, "While significant progress has been made in unifying computations under the workflow umbrella, workflow systems are notoriously hard to use. They require a steep learning curve: users need to learn programming languages, programming environments, specialized libraries, and best practices for constructing workflows" [143].

Although these tools are useful for specific tasks, they are not well-suited for the general needs of research programming, which usually occur in heterogeneous environments involving a mix of scripts, libraries, and 3rd-party tools written in different general-purpose languages. Moreover, they require user "buy-in" at the beginning of an experiment and cannot be easily retrofitted into an existing workflow.

3.7 Electronic Notetaking

For hundreds of years, the pace of scientific research was relatively slow, limited by the need to set up, run, and debug experiments on physical apparatus. Throughout their experiments, researchers took the time to write meticulous accounts of their hypotheses, observations, analyses, and reflections in handwritten *lab notebooks*. However, paper notebooks are not well-suited for the pace of modern computational research:

> Laboratory notebooks have been the traditional mechanism for maintaining such information, but because the volume of data manipulated in computational experiments has increased along with the complexity of analysis, manually capturing provenance and writing detailed notes is no longer an option — in fact, it can have serious limitations. [61]

Most research programmers I observed write notes on the computer in plain text files, “sticky notes” desktop widgets, or specialized notetaking applications. To keep a log of their experimental trials, they copy-and-paste the parameters and textual outputs of executed commands into text files and add notes to provide context. Some also copy-and-paste output images (e.g., graphs) into multimedia notetaking applications such as Microsoft PowerPoint, OneNote, or Evernote. Some scientific research organizations purchase or custom-build electronic lab notebook (ELN) software [99]. ELNs are notetaking applications enhanced with domain-specific templates to ease manual data entry for particular experimental protocols (e.g., bioinformatics, chemistry, genomics).

A shortcoming of all of the aforementioned general-purpose notetaking applications is that notes are not linked with the source code and data files to which they refer. Domain-specific environments can provide such links. For example, integrated development environments for languages such as Mathematica [14] and Python [131] offer a *notebook view* where executable code and output graphs are displayed alongside the programmer’s textual notes and always kept in sync. An ideal notetaking solution for research programming would combine the flexibility and application-independent nature of general-purpose notetaking software with the in-context linking provided by these domain-specific environments.

3.8 File-Based Provenance

Several systems enable research programmers to track forms of *provenance* including the origin of raw data files, the relationships between raw and cleaned data files, script execution parameters, and the relationships between the inputs and outputs of scripts. PASS [119, 121] collects such provenance by tracking Linux system calls within a modified kernel and filesystem, while ES3 [62] uses the Linux `ptrace` mechanism to accomplish a similar goal. Ideas from these tools could be combined with notetaking applications from Section 3.7 to link notes with code and data sets located throughout a user’s filesystem.

3.9 Distributing Research Software

Software companies devote considerable resources to creating and testing one-click installers for products such as Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers meticulously specify and update the proper dependencies in order to integrate their software into package management systems [11]. Despite these efforts, online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring software and dependencies.

In particular, research programmers are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their job is not to release production-quality software. Instead, they usually "release" their experimental software by uploading their source code and data files to a server and maybe writing up some informal installation instructions. There is a slim chance that their colleagues will be able to run their research code "out-of-the-box" without some technical support. Also, "releases" of research code often fail to run when the operating system gets upgraded: The team leader of the Saturn [35] static analysis project admitted in a public email, "As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release" [21].

I will now survey existing tools in these categories in more detail:

One-click installers: Existing tools for creating one-click installers and self-contained applications all require the user to manually specify dependencies at package creation time. For example, Mac OS X programmers can create application bundles using Apple's Developer Tools IDE [13]. Research prototypes such as PDS [36], which creates self-contained Windows applications, and The Collective [141], which aggregates a set of software into a portable *virtual appliance*, also require the user to manually specify dependencies. VMware ThinApp [32] is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software. It cannot be used to create packages from existing software already installed on a live machine, which is a common use case for research programmers who want to distribute their existing experimental code to colleagues.

Package management systems are often used to install open-source software and their dependencies. Generic package managers exist for all major operating systems (e.g., RPM for Linux, MacPorts for Mac OS X, Cygwin for Windows), and specialized package managers exist for ecosystems surrounding many programming languages (e.g., CPAN for Perl, RubyGems for Ruby) [11].

From the package creator's perspective, it takes time and expertise to manually bundle up one's software and list all dependencies so that it can be integrated into a specific package management system. A banal but tricky detail that package creators must worry about is adhering to platform-specific idioms for pathnames and avoiding hard-coding non-portable paths into their programs [147].

From the user's perspective, package managers work great as long as the *exact* desired versions of software exist within the system. However, version mismatches and conflicts are common frustrations, and installing new software can lead to a library upgrade that breaks existing software [54]. The Nix package manager is a research project that tries to eliminate dependency conflicts via stricter versioning, but it still requires package creators to manually specify dependencies at creation time [54]. Also, users need to have root permissions to install software using these solutions; potential users might not have root permissions on centrally-managed corporate machines or scientific compute clusters. A more convenient packaging solution should allow programs to be run without any root permissions, installation, configuration, or risk of breaking existing software.

Virtual machine snapshots achieve the goal of capturing all dependencies required to execute a set of programs on another machine, regardless of what operating system is installed on there. However, this solution requires the user to always be working within a virtual machine (VM) from the start of a project, or else re-install all of their software within a new VM. Also, VM snapshot disk images are usually tens of gigabytes in size since they contain the OS kernel and *all* applications, not just a single application that a research programmer wants to distribute. Thus, there is a need for a more lightweight solution that enables users to create and run packages natively on their own machines rather than through a VM layer.

3.10 Challenges Addressed By This Dissertation

This dissertation presents five new software tools that I have built to address research programming challenges that have not yet been solved by prior work. I now summarize how my tools improve upon the state-of-the-art presented throughout this chapter:

- Proactive Wrangler (Chapter 4) helps research programmers reformat and clean data. Unlike prior data cleaning and reformatting systems (see Section 3.2), Proactive Wrangler has a larger vocabulary of table-level transformations (e.g., reshaping), offers proactive suggestions to help guide users toward relevant transformations, does not require users to provide sample outputs, and can export transformations as Python scripts to facilitate code re-use and downstream integration.

- IncPy (Chapter 5) speeds up the data analysis scripting cycle and helps research programmers manage code and data dependencies. To the best of my knowledge, this is the first attempt to integrate automatic memoization and persistent dependency management into a general-purpose programming language. IncPy operates with low run-time overheads and also eliminates the user overhead of creating, naming, and managing intermediate data files. The speed-ups that IncPy provides are complementary to those provided by JIT compilers and parallel code execution, so these techniques can all be combined.

- SlopPy (Chapter 6) automatically prevents scripts from crashing due to errors, thereby speeding up the data analysis scripting cycle. Unlike prior work, SlopPy does not silently hide errors. Instead, SlopPy taints erroneous values as a special NA type, logs warning messages, and propagates NA values so that the programmer knows which portions of results are derived from error-inducing input records. It also stores exception contexts (i.e., values of local variables), which aid in debugging and allow the programmer to incrementally re-process only the error-inducing records and then merge them with the original results.

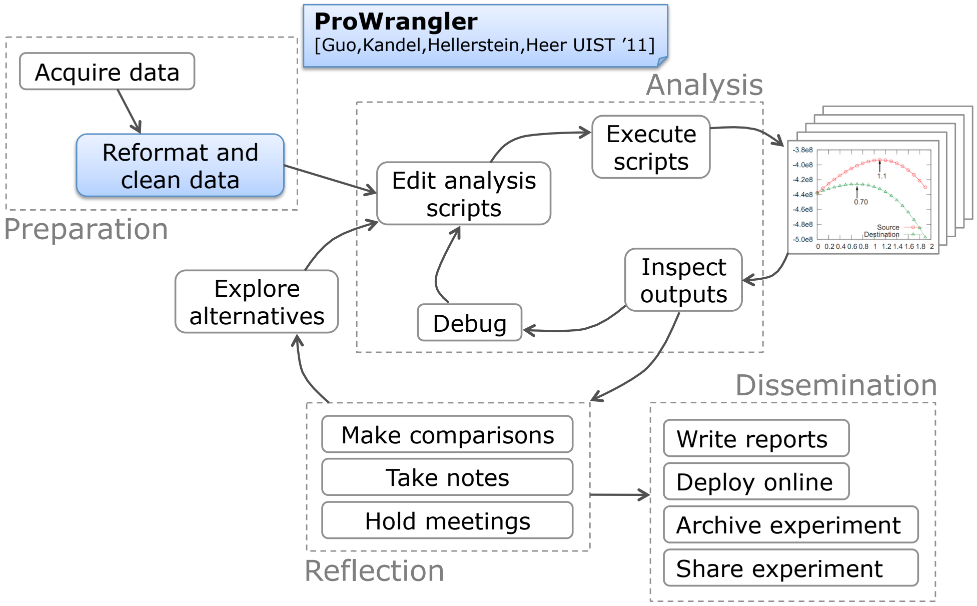
- Burrito (Chapter 7) helps research programmers organize, annotate, and recall past insights about their experiments. It combines the generality of file-based provenance systems (see Section 3.8) with the notetaking and context-linking features of certain domain-specific research programming environments (see Section 3.6). Burrito also enables users to track activities such as reading documentation, copying-and-pasting code, making result comparisons, and managing file versions (see Section 3.3) while working in a heterogeneous UNIX-based environment of their liking.

- CDE (Chapter 8) helps research programmers easily deploy, archive, and share their prototype software. Unlike prior work on software distribution systems (see Section 3.9), CDE allows users to create portable self-contained packages *in situ* from a live running Linux machine. CDE packages are much smaller in size than VM images and can be executed on a target Linux machine with no required installation, configuration, or root permissions.

Finally, Chapter 9 discusses some remaining challenges that this dissertation does not address and concludes with proposals for future research directions.

Chapter 4

Proactive Wrangler: Simplify Data Cleaning



This chapter presents a web-based interactive graphical tool called Proactive Wrangler, which helps research programmers reformat and clean data prior to analysis. Its contents were adapted from a 2011 conference paper that I co-authored with Sean Kandel, Joseph Hellerstein, and Jeffrey Heer [76]. All uses of "we", "our", and "us" in this chapter refer to all four authors.

4.1 Motivation

The increasing scale and accessibility of data — including government records, web data, and system logs — provides an under-exploited resource for improving governance, public policy, organizational strategy, and even our personal lives [154]. However, much of this data is not *suitable* for use by analysis tools. Data is often stored in idiosyncratic formats or designed for human viewing rather than computational processing (e.g., cross-tabulations within spreadsheets).

These hurdles require that analysts engage in a tedious process of data transformation (or *data wrangling*) to map input data to a form consumable by downstream tools. Both prior work [51] and our own conversations with analysts indicate that wrangling is one of the most time-consuming aspects of analysis. As a result, domain experts regularly spend more time manipulating data than they do exercising their specialty, while less technical audiences may be excluded.

Analysts typically wrangle their data by writing scripts in programming languages such as Python or Perl or by manually editing the data within spreadsheets. To assist this process, researchers have developed novel interactive tools. Potter's Wheel [134] and Google Refine [91] are menu-driven interfaces that provide access to common data transforms. Wrangler [96] extends this approach by (a) automatically suggesting applicable transforms in response to direct manipulation of a data table and (b) providing visual previews to aid transform assessment. Each of these tools enable skilled practitioners to rapidly specify transformation workflows; however, they fall short in helping users formulate data transformation strategies. Given an input data set, what is the desired output state? What operations are possible and which sequence of operations will result in suitable data?

One complication is that there is no single "correct" output format. A data layout amenable to plotting in Excel is often different from the format expected by visualization tools such as Tableau. That said, one format is required by many database, statistics and visualization tools: a *relational data table*. In this format, each row contains a single data record and each column represents a variable of a given data type. A relational format contrasts with matrix or cross-tabulation layouts, where both rows and columns represent different variables. Cross-tabs are more convenient for human viewing and thus regularly used for reporting. Consequently, a large number of data sets are stored in this format and analysts may find it difficult to map between formats.

4.2 Contributions

We augmented the Wrangler transformation tool [96] to aid transform discovery and strategy formation, thereby creating a tool called Proactive Wrangler. We take a mixed-initiative [90] approach to addressing this aspect of research programming by generating *proactive suggestions* to improve the suitability of a data set for downstream tools. In other words, we want the computer to help analysts get data into the format that computers expect. This work makes three contributions:

- **A model of data table *"suitability"*** that enables generation of proactive transform suggestions (Section 4.4). We introduce a metric that indicates the degree to which a table adheres to a relational format usable by downstream tools. We use this metric to guide automated search through the space of possible operations. As the search space is too large to evaluate at interactive rates, we couple our metric with empirically-derived heuristics for pruning the space of candidate transforms.

- **An analysis of algorithm behavior** assessing the strengths and limitations of our suggestions (Section 4.5). Across a set of realistic transformation tasks, we find that our method suggests over half (53%) of the steps used in "ideal" transformations. In these cases, our suggestions exhibit high precision: the top-ranked suggestion is the preferred option 77% of the time.

- **A user study** examining how end users apply proactive suggestions (Section 4.6). We find that our method can help users complete transformation tasks, but not always in the manner that we expected. Many users express a desire to maintain control and roundly ignore suggestions presented prior to interaction with the data table. However, users do not express similar concerns when suggestions generated by our proactive model are inserted among "reactive" suggestions specific to an initiating interaction. User assessment of suggestions appears to improve when users see themselves as the initiators.

4.3 Background: The Base Wrangler System

In this work, we extend an existing research prototype called Wrangler [96], which combines multiple end-user programming strategies to facilitate specification of data transformation scripts. Underlying Wrangler is a *declarative domain-specific language* that includes a comprehensive set of transformation operators and enables code-generation of reusable programs (e.g., Python and JavaScript code). Wrangler provides natural language descriptions and *visual previews* with which users can assess and verify transform behaviors. Wrangler uses *programming-by-demonstration methods* (c.f., [69, 118]) to specify regular expression patterns and row predicates. In addition, Wrangler generates *automatic transform suggestions* in response to user interactions with a data table.

4.3.1 The Wrangler Transformation Language

The transformation scripts produced by Wrangler consist of a linear sequence of individual transforms, expressed in an underlying declarative language. The transforms support common data cleaning and reformatting tasks such as splitting or extracting text values, deleting or merging columns, batch text editing, interpolating values, and reorganizing cell layout (e.g., table *reshaping*, as shown in Figure 4.1). Table 4.1 summarizes the most common transforms surfaced in the user interface. The language is the direct descendant of the Potter's Wheel language [134] and draws on concepts introduced in SchemaSQL [106]. This prior work [106, 134] also provides formal proofs of the language's expressiveness, showing it is capable of expressing any one-to-one or one-to-many transform of cell values.

In addition, the declarative nature of the language facilitates implementation

Transform	**Description**
Cut	Remove selected text from cells in specified columns.
Delete	Remove rows that match given indices or predicates.
Drop	Remove specified columns from the table.
Edit	Edit the text in each cell of the specified columns.
Extract	Copy text from cells in a column into a new column.
Fill	Fill empty cells using values from adjacent cells.
Fold	Reshape a table into columns of key-value sets; selected rows map to keys, selected columns to values.
Merge	Concatenate multiple columns into a single column.
Promote	Promote row values to be the column names.
Split	Split a column into multiple columns by delimiters.
Translate	Shift the position of cell values by a given offset.
Transpose	Transpose the rows and columns of the table.
Unfold	Reshape a table by mapping key-value sets to a collection of new columns, one per unique key.

Table 4.1: The Wrangler Transformation Language. Each transform accepts as parameters some combination of enumerable values and text, row, or column selection criteria. For further discussion, see the Wrangler [96] and Potter's Wheel [134] papers.

across a variety of platforms. Thus, the Wrangler tool can generate executable code for multiple runtimes, including Python and JavaScript. For example, we often wrangle a data subset within the user interface and then export a resulting Python script to transform much larger databases (millions or more rows) on a server machine.

A critical feature of the language design is its compactness: with only a dozen operators, analysts can complete a wide variety of data cleaning tasks. Moreover, most operators accept identical parameter types, namely row, text and column selections. The compact language design is intended to facilitate exploration and learning by limiting the number of possible operations considered. An additional benefit is that this reduced set of formally-defined operators lends itself to computational search over operation sequences.

	Boys	Girls
Australia	1	2
Austria	3	4
Belgium	5	6
China	7	8

Australia	Boys	1
Australia	Girls	2
Austria	Boys	3
Austria	Girls	4
Belgium	Boys	5
Belgium	Girls	6
China	Boys	7
China	Girls	8

Figure 4.1: An example of table reshaping. A *Fold* operation transforms the table on the left to the one on the right; an *Unfold* operation performs the reverse.

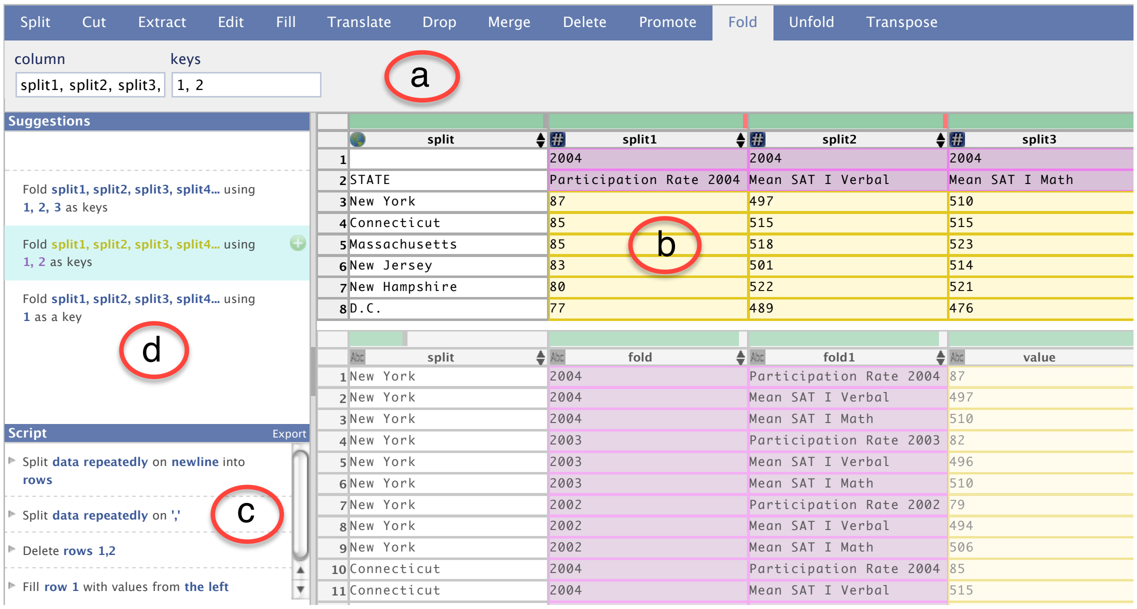
4.3.2 The Wrangler User Interface

The Wrangler user interface (shown in Figure 4.2) allows analysts to specify transforms in the underlying language. The interface contains four primary components: along the top of the display (labeled "a") is a tool bar for transform specification, the right panel ("b") contains an interactive data table display, and the left panel contains an interactive history viewer describing the current script ("c") and automated transform suggestions ("d"). Similar to other transformation tools [91, 134], a transform can be specified manually by selecting a transform type from the tool bar and then entering in desired parameter values. However, Wrangler provides additional facilities to aid transform specification and verification:

Automated Transform Suggestions

Wrangler's data table display supports a set of common interactions, such as selecting rows and columns by clicking their headers or selecting and manipulating text within table cells. While these actions can be used to specify parameters for a selected transform type, manual specification can be tedious and requires that users have a firm command of the available transforms. To facilitate rapid specification, Wrangler infers applicable transforms directly from an initiating interaction.

After the user makes a selection on the data table, Wrangler generates automatic suggestions in a three-phase process: First, Wrangler infers a set of possible parameter



column: split1, split2, split3,

keys: 1, 2

Suggestions

Fold split1, split2, split3, split4... using 1, 2, 3 as keys

Fold split1, split2, split3, split4... using 1, 2 as keys

Fold split1, split2, split3, split4... using 1 as a key

Script

- Split data repeatedly on newline into rows
- Split data repeatedly on ','
- Delete rows 1,2
- Fill row 1 with values from the left

	split	split1	split2	split3
1		2004	2004	2004
2	STATE	Participation Rate 2004	Mean SAT I Verbal	Mean SAT I Math
3	New York	87	497	510
4	Connecticut	85	515	515
5	Massachusetts	85	518	523
6	New Jersey	83	501	514
7	New Hampshire	80	522	521
8	D.C.	77	489	476

	split	fold	fold1	value
1	New York	2004	Participation Rate 2004	87
2	New York	2004	Mean SAT I Verbal	497
3	New York	2004	Mean SAT I Math	510
4	New York	2003	Participation Rate 2003	82
5	New York	2003	Mean SAT I Verbal	496
6	New York	2003	Mean SAT I Math	510
7	New York	2002	Participation Rate 2002	79
8	New York	2002	Mean SAT I Verbal	494
9	New York	2002	Mean SAT I Math	506
10	Connecticut	2004	Participation Rate 2004	85
11	Connecticut	2004	Mean SAT I Verbal	515

Figure 4.2: The user interface of the Wrangler system, implemented as a rich web application. Clockwise from the top: (a) tool bar for transform specification, (b) data table display, (c) history viewer containing an exportable transformation script, (d) suggested transforms. In this screenshot, the effect of the selected *Fold* transform is previewed in the data table display using before (top) and after (bottom) views.

values in response to the user's selection. Example parameters include regular expressions matching selected text substrings and row predicates matching selected values, generated using programming-by demonstration methods [96, 118]. Second, Wrangler enumerates a list of transforms that accept the inferred parameters. Third, Wrangler filters and ranks the resulting transforms according to historical usage statistics and heuristics intended to improve result quality. These transforms then appear as a ranked list of suggestions in the interface, where users can preview their effects and modify their parameters. For more details regarding the Wrangler inference engine, we refer interested readers to Kandel et al. [96].

Assessing Transform Effects

When creating data transformation scripts, users often find it difficult to understand a transform's effect prior to executing it [96]. To aid in transform assessment, Wrangler presents suggestions using natural language descriptions to aid quick scanning of the suggestions list. When a transform is selected, Wrangler uses visual previews to enable users to quickly evaluate its effects without executing it. When feasible, Wrangler displays in-place previews in the data table, drawing users' attention to the effect of the transform in its original context. For complex reshaping operations, Wrangler shows before and after images of the table and uses color coding to help users perceive the correspondence of cells between the two states (Figure 4.2b).

4.3.3 Opportunities for Improvement

Wrangler's automated suggestions and visual assessment features help analysts iteratively hone in on a desired transformation. However, observing user activity with Wrangler via both a formal user study and informal anecdotes revealed that analysts have difficulty formulating data cleaning strategies [96]. Novice users are often unsure which target data state is the one best suited for subsequent analysis. Even when the target data state is known, users are often unsure which sequence of operations is needed to reach that state. We have observed that both novice and expert users consequently resort to blind exploration of the available transforms. This process is hampered by the fact that some operations remain difficult to specify despite the aid of automated suggestions. In particular, reshaping operations that significantly alter the arrangement of table cells (e.g., *fold* and *unfold*) are hard to conceptualize and apply. However, it is exactly these operations that are often necessary to map input data to a format usable by analysis tools.

We attempt to address these issues by extending Wrangler to provide *proactive suggestions* in addition to the ordinary "reactive" suggestions initiated by user actions. Our proactive suggestions are intended to lead users towards effective cleaning strategies and facilitate the specification of complex reshaping operations. After describing the details of our proactive suggestion algorithm (Section 4.4), we go on to analyze the behavior of our algorithm (Section 4.5) and evaluate our approach in a user study (Section 4.6).

4.4 Proactive Data Wrangling

We now describe extensions to Wrangler to automatically generate proactive suggestions. In addition to the previous method of suggesting transforms in response to user-initiated actions (e.g., selecting a row or column), *Proactive Wrangler* continually analyzes the current state of the data table and provides suggestions to make the table more suitable for import into a relational database or analytic tool. These suggestions are intended to both accelerate the work of experts and guide novices towards a desired outcome. For example, we wish to surface complex reshaping operations (*fold* and *unfold*) within a "recognition" task, rather than a harder "recall" task in which the user needs to determine which operation they require and manually initiate its specification.

4.4.1 A Proactive Data Cleaning Scenario

To give a sense of how proactive suggestions ease the data cleaning process, Figure 4.3 illustrates a complete scenario where a user transforms data imported from an Excel spreadsheet (Table 1 within Figure 4.3) into a dense relational format (Table 6). First, 1.

Bureau of I.A.	
Regional Director	Numbers
Niles C.	Tel: (800)645-8397
	Fax: (907)586-7252
Jean H.	Tel: (918)781-4600
	Fax: (918)781-4604
Frank K.	Tel: (615)564-6500
	Fax: (615)564-6701

Manually select and delete the first two rows

Proactive suggestions (none chosen):
1. Fill column 1 from above
2. Split column 2 on ':'
3. Delete empty rows

2.

Niles C.	Tel: (800)645-8397
	Fax: (907)586-7252
Jean H.	Tel: (918)781-4600
	Fax: (918)781-4604
Frank K.	Tel: (615)564-6500
	Fax: (615)564-6701

Proactive suggestions:
1. Fill column 1 from above
2. **Split column 2 on ':'**
3. Delete empty rows

3.

Niles C.	Tel	(800)645-8397
	Fax	(907)586-7252
Jean H.	Tel	(918)781-4600
	Fax	(918)781-4604
Frank K.	Tel	(615)564-6500
	Fax	(615)564-6701

Proactive suggestions:
1. **Delete empty rows**
2. Fill column 1 from above

4.

Niles C.	Tel	(800)645-8397
	Fax	(907)586-7252
Jean H.	Tel	(918)781-4600
	Fax	(918)781-4604
Frank K.	Tel	(615)564-6500
	Fax	(615)564-6701

Proactive suggestions:
1. **Fill column 1 from above**

5.

Niles C.	Tel	(800)645-8397
Niles C.	Fax	(907)586-7252
Jean H.	Tel	(918)781-4600
Jean H.	Fax	(918)781-4604
Frank K.	Tel	(615)564-6500
Frank K.	Fax	(615)564-6701

Proactive suggestions:
1. Unfold column 1 on column 3
2. **Unfold column 2 on column 3**

6.

	Tel	Fax
Niles C.	(800)645-8397	(907)586-7252
Jean H.	(918)781-4600	(918)781-4604
Frank K.	(615)564-6500	(615)564-6701

Result: cleaned table ready for DB import

Figure 4.3: Using Proactive Wrangler to reformat raw data from a spreadsheet. The proactive suggestion chosen at each step is highlighted in **bold**. In the Wrangler UI, proactive suggestions are displayed in the suggestions panel (Figure 4.2d).

Wrangler analyzes the initial table state and makes 3 proactive suggestions. The user ignores them and instead manually selects and deletes the first two rows, turning Table 1 into Table 2. Then the user chooses to execute the 2nd suggestion (shown in **bold**), *Split column 2 on ':'*, which transforms the data into Table 3. Wrangler continues proactively suggesting additional transforms, and the user can simply pick the desired ones from the list (a "recognition" task) rather than manually specifying each one (a "recall" task).

Although the above example is small, it is indicative of a common usage pattern when cleaning data with Proactive Wrangler. In particular, cleaning often involves executing a mix of human-initiated and proactively-suggested transforms. Here the user must recognize that rows 1 and 2 are extraneous headers that should be deleted, but Wrangler proactively suggests all other necessary transforms. In the remainder of this section, we describe how Wrangler generates, filters and ranks these proactive suggestions.

4.4.2 A Metric for Data Table "Suitability"

We define a *suitable* data table as one that can be loaded and effectively manipulated by an analytic tool or relational database. These tools often expect data in a relational format where each row contains a single data record and each column represents a variable of a given data type. Although the raw data in Table 1 of Figure 4.3 is human-readable, it is not suitable for analytic tool or database import because of its irregular structure: it contains extraneous header rows, empty rows, and telephone/fax numbers are located in cells mixed with other strings, separated by a colon delimiter (e.g., "Tel: (800)645-8397"). In contrast, the compact relational result in Table 6 is more readily usable by downstream tools.

We created a metric that assesses how amenable a table is for import into downstream tools. Our metric rewards column type homogeneity (H), fewer empty values (E), and lack of delimiters (D). A *lower* score indicates a more suitable table. A homogeneous, dense, delimiter-free table (e.g., Table 6 in Figure 4.3) has a "perfect" score of zero; tools expecting relational tables should be able to import it.

For a table T with rows R and columns C, we define the table suitability S as

$$S(T) = \left(1 - \frac{\sum_{c \in C} H_c(T)}{|C|}\right) + \frac{E(T) + D(T)}{|R|\,|C|} \tag{4.1}$$

where $|R|$ is the number of rows, $|C|$ is the number of columns, and each component function is defined as follows:

- H_c is the **homogeneity** of column c, the sum of squares of the proportions of each data type *Type* present in that column:

  $$H_c = \sum_{\mathit{Type}} \left(\frac{|i \in R : c_i \in \mathit{Type}|}{|R|}\right)^2 \tag{4.2}$$

  Wrangler parses each cell's contents into the most specific possible type, which can be a built-in such as *number* or *date*, a user-defined type such as *zip code* or *country name*, or a generic *string* type for those that fail to match a more specific type (for more details regarding Wrangler's type inference, see Kandel et al. [96]). If 75% of values in a column are numbers and 25% are dates, then the column's homogeneity is $(0.75)^2 + (0.25)^2 = 0.625$. If 50% are numbers and 50% are dates, then the column is less homogeneous: $(0.5)^2 + (0.5)^2 = 0.5$. A column that contains only values of a single type has the maximum possible homogeneity score of 1. Our table suitability score averages the homogeneity of all columns and subtracts that average from 1, so that higher homogeneity leads to a *lower* (more "suitable") score.

- E is the count of **empty cells**. We favor denser tables, as missing values may indicate a sub-optimal data layout possibly containing replicated values in adjacent columns.

- D is the count of **delimiters**. A delimiter is a comma, colon, pipe, or tab character. We favor tables with fewer delimiters, as those tables are more likely to consist of atomic data rather than compound elements that could split into specific values.

Our suitability score is invariant of table size, since its terms are all normalized by the numbers of rows and columns. However, one of its limitations is that it does not penalize *data loss*. In the extreme pathological case, deleting all cells except for one can lead to a perfect score of 0 (assuming that last cell contains no delimiters). In the next section, we describe how we designed Proactive Wrangler to avoid suggesting such undesirable transforms.

Poor scores in each component of Equation 4.1 make a table ill-suited for tool import. In databases and analytic tools, a column with a mixture of data types typically can only be imported as uninterpreted strings. A column with many delimiters may be treated as strings rather than composite values of more specific types. A column with many empty cells often indicates replicated values in adjacent columns. All these situations can limit downstream analyses, which often work best with sets of well-identified values.

4.4.3 Generating Candidate Transforms

Proactive Wrangler automatically suggests transforms that improve a table's suitability score by first generating candidate suggestions, and then evaluating, filtering, and ranking these candidates before presenting them to the user.

One way to avoid suggesting undesirable lossy transforms is to explicitly penalize data loss when ranking suggestions; however, this suffers from efficiency problems because all possible suggestions must still be generated, evaluated, and ranked. Instead, we adopt a simpler approach: we prune the space of suggestions to ensure that Proactive Wrangler never makes suggestions that involve significant data loss. We limit lossy transforms to deletions of mostly-empty rows (which often appear in spreadsheet headers or footnotes). We contend that data deletion should be left to the user's discretion, and so Proactive Wrangler should only help to compact (fill and delete empty cells), parse (split on delimiters), and reshape (fold and unfold) the table. Thus, it generates the following types of transforms as candidates at each step:

- **Fold** columns $C_1, \ldots, C_n$ using rows $R_1, \ldots, R_n$ as keys
- **Unfold** column C_1 on column C_2
- **Split** column C into multiple columns using a delimiter (comma, colon, pipe, or tab)
- **Fill** all empty cells of row R with values from the left
- **Fill** all empty cells of column C with values from above
- **Delete** all mostly-empty rows ($\geq 75\%$ empty cells)
- **Delete** all empty rows
- **Delete** all empty columns

Since most of these transforms take row and/or column indices as parameters, enumerating all possibilities is both infeasible and undesirable. Assuming that a table has $|R|$ rows and $|C|$ columns, there are $O(|R|)$ possible row fills, $O(|C|)$ possible column fills and splits, $O(2^{|R|} \cdot 2^{|C|})$ possible folds, and $O(|C|^2)$ possible unfolds. It would take far too much time to evaluate all possible parameterizations, precluding interactive response rates. Even if efficiency were not a concern, many parameterizations are undesirable since they trigger degenerate cases of certain transforms.

Informed by our data cleaning experiences and empirical observations from a corpus of public data sets (described in Section 4.5), we developed rules to reduce the set of transform parameters and eliminate undesirable candidates:

- ***Fill.*** Proactive Wrangler only generates *fill* candidates if the majority ($\geq 50\%$) of cells in a given row or column are empty. This type of transform is useful for filling in sparse key rows (or columns) before doing a fold (or unfold).
- ***Fold.*** Proactive Wrangler only generates *fold* candidates using all columns except for the leftmost one (i.e., Columns 2 through $|C|$), and using at most the first 3 rows as keys (i.e., Row 1, Rows 1–2, or Rows 1–3).

The above rule reduces the number of possible folds from $O(2^{|R|} \cdot 2^{|C|})$ to 3 and covers the majority of observed use cases for transforming a cross-tabulation into a relational format. To quantify our intuition, we manually inspected all 114 tables in the 2010 OECD Factbook, an annual publication of world socioeconomic data sets from `oecd.org`. For all except two of the tables, the *only* sensible fold used columns 2 through $|C|$, and for all except one table, the only sensible fold used at most the first 3 rows as keys. A cursory inspection of cross-tabulation data sets from Wikipedia, US Bureau of Justice, and `data.gov` showed no tables that could be sensibly folded using columns other than 2 through $|C|$ or using more than the first 3 rows as keys.

Also, Proactive Wrangler only generates fold candidates if the leftmost column contains all unique and non-null values in rows other than the key row(s); otherwise the folded table will have conflicts. For example, if there were a duplicate "Australia" row in the table on the left of Figure 4.1, then the results of a fold would be ambiguous.

- ***Unfold.*** Proactive Wrangler only generates *unfold* candidates of the form "Unfold C_1 on C_2" if all columns except for C_2 are completely homogeneous (single-typed) and non-empty; otherwise the headers in the resulting cross-tab will not be well-typed. (The values in C_2 become the data matrix at the core of the resulting table, so it can have empty cells.) Also, Proactive Wrangler only generates unfold candidates if the table has exactly 3, 4, or 5 columns, which corresponds to 1, 2, or 3 key columns, respectively. Our intuition for imposing this limit is similar to why we only restrict folds to using at most the first 3 rows as keys: it covers the majority of use cases, as indicated by our data corpus.

The aforementioned rules reduce the number of candidate transforms from $O(2^{|R|} \cdot 2^{|C|})$ to $O(|R| + |C|)$. At each step (e.g., in Figure 4.3), Proactive Wrangler generates all of these candidates at once and then evaluates, filters, and ranks them before presenting them to the user.

4.4.4 Ranking and Presenting Proactive Suggestions

After Proactive Wrangler generates a set of candidate transforms, it executes each of them on the current table and calculates the suitability score for the resulting table. It removes all candidates that result in tables with scores higher ("worse") than the current table. We use a slightly modified scoring method for *unfold* transforms. The operation "Unfold C_1 on C_2" transforms the values in column C_2 into a 2D data matrix at the core of the resulting table. Thus, that matrix should be scored as an atomic unit rather than as disparate columns. Proactive Wrangler calculates the score improvement of an unfold by the change in homogeneity and density between C_2 and the resulting data matrix.

Proactive Wrangler then presents the remaining candidates to the user in the suggestions panel (Figures 4.2d and 4.3), sorted by the amount of improvement ($-\Delta S$). A user can preview and execute one of these proactive suggestions or can choose to ignore them, and instead make a selection on the data table to surface the ordinary context-dependent suggestions.

4.5 Characterizing Proactive Algorithm Behavior

To assess the quality of our proactive suggestions, we analyzed the behavior of our algorithm across a collection of diverse data sets. For each, we compared the suggestions produced by our algorithm with "ideal" transformation scripts constructed by hand. As there is often more than one feasible "clean" goal state, we strove to transform each data table into a relational format amenable for import into a database or analytic tool and also sought to remove extraneous tokens from fields (e.g., parentheses, leading dashes, etc.).

We compared 20 distinct data sets in our analysis. While this collection is relatively small, each data set was chosen because it is representative of a much larger class of data we have collected. For example, although we only included three tables from the 2010 OECD Factbook of world socioeconomic data, all 114 tables from that source had a format that matched one of these three canonical tables. Although we cannot claim that our corpus is a representative sample of all data that users might want to clean, we strove to select data of diverse origins, formats, and complexity:

- ***Origins.*** Our corpus consists of a mix of government and NGO data (e.g., OECD, US Department of Justice, data.gov), scientific data (e.g., flower classifications, audiology experiments) and data scraped from websites (e.g., Wikipedia reference tables, Craigslist apartment rental listings).
- ***Formats.*** Data formats range over cross-tabulations (e.g., left half of Figure 4.1), Excel reports intended for human consumption (e.g., Table 1 in Figure 4.3), line-oriented records with custom token separators and ad-hoc formats generated by scientific experiments.
- ***Complexity.*** The most concise cleaning script we constructed for data sets in our corpus ranged from 1 to 7 transforms in length (i.e., some data sets were noticeably "dirtier" than others). The mean script length was 3.6 transforms.

We now summarize the results of using Proactive Wrangler to transform the 20 data sets in our corpus. When available, we always selected the highest-ranking proactive suggestion that would lead us towards the goal state; otherwise we initiated a selection on the data table and chose to execute the appropriate context-specific suggestion. Of all executed transforms, 53% (39/73) were proactive suggestions; the remaining 47% (34/73) resulted from human-initiated actions. When appropriate proactive suggestions appeared, the top-ranked (#1) suggestion was the preferable choice 77% of the time (30/39). The lowest rank of any chosen proactive suggestion was 6, and the mean rank was 1.6.

For reference, in the ideal case 100% of all executed transforms would come from the top-ranked proactive suggestion. In reality, our scoring metric is not perfect and there are certain classes of transforms that Proactive Wrangler is designed specifically *not* to offer. For our corpus, the following transforms required human intervention: splitting on non-standard delimiters, extracting and cutting substrings, and deleting non-sparse rows and columns. Across our collection, 6 of the corpus tables can be cleaned solely using proactive suggestions, 9 require a mix of human-initiated selections and proactive suggestions, and 5 require exclusively human selections. We now discuss each category in turn.

4.5.1 Fully Proactive

Six of the data tables in our corpus (30%) exhibited the best-case performance for Proactive Wrangler: all of the required transforms appear as proactive suggestions. This typically occurs for tables whose individual elements are properly formatted but the table itself requires re-structuring.

For example, all 114 tables in the 2010 OECD Factbook and all 50 state-level crime data sets from the Department of Justice are cross-tabulations embedded within Excel spreadsheets. Even though we only included a few samples in our corpus, the rest of the tables from those sources are identically formatted. For all of these tables, Proactive Wrangler suggests to delete the mostly-empty title and footnote rows, fill in the header key row(s), and then perform the appropriate fold to get the data into a relational format.

4.5.2 Hybrid

In the common case (9/20 tables, 45%), data must be cleaned using a combination of user-initiated transforms and proactive suggestions. For these kinds of tables in our corpus, 55% of executed transforms (23/42) came from proactive suggestions. By design, Proactive Wrangler does not offer certain types of suggestions, namely of transforms that lead to data loss. This limitation causes Wrangler to sometimes get "stuck" at a local optimum point when it cannot generate any more proactive suggestions. In those cases, human intervention can allow Wrangler to escape local optima and resume generating useful proactive suggestions.

For example, in a veteran's hospital data set, we first chose a proactive suggestion to delete all empty rows, but then we had to manually delete an unnecessary column and remove (cut) the trailing '%' character from all numbers representing percentages. After those user-initiated clean-ups, Wrangler proactively offers the proper fold suggestion, which we executed to complete our cleaning task.

4.5.3 Manual

Wrangler cannot offer useful proactive suggestions for data that require extensive string parsing within individual cells, which occurred in a quarter of our examples (5/20 tables, 25%). These data sets all require the user to perform custom string splits, cuts, or extracts. For two of these data sets, Wrangler "stayed out of the way" and did not offer any proactive suggestions. For the other three, Wrangler offered on average two (unhelpful) proactive suggestions. Our findings from the user study (see next section) indicate that users find it easy to ignore such unhelpful suggestions.

For example, each textual line in a Craigslist apartment listings data set is a string in this format:

```
$2475/2br - Superb location - (palo alto) pic
```

To create a table of rental prices, bedroom count, and locations, one can split each line on the '/' character, cut the '$' from the price to make it into an integer, and extract the location (e.g., "palo alto") from within the parentheses. These types of string manipulations are easier for a human to initiate using Wrangler's reactive suggestions; it is difficult for an algorithm to suggest these manipulations *a priori* with both high precision and recall. Consequently, Wrangler does not currently attempt to make proactive suggestions for these manipulations.

4.6 User Study

We conducted a user study comparing versions of Wrangler with and without proactive suggestions. We originally hypothesized that proactive suggestions would simplify specification of commonly-used, but hard-to-specify reshaping transforms (e.g., fold, unfold), while causing limited distraction to users specifying other types of transforms. Furthermore, we hypothesized that by lowering the cost of finding applicable transforms, users would more quickly complete data cleaning tasks requiring multi-step transformations.

As we will discuss, our results indicate that suggestions generated by our proactive model can help users discover applicable transforms, but not always in the manner we anticipated. Users often dismissed proactive suggestions presented prior to an initiating interaction, but made use of those exact same suggestions when presented later in the session.

4.6.1 Participants

We recruited 16 participants (11 male, 5 female) who had never used Wrangler before. All were university students with at least moderate experience programming and using data analysis tools such as Microsoft Excel or Matlab; all came from Computer Science or related majors. Participants received a $15 gift certificate for one hour of their time.

4.6.2 Methods

We evaluated the impact of proactive suggestions by comparing Proactive Wrangler to the original baseline version of Wrangler that generates suggestions only in reaction to user-initiated interactions. First, we guided each subject through a 15-minute tutorial of Wrangler, highlighting both proactive and reactive modes.

Next, we had each subject perform data cleaning tasks on four data sets that were miniature versions of data sets from our corpus. We designed each task so that, in proactive mode, helpful proactive suggestions were available at most but not all steps (the “Hybrid” category from the algorithm behavior study of Section 4.5). We imposed a 10-minute limit per task. The four cleaning tasks were divided into two pairs, each requiring structurally similar transformations. We gave participants a paper printout of the desired goal table state for each task. For a given pair of tasks, we assigned subjects one task using Proactive Wrangler and the other task using the baseline “reactive” Wrangler. We counterbalanced task order and Wrangler versions across subjects.

After completing the four tasks, we asked subjects to describe their preferences for reactive versus proactive suggestions. Each study session was administered on a 15” MacBook Pro laptop and recorded using screen capture software.

The first eight subjects used a version of Proactive Wrangler that displayed proactive suggestions only when there was no active selection on the data table; as soon as the user makes a selection, the proactive suggestions were replaced by the standard reactive suggestions. However, we observed that most subjects largely ignored the proactive suggestions in this configuration. After interviewing those subjects about their rationale, we updated Wrangler to append the top three proactive suggestions to the list of reactive suggestions normally generated from a user interaction. In other words, Wrangler now displayed proactive suggestions regardless of how the user interacted with it. This modified *embedded proactive* interface was seen by the last eight subjects.

4.6.3 Results

We initially expected subjects to complete tasks faster using Proactive Wrangler, but in fact we found no significant difference in completion times between proactive, embedded, or reactive variants ($F_{2,51} = 0.059$, $p = 0.943$ according to repeated measures ANOVA). We compared within individual tasks and controlled for the effects of task order, but found no significant differences. The average task completion time was 231 seconds; in only one instance did a user fail to complete a task by going over the 10-minute time limit.

Upon reviewing the usage logs, we found that subjects made sparing use of proactive suggestions. Most users (11/16, 69%) executed a proactive suggestion at least once. However, the total number of such transforms was small: in proactive conditions, subjects executed 1.16 proactive suggestions per task on average, compared to an average of 4.9 reactive suggestions on those same tasks.

To understand why subjects were reluctant to use the proactive suggestions, we reviewed our recorded experiment sessions to identify critical incidents. We identified three recurring patterns: subjects ignoring useful proactive suggestions, previewing but dismissing suggestions, and viewing suggestions as a last resort. We now discuss each type of incident in turn:

Ignoring proactive suggestions

Nearly one-third of the subjects (5/16) simply never previewed any proactive suggestions, despite having used them in the tutorial. All of these subjects ignored at least one proactive suggestion that they later executed after making a selection and choosing an *identical* reactive suggestion. For example, after 137 seconds of work, one subject was almost done with a task and only needed to perform a final unfold. The necessary unfold was displayed as the sole proactive suggestion, but the subject ignored it and spent the next 86 seconds experimenting with selecting columns and specifying various folds and unfolds. He finally chose the correct unfold and completed his task, at which point he said, *"A proactive suggestion would have really helped me here!"*

Once we integrated proactive suggestions among reactive suggestions, 4 out of the 8 subjects who used the embedded version ignored a useful proactive suggestion only to click on it immediately after selecting an unrelated part of the table. One subject wanted to do a fold when in fact he needed an unfold (which was proactively suggested but ignored). When he selected some columns and attempted a fold, he noticed the proactive unfold suggestion at the bottom of the list and realized he needed to unfold, so he selected it and completed his task. This unfold would not have surfaced if he had not been using the embedded proactive interface.

Previewing, then ignoring, proactive suggestions

Almost half of our subjects (7/16) actually previewed the effects of a useful proactive suggestion, dismissed it, and a short time later made a selection and executed the *identical* reactive suggestion. As a typical example, one subject previewed a proactively-suggested unfold transform for a few seconds. But instead of executing it, he immediately selected the two columns in the table necessary to have Wrangler reactively suggest the *same* unfold. He then navigated directly to that unfold transform, previewed, and executed it.

Proactive suggestions as a last resort

Three subjects used proactive suggestions only when they ran out of ideas for triggering reactive suggestions. For example, out of the 167 seconds that one subject spent on a task, she was stuck staring at the table for 60 seconds (36% of total time) before she looked at the top-ranked proactive fold, previewed, and then executed it. This was the first (and only) time that she paused for long enough to actually consider using proactive suggestions; for the remainder of her session, she was continually making selections on the table, which surfaced reactive suggestions and hid all of the proactive ones.

4.6.4 Study Limitations

All subjects had some programming background, and this might make them more likely to want to explore rather than immediately follow proactive suggestions. One subject remarked, *"Maybe it's just me, but I like to try [initiating Wrangler actions] myself to see how they work."* Also, we provided specific target end states to subjects and timed them. Although many analysts have some end state in mind when working with data, discovering an appropriate end state is often an exploratory task. Several subjects reported that given a more ambiguous task, they believe they would be more likely to explore proactive suggestions, since these may inform what types of end states are possible.

4.6.5 Summary

Our user study largely disconfirms our original hypotheses. Users initially ignored proactive suggestions and on average did not complete tasks more quickly with them. However, proactive suggestions did aid the transformation process. Over two-thirds of subjects executed a proactive suggestion at least once. Suggestions generated by our proactive model proved helpful when embedded among reactive suggestions and when users failed to complete tasks by other means. At minimum, proactive suggestions are occasionally helpful and do not disrupt users. Moreover, our results raise some future research questions for the design of mixed-initiative interfaces. We discuss these in the next section.

4.7 Discussion and Future Work

We presented a metric for table "suitability" that we apply to generate suggested data transforms. When compared with idealized transformation scripts, our model produces helpful suggestions for a majority of transform steps and ranks the preferred suggestion highly. In our own use, we have found proactive suggestions to be of great assistance: we regularly use the tool for data wrangling and find proactive suggestions to be a valuable complement. As a result, we believe our proactive suggestion model helps advance the state-of-the-art in interactive data transformation.

However, our user study results cast a shadow on our initial motivations. We find little evidence that proactive suggestions help novice users complete transformation tasks more quickly or learn data cleaning strategies. Upon review of our notes and recordings, we hypothesize multiple factors shape the observed usage patterns: *attentional blindness* to proactive suggestions, users' desire to *sustain initiative*, and *insufficient expertise* to recognize the value of suggestions.

4.7.1 Attention Blindness

Some subjects reported that they did not attend to proactive suggestions, even after using them in the tutorial. Many instead clicked on the table to initiate reactive suggestions. On the one hand, subjects expressed appreciation that proactive suggestions did not interrupt their workflow: annoying distractions are a classic complaint for mixed-initiative interfaces. However, insufficient visibility also seems to be a problem. Wrangler and similar tools might benefit from cues that emphasize suggested operations. For instance, subtle animation or colored backgrounds might draw attention to high-confidence suggestions without interrupting users' flow.

4.7.2 User Agency and Sustained Initiative

Many users expressed a pre-existing disdain for proactive interfaces. One said: *"I hate suggestions popping up at me on the computer ... I just want to get them out of my way."* Users described a distrust of automated suggestions in productivity software, citing examples such as Clippy, the infamous Microsoft Office "paper clip" assistant. Another user, describing his reluctance to use Wrangler's proactive suggestions, stated *"I knew what I wanted to do with the data, so I just did it myself."* Ironically, users voiced no such qualms regarding reactive suggestions, which were regularly employed to transform data.

During the first half of our study, only 4/8 subjects (50%) executed proactive suggestions. After we deployed the embedded proactive interface, 7/8 subjects (88%) executed suggestions generated by our proactive model, including those inserted into reactive suggestions. It is possible that user assessment of suggested transforms improves when users perceive of themselves as the initiator. We observed several instances of subjects making a selection, choosing an embedded proactive suggestion, and executing it *after* the same suggestion had already been proactively presented and viewed.

Subjects seem to value suggestions more when they are offered in response to an initiating action, even if the suggestions are generated independently. While speculative, this observation suggests that future research might examine not only the utility of suggestions to the stated task goal [90], but also how interface design and turn-taking affects user receptiveness to those suggestions. What design decisions might help sustain users' sense of control in mixed-initiative UIs?

4.7.3 Expertise for Transform Assessment

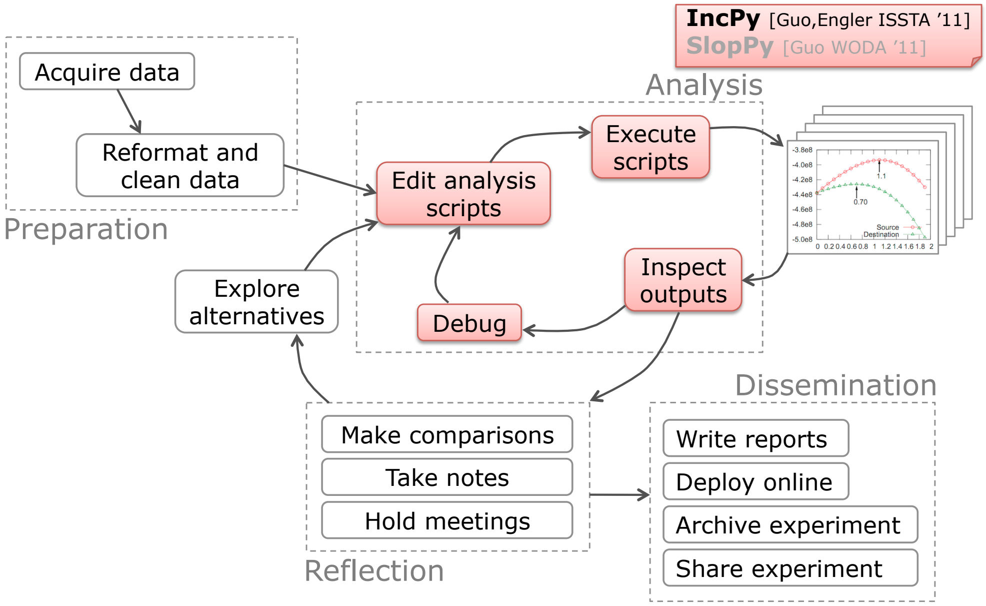
Our own internal use of Proactive Wrangler and the results of our study with novice users are at a disconnect. In our experience, proactive suggestions can reduce the cognitive effort and also the time required to specify transformations. Initially we hypothesized that proactive suggestions can speed transformation by casting transform specification as a "recognition" task, in which users recognize a desired transform among the list of suggestions. Our user study results suggest otherwise. Prior knowledge of transformation strategies may be needed to rapidly “recognize” the value of a transform. Thus, proactive suggestions may actually be more effective for expert users than for novices. New preview mechanisms might better facilitate user assessment of suggestions. For example, thumbnails of a transform’s effects might be easier to comprehend than the current textual descriptions.

More sophisticated approaches may be needed to foster learning of multi-step transformation strategies. In most cases subjects followed a greedy approach to discovering an appropriate chain of transformations. Some complex transformations may require steps in which it is difficult to gauge progress from an intermediate state (e.g., a fold transform followed by a subsequent unfold). Subjects stated that they were worried that proactive suggestions might lead them down the wrong path. However, they often did later select and execute transforms that were identical to those previously suggested by Proactive Wrangler. By this point, the subject had explored more alternatives and used interactions to communicate their intent. It is unclear which of these two factors gave the user more trust in the system.

Ultimately, users may be the most successful once they are able to articulate and evaluate their own transformation strategies. Towards this aim, future work might examine alternative approaches to fostering expertise. Tutorial generation may be one means. Another possibility is to examine the effects of multi-step suggestions. New user interface techniques for suggesting and previewing multiple steps may aid these situations; can a different design prompt multi-step reasoning rather than greedy search? More broadly, designing mixed-initiative interfaces to foster expertise through example presents a compelling challenge for future work.

Chapter 5

IncPy: Accelerate the Data Analysis Cycle



This chapter presents an enhanced Python interpreter called IncPy, which speeds up the data analysis scripting cycle and helps researchers manage code and data dependencies. Its contents were adapted from a 2011 conference paper that I co-authored with Dawson Engler [75], which is an extended version of a 2010 workshop paper [73]. All uses of "we", "our", and "us" in this chapter refer to both authors.

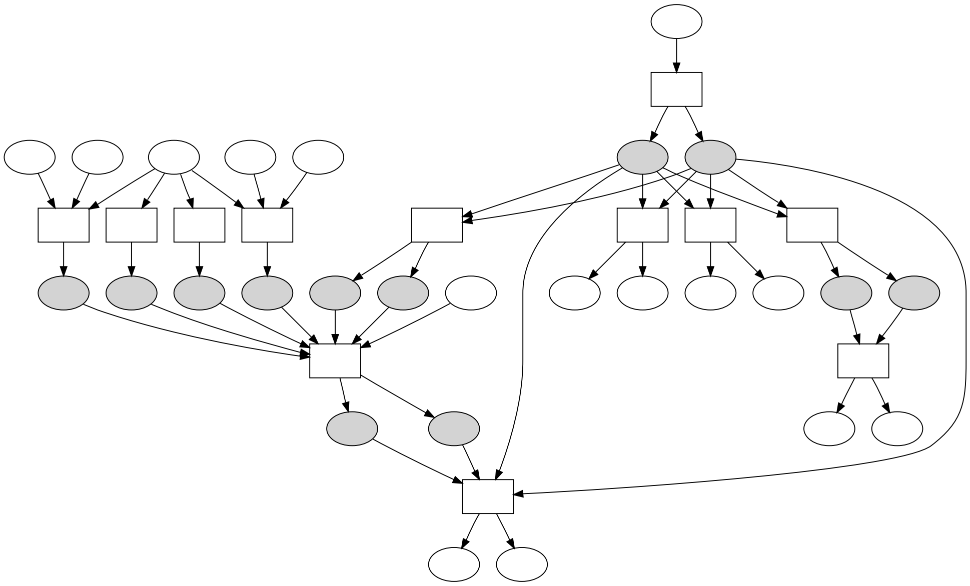


Figure 5.1: A data analysis workflow from my Summer 2009 internship project [78] comprised of Python scripts (boxes) that process and generate data files (circles). Gray circles are intermediate data files, which IncPy can eliminate.

5.1 Motivation

To illustrate a common problem that arises during data analysis scripting, I will recount my experiences during a 2009 summer internship at Microsoft Research. My project was to analyze software bug databases and employee personnel datasets to quantify people-related factors that affect whether bug reports are fixed, reassigned, or reopened. This project was later published as three conference papers [78, 79, 156]. During that summer, I wrote all scripts in one language (Python), but my datasets were stored in diverse file formats (e.g., semi-structured plaintext, CSV, SQL database, serialized objects), which is a typical heterogeneous setup for data analysis workflows [42].

I first wrote a few scripts to process the primary datasets to create output charts and tables. However, since those scripts took a long time to run (tens of minutes to several hours), I split up my analysis into multiple stages and wrote serialization code to output the intermediate results of each stage to disk. Breaking up my scripts into stages — implemented as functions or separate scripts — improved performance and sped up iteration times: When I edited and re-executed later stages, those stages could re-use intermediate results from disk rather than waste time re-executing unchanged earlier stages. However, my hard disk was now filled with dozens of scripts and intermediate data files.

Upon inspecting my output charts and tables, my supervisor often asked me to adjust my scripts or to fork my analyses to explore multiple alternative hypotheses (e.g., *"Please explore the effects of employee location on bug fix rates by re-running your analysis separately for each country."*). Thus, I parameterized my scripts to generate multiple output datasets based on command-line parameters, which created even more output data files.

Having to manually keep track of dozens of scripts and data files led to frustrating bugs. For example, I would update certain datasets but forget to re-run the scripts that analyzed them, which meant that some other data files were now incorrect. Or I would delete scripts but forget to delete the data files they generated, leaving "orphan" data files that might erroneously be processed by subsequent scripts. Or I would forget which scripts and datasets generated which charts, and whether those charts were up-to-date. Since these bugs all manifested as incorrect outputs and *not as crashes*, it was difficult to actually determine when a bug occurred. To be safe, I would periodically re-run all of my scripts, which eliminated the performance benefits of splitting up my workflow into multiple stages in the first place.

One possible solution would be to write all of my code and dataset dependencies in a Makefile [57], so that invoking `make` would only re-run the scripts whose dependent datasets have changed. However, since I rapidly added, edited, and deleted dozens of scripts and datasets as I explored new hypotheses, it was too much of a hassle to also write and update the dependencies in parallel in a Makefile. At the end of my internship, I finally created a Makefile to document my workflow, but some dependencies were so convoluted that my Makefile likely contains errors. Figure 5.1 shows the dependencies between my Python scripts (boxes) and data files (circles), extracted from my end-of-internship Makefile.

My colleagues also experienced similar frustrations throughout the data analysis scripting process. A literature search revealed that even veteran computational scientists acknowledged that having to organize code, data, and their dependencies was an impediment to productivity [98, 124].

5.2 Contributions

Problem: General-purpose programming languages provide no support for managing the myriad of dependencies between code and data sets that arises throughout the data analysis process. The "state-of-the-art" solution that veteran data analysts suggest is to simply use disciplined file naming conventions and Makefiles as the "best practices" for coping with these dependencies [98, 124].

Our solution: To enable programmers to iterate quickly without the burden of managing code and file dependencies, we added dynamic analyses to the programming language interpreter to perform automatic memoization and dependency management. Our technique works as follows:

1. The programmer's script runs in a custom interpreter.
2. The interpreter automatically memoizes [117] (caches) the inputs, outputs, and dependencies of certain function calls to disk, only doing so when it is safe (pure and deterministic call) and worthwhile (faster to re-use cached results than to re-run) to do the memoization.
3. During subsequent runs of the same script (possibly after the programmer edits it), the interpreter skips all memoized calls and re-uses cached results if the code and data that those results depend on are unchanged.
4. The interpreter automatically deletes on-disk cache entries when their code or data dependencies are altered.

We implemented our technique as a custom open-source Python interpreter called IncPy (**Inc**remental **Py**thon) [9]. However, our technique is not Python-specific; it can be implemented for similar languages such as Perl, Ruby, or R.

Benefits: IncPy improves the experience of writing data analysis scripts in three main ways:

- **Less code**: Programmers can write data analysis stages as pure functions that return ordinary program values and connect stages together using value assignments within their scripts. IncPy automatically memoizes function inputs/outputs to a persistent on-disk cache, so programmers do not need to write serialization and deserialization code. Less code means fewer sites for bugs. Although programmers get the most memoization benefits when they write code in a modular and functional style, IncPy does not enforce any particular style. Programmers can use the full Python language and perform impure actions when convenient.

- **Automated data management**: IncPy manages the dependencies between code and datasets so that the proper data can be updated when the code they depend on changes, thus preventing stale data bugs. IncPy tracks datasets that already exist on disk (e.g., CSV files and SQL databases) as well as those that it creates by memoizing function calls. For example, I could have eliminated all of the gray circles (intermediate data files) in Figure 5.1 if I had used IncPy. Each analysis stage (box in Figure 5.1) could directly operate on the return values from upstream stages, and IncPy would automatically create and manage the intermediate datasets (cache entries).

- **Faster iteration times**: IncPy allows programmers to iterate and explore ideas faster because when they edit and re-run scripts, memoized results from unchanged stages can be loaded from the on-disk cache rather than re-computed. Programmers get these performance benefits without having to write any annotations, caching code, or manage intermediate datasets.

Because IncPy works with ordinary Python scripts, it is well-suited for programmers who want to focus on analyzing their data without needing to invest the effort to learn new language features, domain-specific languages, or other tools.

5.3 Example

We use an example Python script to illustrate the basic features of IncPy. Figure 5.2 shows a script that processes a `queries.txt` file using three functions. Assume that the script takes 1 hour to process a 59-line `queries.txt` file, where each line contains an SQL query. `stageA` makes 59 calls to `stageB` (one call for each line in `queries.txt`) followed by 1 call to `stageC`, where each of those calls lasts for 1 minute. The rest of `stageA` terminates instantaneously.

When we run this script for the first time, IncPy tracks the names and values of the global variables and files that each function reads and the code that each function calls. It dynamically generates the dependency graph shown on the bottom half of Figure 5.2, which contains three types of dependencies: For example, the function `stageA` has a *code dependency* on `stageB`; `stageB` has a *global variable dependency* on `MULTIPLIER`; `stageB` also has a *file read dependency* on the `masterDatabase.db` file. As each function call finishes, IncPy memoizes the arguments, return values, and dependencies of each call to a persistent on-disk cache.

Now when we edit some code or data and run the same script again, IncPy can consult the memoized dependencies to determine the minimum number of functions that need to be re-executed. When IncPy is about to execute a function whose dependencies have not changed since the previous execution, it will skip the call and directly return the memoized return value to its caller. Here are some ways in which a subsequent run can be faster than the initial 1-hour run:

- If we edit the end of `stageA` to return, say, the product of `transformedLst` elements rather than the sum, then re-executing is nearly instantaneous since IncPy can re-use all the memoized results for `stageB` and `stageC`.

- If we fix a bug in `stageC`, then re-executing only takes 1 minute: We must re-run `stageA` and `stageC` but can re-use memoized results from all 59 calls to `stageB`.

- If we modify a line in `queries.txt`, then re-executing takes 2 minutes since `stageB` only needs to re-run on the SQL query string specified by the modified

```
MULTIPLIER = 2.5 # global variable

# Input: name of file containing SQL queries
def stageA(filename):
  lst = [] # initialize empty list
  for line in open(filename, 'r'):
    lst.append(stageB(line))
  transformedLst = stageC(lst)
  return sum(transformedLst) # returns a number

# Input: an SQL query string
def stageB(queryStr):
  db = open_database('masterDatabase.db')
  q = db.query(queryStr)
  res = ... # run for 1 minute processing q
  return (res * MULTIPLIER) # returns a number

# Input: a list of numerical values
def stageC(lst):
  res = ...  # run for 1 minute processing lst
  return res # returns a list of numbers

print stageA(”queries.txt”) # top-level call
```

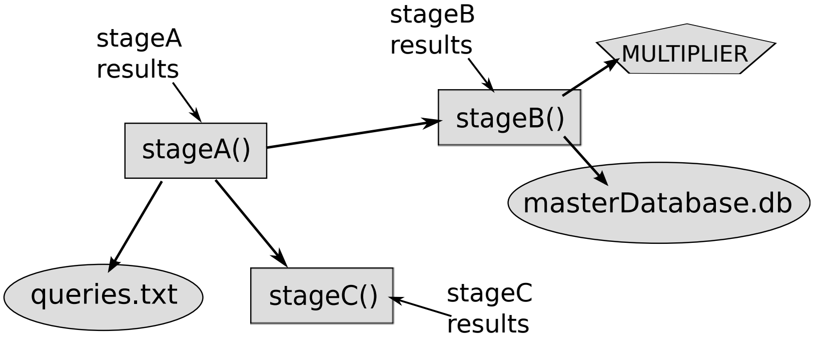


Figure 5.2: Example Python data analysis script (top) and dependencies generated during execution (bottom). Boxes are code, circles are file reads, and the pentagon is a global variable read.

line, and `stageC` must also re-run since its input is different.

- If we append a new line to `queries.txt`, then re-executing takes 2 minutes since `stageB` only needs to run on the new line, and `stageC` must also re-run.

For comparison, if we ran this script in the regular Python interpreter, then *every subsequent run would take 1 hour*, regardless of how little we edited the code or the `queries.txt` file. If we wanted to have our script run faster after minor edits, we would need to write our own serialization and deserialization code to save and load intermediate data files, respectively. Then we would need to remember to re-generate particular data files after their dependent functions are edited, or else risk getting incorrect results. IncPy automatically performs all of this caching and dependency management so that we can iterate quickly without needing to write extra code or to manually manage dependencies.

5.4 Design and Implementation

IncPy consists of dynamic analyses that perform dependency tracking and function memoization (Section 5.4.1), persistent cache management (Section 5.4.2), function profiling and impurity detection (Section 5.4.3), and object reachability detection (Section 5.4.4).

We created IncPy by adding ∼4000 lines of C code to the official Python 2.6.3 interpreter. IncPy is fully compatible with existing Python scripts and 3rd-party extension modules already installed on the user's machine.

5.4.1 Memoizing Function Calls

IncPy's main job is to automatically memoize certain function calls to a persistent on-disk cache when the target program is about to exit the call. Each cache entry represents one memoized call and contains the fields in Table 5.1.

IncPy updates the fields of Table 5.1 in each function's stack frame as it is executing. It does so by interposing on the interpreter's handlers for the corresponding

Full name	Function's name and enclosing filename (for methods, also add the enclosing class's full name)
Arguments	Argument values for this call
Return value	Return value for this call
Terminal output	Contents of text printed to `stdout` and `stderr` buffers during this call
Global var. dependencies	Names and values of all global variables, variables in enclosing lexical scopes, and static class fields read during this call
File read dependencies	Names and last modified times of files read in this call
File write dependencies	Names and last modified times of files written in this call
Code dependencies	Full names and bytecodes of this function and of all functions that it transitively called

Table 5.1: Contents of a persistent on-disk cache entry, which represents one memoized function call.

program events. For example, we inserted code in the interpreter's handler for file I/O to add a file read/write dependency to all functions on the stack whenever the target program performs a file read/write. Thus, if a function `foo` calls `bar`, and some code in `bar` reads from a file `data.txt`, then *both* `foo` and `bar` now have a *file read dependency* on `data.txt`. Similarly, IncPy adds a global variable dependency to all functions on the stack whenever the program reads a value that is reachable from a global variable (see Section 5.4.4 for details).

Although interposing on every file access, value access, and function call might seem slow, performance is reasonable since the bookkeeping code we inserted is small compared to what the Python interpreter already executes for these program events. For example, the interpreter executes a few hundred lines of C code to initiate a Python function call, so the few extra lines we inserted to fetch its argument values and update code dependencies has a minimal performance impact. We evaluate performance in Section 5.5.1.

When the target program finishes executing a function call, if IncPy determines that the call should be memoized (see Section 5.4.3 for criteria), it uses the Python standard library `cPickle` module to serialize the fields in Table 5.1 to a binary file. For arguments, global variables, and function return values, IncPy serializes the entire object that each value refers to, which includes all objects transitively reachable from it. Since IncPy saves these objects to disk at this time, it does not matter if they are mutated later during execution. All cached arguments and global variables have the same values as they did when the function call began; otherwise, the call would be impure and not memoized.

IncPy stores each serialized cache entry on disk as a separate file, named by an MD5 hash of the serialized argument values (collisions handled by chaining like in a hash table). Each file is written atomically by first writing to a temporary file and then doing an atomic rename; doing so allows multiple processes to share a common cache, since no process can ever see a cache file in an inconsistent state while it is being written. All cache entries for a function are grouped together into a sub-directory, named by an MD5 hash of the function's *full name* (see Table 5.1).

IncPy does not limit the size of the on-disk cache; it will keep writing new entries to the filesystem as long as there is sufficient space. IncPy automatically deletes cache entries when their dependencies are altered (see Section 5.4.2). Also, since each cache entry is a separate file, a user (or script) can manually delete individual entries by simply deleting their files.

5.4.2 Skipping Function Calls

When the target program calls a function:

1. **Cache look-up**: IncPy first looks for an on-disk cache entry that matches its full name and the values of its current arguments and global variable dependencies, checking for equality using Python's built-in object equality mechanism. If there is no matching cache entry, then IncPy simply executes the function.

2. **Checking dependencies and invalidating cache**: If there is a match, then IncPy checks the file and code dependencies in the matching cache entry. If a dependent file has been updated or deleted, then the cache entry is deleted. If

the bytecode for that function or any function that it called has been changed or deleted, then all cache entries for that function are deleted. When these dependencies are altered, we cannot safely re-use cached results, since they might be incorrect. For example, if a function `foo` calls `bar` and returns `42`, then if someone modifies `bar` and re-executes `foo`, it might no longer return `42`.

3. **Skipping the call**: If there is a matching cache entry and all dependencies are unchanged, then IncPy will skip the function call, print out the cached `stdout` and `stderr` contents, and return the cached return value to the function's caller. This precisely emulates the original call, except that it can be much faster.

One practical benefit of IncPy atomically saving each cache entry to a file as soon as the function exits, rather than doing so at the end of execution, is that if the interpreter crashes in the middle of a long-running script, those cache entries are already on disk. The programmer can fix the bug and re-execute, and IncPy will skip all memoized calls up to the site of the bug fix. For example, some of our users have encountered annoying bugs where their scripts successfully processed data for several hours but then crashed at the very end on a trivial bug in the output printing code. IncPy was able to memoize intermediate results throughout execution, so when they fixed those printing bugs and re-executed, their scripts ran much faster, since IncPy could skip unchanged function calls and load their results from the cache.

5.4.3 Which Calls Should Be Memoized?

IncPy automatically determines which function calls to memoize without requiring any programmer annotations. However, a programmer can force IncPy to always or never memoize particular functions by inserting annotations.

Which Calls Are Safe To Memoize?

It is only safe to memoize function calls that are pure and deterministic, since only for those calls will the program execute identically if they are later skipped and replaced with their cached return values.

Pure calls: Following the definition of Salcianu and Rinard, we consider a function call *pure* if it never mutates a value that existed prior to its invocation [139]. In Python (and similar languages), all objects reachable from global variables[1] and a function's arguments might exist prior to its invocation. Thus, when a program mutates a globally-reachable object, IncPy marks *all functions* on the stack as impure. For example, if a function `foo` calls `bar`, and some code in `bar` mutates a global variable, then IncPy will mark *both* `foo` and `bar` as impure, since both functions were on the stack when an impure action occurred. When a program mutates an object reachable from a function's arguments, IncPy marks only that function as impure (see Section 5.4.4 for details). Note that an object might be reachable from more than one function's arguments: If a function `foo` accepts an argument `x` and directly passes it into `bar`, then the object to which `x` refers is an argument of both `foo` and `bar`. Section 5.4.4 describes how we efficiently implement reachability checks.

IncPy does *not* mark a function as impure if it writes text to the terminal; instead, it separately captures `stdout` and `stderr` outputs in the cache (see Table 5.1) and prints those cached strings to the terminal when the function is skipped.

Unlike static analysis [139], IncPy dynamically detects impurity of individual execution paths. This is sufficient for determining whether a particular call is safe to memoize; a subsequent call of a pure deterministic function with the same inputs will execute down the same path, so it is safe to skip the call and re-use memoized results. If a function is pure on some paths but impure on others, then calls that execute the pure paths can still be memoized.

Deterministic calls: We consider a function call *deterministic* if it does not access resources such as a random number generator or the system clock. It is difficult to automatically detect all sources of non-determinism, so we have annotated a small number of standard library functions as non-deterministic (e.g., those related to randomness, time, or `stdin`). However, users of IncPy do not have to make any annotations in their programs. IncPy marks all functions on the stack as impure when the target program calls one of these non-deterministic functions.

[1] includes variables in enclosing scopes and static class fields

In theory, memory allocation is a source of non-determinism if a program makes control flow decisions based on the addresses of dynamically-allocated objects. For the sake of practicality, IncPy does not treat memory allocation as non-deterministic, since if it did, then almost all functions would be impure. In our experience, it is rare for programs written in memory-safe languages such as Python to branch based on memory addresses, since pointers are not directly exposed. For example, none of the scripts in our benchmark suite (see Section 5.5.1) branch based on memory addresses.

Finally, IncPy does not handle non-determinism arising from thread scheduling.

Self-contained file writes: Writing to a file might seem like an impure action, since it mutates the filesystem. While that is technically true, we make an exception for a kind of idempotent file write that we call a *self-contained write.* A function performs a self-contained write if it was on the stack when the file was opened in pure-write (not *append*) mode, written to, and then closed. We observed that data analysis scripts often perform self-contained writes: An analysis function usually processes input data, opens an output file, writes data to it, and then closes it. For example, although only one set of scripts in our benchmark suite performed file writes, all of its 17 writes were self-contained (see Section 5.5.2 for a case study of that benchmark).

For example, if a function `foo` does a self-contained write to `data.txt` (open → write → close), then each call to `foo` creates a new and complete copy of `data.txt`. Thus, IncPy still considers `foo` to be pure and records `data.txt` as a *file write dependency.* As long as `foo` and all its dependencies remain unchanged, then there is no need to re-run `foo` since it will always re-generate the same contents for `data.txt`.

However, if a function call writes to a file in a non-self-contained way (e.g., by opening it in *append* mode or not closing it), then IncPy marks that particular call as impure and does not memoize it.

Object-oriented programming support: For object-oriented programs, IncPy memoizes pure deterministic method calls just like ordinary function calls: The receiver (`this` object) is memoized as the first argument, and static class fields are memoized as global variables. Since IncPy tracks methods and objects at run time, it correctly handles inheritance and polymorphism.

```
myA = A()    # new instance of class A
myA.b = B()  # new instance of class B

def foo():
  localA = A()        # new instance of class A
  localA.b = myA.b    # externally-mutable
  return localA

newA = foo()
myA.b.x = 50
newA.b.x = 100
assert myA.b.x == newA.b.x   # both equal to 100
```

Figure 5.3: The function `foo` returns an externally-mutable value, so it cannot safely be memoized.

Externally-mutable return value: It is not safe to memoize a function call if its return value contains any mutable component that is referenced (*aliased*) by an object that existed before that call. We call these values *externally-mutable* since an object from another part of the program can mutate them. Figure 5.3 demonstrates this scenario, which we have never personally observed but is nonetheless legal Python code: After `foo` executes and returns `localA` to its caller, `myA.b` and `newA.b` refer to the same object, so the `assert` always succeeds. However, if IncPy were to memoize `foo`, skip its calls, and replace its return value with a *copy* of `localA` retrieved and deserialized from the on-disk cache, then `myA.b` and `newA.b` now refer to different objects in memory. Therefore, the `assert` now fails because `myA.b.x` is 50 and `newA.b.x` is 100. To preserve correctness, IncPy marks `foo` as impure and does not memoize its calls.

At function exit time, IncPy determines whether its return value contains an externally-mutable component by traversing inside of it and looking for mutable objects. We conservatively assume that any object that is not an immutable built-in Python type (e.g., `int`, `float`, `string`) is mutable. If one of these mutable objects has a *reference count* greater than 1 (i.e., more than one other object is referencing it), then we conservatively mark the return value as externally-mutable.

Which Calls Are Worthwhile To Memoize?

It is only worthwhile to memoize a function call if loading and deserializing the cached results from disk is faster than re-executing the function. A simple heuristic that works well in practice is to have IncPy track how long each function call takes and only memoize calls that run for more than 1 second. The vast majority of calls (especially in library code) run for far less than 1 second, so it is faster to re-execute them rather than to save and load their results from disk.

There are pathological cases where this heuristic fails. For example, if a function runs for 10 seconds but returns a 1 GB data structure, then it might take more than 10 seconds to load and deserialize the 1 GB data structure from the on-disk cache. Our experiments show that it takes 20 seconds to load and deserialize a 1 GB Python list from disk (Section 5.5.1). If IncPy memoized the results of that function, then skipping future calls would actually be slower than re-executing (20 seconds vs. 10 seconds). We use a second heuristic to handle these pathological cases: IncPy tracks the time it takes to serialize and save a cache entry to disk, and if that is longer than the function's original running time, then it issues a warning to the programmer and does not memoize future calls to that function (unless its code changes). Our experiments in Section 5.5.1 indicate that it always takes more time to save a cache entry than to load it, so the cache save time is a conservative approximation for cache load time.

5.4.4 Dynamic Reachability Detection

When a program is about to read from or write to some Python object in memory, how does IncPy determine whether that object is reachable from a global variable or a function's argument? IncPy needs this information to determine when to add a global variable dependency (Table 5.1) and when to mark function calls as impure.

The Python interpreter represents every run-time value in memory as an object, so IncPy augments every object with two fields: the name of a global variable that reaches this object (`globalName`), and the starting "time" of the outermost function call on the stack whose arguments reach this object (`funcStart`). IncPy measures "time" by the number of function calls that the interpreter has executed thus far. These fields are null for objects that are not reachable from a global or a function argument.

When the program loads a global variable, IncPy sets the `globalName` field of its value to the variable's name. When the program calls a function, IncPy sets the `funcStart` field of all its argument values to the current "time" (number of executed function calls), only for values whose `funcStart` has not already been set by another function currently on the stack. When the program executes an object field access (e.g., `my_obj.field`) or element access (e.g., `my_list[5]`), IncPy copies the `globalName` and `funcStart` fields from the parent to the child object. For example, if a program executes a read of `foo.bar[5]` where `foo` is a global variable, then the objects referred to by `foo`, `foo.bar`, and `foo.bar[5]` would all have the name "`foo`" in their `globalName` fields. When the program is about to read from or write to an object, IncPy can do a fast lookup of its `globalName` and `funcStart` fields to determine whether it is reachable from a global variable or a function argument, respectively.

The `funcStart` field enables IncPy to efficiently determine which functions to mark as impure when an argument is mutated. For example, if a function `foo` accepts an argument `x` and passes it into `bar`, then `x` is an argument of both `foo` and `bar`. Assume that the call to `foo` started at time 5 and `bar` started at time 6. The `funcStart` field of `x` is 5, since that is the start time of `foo`, the outermost function call where `x` is an argument. If code within `bar` mutates any component of `x`, then IncPy sees that its `funcStart` field, 5, is less than or equal to the start time of both `foo` and `bar`, so it marks both functions as impure.

Implementation: We initially implemented reachability detection by directly adding two extra fields to the Python object datatype: `globalName` and `funcStart`. This worked fine in development, but when we started getting users, they complained that IncPy did not work with 3rd-party Python extension modules already installed on their machines. Extension modules consist of compiled C/C++ code that rely on the Python object datatype to be of a certain size. Since IncPy augmented that datatype with two extra fields, extension module code no longer worked. To use IncPy with their extensions, users would need to re-compile extension code with the IncPy headers, which can be difficult due to compile-time dependencies.

To make IncPy work with users' already-installed extensions, we re-implemented using a shadow memory approach [123]. We left the Python object datatype unchanged and instead maintain `globalName` and `funcStart` fields for each object in a sparse table. To do a table look-up, IncPy breaks up an object's memory address into 16-bit chunks and uses them to index into a multi-level table similar to an OS page table. We use a two-level table for 32-bit architectures and a four-level table for 64-bit. For example, to look up an object at address `0xdeadbeef`, IncPy first looks up index `0xdead` in the first-level table. If that entry exists, it is a pointer to a second-level table, so IncPy looks up index `0xbeef` in that second-level table. If that entry exists, then it holds the `globalName` and `funcStart` fields for our target object. This mapping works because Python objects never change memory locations. When an object is deallocated, IncPy clears its corresponding table entry. IncPy conserves memory by lazily allocating tables. However, memory usage is still greater than if we had inlined the fields within Python objects, but that is the trade-off we made to achieve binary compatibility with already-installed extension modules.

5.4.5 Supporting File-Based Workflows

Figure 5.4 shows a script that implements a two-stage data analysis workflow. The programmer wrote extra code to save the results of `stage1` to an intermediate data file `stage1.out`, so that when `stage2` is edited, the code for `stage1` does not have to re-run. However, if the programmer changes the code for `stage1`, then it must be re-run to generate a new `stage1.out`, or else the input to `stage2` will be incorrect.

By using IncPy, the programmer can simplify that script into the one shown in Figure 5.5. There is no more need to write code to save and load data files, or to manually manage their dependencies. The on-disk cache entries that IncPy creates after it memoizes `stage1` and `stage2` are the substitutes for the `stage1.out` and `stage2.out` files, respectively.

```
def stage1():
  infile = open('input.dat', 'r')
  ... # parse and process infile
  outfile = open('stage1.out', 'w')
  outfile.write( ... ) # write output to file
  outfile.close()

def stage2():
  infile = open('stage1.out', 'r')
  ... # parse and process infile
  outfile = open('stage2.out', 'w')
  outfile.write( ... ) # write output to file
  outfile.close()

# top-level script code:
stage1()
stage2()
```

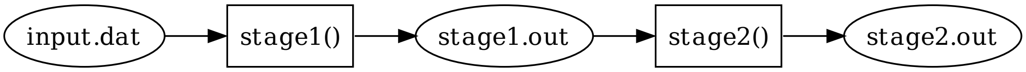


Figure 5.4: Example Python script that implements a file-based workflow, and accompanying dataflow graph where boxes are code and circles are data files.

```
def stage1():
  infile = open('input.dat', 'r')
  out = ... # parse and process infile
  return out

def stage2(dat):
  out = ... # process dat argument
  return out

# top-level script code:
stage1_out = stage1()
stage2_out = stage2(stage1_out)
```

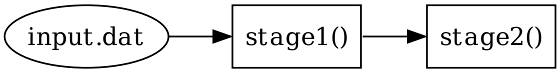


Figure 5.5: The Python script of Figure 5.4 refactored to take advantage of IncPy's automatic memoization. Data flows directly between the two stages without an intermediate data file.

Despite the benefits of a pure-Python workflow (Figure 5.5), some of our users still choose to create intermediate data files (Figure 5.4) for performance reasons. IncPy serializes entire Python data structures, which is convenient but can be slow, especially for data larger than a few gigabytes. Scripts often run faster when accessing data files in a format ranging from a specialized binary format to a database.

IncPy supports these file-based workflows by recording file read and write dependencies. After it executes the script in Figure 5.4, `stage1` will have a read dependency on `input.dat` and a write dependency on `stage1.out`; `stage2` will have a read dependency on `stage1.out` and a write dependency on `stage2.out`. IncPy can use this dependency graph to skip calls and issue warnings. For example, if only `stage2` is edited, IncPy can skip the call to `stage1`. If `stage1.out` is updated but `stage1` did not change, then IncPy will issue a warning that some external entity modified `stage1.out`, which could indicate a mistake. In contrast, the regular Python interpreter has none of these features.

5.5 Evaluation

Our evaluation addresses three main questions:

- What are the performance impacts of dynamic dependency tracking and automatic memoization (Section 5.5.1)?
- How can IncPy speed up iteration times on real data analysis tasks without requiring programmers to write caching code (Section 5.5.2)?
- How have users found IncPy useful when doing data analysis (Section 5.5.3)?

We ran all experiments on a Mac Pro with four 3GHz CPUs and 4 GB of RAM, with Python 2.6.3 and IncPy both compiled as 32-bit binaries for Mac OS X 10.6 using default optimization flags. For faster performance, IncPy did not track dependencies within Python standard library code, because we assume users do not ever change that code.

5.5.1 Performance Evaluation

Running a script with IncPy when the cache is empty will be slower than running with the regular Python interpreter for two reasons: First, IncPy needs to dynamically track dependencies, global reachability, and function impurity to determine when it is safe and worthwhile to memoize. Second, IncPy must save and later load memoized data from the on-disk cache.

Overall Slowdown

To quantify IncPy's overall slowdown on typical data analysis scripts, we compared running times and memory usage when executing six scripts with regular Python and IncPy. We obtained the following scripts from researchers who had written them to analyze data for research that they intended to publish in peer-reviewed papers:

- **linux** — A script we wrote in 2007–2008 to mine data about the Linux kernel project's revision control history for an empirical software engineering paper [72]. We present a case study of this script in Section 5.5.2.

- **tags** — A set of information retrieval scripts for a paper contrasting crowd-sourced tags with expert annotations for keywords describing books [89]. It consists of three stages, named *tags-1*, *tags-2*, and *tags-3*, respectively. We present a case study in Section 5.5.2.
- **vr-log** — A set of scripts that process event logs from a virtual world for a distributed systems paper [45]. We present a case study in Section 5.5.2.
- **sys-log** — A script that processes a 2.5 GB supercomputer error log for an anomaly detection paper [126].
- **mmouse** — A script that post-processes and graphs synchronized mouse input events for an HCI paper [87].
- **biology** — A bioinformatics script that uses a hidden Markov model to analyze human genome data.

Table 5.2 and Figure 5.6 show IncPy's run-time and memory overheads. The left half of Figure 5.6 shows a mean slowdown of 16% relative to regular Python on the data analysis script benchmarks, most of which is due to IncPy dynamically tracking dependencies, global reachability, and impurity. IncPy only memoizes the few long-running function calls corresponding to the data analysis stages in each script, and memoization only takes a few seconds (see Section 5.5.1). Table 5.2 shows that total script running times range from 1 minute to 8 hours. Memory overheads range from negligible to 2X, mainly due to maintaining object metadata (see Section 5.4.4).

We could not find data analysis scripts larger than ∼1000 lines of code, so to demonstrate IncPy's scalability, we used it to run the test suite of the Django project. Django is a popular framework for building web applications and is one of the largest Python projects, with 59,111 lines of Python code [16]. To estimate worst-case behavior, we measured IncPy's run-time slowdown on the Django test suite. The right half of Figure 5.6 shows our results. All 151 tests passed when running with IncPy. The mean running time for an individual test case was 1.1 seconds (median was 0.57 sec). The mean slowdown relative to regular Python was 88% (maximum slowdown

Script	# lines	Running time		Peak RAM (MB)	
name	of code	Python	IncPy	Python	IncPy
mmouse	230	0:51	1:00	50	101
tags-3	1200	2:29	3:09	371	374
tags-1	1200	3:00	3:17	440	444
linux	200	4:58	5:10	8.6	15
tags-2	1200	6:07	7:57	486	488
vr-log	700	24:42	28:30	323	694
sys-log	200	33:13	37:56	67	73
biology	250	8:11:27	8:54:46	1884	1966

Table 5.2: Running times and peak memory usage of data analysis scripts executed with Python and IncPy, averaged over 5 runs (variances negligible). Figure 5.6 shows run-time slowdown percentages.

was 118%). These short runs elicit IncPy's worst-case behavior because IncPy has a longer start-up time than the regular Python interpreter; this is because IncPy must set up data structures for dependency tracking. In reality, it would be impractical to use IncPy on short-running applications such as Django, since its start-up time nullifies any potential speed-ups. (Surprisingly, on two of the test cases, IncPy's memoization led to minor speed-ups!) However, for its intended use on long-running data analysis scripts, IncPy has a reasonable slowdown of ∼16%.

Automatic Memoization Running Times

To measure cache save and load times in isolation, we created a microbenchmark consisting of one Python function that allocates a list of N random integers and returns that list to its caller. We annotated that function to force IncPy to memoize it; otherwise it would not be memoized since it is both non-deterministic and short-running. We ran the function once and measured the time it takes for IncPy to save its memoized results to disk. Then we ran it again and measured the time it takes for IncPy to load the matching cache entry from disk and skip the original call.

Figure 5.7 shows cache save and load times for lists ranging from 1 to 100 million elements. Running times are instantaneous for lists of less than 100,000 elements

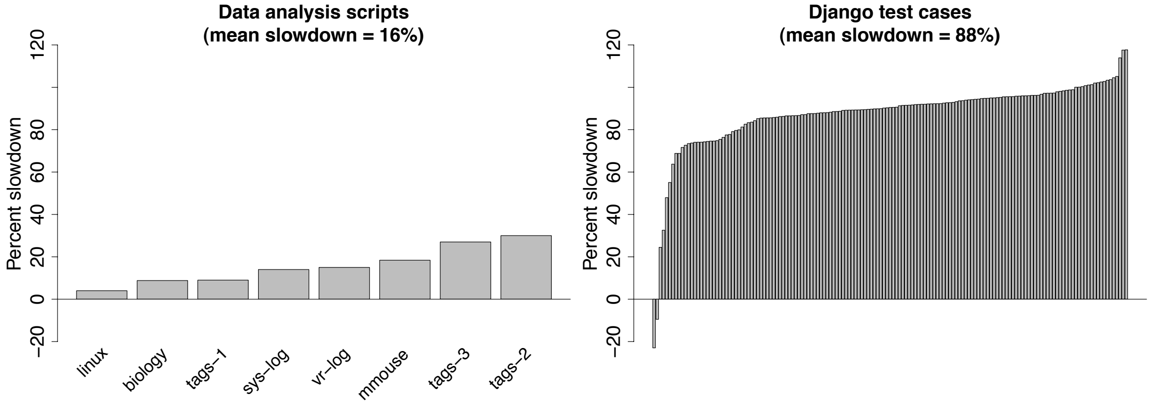


Figure 5.6: Percent slowdowns when running with IncPy in the typical use case (data analysis scripts on left) and estimated worst case (151 Django test cases on right), relative to running with regular Python.

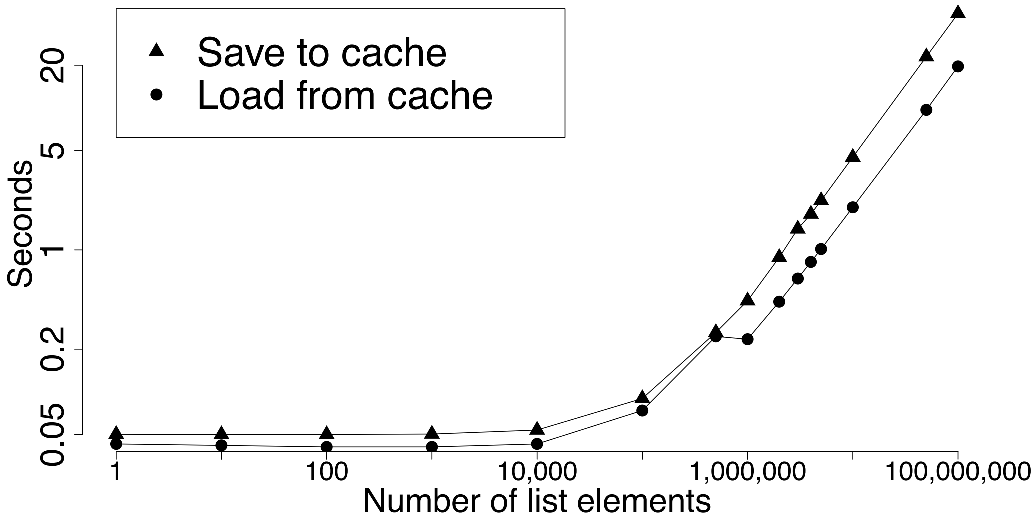


Figure 5.7: Number of seconds it takes for IncPy to save/load a Python list of N integers to/from the persistent cache.

and then scale up linearly. At the upper extreme, a list of 100 million integers is 1 gigabyte in size and takes 20 seconds to load from the cache. Cache load times are end-to-end, taking into account the time to start up IncPy, find a matching on-disk cache entry, load it from disk, deserialize it into a Python object, and finally skip the original function call.

In sum, memoization is worthwhile as long as cache load time is faster than re-execution time, so Figure 5.7 shows that IncPy can almost always speed up functions that originally take longer than ∼20 seconds to run.

5.5.2 Case Studies on Data Analysis Scripts

To demonstrate how IncPy can provide the benefits of less code, automated data management, and faster iteration times when writing data analysis scripts, we present brief case studies of three scripts from Table 5.2: **linux**, **tags**, and **vr-log**. IncPy provides similar benefits for the other three scripts used in our experiments.

Speeding Up Exploration of Code Variants

To show how IncPy enables faster iteration when exploring alternative hypotheses in a data analysis task, we studied a Python script we wrote in 2007–2008 to mine Linux data for a paper [72]. Our script computes the chances that an arbitrary file in the Linux kernel gets modified in a given week, by querying an SQLite database containing data from the Linux project's revision control history. It retrieves the sets of files present in the Linux code base ("alive") and modified during a given week, using the `get_all_alive_files(week)` and `get_all_modified_files(week)` functions, respectively, and computes probabilities based on those sets. Both functions perform an SQL query followed by post-processing in Python (this mix of declarative SQL and imperative code is common in data analysis scripts). Each call takes 1 to 4 seconds, depending on the queried week.

Our script computes probabilities for 100 weeks and aggregates the results. It runs for 4 minutes, 58 seconds (4:58) with regular Python and 5:10 with IncPy (4% slower). IncPy memoizes all 100 calls to those two functions and records a file dependency on the SQLite database file.

To see how we edited this script throughout its development process, we checked out and inspected all historical versions from its revision control repository. The first version only did the original computation, but as we delved deeper into our research questions, we augmented subsequent versions to do related computations. The final version of our script (dated Feb. 2008) contains code for all 6 variants; a command-line flag controls which gets run. Here are the variants and their running times with regular Python:

1. Original computation: Chances that a Linux source code file gets modified in a given week (4:58)

2. Chances that a Linux file gets modified, conditioned on the modification type (e.g., bug fix, refactoring) (5:25)

3. Chances that a Linux file gets modified, conditioned on the time of day (e.g., morning vs. evening) (5:25)

4. Chances that a Linux file gets modified by the person who has modified it the most in the past (5:15)

5. Chances that a Linux file gets modified by someone above 90^{th} percentile for num. edits to any file (5:25)

6. Chances that a Linux file gets modified by a person with a `.com` email address (5:25)

After taking 5:10 to make an initial run of the original computation with IncPy, running *any variant* only takes 30 seconds, a **10X speed-up** over regular Python. This speed-up occurs because each variant calls the same 2 functions:

```
alive_set = get_all_alive_files(week)
mod_set   = get_all_modified_files(week)
# <do fast-running computations on these sets>
```

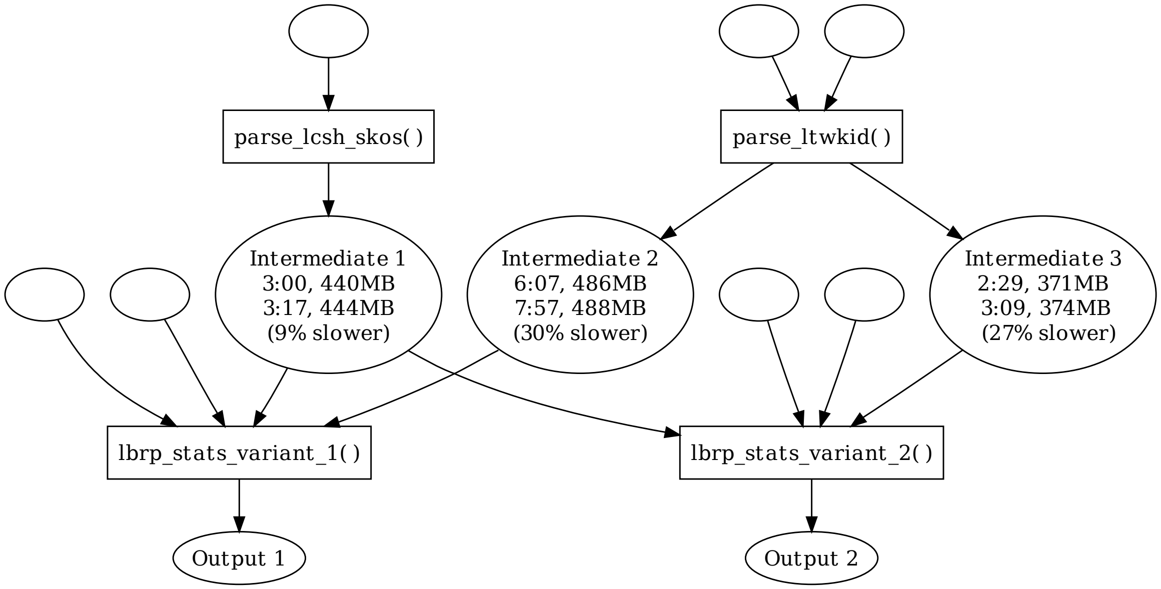


Figure 5.8: Datasets (circles) and Python functions (boxes) from Paul's information retrieval scripts [89]. In each "Intermediate" circle, the 1st row is run time and memory usage for generating that data with Python, and the 2nd row for IncPy.

The computations that differ between variants take much less time to run than the 2 functions that they share in common; IncPy memoizes those calls, so it can provide a 10X speed-up regardless of which variant runs. This idiom of slow-running (but rarely-changing) code followed by fast-running (but frequently-changing) code exemplifies how data analysts explore variations when prototyping scripts. IncPy automatically caches the results of the slow-running functions, so that subsequent runs of any variants of the fast-running parts can be much faster.

Removing Existing Ad-Hoc Caching Code

To show how we can remove manually-written caching code and maintain comparable performance, we studied information retrieval scripts written by our colleague Paul [89]. Figure 5.8 shows his research workflow, which consists of 4 Python functions (boxes) that process data from 7 input datasets (empty circles). His functions are arranged in 2 script variants:

```
# Script variant 1:
x = parse_lcsh_skos()
y1 = parse_ltwkid('dataset1')
print lbrp_stats_variant_1(x, y1)

# Script variant 2:
x = parse_lcsh_skos()
y2 = parse_ltwkid('dataset2')
print lbrp_stats_variant_2(x, y2)
```

Paul frequently edited the final function in each script: `lbrp_stats_variant_1` and `lbrp_stats_variant_2`. Since he rarely edited `parse_lcsh_skos` and `parse_ltwkid`, he grew tired of waiting for them to re-run on each execution and simply produce the same results, so he wrote extra code to save their intermediate results to disk and skip subsequent calls. Doing so cluttered up his script with code to serialize and deserialize results but sped up his iteration times.

We refactored Paul's script to remove his caching code and ran it with IncPy, which performed the same caching automatically. The numbers in the "Intermediate" circles in Figure 5.8 compare performance of running the original functions to generate that data with Python versus the refactored versions with IncPy (they correspond to the tags-1, tags-2, and tags-3 rows in Table 5.2). Both versions took less than a second to load cached results on a subsequent run.

Besides eliminating the need to write boilerplate caching code, IncPy also automatically tracks dependencies between Paul's code and his 7 input datasets (empty circles in Figure 5.8). This can prevent errors such as manually updating a dataset and forgetting to re-run the dependent functions.

From File-Based to Pure-Python Workflows

To show how IncPy can benefit existing file-based workflows (Section 5.4.5) and provide an easy transition to a pure-Python workflow, we studied Python code written

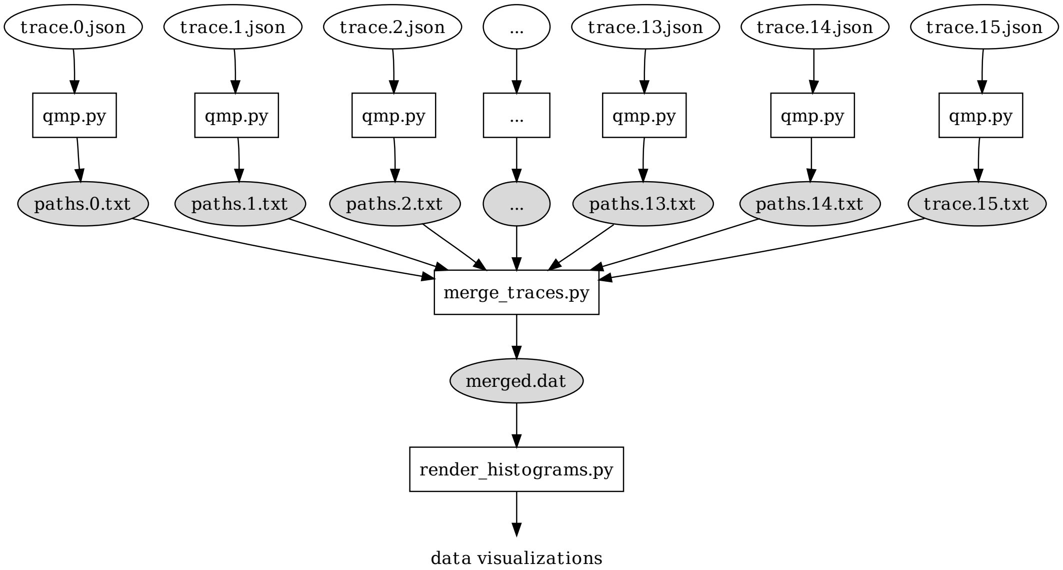


Figure 5.9: Python scripts (boxes) and data files (circles) from Ewen's event log analysis workflow [45]. Gray circles are intermediate data files, which are eliminated by the refactoring shown in Figure 5.10.

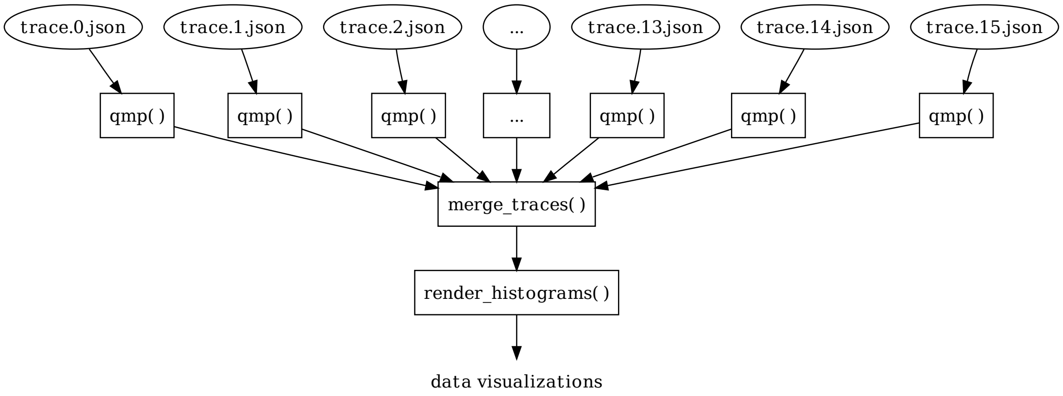


Figure 5.10: Refactored version of Ewen's event log analysis workflow [45], containing only functions (boxes) and input data files (circles)

by our colleague Ewen to process event logs from a virtual world system [45]. Figure 5.9 shows his 3 Python scripts as boxes: The `qmp.py` script takes two filenames as command-line arguments, reads data from the first, processes it, and writes output to the second. The `merge_traces.py` script first invokes `qmp.py` 16 times (once for each input `trace.X.json` file), then aggregates all resulting `paths.X.txt` files into a `merged.dat` file. Finally, `render_histograms.py` creates histograms and other data visualizations from the contents of `merged.dat`.

Ewen's top-level workflow calls 3 Python scripts and creates 17 intermediate data files (16 `paths.X.txt` and `merged.dat`). The main reason why he broke up his analysis into several scripts was so that he could tweak and debug each one individually without having to re-run everything. He frequently modified the visualization code in `render_histograms.py` and occasionally the aggregator in `merge_traces.py`.

IncPy can benefit Ewen's current scripts by dynamically recording all code and file read/write dependencies to create the dependency graph of Figure 5.9. IncPy is 15% slower than regular Python on the initial empty-cache run (28:30 vs. 24:42). But IncPy's caching allows some subsequent runs to be faster, because only script invocations whose dependent code or data have changed need to be re-run:

- If no input `trace.X.json` files change (or some are deleted), then none need to be re-processed by `qmp.py`
- If new `trace.X.json` files are added, then only those files need to be processed by `qmp.py`
- If an individual `trace.X.json` file changes, then only that file needs to be re-processed by `qmp.py`

Even though IncPy provides these speed-ups, Ewen's code is still cluttered with boilerplate to save and load intermediate files. For example, each line in `paths.X.txt` is a record with 6 fields (4 floats, 1 integer, 1 string) separated by a mix of colons and spaces, for human-readability. `qmp.py` contains code to serialize Python data structures into this ad-hoc textual format, and `merge_traces.py` contains code to deserialize each line back into Python data.

It took us less than an hour to refactor Ewen's code into the workflow of Figure 5.10, where each script is now a function that passes Python data into the next function via its return value *without explicitly creating any intermediate data files.* IncPy still provides the same speed-up benefits as it did for Ewen's original scripts, but now the code is much simpler, only expressing the intended computation without boilerplate serialization/deserialization code. For an initial empty-cache run, IncPy is 23% slower than regular Python (29:36 vs. 24:02). A subsequent run takes 0.6 seconds if no dependencies change.

Ewen's workflow is a smaller version of my workflow shown in Figure 5.1. We could have refactored that workflow in the same way, but we no longer have access to its code or data sets since that work was done within Microsoft Research.

5.5.3 Real-World User Experiences

We now describe the experiences of some research programmers who found IncPy on our website and used it to run their data analyses. Although these anecdotes lack the rigor of a user study, they show IncPy's real-world applicability because they come from people whom we have never met and who voluntarily chose to use IncPy because it fulfilled a real need.

Eric J. is a neuroscience Ph.D. student at MIT whose thesis involves creating stochastic hardware and software systems for accelerated probabilistic inference. His Python scripts would run for several hours generating hundreds of VHDL (hardware description) source files, calling external tools to process those files and create output logs, and then parsing those logs to create data visualizations. Using IncPy, he was able to refactor his scripts to pass data between stages as ordinary Python strings, thereby eliminating all intermediate VHDL and log files. He told us via email:

> IncPy has enabled me to do these runs without worrying about the coding overhead associated in serialization of intermediate results to disk, saving me considerable development time and effort. IncPy also makes it much easier for me to explore my results interactively, as the automatic memoization even works for ad-hoc data-analytic functions.

Eric R. is a masters student in media informatics at LMU Munich in Germany whose thesis involves using computer vision to automatically recognize how people grasp mobile devices. This email snippet shows how he uses IncPy:

> In order to recognize which part of the hand is touching which fingerprint sensor, I need to compare the taken samples to the full handscan. This is extremely slow on [sic] pixel level, so I need to extract certain features which is very slow for the full handscan. This is where IncPy comes into play, so I don't have to wait for half an hour (or so it feels) at the beginning of each test run.

Eric's workflow consists of several image processing stages implemented as Python functions. Without IncPy, he would need to save the outputs of each stage to temporary image files or else endure long wait times. IncPy allowed him to pass image data between stages as raw integer arrays, which are automatically memoized.

Ryan is a masters student in linguistics at the University of Tromsø in Norway who used IncPy to prototype natural language processing scripts for a machine translation project. When we apologized for the lack of detail in IncPy's documentation, he responded via email:

> But I think what's awesome about IncPy is you just can just switch to using it with no code changes and it just works with no pain ... Which is quite ideal for the way I've been using it. I guess for that reason maybe it's not a huge problem that your documentation is only in progress, you really don't need to know a whole lot to make it work. :)

Ryan's quote exemplifies the usability advantages of IncPy not requiring programmers to learn any new language features or external tools. He is a linguist who learned just enough about Python to work on his project, and he can transition to using IncPy without any prior training.

5.6 Discussion

5.6.1 Complementary Techniques

The goals of IncPy are complementary to those of JIT compilers and parallel computation, so all three techniques can be combined for greater speed-ups. Whereas JIT compilers excel at speeding up CPU-bound code, IncPy can also speed up I/O and network-bound code. In addition, eliminating redundant computations can be useful for speeding up parallel code running on a cluster. For instance, Cory, a bioinformatics graduate student we interviewed, lamented how a postdoc in his research group was constantly using up all the compute-power on the group's 124-CPU cluster. This postdoc's data-parallel Perl script consisted of several processing stages that took dozens of hours to run on each cluster node (on a slice of the input genomic data). He would often make tweaks to the second stage and re-run the entire script, thus needlessly re-computing all the results of the first stage. After a few weeks of grumbling, Cory finally inspected the postdoc's code and saw that he could refactor it to memoize intermediate results of the first stage, thus dramatically reducing its running time and freeing up the cluster for labmates. The postdoc, who was not an adept programmer, perhaps did not have the expertise or willingness to refactor his code; he was simply willing to wait overnight for results, to the chagrin of his labmates. With an interpreter such as IncPy installed on their cluster machines, there would be no need to perform this sort of manual code refactoring.

5.6.2 Limitations

IncPy's main limitation is that it cannot track impurity, dependencies, or non-determinism in non-Python code such as C/C++ extension modules or external executables. Similarly, IncPy does not handle non-determinism or remote dependencies arising from network accesses. Users can make annotations to manually specify impurity and dependencies in external code. Fortunately, most C functions in the Python standard library are pure and self-contained, but we have annotated a few dozen as impure (e.g., list `append` method) and non-deterministic (e.g., time, randomness).

Another limitation is that IncPy is not designed to track fine-grained changes in code or data. If even one line in a function or dataset changes, then IncPy deletes cache entries that depend on that code or data, respectively. Implementing finer-grained dependency tracking would increase IncPy's baseline run-time slowdown.

IncPy cannot memoize function calls whose cache entries contain objects that the Python standard library `cPickle` module is unable to serialize. A workaround is to create serializable proxy objects and then write code to convert to and from the proxies. IncPy automatically creates proxies for three common types: file objects, function objects, and SQLite database cursor objects.

Finally, IncPy's performance is limited by the Python standard library `cPickle` module and the default object equality checking mechanism: IncPy uses `cPickle` to perform serialization and object equality checks to find matching cache entries. We have not yet attempted to create faster versions of these checks.

5.6.3 Future Work

- **Intraprocedural analysis**: Currently, IncPy memoizes at the function level, but many scripts contain memoization opportunities within individual functions or are written as one monolithic top-level `main` function. For instance, a long-running loop that populates a list with values could be automatically refactored to its own function, so that its results can be memoized and re-used.

- **Database-aware caching**: Several researchers we interviewed stored their datasets in a relational database, used Python scripts to make simple SQL queries, and then performed sophisticated data transformations using Python code. They found it easier and more natural to express their algorithms in an imperative manner rather than in SQL. If IncPy were augmented to intercept Python's database API calls, then it could track finer-grained data dependencies between database entries and Python code.

- **More pure memoization**: The current version of IncPy invalidates a cache entry if any of its file or code dependencies have changed, so it only skips calls when the relevant parts of the environment are unchanged from the previous run. IncPy could easily be extended to keep a hash of file contents for dependencies rather than last modified times, so that it can skip calls when the environment matches a state that existed at *some* time in the past, not necessarily just after the previous run.

- **Network-aware caching**: Several CS researchers who extract large amounts of data from the Internet for their research told us that they found it annoying to have to re-run their extractor scripts after fixing bugs or making tweaks. If IncPy were augmented to intercept Python's networking API calls, then it could transparently cache data retrieved from the Internet on the researcher's machine, maintain the proper dependencies, and speed up subsequent runs by eliminating unnecessary network activity.

- **Provenance browsing**: Some researchers would like ways to interactively browse (or even manually edit) the saved intermediate results of their scripts. Allowing a user to directly view the data and its provenance could help them manually verify the correctness of intermediate results, debug more effectively, and predict how long it might take to re-execute portions of their workflow.

5.6.4 Opportunities for Generalizing

The technique underlying IncPy can be implemented in a straightforward manner for other interpreted languages commonly used for data analysis scripting, such as Perl, Ruby, R, or MATLAB. An implementation for these languages could interpose on the interpreter's handlers for function calls, run-time value accesses, and file I/O in the exact same way as IncPy does. Implementing for a compiled language (e.g., Java, C++) is less straightforward but still feasible: One could create a source-to-source translator or static bytecode rewriter that augments the target program with the appropriate interposition callbacks prior to execution. However, performance now becomes a concern. IncPy's run-time overhead is reasonable (mean of 16% on our benchmarks) because interpreters are already slow compared to executing compiled code. Static purity and escape analyses [139], and other compile-time optimizations, might be needed to achieve reasonable overheads on compiled code.

One could further generalize the ideas in this chapter by implementing an IncPy-like tool that works across programs written in multiple languages. Some data analysis workflows consist of separate programs that coordinate with one another using intermediate data files [98, 124]. A tool could use system call tracing (e.g., `ptrace` on Linux) to dynamically discover dependencies between executed programs and the data files that they read/write, in order to eliminate unnecessary program executions.

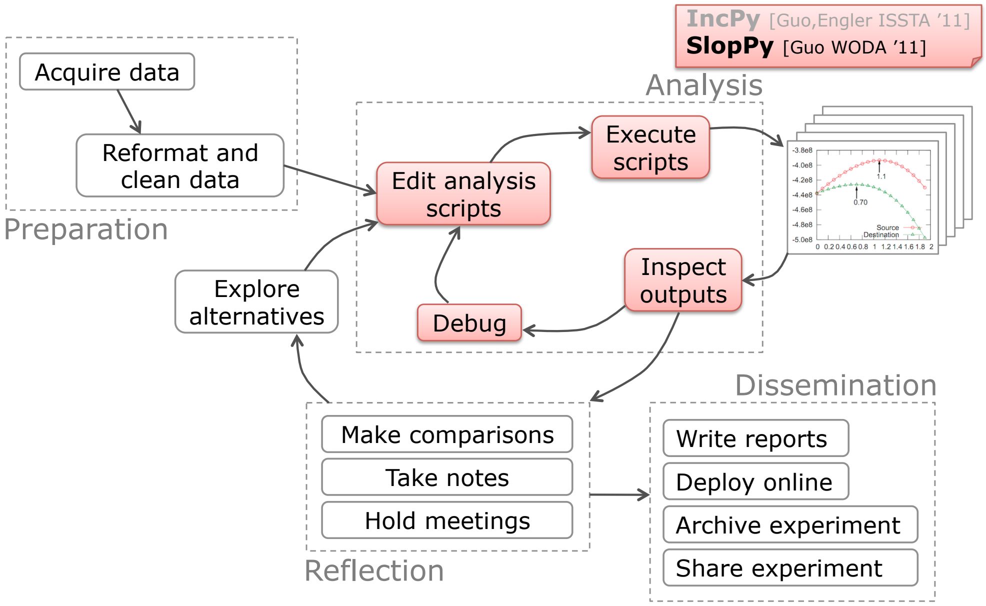
IncPy can also be useful in domains beyond data analysis. For instance, it could speed up regression testing by ensuring that after the programmer makes a code edit, only the regression tests affected by that edit are re-run. This might make it feasible for tests to run continuously during development, which has been shown to reduce wasted time and allow bugs to be caught faster [137]. Continuous testing is most effective for short-running unit tests tightly coupled with individual modules (e.g., one test per class). However, many projects only contain system-wide regression tests, each covering code in multiple modules. Manually determining which tests need to re-run after a code edit is difficult, and waiting for several minutes for *all* tests to re-run precludes interactive feedback. Instead, IncPy could use function-level dependency tracking and memoization to automatically incrementalize regression testing.

5.6.5 Conclusion

In this chapter, we have presented a practical technique that integrates automatic memoization and persistent dependency management into a general-purpose programming language. We implemented our technique as an open-source Python interpreter named IncPy [9]. IncPy works transparently on regular Python scripts with a modest run-time slowdown, which improves both usability and reliability: IncPy is easy to use, especially by novice programmers, since it does not require programmers to insert any annotations or to learn new language features or domain-specific languages. IncPy improves reliability by eliminating human errors in determining what is safe to cache, writing the caching code, and managing code and dataset dependencies. In sum, IncPy allows programmers across a wide range of disciplines to iterate faster on their data analysis scripts while writing less code.

Chapter 6

SlopPy: Make Analysis Scripts Error-Tolerant



This chapter presents an enhanced Python interpreter called SlopPy, which automatically makes existing scripts error-tolerant, thereby speeding up the data analysis scripting cycle. Its contents were adapted from a 2011 workshop paper [71].

6.1 Motivation

Research programmers often write scripts to parse, transform, process, analyze, and extract insights from data. In this chapter, I refer to these as *ad-hoc data processing scripts* because both the scripts and the data they operate on are ad-hoc in nature:

- **Ad-hoc scripts**: Programmers write these scripts in a "quick-and-dirty" manner to explore their datasets and to formulate, test, and refine hypotheses. They do not spend much time making these scripts robust, error-tolerant, or modular, since their primary goal is to discover insights from their data, not to produce a well-engineered piece of production software.
- **Ad-hoc data**: Vast amounts of real-world data are stored in ad-hoc formats [59]. Examples of such semi-structured and unstructured data are those scraped from web pages, mined from emails, and logs produced by computer systems and scientific lab equipment. Ad-hoc data often contains inconsistencies and errors due to lack of well-defined schemas, malfunctioning equipment corrupting logs, non-standard values to indicate missing data, or human errors in data entry [59].

Although domain-specific data processing languages are being developed in research labs [59, 128, 132], most modern programmers write ad-hoc data processing scripts in general-purpose languages such as Python, Perl, and Ruby due to their flexibility, support for rapid prototyping, and active ecosystems of open-source libraries.

I have heard my colleagues gripe about the following problem regarding the brittleness of their scripts:

1. They start executing a script that is expected to take a long time to run (e.g., tens of minutes to a few hours).
2. They work on another task or go home for the evening.
3. When they return to their computer, they see that their script crashed upon encountering the first uncaught exception and thus produced no useful output.

The crash could have occurred due to a bug in either the script or the data: Since programmers write scripts in a “quick-and-dirty” manner without carefully handling edge cases, many bugs manifest as uncaught run-time exceptions. Also, ad-hoc data sources often contain badly-formatted “unclean” records that cause an otherwise-correct script to fail [59]. Script executions are mostly “all-or-nothing” events: Only a bug-free script will successfully terminate with a result; any imperfection leads to a crash that only indicates the first error, likely masking additional errors.

Regardless of cause, the programmer gets frustrated because he/she has waited for a long time and still cannot see any results. The best he/she can do now is to try to fix that one single visible error and re-execute the script. It might take another few minutes or hours of waiting before the script gets past the point where it originally crashed, and then it will likely crash again with another uncaught exception. The programmer might have to repeat this debugging and re-executing process several times before the script successfully finishes running and actually produces results.

For another scenario that exhibits this problem, consider an online real-time data analytics script (e.g., deployed on a server) that continuously listens for input and incrementally processes incoming data. Unless the programmer meticulously handled all edge cases, that script will inevitably encounter some data it cannot process and crash with an uncaught exception. Discovering that the script crashed, debugging and fixing the error, and re-deploying it might lead to missing a few hours’ or days’ worth of incoming data, all just because of one unexpected crash-causing input.

6.2 Contributions

To address this problem of script brittleness, I developed a dynamic program analysis technique that automatically adds error tolerance to existing scripts without requiring any programmer effort. It works as follows:

1. The script executes in a custom interpreter.

2. When an expression throws an uncaught exception, the interpreter creates an NA (**N**ot **A**vailable) object representing the exception, assigns it to the expression's target, saves the NA object and local variable values to a log file, and continues executing normally.

3. Whenever an NA object appears in an expression, the interpreter handles it according to special rules and logs a warning. For example, all unary and binary operations involving an NA object will return NA.

My technique guarantees that scripts will never crash. As soon as a script throws an exception when processing a record in the dataset, the interpreter marks values derived from that record as NA and continues processing subsequent records. Since scripts usually process each record independently, NA objects do not propagate to taint values derived from neighboring records (i.e., they are confined to the error-inducing records). When the script finishes running, it outputs results for the successfully-processed records, which often provides more information than prematurely crashing.

The interpreter also maintains a log of what went wrong when processing dataset records that caused the script to throw exceptions. This log can be used to debug errors and can even be processed to incrementally recover from errors without re-executing the script on the entire dataset. I demonstrate an incremental error recovery example in Section 6.5.3.

Main benefits: My technique allows programmers to iterate faster when developing ad-hoc scripts in three ways:

- **Faster coding**: Programmers can write "quick-and-dirty" scripts, execute them, and see partial results (and all errors) without being slowed down by the burden of worrying about full correctness. They also do not need to write extra code to handle exceptions that could arise in *almost every line of code* (e.g., every array operation could throw an out-of-bounds exception). A study found that 50–70% of the code in reliable systems software was for handling edge cases [68]; although that study was not on data processing scripts, it shows that programmers spend lots of effort on error handling in general. My technique acts as a safety net to guard against unexpected edge cases without any programmer effort. As their scripts mature, programmers can write specialized error handling (or even recovery) code to wean themselves off of this safety net.

- **Fewer edit/run/debug iterations**: After every script execution, programmers can see a log of all run-time errors and detailed information about their context. Thus, they can fix multiple bugs in one round of code edits rather than fixing one at a time and waiting for minutes or hours before seeing whether there are additional errors. As an analogy, if a compiler only displayed the first syntax error when compiling code, then the programmer can only fix one error at a time and must re-compile to see the next error. Fortunately, modern compilers display as many useful errors and warnings as possible per invocation.

- **More insights per iteration**: When a script has bugs, partial results always provide more insights than no results, thus helping to inform code edits for the next iteration. In a mailing list discussion about manually adding error tolerance to a Python bioinformatics library, a researcher echoed these sentiments:

> I work with millions of [genetic] sequences at a time, and if 5,000 or 50,000 are badly formatted (or problematic due to a parser bug), I would rather make a note of it and move on, coming back later to fix the problem. The alternative would be — well, and [sic] ugly hack, which will cause loss of time and research momentum. [63]

In addition, partial results can actually be accurate when computing aggregate statistics: For instance, if the true mean housing price in a large dataset is \$200k, a script might return a mean of \$198k if it can only process 95% of the data (the other 5% might require more detailed parsing logic). They can even sometimes provide exact answers: For instance, Condie et al. found the top 10 most frequently occurring words in a 5.5GB corpus of Wikipedia article text after only processing about half of the dataset [47].

6.3 Technique

My technique involves altering run-time exception handling by creating special NA objects rather than crashing. I find it easiest to explain and implement in the context of a programming language interpreter (e.g., for Python), but it can also be implemented in a compiler or bytecode transformation tool.

6.3.1 Creating NA from Uncaught Exceptions

When a regular interpreter executes a script, as soon as an expression throws an exception, the interpreter inspects all functions on the stack to try to find an exception handler (i.e., code in a try-catch block). If it can find a handler, then it transfers control to that handler and keeps executing. If it cannot find a handler, then it will crash the script and print an error message. For example, executing `x = 1 / 0` will crash the script with a divide-by-zero exception.

My technique alters the interpreter so that, instead of crashing the script on an uncaught[1] exception, it creates a special NA (Not Available) object that becomes the result of the expression that threw the exception. An NA object has three fields:

- **Exception type** (e.g., array out-of-bounds)
- **Exception details** (e.g., *"index 25 is out of bounds of the array named x"*)
- **Stack backtrace** indicating all currently-executing function names and line numbers

When my modified interpreter executes `x = 1 / 0`, it creates a new NA object to represent the exception, assigns it to the variable `x`, and continues normal execution.

6.3.2 Logging Exception Context for Debugging and Incremental Error Recovery

When the interpreter creates a new NA object, it saves a snapshot of that object and the current values of all local variables to a log file. For example, consider this Python

[1] If the script provides an exception handler, then the handler executes as usual.

script, which iterates through each line in big_data.txt, splits it into tokens based on whitespace, converts the first two tokens into numbers, and prints their ratio:

```
line_number = 1
for line in open('big_data.txt'):
  tokens = line.split(' ')
  ratio = float(tokens[0]) / float(tokens[1])
  print ratio
  line_number += 1
```

Although this script is tiny, it can throw three possible exceptions when processing big_data.txt:

1. If a line contains fewer than two tokens, then accessing tokens[1] will throw an out-of-bounds exception.

2. If a token does not represent a floating-point number, then converting it to a float will throw an exception.

3. If float(tokens[1]) is 0, then there is a divide-by-zero exception.

If big_data.txt has 1 million lines, then when the script finishes processing it, the warning log file might look like:

```
Divide-by-zero exception: 5 / 0
    line_number=291818, line='5 0', tokens=['5', '0']
    < ... full stack backtrace ... >
Invalid float exception: 'six' is not a valid float
    line_number=320183, line='5 six', tokens=['5', 'six']
    < ... full stack backtrace ... >
Out-of-bounds exception: tokens[1] is out of bounds
    line_number=983012, line='2', tokens=['2']
    < ... full stack backtrace ... >
```

The above example log file indicates that the script failed to process three records in big_data.txt. The exception object, stack backtrace, and values of local variables pinpoint the exact cause of each failure. The programmer now knows exactly where to insert error handling or recovery code to avoid triggering those exceptions in future runs. Alternatively, he/she could edit the dataset to alter or remove those problematic records (located at `line_number`). In contrast, a regular Python interpreter would crash this script at the first exception (when processing line 291818), so the programmer would not see the other two exceptions until he/she edited either the script or dataset and re-executed, which could take anywhere from minutes to hours.

Finally, in some cases, the programmer can write an amended version of the script to *process the log file itself* and merge the results with those from the original computation. Since the log file contains partially-processed records stored in local variables, an amended script can directly process that file (a trick colloquially known as "sloppy seconds"), which can be much faster than re-executing on the entire original dataset. In the above example, an amended script would only need to process the three error-inducing records in the log file rather than the 1 million in the original dataset. In Section 6.5.3, I demonstrate this form of incremental error recovery.

6.3.3 Treatment of NA Objects

The interpreter treats NA objects in the following special ways throughout execution:

- A unary operation on an NA object (e.g., negation) returns itself (the original NA object).

- A binary operation involving an NA object (e.g., addition, subtraction) returns itself[2].

- A comparison involving an NA object (e.g., less than, equal to, greater than) returns itself[2], since the result is unknown (neither TRUE nor FALSE).

- Calling an NA object as a function returns itself.

- Accessing an element of an NA object (e.g., via array indexing) returns itself.

- Accessing a named field of an NA object returns itself.

[2] If the operation involves two NA objects, then it returns the first one.

- NA `and` False (conjunction) returns False.
- NA `or` True (disjunction) returns True.
- Using an NA object as an iterator is like iterating over an empty collection.
- Mutating an NA object does nothing.

In order to inform the user about how NA objects propagate during execution, the interpreter logs all NA-related operations (e.g., a comparison involving an NA object) to the log file.

When an NA object appears in most types of expressions, the interpreter simply propagates it to the expression's result. The intuition here is that if an operand has an unknown (NA) value, then the result should also be unknown. For example, if `x` is NA, then after executing `z = x + 5`, the value of `z` is also NA[3]. Note that `x` and `z` refer to the same NA object since the interpreter re-uses (aliases) the NA operand of the addition as the result. This is not only more efficient than creating a new object, but it also propagates information about the original error source to aid in debugging.

Note that this technique is most effective on scripts with short error propagation distances, like those that process each record independently of one another [136]. If script execution causes NA objects to taint all subsequent values, then the interpreter cannot produce much useful output.

Iterators: Scripts often compute aggregate statistics by iterating over a collection (e.g., a list) and performing computations such as summing up the elements. If a collection contains even one NA object, then that will taint the computation's result as NA. Thus, when iterating over a collection, the interpreter should skip over NA objects rather than emitting them. This provides the illusion that a collection contains no NA elements, so iterator-based aggregate computations can produce partial results rather than NA. At the same time, though, NA objects remain in the collection, so that direct indexing still returns the correct element.

[3] This behavior can be directly implemented for a dynamically-typed language such as Python. One would need to hack the type system to implement for a statically-typed language such as Java.

Branch conditions: When a script must branch based on an NA condition value (e.g., in an `if` statement), which side should the interpreter take? Technically, neither is correct, since an NA value is neither TRUE nor FALSE. Thus, the interpreter can pick either side (my implementation always takes the `if` side rather than the `else` side). At the end of execution, the user can consult the log file to see which branches had NA conditions, so that he/she can be more suspicious about results dependent on those branches. One potential area for future work is a mode where the interpreter forks to take *both* sides of NA-based branches, but then the obvious challenges of exponential path explosion [41] must be addressed.

6.3.4 Special Handling for Assertion Failures

Programmers write assertions to specify constraints on their scripts' data. A regular interpreter crashes with an assertion failure as soon as one of these constraints is violated, since it is incorrect to continue executing with invalid data.

My technique alters the interpreter so that when it encounters an assertion failure, rather than crashing, it creates an NA object and assigns it to all variables involved in the assertion. The intuition here is that when an assertion fails, only the data involved in the assertion is invalid, so it is safe to continue executing as long as they are marked as NA.

Here is the example from Section 6.3.2 augmented with an assertion that all computed ratios must be less than 1:

```
line_number = 1
for line in open('big_data.txt'):
  tokens = line.split(' ')
  ratio = float(tokens[0]) / float(tokens[1])
  assert(ratio < 1.0)
  print ratio
  line_number += 1
```

This assertion indicates that the programmer assumes all ratios in `big_data.txt` are less than 1. If some records violate this assumption, then the interpreter assigns `ratio` to a new NA object and prints “`<NA>`” when processing those records. At the end of execution, the log file shows which records led to assertion failures, so that the programmer can either fix the dataset or reconsider his/her assumptions.

6.3.5 The Benefits of Precision

The simple example I have shown so far might give the impression that my technique merely skips over bad records. A programmer could accomplish this same goal by simply wrapping the main loop body in a try-catch block and using a `continue` statement as the exception handler (to continue onto the next record). Also, frameworks such as MapReduce [52] can automatically skip over bad records.

My technique is actually more precise, since it catches exceptions at the expression level. This means that if a record contains, say, 30 fields and the script throws an exception when processing one particular field, then only data from that field becomes NA, but the remaining 29 fields are successfully processed. Thus, my technique can automatically skip over portions of bad records rather than entire records. To mimic this level of precision, a programmer would need to wrap *every single program expression* in a try-catch block.

6.4 Python Implementation

My technique can be implemented for any general-purpose programming language, regardless of whether it is compiled or interpreted, statically or dynamically typed. Although data processing scripts written in dynamically-typed languages (e.g., Python) throw more run-time type errors, analogous programs in statically-typed languages (e.g., Java, the language for writing Hadoop data processing jobs [1]) still throw plenty of run-time errors such as array out-of-bounds or null pointer exceptions.

I implemented my technique for Python, since it is a popular language for writing data analyses [133]. I created a prototype open-source interpreter named SlopPy (**Slop**py **Py**thon) [22] by adding 500 lines of C code to the Python 2.6 interpreter. SlopPy passes all of the Python regression tests and works on existing scripts and

3rd-party libraries without any code modifications. The behavior of SlopPy is identical to that of regular Python during normal (exception-free) execution, so scripts run at the same speed.

I implemented all features in Section 6.3 in a straightforward way by modifying how the Python interpreter handles uncaught exceptions and by defining a new NA built-in type. In addition, I also hacked the interpreter's handling of iterators and Python generators [18] (generalized form of iterators) to skip over NA objects rather than emitting them during iteration.

When SlopPy encounters an uncaught exception, it unwinds the stack until it finds the first function *not* in the Python standard library and creates the NA object in that function's frame. In my experiments, this provided a better user experience since the user can more easily reason about exceptions in his/her own code rather than in library code.

SlopPy simultaneously produces two warning log files: a human-readable text log, and a binary log of serialized Python objects which an incremental recovery script can directly process without needing to parse the text log (see Section 6.5.3).

6.5 Evaluation

To demonstrate some of SlopPy's capabilities, I performed three case studies on a 3 GHz Mac Pro with 4 GB of RAM, with regular Python 2.6 and SlopPy both compiled as 32-bit binaries for Mac OS X 10.6.

6.5.1 Supercomputer Event Log Analysis

To show how SlopPy allows programmers to write simple scripts without worrying about error handling, I wrote a script to analyze an event log from the Spirit supercomputer installed in Sandia National Laboratories [25]. In 2007, Oliner and Stearley released event logs from 5 supercomputers [126]; for this experiment, I used the log for Spirit since it is the largest in size (37 GB). This log file contains 272,298,969 lines, where each line documents an event such as an incoming network service request,

```
- 1104594301 2005.01.01 sadmin1 Jan 1 07:45:01 sadmin1/sadmin1 xinetd[2228]: \
    START: rsync pid=616 from=172.30.80.251
- 1106166706 2005.01.19 sadmin1 Jan 19 12:31:46 sadmin1/sadmin1 xinetd[7746]: \
    START: rsync pid=439 from=172.30.73.8
- 1107350922 2005.02.02 sadmin1 Feb 2 05:28:42 sadmin1/sadmin1 xinetd[7746]: \
    START: tftp pid=8381 from=172.30.72.163
```

Figure 6.1: Three example records for xinetd sessions in the Spirit supercomputer event log file [25]. There are only three lines; the '\' character is a line continuation.

kernel panic, or hardware failure.

System administrators routinely write ad-hoc scripts to query supercomputer event logs to monitor system health, discover aberrations, and diagnose failures. Oliner and Stearley describe how log files contain inconsistent or ill-defined structure, corrupted records, and duplicated records, all of which make it harder to write robust log analysis scripts [126].

For this experiment, I emulated a system administrator and wrote a script to print out the IP address of each machine that requested services from the Spirit supercomputer via the UNIX `xinetd` daemon. A sysadmin might use this data to plot a histogram and inspect the distribution of requests by IP addresses or corresponding geographic region; addresses with unusually high activity might indicate either an internal system malfunction or an intrusion attempt.

From a cursory glance at the log file, I saw that `xinetd` events seemed to obey a straightforward format, as shown in Figure 6.1. I wrote the simplest possible script to extract and print the IP addresses: It iterates over all lines of the log file, splits each into whitespace-separated tokens, finds lines whose 8th token (0-indexed) starts with "`xinetd`", then extracts the IP address from the 12th token, which should be formatted like "`from=172.30.80.251`". To avoid biases due to duplicated records, my script coalesces all events within a 5-second interval into one, which Oliner and Stearley also do in their analyses [126]. Figure 6.2 shows my entire script.

Running my script using SlopPy took 20 minutes, 4 seconds and printed 39,225 IP addresses. The SlopPy warning log showed that it caught 11,076 exceptions, which indicates that my script could not process 11,076 lines (out of 272,298,969 total lines, which is only 0.004%).

```
cur_time = -99999

for line in open('spirit.log'):
  tokens = line.split(' ')
  utime = int(tokens[1])
  component = tokens[8]

  if component.startswith('xinetd'):
    # coalesce events within 5-second interval
    if (utime - cur_time) <= 5: continue
    else: cur_time = utime

    ip_addr = tokens[12].split('=')[1]
    ip_lst = ip_addr.split('.')
    ip_byte0 = int(ip_lst[0])
    ip_byte1 = int(ip_lst[1])
    ip_byte2 = int(ip_lst[2])
    ip_byte3 = int(ip_lst[3])
    print ip_byte0,ip_byte1,ip_byte2,ip_byte3
```

Figure 6.2: Python script to extract IP addresses from xinetd sessions in the Spirit supercomputer log, where each line is expected to look like Figure 6.1.

Since the SlopPy warning log contains the context of each exception (see Section 6.3.2), the values of the `line` local variable at each exception show the contents of all lines my script could not process. I searched through the `line` values and could not find any IP addresses, which means that my script extracted all of them (100% precision and recall). The 11,076 exceptions were all due to lines that looked similar to a syntactically-correct `xinetd` request record but did not contain an IP address. Example lines contained:

- Bizarre numbers that are not valid IP addresses: e.g., `from=#564#`
- Null sentinel value: e.g., `from=<no address>`
- Various error messages: e.g., `xinetd[14743]: warning: can't get client address: Connection reset by peer`

Since the Spirit event log file was huge, I did not see these aberrant records when I manually looked through it to learn the schema in preparation for writing my script. All the `xinetd` records I saw followed the format in Figure 6.1, so I wrote a simple script that was only sufficient to process records in that exact format. SlopPy freed me from having to think about error handling. In this case, my script achieved 100% precision and recall when extracting IP addresses, without requiring any error-handling code.

In contrast, when I run my script with regular Python, every time it throws an exception, the interpreter crashes and I need to edit the script to handle that exception and then re-execute. Depending on when the next exception occurs, each run can take from seconds to tens of minutes.

6.5.2 Computational Biology

To show how SlopPy can add error tolerance to an existing script, I used it to run a computational biology script written by Peter B., a Ph.D. student in my department.

When I first told Peter about my project, he immediately showed me a script where SlopPy would have saved him a day of labor. Peter's 200-line Python script runs the Viterbi dynamic programming algorithm [31] on human genomic data and prints out a textual table of results. When he first ran the script, it computed for 7 hours (overnight); when he returned to his computer the next morning, he saw that it had crashed with an exception caused by taking the logarithm of zero[4]. He was especially frustrated since the exception occurred *at the very end of the run* when the script was post-processing and printing out the results table. In other words, all the real 7-hour computation was already done, but the script crashed while formatting the final output.

After seeing that exception, Peter made a simple fix: He defined a custom `log` function that returns a sentinel "negative infinity" value for `log(0)`. When he edited and re-ran his script, everything worked fine, but he lost a day of productivity just because of an unexpected exception.

When I ran the original version of Peter's script using SlopPy (before he patched the `log` function), it finished in 7.5 hours, printing a results table with 328,879 rows and 16 numeric columns (5.2 million total numbers). The SlopPy warning log showed that it caught 35 exceptions, all of which arose from taking the logarithm of zero. The 35 corresponding NA values appeared in the results table as entries that printed as an "`<NA>`" string rather than an actual number.

Since the exceptions occurred at the end of execution after the input data had been transformed and mixed across several matrices, there were no individual "bad records" to blame for the crashes. It just happened that in 35 rare cases, running the Viterbi algorithm and post-processing on some combination of fields within input records led to `log(0)` being performed. Peter developed and tested his script on a small dataset so that each run only took a few seconds. He never saw the `log(0)` exception during testing; it only occurred after running for 7 hours on the full dataset.

If Peter had run his script using SlopPy rather than regular Python, he would be able to see full results after the first run. Even though the results table contains 35 NA values (out of 5.2 million total values), he knows that all NA values represent `log(0)` from inspecting the NA exception type/details, so he does not lose any information.

[4] The Python math library throws an exception for `log(0)`

6.5.3 Incremental Recovery for HTML Parsing

To show how SlopPy enables incremental error recovery, I wrote a simple script to compute a reverse webpage link graph from a corpus of HTML files. When I ran the script, it failed to parse 1% of the files, but I was able to process the SlopPy warning log to recover some data from those unparsable files and merge them back into the original graph.

A reverse web-link graph associates each HTML webpage with a set of webpages that link to it. A search engine can use this graph to compute reputation metrics like PageRank. I took this example from the MapReduce paper [52].

For simplicity, I wrote a sequential script to compute the reverse web-link graph, but SlopPy should provide the same benefits for a MapReduce-style Python script. My script iterates over 71,768 HTML files that I downloaded from a 1.3GB public corpus [27], parses each using the HTML parser from the Python standard library, finds all outgoing links, and builds a graph mapping target URLs to source URLs.

My script takes 10 minutes, 19 seconds to run to completion with SlopPy and generates a graph with 246,932 nodes and 1,256,808 edges. The warning log showed that the standard Python HTML parser threw an exception on 736 out of the 71,768 HTML files in my corpus (1%). HTML files often do not conform to W3C standards, so they crash parsers with messages such as these that appeared in my log:

```
Error: unexpected ’\\xa9’ char in declaration
Error: expected name token at \’<!:iB;@ i39#.6o;H8g" \’
```

SlopPy allows my script to tolerate these parse errors and continue processing rather than crashing; in the end, 99% of the files were properly processed in the initial run.

The warning log contains the values of local variables at the time each exception was thrown: One of the variables contains the string contents of the unparsable HTML file, and another contains its filename. A script could directly process that log to access the contents of the 736 HTML files (1%) that my original script could not parse.

I modified my script to not use an HTML parser (since that would just crash again) but instead to use a regular expression to extract outgoing links. The reason why I did not originally use a regular expression was because it is less accurate than a real parser, so it might miss some links. However, a regular expression is more robust, since it does not care whether the HTML conforms to W3C standards.

I ran my modified script on the warning log and merged the newly-found links into the existing reverse web-link graph. This recovery run completed in 6 seconds and added 1116 new nodes and 4987 new edges to the graph, which only made it 0.4% larger. I call this run *incremental* because it did not re-process the entire corpus, which would have taken the full 10 minutes. Although 10 minutes is not too long to wait, if I had access to a larger HTML corpus, then a full run could have taken hours, even when parallelized.

6.6 Discussion and Future Work

My design philosophy underlying SlopPy is that programmers doing ad-hoc data processing tasks can be more productive if the run-time system allows their imperfect, buggy programs to run to completion and produce partial results rather than mercilessly crashing them. I want programmers to be able to rapidly hack on ad-hoc scripts and discover insights about their data without worrying about the mundane details of error handling. Here are some directions for future work:

- Since SlopPy allows incorrect programs to keep running, it is vital to have their partial results be accompanied by a precise explanation of what went wrong. The warning log file that SlopPy produces is a first step towards this goal, but there is still plenty of room for improvement. I would like to accurately track both control- and data-flow dependencies of NA values and report this provenance data in such a way that the programmer can easily find out what went wrong, why, and how it can be fixed. I would also like the programmer to get a realistic sense of how much he/she can “trust” particular components of the output, and perhaps even get probabilistic bounds on how badly a particular NA value might corrupt indirectly-affected numerical results.

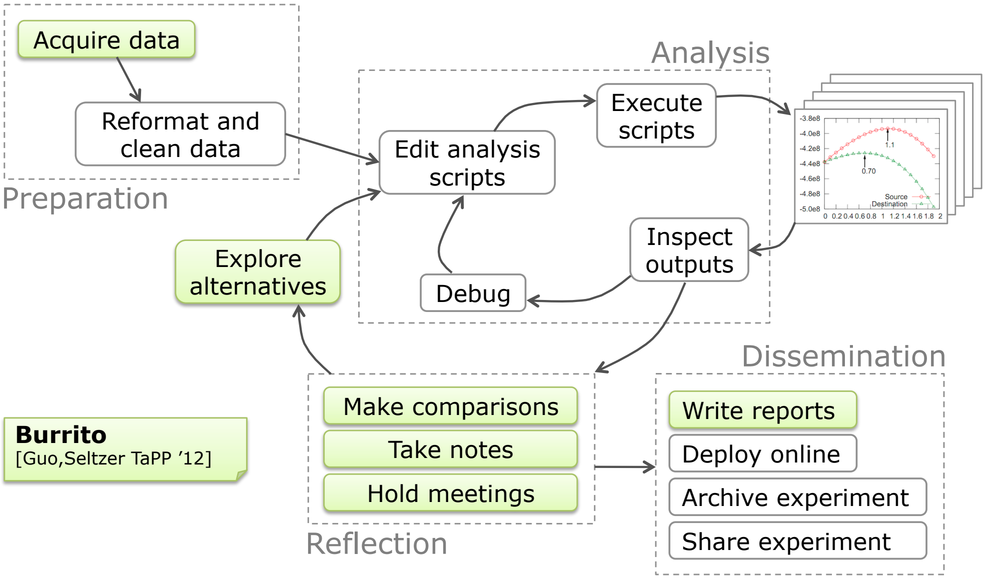
- I want to explore more principled ways of isolating the buggy parts of execution from the correct parts, in order to preserve the validity of partial results. For example, if NA-tainted execution were about to delete previously-computed results, then it might be better to actually terminate execution rather than continuing to execute.

- Since more and more data processing is being done in the cloud (e.g., via Amazon EC2 and MapReduce), could ideas from SlopPy be generalized to work on parallel and distributed data processing systems? One unique challenge in this space is that these systems are often comprised of multiple layers implemented in different languages (e.g., SQL, Hive [151], Pig [128], Java, Python), so error handling, propagation, and reporting must cross language boundaries.

In closing, although I have presented SlopPy in the context of data processing scripts, it is a general-purpose interpreter that can protect *any* Python application (e.g., server or web application) from unanticipated crashes due to latent bugs. Broader directions for future work include thinking about other domains where this “sloppy run-time system” idea can improve programmer productivity and, paradoxically, perhaps even software reliability.

Chapter 7

Burrito: Manage Experiment Notes and Files



This chapter presents a Linux-based system called Burrito, which helps research programmers organize, annotate, and recall past insights about their experiments. Its contents were adapted from a 2012 workshop paper that I co-authored with Margo Seltzer [77]. All uses of "we", "our", and "us" in this chapter refer to both authors.

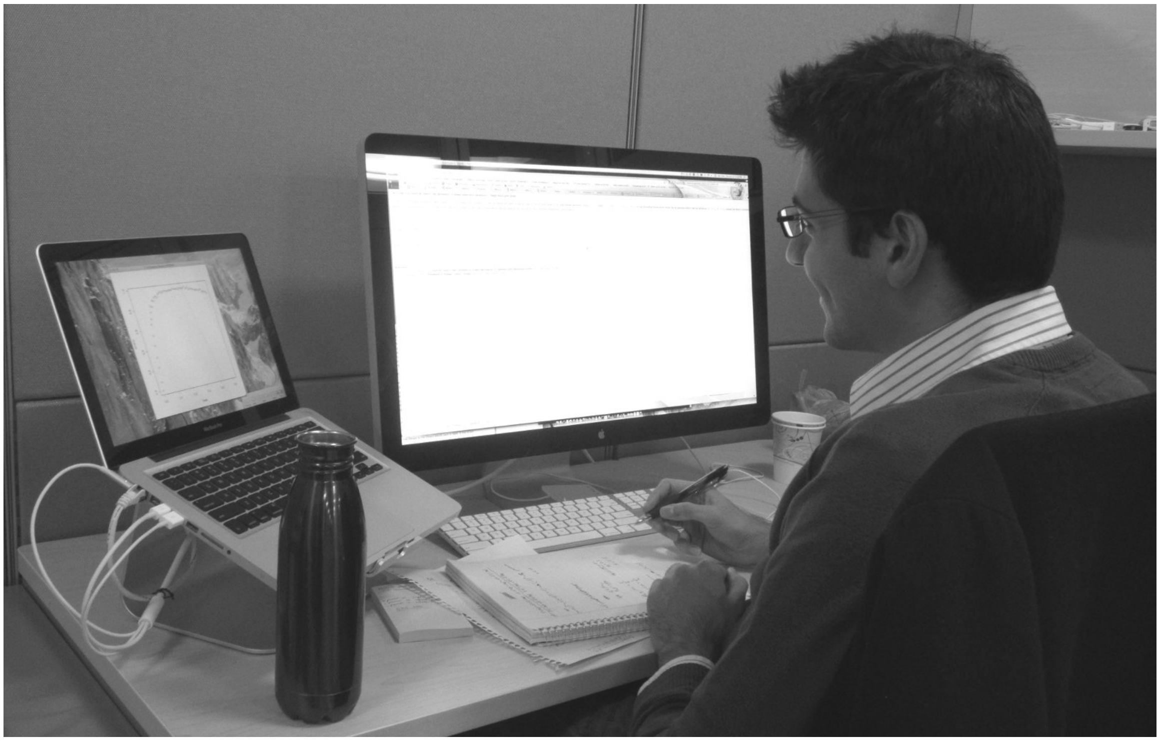
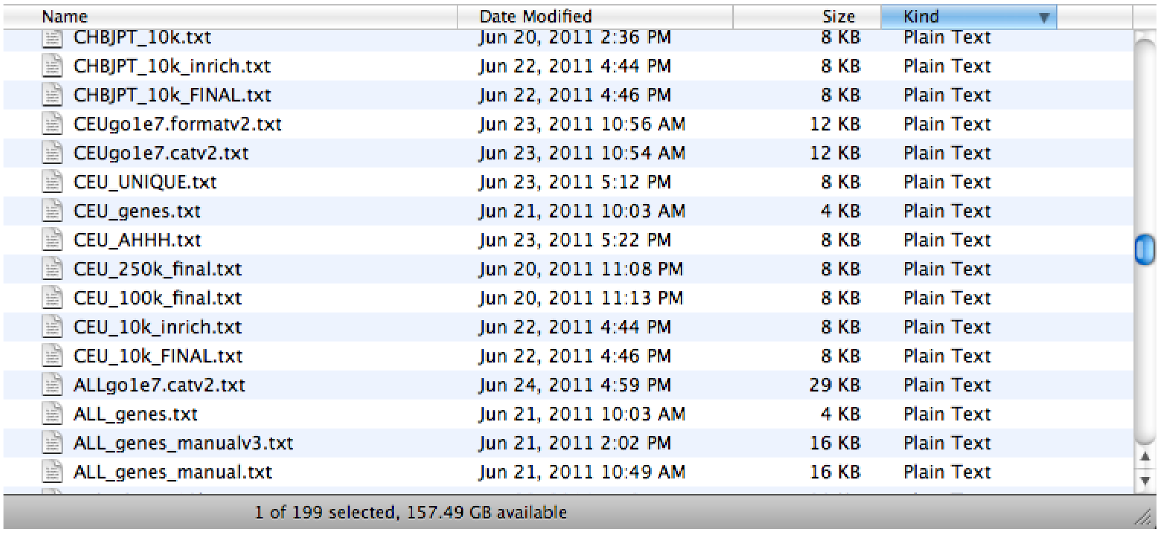


Figure 7.1: A computational biologist at work, viewing code and documentation on his main monitor, graphs on his laptop screen, and taking notes in a paper notebook.

7.1 Motivation

Recall from Chapter 2 that research programmers have trouble keeping track of numerous code and data files, comparing the results of trials executed with different parameters, and remembering what they learned from past successes and failures. These problems persist because the current methods for managing and documenting experiments require too much *user overhead* (effort) and provide too little *context*:

1. **Managing file versions**: Researchers are reluctant to use version control systems (e.g., SVN, Git) due to the *user overhead* of setting up a repository and deciding when to add and commit file versions. Instead, we observed that many researchers manually perform "versioning" by making multiple copies of selected files and appending version numbers, script parameter values, and other metadata onto their filenames (see Figure 7.2). These cryptic filenames provide some *context* for, say, which execution parameters generated an output file, but



Name	Date Modified	Size	Kind
CHBJPT_10k.txt	Jun 20, 2011 2:36 PM	8 KB	Plain Text
CHBJPT_10k_inrich.txt	Jun 22, 2011 4:44 PM	8 KB	Plain Text
CHBJPT_10k_FINAL.txt	Jun 22, 2011 4:46 PM	8 KB	Plain Text
CEUgo1e7.formatv2.txt	Jun 23, 2011 10:56 AM	12 KB	Plain Text
CEUgo1e7.catv2.txt	Jun 23, 2011 10:54 AM	12 KB	Plain Text
CEU_UNIQUE.txt	Jun 23, 2011 5:12 PM	8 KB	Plain Text
CEU_genes.txt	Jun 21, 2011 10:03 AM	4 KB	Plain Text
CEU_AHHH.txt	Jun 23, 2011 5:22 PM	8 KB	Plain Text
CEU_250k_final.txt	Jun 20, 2011 11:08 PM	8 KB	Plain Text
CEU_100k_final.txt	Jun 20, 2011 11:13 PM	8 KB	Plain Text
CEU_10k_inrich.txt	Jun 22, 2011 4:44 PM	8 KB	Plain Text
CEU_10k_FINAL.txt	Jun 22, 2011 4:46 PM	8 KB	Plain Text
ALLgo1e7.catv2.txt	Jun 24, 2011 4:59 PM	29 KB	Plain Text
ALL_genes.txt	Jun 21, 2011 10:03 AM	4 KB	Plain Text
ALL_genes_manualv3.txt	Jun 21, 2011 2:02 PM	16 KB	Plain Text
ALL_genes_manual.txt	Jun 21, 2011 10:49 AM	16 KB	Plain Text

1 of 199 selected, 157.49 GB available

Figure 7.2: Metadata such as script parameter values and version numbers are often encoded in output filenames.

researchers often forget their own ad-hoc naming schemes.

2. **Copying-and-pasting**: To keep a record of their trials, researchers copy-and-paste the parameters and textual outputs of executed commands into text files. Some also copy-and-paste output images (e.g., graphs) into Microsoft PowerPoint, OneNote, or other electronic notebook software. These actions are purely *overhead*, disrupting the researcher's workflow while manually duplicating information that the computer ought to be able to track automatically.

3. **Writing notes**: Most researchers we observed write notes in plain text files or "sticky notes" desktop widgets. Notes provide some *context* for what the researcher is thinking at the present moment, but they are not linked with the source code and data files to which they refer.

4. **Organizing notes files**: The problem with keeping one central notes file is that it is hard to search through it, and the problem with keeping many specialized files is that it is hard to name and locate them throughout the filesystem. Like copying-and-pasting, the burden of coming up with a scheme to organize notes files is purely *overhead*.

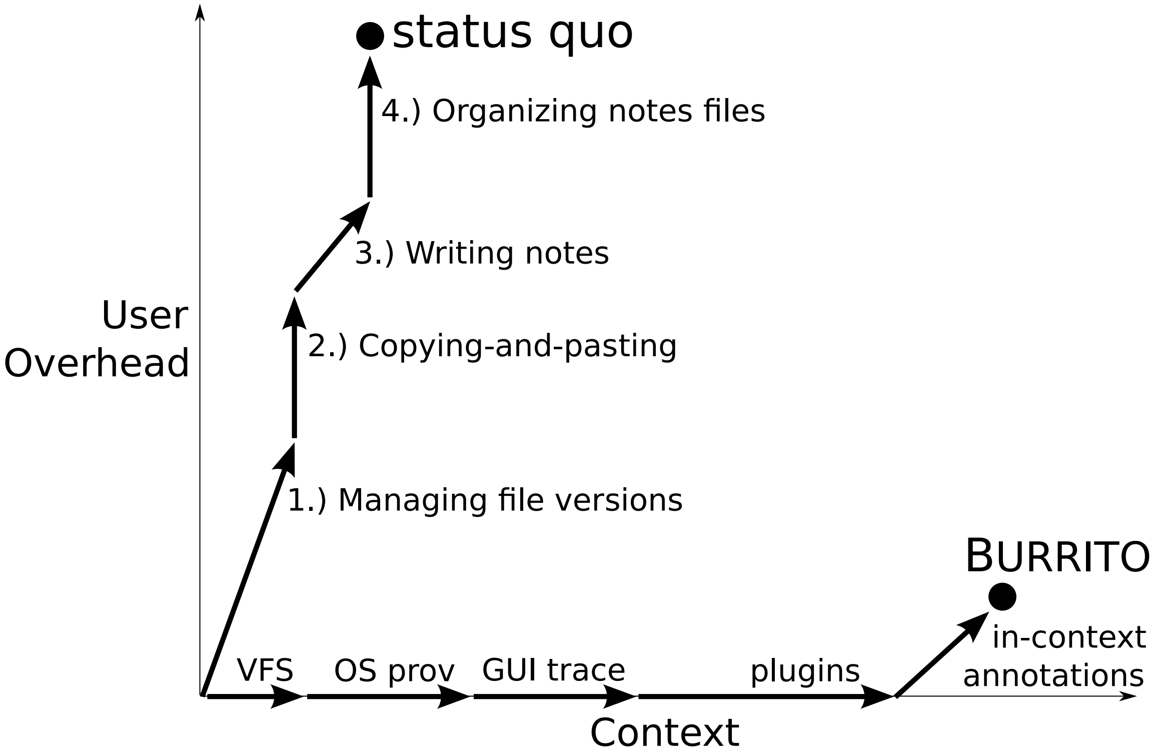


Figure 7.3: The amounts of context and user overhead added by each component of current organization methods ("status quo") and by the Burrito system.

The sum total of these four activities leads to the "status quo" dot in the upper-left corner of Figure 7.3. Today, researchers are frustrated by the excessive *user overhead* that detracts from their goal of running experiments, and they have trouble remembering what they learned from past trials due to a lack of *context* in their notes.

7.2 Burrito System Overview

We now present Burrito, a Linux-based system which enables researchers to work in the low-overhead, high-context state in the lower-right corner of Figure 7.3. Burrito consists of two parts: an extensible platform that automatically captures the user's activity context (Section 7.3), and a set of applications that allow the user to annotate and query the collected context (Section 7.4).

Suppose that Alice is a Ph.D. student doing computational research on a Linux machine with the Burrito platform installed. A versioning filesystem ("**VFS**" in Figure 7.3) preserves all old versions of her source code and data files, so that she no longer needs to use version control or make multiple weirdly-named copies of her files. An OS-level provenance collection daemon (**OS prov**) captures execution context such as which processes read from and wrote to which files. A **GUI trace** daemon captures her GUI interactions, which provides context such as which application windows she was viewing at particular times. Finally, a set of platform **plugins** capture her activities within specific applications (Section 7.3.2). For example, the Bash plugin records her executed Bash shell commands, and the Firefox plugin records her web browsing history. Alice can also write plugins for domain-specific scientific software that she uses. In sum, the Burrito platform automatically captures Alice's work activities and their context with *no user overhead*, no perceptible run-time slowdowns, and disk space usage of ∼2 GB per month (see Section 7.5 for details).

A GUI application called the *Activity Feed* allows Alice to view her recent activities in real-time and make **in-context annotations** (Section 7.4.1). For example, as soon as she executes a Bash command, it appears as an event in her feed (see Figure 7.5). She can click on that event to annotate it with text notes. In-context annotations eliminate most of the user overhead in notetaking: Alice no longer needs to copy-and-paste command parameters, output files, and other data into scattered notes files; she also no longer needs to name, organize, and locate her notes, since all annotations are linked with their referent events. The only unavoidable overhead is writing the note itself.

The Burrito system "wraps" a layer of computational infrastructure around the user's normal work environment. Its combination of automatic activity capture and in-context annotations enables researchers like Alice to work in the lower-right corner of Figure 7.3: *Low overhead* means that Alice can spend more of her time doing actual experiments rather than managing files and notes. *High context* means that Alice can make queries about her past activities using applications built atop the Burrito platform. For example:

- She can compare the results of running different variants of her experiment by viewing how changes to her source code files and command parameters led to corresponding changes in output files (Section 7.4.2).
- When she returns to work after a vacation, she can remind herself of what web pages, documents, and other code she was consulting when she was last hacking on a specific part of her codebase (Section 7.4.3).
- She can retrieve old versions of files by context rather than just by date (e.g., *"show me the version of the graph generated when I was editing this script while viewing that particular tutorial web page."*).
- She can generate an HTML report of her past week's experimental trials to help prepare for her weekly meeting with her Ph.D. advisor (Section 7.4.4).
- She can generate a summary of her past few years' worth of activities and notes to prepare for writing her Ph.D. dissertation (Section 7.4.4).

Burrito strives to automate as much of the experiment management and notetaking process as possible, thus freeing researchers to focus on their actual work.

7.3 The Burrito Platform

The Burrito platform (Figure 7.4) consists of a core (Section 7.3.1) and a set of plug-ins (Section 7.3.2) that integrate a rich trace of user activity into a master database.

7.3.1 Core Platform

NILFS versioning filesystem

The base layer of the Burrito core platform is NILFS, a log-structured filesystem that performs continuous snapshotting of every file write [101]. NILFS allows the user to view old versions of files by mounting a snapshot of a specific past time as a read-only partition. It has been in active development for the past decade and officially entered the Linux kernel in June 2009 (version 2.6.30).

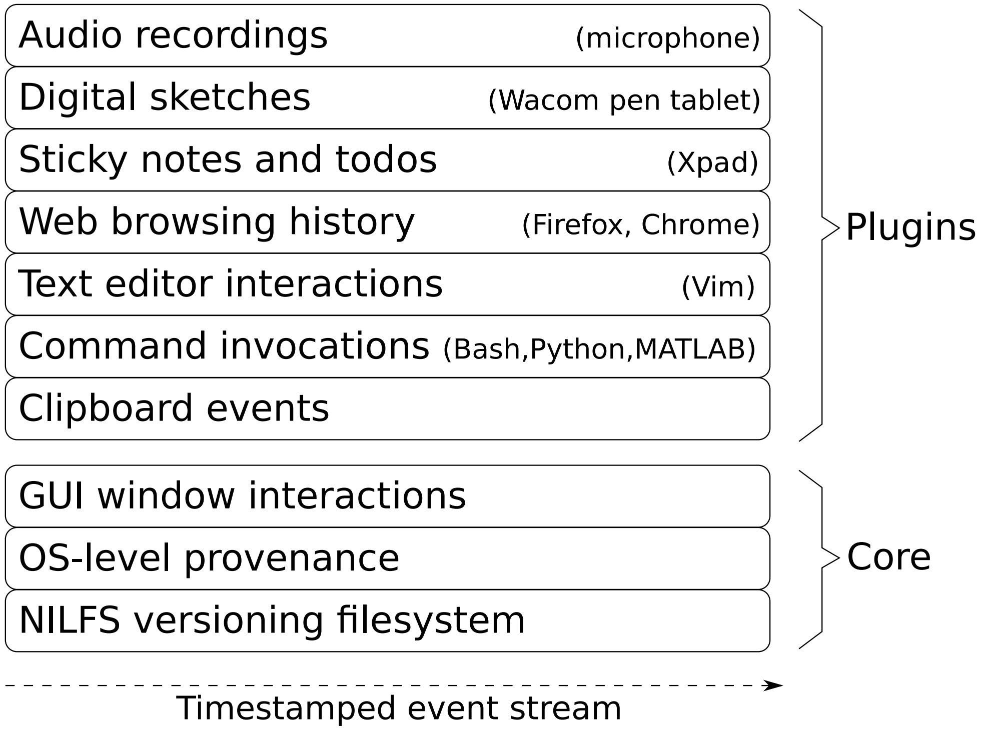

Audio recordings (microphone)
Digital sketches (Wacom pen tablet)
Sticky notes and todos (Xpad)
Web browsing history (Firefox, Chrome)
Text editor interactions (Vim)
Command invocations (Bash,Python,MATLAB)
Clipboard events
Plugins
GUI window interactions
OS-level provenance
NILFS versioning filesystem
Core
Timestamped event stream

Figure 7.4: Burrito platform overview: Each component ("layer") logs a timestamped stream of events to a master database.

We format and mount the user's `/home` directory as a NILFS partition, configured to retain all snapshots. This setup essentially turns the user's `/home` directory into a version control system, where *all* edits are automatically committed. While we could have automatically versioned all files on a machine, we chose to mount only `/home` as NILFS for two reasons: First, it improves performance and disk space utilization, and second, researchers we interviewed did not seem interested in versioning system files outside of `/home`. On an externally-administered shared machine, one could imagine also mounting system partitions as NILFS, so that system configuration changes would be visible to Burrito.

We have added `NOPASSWD` directives to `/etc/sudoers` so that NILFS snapshots can be mounted and unmounted without needing to enter a password (although the user still needs sudo privileges).

NILFS provides userspace tools to query and mount old snapshots. We wrote a Python library to interface with these tools. Given a target timestamp, our library can mount a read-only snapshot of `/home` as it existed at that time, placing it into a sub-directory within `/tmp`.

OS-level provenance collection

The middle layer in the Burrito core collects OS-level *provenance* [119], which includes information such as which files were read, written, and renamed by each process, and which sub-processes were forked and exec'ed.

We use the SystemTap tool [26] to collect OS-level provenance. SystemTap allows users to write system monitoring scripts in a C-like language, which it then compiles into a kernel module and injects into a live kernel. It is under active development and supported in all modern Linux distributions. We wrote a SystemTap script that logs the timestamps and parameter values of selected system calls. When the user logs in, our script starts in the background, opens a log file in `/var/log/burrito`, and waits for system call events.

Each time any process issues a system call, our script prints a one-line log entry containing the current timestamp (milliseconds since the Unix epoch), process ID (PID), parent process ID (PPID), user ID (UID), executable name, and the following fields specific to each system call type:

- `open`: the opened file's absolute path, file descriptor number, and mode (read, write, or read-write)
- `mmap`: the opened file descriptor number and protection mode (read, write, or read-write)
- `read` and `write`: the file descriptor that performed the read/write. Our script only logs the *first* read and write made by a process to a particular file descriptor. This way, each time a program saves (or loads) a file, only one write (or read) event is logged.
- `close`: file descriptor of the closed file
- `pipe`, `dup`, `dup2`: pair of file descriptors
- `fork`: child PID
- `execve`: current working directory, executable filename, and command-line arguments
- `exit_group`: exit code
- `rename`: absolute paths of old and new filenames

GUI interaction tracing

The final layer of the Burrito core platform records the user's interactions with the Linux desktop GUI using the Assistive Technology Service Provider Interface (AT-SPI) [2]. Developers of accessibility tools (e.g., screen readers) use the AT-SPI library to programmatically access GUI widgets and events. The AT-SPI is supported by GNOME and KDE, the two main Linux desktop environments, and by their native GTK and Qt GUI toolkits. We implemented our GUI tracer as a Python script that calls the AT-SPI via its Python bindings.

When the user logs in, our GUI tracer script starts in the background and opens a log file in `/var/log/burrito`. Each time a significant GUI event occurs (see next paragraph for details), our script appends an entry to the log that contains the current *desktop state*, obtained via AT-SPI calls. Each desktop state object contains a list of open GUI applications, the GUI windows belonging to each application, and the current title, size, location, and status of each window. A window's status can either be normal, minimized, or *active*. A desktop state can have at most one active window: the one currently in focus.

Our script registers callbacks with the AT-SPI so that it writes the current timestamp and desktop state to the log when the following GUI events occur:

- The user selects a window as the active window.
- The user creates, destroys, moves, resizes, maximizes, or minimizes a window.
- The user causes a window's title to change, which can occur when a web browser loads a different web page or when a PDF viewer loads a new document.

These raw log files contain the full history of a user's interactions with windows in the Linux desktop GUI.

Continuous data integration

We wrote a daemon that periodically polls the raw logs, parses and cleans their data, and integrates them together into a master database. We use MongoDB [15], a popular schema-free document-oriented database, as the master store for Burrito platform data. We found MongoDB to be a good fit because our daemon's Python code can easily serialize Python objects into the JSON format that MongoDB requires rather than having to "flatten" fields into relational tables. JSON [10] is a concise text-based format capable of expressing key-value associations and nested data (e.g., a key mapping to a list of values, which can themselves be mappings). MongoDB's schema-free nature makes it easy for Burrito applications (see Section 7.4) to add new fields, such as annotations, to stored data. To facilitate fast time-based search, we created indices on all timestamp fields such as file read/write/rename times, process creation/exit times, and desktop state timestamps; users can easily create additional indices in the MongoDB shell.

The data integration daemon wakes every 10 seconds and *incrementally* fetches only the new log entries that have appeared since the last round of processing. It dispatches those new entries to the appropriate *parser*, which curates, serializes, and saves the resulting objects into the master database.

The OS-level provenance log parser translates a series of flat log entries such as *"Process ID 123 opened a file foo using file descriptor 5"* into a *provenance graph* that connects processes to files and to other processes.

When the parser encounters a `fork` log entry, it creates a new *process state* object. Each process state contains these fields: PID, parent PID, UID, creation time, exit time, exit code, and a list of *phases.* A phase begins with an `execve` system call and represents the duration when a process was running a particular executable image. Processes have multiple phases if `execve` was called multiple times without forking. Each process phase contains its creation time, end time, `execve` filename, arguments, current working directory, and most importantly, a timestamped list of files read, written, and renamed.

The parser maintains a set of active (not yet exited) processes in memory. At the end of each processing round, it saves to the database only the active processes whose fields have been updated since the last round. When the parser encounters an `exit_group` log entry, it marks the target process as exited, removes it from the active set, and updates its status in the database. Thus, the master database maintains an up-to-date provenance graph of all active and exited processes and the files that they read/wrote/renamed (with a 10-second polling lag).

The GUI log parser continually cleans new incoming entries in the GUI interaction trace log and saves *desktop state* objects to the master database. Due to the frequency with which AT-SPI fires events, there ends up being redundancy in these logs, so the parser first filters redundant entries. It then tries to match up each GUI window with the process that launched it. Due to the nature of X Window and GUI toolkits, there is no standard way to query a GUI window and find the underlying process that

Bash:	`{"timestamp":93717, "pid":336, "pwd":"/home/bob", "command":["rm", "foo.txt"]}`
MATLAB:	`{"timestamp":94023, "command":"plot(x,abs(true-y),’mx’)"}`
Vim:	`{"timestamp":95852, "pid":359, "event":"editing", "filename":"/home/bob/foo.txt"}`
Clipboard:	`{"timestamp":96128, "copy_time":91283, "x":577, "y":622, "contents":"hello world"}`

Table 7.1: Example JSON log entries produced by Burrito plugins. Timestamps have been shortened for readability.

controls it. Thus, the parser uses the following heuristic: When the first GUI window is created for an application, query the master database for all processes that were spawned in the past few seconds (8 by default), and if any of those match the name of the GUI application, then associate its PID with that application’s windows.

7.3.2 Platform Plugins

Since it is hard to predict what set of applications a particular researcher prefers to use, we made it easy for developers (who are either the researchers themselves or programmers they hire) to write plugins to integrate application-specific data into the master database: Each plugin simply appends lines of JSON-formatted text into a log file within `/var/log/burrito`. The only mandatory field that each JSON log entry must include is a timestamp (see Table 7.1 for examples). Since JSON is plain text, plugins can be written in any programming language capable of writing to a file. There is no need for custom APIs or language bindings.

To install a plugin, the developer registers its log file path and a parser with the data integration daemon. Most plugins use the **Standard JSON parser**, which parses each line in the log as a JSON object and writes it into the database (MongoDB expects JSON as input). The integrator saves data from each plugin into a separate database table and indexes by timestamp.

Although the developer can modify an application’s code to insert the logging calls required for a Burrito plugin, doing so can be cumbersome since it involves re-compilation and re-installation. We now show how it is possible to create a variety of plugins without modifying or re-compiling the original application. Most of our plugins were less than 30 lines of Python code, as shown in Table 7.2.

Plugins using GUI event monitoring

We wrote three Burrito plugins by using the AT-SPI to monitor GUI events and create log entries:

- **Sticky notes**: Many researchers we observed used a "sticky notes" application to write notes and todo list items. We wanted to keep a full history of all notes, so we wrote a Burrito plugin for the Xpad sticky notes application. Our plugin uses AT-SPI to monitor when each note window's text buffer changes and saves a JSON log entry with the current timestamp and buffer contents.

- **Clipboard tracing**: To track the data that researchers copy and paste between applications (e.g., copying sample code from a tutorial web page), we wrote a Burrito plugin that logs clipboard events. Our plugin logs when text is copied to the primary X clipboard and when it is later pasted via a middle mouse button click. We wrote a custom parser that finds the active GUI window at copy and paste times and saves the source and destination application PIDs, respectively, to the database.

- **Web browser tracing**: Researchers use the web as a primary resource for technical documentation, so we want to track the web pages that they visit. We wrote a plugin that monitors web browser windows for title changes, which occur when the user switches to view a different web page. At those times, the plugin scrapes the URL from the URL text field in the GUI and saves a log entry with the timestamp, URL, and web page title. This form of logging is more precise than web browser histories because there is no deduplication; it records the exact page that the user was viewing at every moment the web browser window was active. We have implemented plugins for Google Chrome and Firefox; the same technique can be used for any other GUI web browser.

The advantage of GUI scraping is that it enables creating plugins without modifying applications, but the disadvantage is that it is potentially brittle. If an application changes the layout of its GUI widgets in a future version, then the corresponding plugin needs to adjust as well.

Plugins using application scripting

We wrote three plugins for applications with support for scripting. This technique is robust as long as the target application's scripting API remains unchanged.

- **Scriptable text editor**: Researchers spend most of their work time in a text editor writing either code or prose. They often have multiple files open within their editor, but the GUI tracer can only detect when the text editor window is active, not which file is currently being edited. To improve precision, we wrote a plugin for the Vim text editor by adding scripting code to `.vimrc`. Our plugin logs an event whenever the user switches to edit a different file within Vim. It also allows the user to select and "bookmark" specific passages of text, recording the marks and timestamps to the log. It is straightforward to write a similar plugin for other text editors, such as Emacs and gedit, and scriptable IDEs, such as Eclipse.

- **Bash and Python command invocations**: We wrote a plugin for the Bash shell that records the time, present working directory, and command-line arguments of all executed Bash commands. Our plugin adds a debug trap line (`trap '<log $PWD and $BASH_COMMAND>' DEBUG`) to the `.bashrc` init script, which executes logging code before each Bash command is run. We wrote a similar plugin for the Python shell to capture executed Python commands.

Developers can write similar plugins for other scriptable apps such as the Pidgin IM chat client, the GIMP image editor, and event hooks for version control systems such as Git and SVN.

Plugins using filesystem notifications

Our final class of plugins logs new events whenever specific applications write certain kinds of files to disk:

- **Digital sketches**: Researchers often make freehand sketches on whiteboards, scrap paper, or notebooks. To integrate sketches into the Burrito timeline, we connected a Wacom Bamboo pen tablet to our machine. This hardware allows the user to draw sketches with a digital stylus in a program such as `gnome-paint`. Our plugin consists of a custom parser that detects when the `gnome-paint` process writes to an image file (by querying OS-level provenance) and then logs the timestamp and filename of the saved sketch. Since NILFS preserves all file versions, Burrito can access all old sketches even if the user saves new sketches to the same file.

- **Audio recordings**: Some researchers want to take voice notes and integrate them into the Burrito timeline. Using the same logic as our digital sketch plugin, this plugin tracks audio files saved by the GNOME Audio Recorder Applet and logs their timestamps and filenames.

- **MATLAB command invocations**: Many researchers use MATLAB as their data analysis environment. We wanted to log all executed MATLAB commands, but we could not take the same approach as we did for Bash and Python, since MATLAB is neither open-source nor scriptable. Instead, we leveraged the fact that MATLAB already writes its command history to a `history.m` file. We configured MATLAB to write to `history.m` after executing every command rather than only writing at exit time. Our custom parser detects writes to `history.m` and creates timestamped log entries for new commands.

7.4 Example Burrito Applications

To demonstrate the capabilities of the Burrito platform, we built four Linux desktop applications on top of it. We designed each application to fulfill common use cases described by our target demographic of research programmers (see Chapter 2).

- The *Activity Feed* allows the user to view recent experiment activities, annotate events with notes, mark significant checkpoints, and monitor, diff, and revert recently-modified files (Section 7.4.1).

Component	Lines of code	Language
Burrito platform core (Section 7.3.1)		
NILFS userspace library	350	Python
OS-level provenance tracer	250	SystemTap
GUI interaction tracer	500	Python
Data integrator	2000	Python
Burrito platform plugins (Section 7.3.2)		
Digital sketches	10	Python
Audio recordings	10	Python
Firefox & Google Chrome	10	Python
Python	10	Python
MATLAB	20	Python
Bash	40	Bash
Sticky notes	50	Python
Vim	80	Python
Clipboard	100	Python
Burrito applications (Section 7.4)		
Activity Feed	1500 (15)	Python
Computational Context Viewer	750 (10)	Python
Activity Context Viewer	1000 (10)	Python
Lab Notebook Generator	300 (10)	Python

Table 7.2: The approximate lines of code and languages used to implement each component in the Burrito system. The numbers in parentheses show approximately how much code was required to interface with the Burrito platform.

- The *Computational Context Viewer* allows the user to view how changes in source code lead to corresponding changes in output data files (Section 7.4.2).
- The *Activity Context Viewer* allows the user to view the activity context surrounding edits to a file (Section 7.4.3).
- The *Lab Notebook Generator* creates an HTML report summarizing the user's experiment activities and notes over a given time period (Section 7.4.4).

Burrito applications can be written in any language with an interface to the MongoDB database. We wrote our applications in Python using the PyMongo and PyGTK libraries to access MongoDB and the GTK GUI toolkit, respectively. Although each application is ∼1000 lines of code, less than a dozen lines are used to interface with the Burrito platform (see the numbers in parentheses in Table 7.2).

7.4.1 Activity Feed

The Activity Feed is a sidebar residing on the left portion of the user's Linux desktop background. It periodically polls the master Burrito database (every 5 seconds by default) and displays a near real-time stream of the user's actions as a list of *feed events* in reverse chronological order. New events appear at the top of the feed and push down older events (Figure 7.5). This UI metaphor is inspired by the Facebook news feed and Twitter tweet stream. The feed currently displays six types of events:

- A **Bash command event** shows a group of Bash shell commands executed in the same directory without any other intervening events. The user can click on any command to copy it to the clipboard and paste it into a terminal to re-execute. (Python and MATLAB commands can also be displayed.)
- A **website visit event** shows a set of web pages visited without any intervening events. The user can click on any page title to open its link in a browser.
- A **file modification event** shows a group of files modified by a particular process. For example, saving a source code file in a text editor will create a new

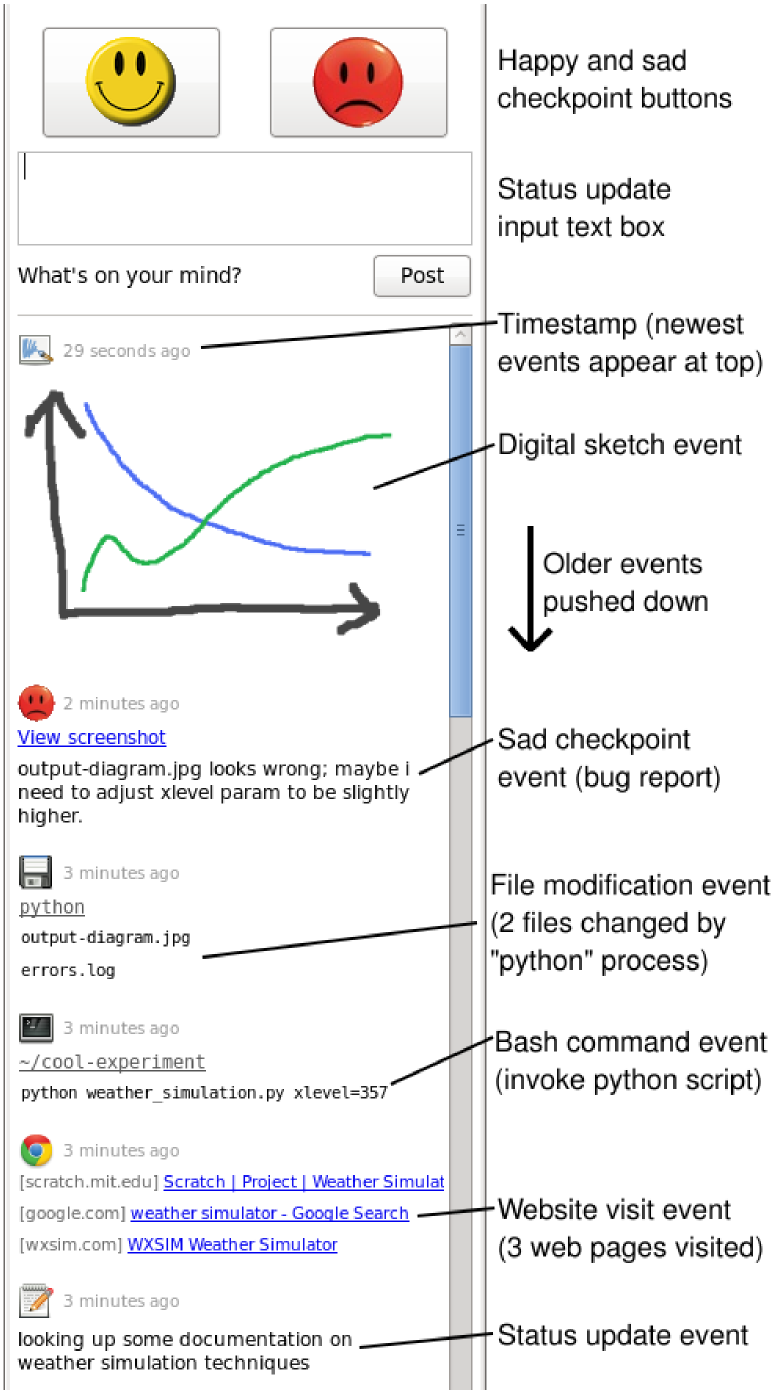
file modification event, as does executing a script to generate an output data file. Only files within the user's home directory are shown here, since NILFS only maintains version histories for `/home`.

- A **digital sketch event** shows a thumbnail view of a sketch that the user has just drawn using, say, a pen tablet. The user can click on the thumbnail to open the full-sized sketch in an external image viewer.

- The user can create a **status update event** by entering text in the status text box and pressing the "Post" button. This is the main way for users to take notes about what they are currently working on at a given moment, which helps place other events in context (e.g., posting *"I'm now trying to optimize my B-tree split algorithm to copy less data"*).

- The user can create a **checkpoint event** by clicking on either the "Happy Face" or "Sad Face" button and then entering a note in the pop-up text box describing why they are happy or sad about the current state of the experiment. The system takes a screenshot and pushes it alongside the note onto the feed. A "happy checkpoint" is like making a *commit* in a version control system, and a "sad checkpoint" is like filing a *bug report* in a bug tracking system.

The Activity Feed allows the user to take notes in the context of recent activities and to monitor recently-modified files. It coalesces events for readability: e.g., if the user ran 5 Bash commands and then visited 10 websites, only two events are displayed. However, since it displays all events in a linear stream, it is not ideal for viewing older events. The applications we describe in Sections 7.4.2, 7.4.3, and 7.4.4 provide more focused views.


What's on your mind?
Post
Happy and sad checkpoint buttons
Status update input text box
Timestamp (newest events appear at top)
Digital sketch event
Older events pushed down
Sad checkpoint event (bug report)
File modification event (2 files changed by "python" process)
Bash command event (invoke python script)
Website visit event (3 web pages visited)
Status update event

29 seconds ago

2 minutes ago

View screenshot

output-diagram.jpg looks wrong; maybe i need to adjust xlevel param to be slightly higher.

3 minutes ago

```
python
output-diagram.jpg
errors.log
```

3 minutes ago

```
~/cool-experiment
python weather_simulation.py xlevel=357
```

3 minutes ago

[scratch.mit.edu] Scratch | Project | Weather Simulat

[google.com] weather simulator - Google Search

[wxsim.com] WXSIM Weather Simulator

3 minutes ago

looking up some documentation on weather simulation techniques

Figure 7.5: The Activity Feed resides on the desktop background and shows a near real-time stream of user actions.

Annotating feed events

Besides writing notes in status update and checkpoint events, the user can also add text annotations to all other types of feed events. After right-clicking on an item in the feed (e.g., a Bash command invocation), the user can choose the "Annotate" option from a pop-up menu. Doing so creates a text box immediately below the event where the user can enter and save a note to the database.

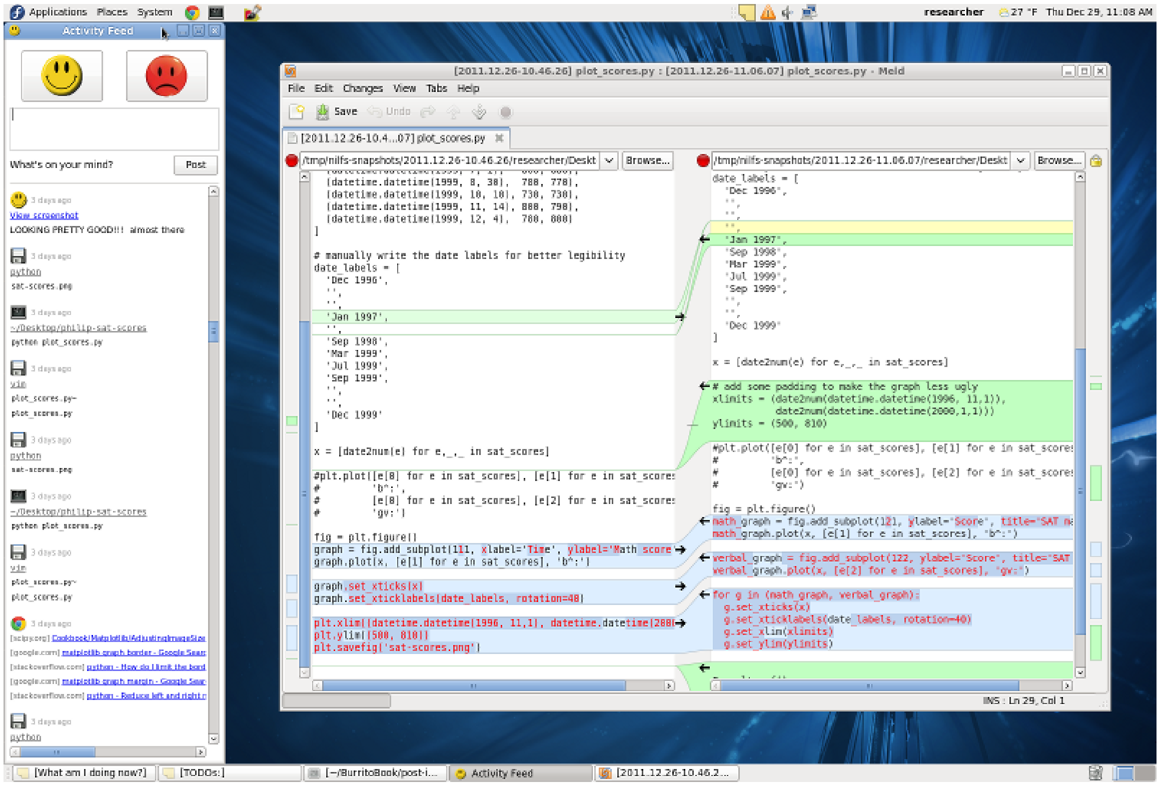
Right-clicking on any feed item also surfaces a "Copy event hashtag" menu option, which copies a unique *hashtag* [8] string identifying this event to the clipboard. For example, a hashtag for a Bash command event might be `#bash-98aaa130f4`. The user can paste hashtags in annotations to add references to specific events, thus creating semantic inter-event links that other tools can parse. For example, a bug tracking application could look for annotations containing hashtags of "sad checkpoint" events (akin to bug reports) and update the bug database accordingly (e.g., some modified file might be annotated as *"fixed bug* `#sad-82fa1273eb` *with this hacky edit"*).

Use case: Researchers currently document their experiments by manually logging selected activities and notes in various text files. They struggle with naming, locating, and later retrieving those files and then associating their notes with the proper context. For instance, when reading months-old notes about tweaks made to a particular script, the user will probably be unable to view the exact version of the script to which the notes refer. In contrast, activity feed annotations allow users to make notes within the most precise context *at the time when relevant events are occurring.* By querying NILFS, the user can later retrieve the exact old version of a file to which an annotation refers, even if that file has been deleted.

Interacting with file modification events

The Activity Feed provides a convenient interface to monitor and access old versions of files. Right-clicking on a file modification event in the feed pops up a menu where the user can make the following types of actions:

- **Open** the version of the file either right before or after the given modification. The system mounts the NILFS snapshot corresponding to the event's timestamp


Applications Places System
researcher
27 °F Thu Dec 29, 11:08 AM
Activity Feed
Post
[2011.12.26-10.46.26] plot_scores.py : [2011.12.26-11.06.07] plot_scores.py - Meld
File Edit Changes View Tabs Help
Save Undo
[2011.12.26-10.4...07] plot_scores.py
Browse...
Browse...
INS : Ln 29, Col 1
[What am I doing now?]
[TODOs:]
[~/BurritoBook/post-i...
Activity Feed
[2011.12.26-10.46.2...

What's on your mind?

3 days ago
View screenshot
LOOKING PRETTY GOOD!!! almost there

3 days ago
python
sat-scores.png

3 days ago
~/Desktop/philip-sat-scores
python plot_scores.py

3 days ago
vim
plot_scores.py~
plot_scores.py

3 days ago
python
sat-scores.png

3 days ago
~/Desktop/philip-sat-scores
python plot_scores.py

3 days ago
vim
plot_scores.py~
plot_scores.py

3 days ago
[scipy.org] Cookbook/Matplotlib/AdjustingImageSize
[google.com] matplotlib graph border - Google Sear
[stackoverflow.com] python - How do I limit the bord
[google.com] matplotlib graph margin - Google Sear
[stackoverflow.com] python - Reduce left and right r

3 days ago
python

/tmp/nilfs-snapshots/2011.12.26-10.46.26/researcher/Deskt

```
  [datetime.datetime(1999, 8, 30),  788, 778],
  [datetime.datetime(1999, 10, 10), 730, 730],
  [datetime.datetime(1999, 11, 14), 888, 798],
  [datetime.datetime(1999, 12, 4),  780, 800]
]

# manually write the date labels for better legibility
date_labels = [
  'Dec 1996',
  '',
  '',
  'Jan 1997',
  '',
  'Sep 1998',
  'Mar 1999',
  'Jul 1999',
  'Sep 1999',
  '',
  '',
  'Dec 1999'
]

x = [date2num(e) for e,_,_ in sat_scores]

#plt.plot([e[0] for e in sat_scores], [e[1] for e in sat_scores
#         'b^:',
#         [e[0] for e in sat_scores], [e[2] for e in sat_scores
#         'gv:')

fig = plt.figure()
graph = fig.add_subplot(111, xlabel='Time', ylabel='Math score'
graph.plot(x, [e[1] for e in sat_scores], 'b^:')

graph.set_xticks(x)
graph.set_xticklabels(date_labels, rotation=40)

plt.xlim((datetime.datetime(1996, 11,1), datetime.datetime(2000
plt.ylim((500, 810))
plt.savefig('sat-scores.png')
```

/tmp/nilfs-snapshots/2011.12.26-11.06.07/researcher/Deskt

```
date_labels = [
  "Dec 1996",
  '',
  '',
  '',
  "Jan 1997",
  "Sep 1998",
  "Mar 1999",
  "Jul 1999",
  "Sep 1999",
  '',
  '',
  "Dec 1999"
]

x = [date2num(e) for e,_,_ in sat_scores]

# add some padding to make the graph less ugly
xlimits = (date2num(datetime.datetime(1996, 11,1)),
           date2num(datetime.datetime(2000,1,1)))
ylimits = (500, 810)

#plt.plot([e[0] for e in sat_scores], [e[1] for e in sat_scores
#         "b^:",
#         [e[0] for e in sat_scores], [e[2] for e in sat_scores
#         "gv:")

fig = plt.figure()
math_graph = fig.add_subplot(121, ylabel="Score", title="SAT ma
math_graph.plot(x, [e[1] for e in sat_scores], "b^:")

verbal_graph = fig.add_subplot(122, ylabel="Score", title="SAT
verbal_graph.plot(x, [e[2] for e in sat_scores], "gv:")

for g in (math_graph, verbal_graph):
  g.set_xticks(x)
  g.set_xticklabels(date_labels, rotation=40)
  g.set_xlim(xlimits)
  g.set_ylim(ylimits)
```

Figure 7.6: Desktop screenshot showing the Activity Feed on the left and the Meld visual diff app in the center displaying a diff of two versions of a Python source file.

and opens the file using `gnome-open`, a utility that dispatches to the appropriate application by file type (e.g., image viewer, text editor).

- **Diff** versions of a text file by launching the Meld visual diff tool (see Figure 7.6). The user can select two old versions of a file to diff, or diff one old version against the current file's contents.

- **Revert** a file to any older version, thus undoing unintended edits. The user can also click on a process name and choose to revert all of the files it modified, which is useful for undoing the effects of, say, a buggy execution that overwrote important data files.

- **Watch** a file for changes. When the user selects a file version to watch, the feed will highlight in red any file modification event that causes that file to differ from the selected version. These selections serve as *regression tests*, which is useful when refactoring to ensure that newer code versions still generate identical output files when re-executed.

- **View the context** surrounding modifications to the chosen file. See Sections 7.4.2 and 7.4.3 for details.

Use case: The Activity Feed subsumes the basic features of version control systems, regression testing frameworks, and sticky notes applets while exposing a unified interface. Rather than restoring files by time or version control commit points, users can access old file versions within the context of all past activities (e.g., executed commands, visited websites, checkpoints, notes).

7.4.2 Computational Context Viewer

The Computational Context Viewer allows researchers to answer a central question in their experimental workflow: *"What effects did changes in my source code files have on my experiment's output files?"*

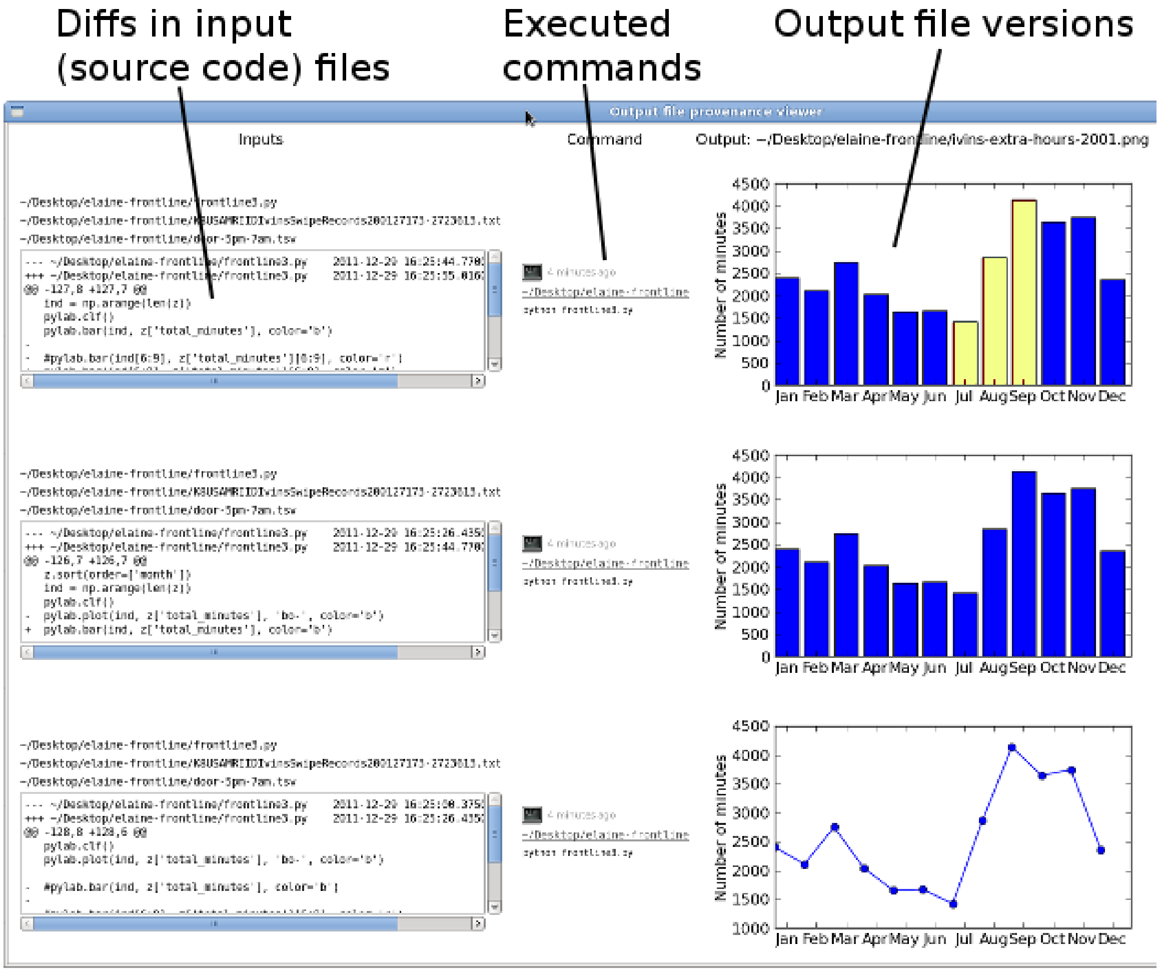
Recall from Chapter 2 that computational experiments consist of repeatedly editing scripts, executing them to produce human-readable output files, and inspecting the outputs to garner insights for the next round of edits. To test variants of their hypotheses, researchers execute scripts repeatedly with different command-line parameters and adjustments to source code. To facilitate comparisons across runs, they often give their script and output files cryptic filenames to indicate which variant each represents (see Figure 7.2). When returning to an experiment at a later date, researchers often forget what exact piece of code generated a particular output table or graph.

The Burrito system solves this common problem by automatically tracking all old versions of source code and output files and providing a user interface to compare and take notes on different variants. Using our system, the user only needs to edit *one* script and generate *one* output file for a given experiment.

To make comparisons across runs, the user launches the Computational Context Viewer GUI with the desired output file (and optional time bound) as an argument. This can be done either from the command-line or from the Activity Feed by clicking on a modified file event.

The Computational Context Viewer displays all versions of the output file in reverse chronological order. It also shows the parameters of the executed command that created each version, thus eliminating the need to encode parameters in cryptic filenames. Finally, the GUI shows the diffs of all source files that led to the creation of the given output file version via the executed command. For example, Figure 7.7 shows three variants of an output graph file generated by a data analysis script: a line graph, a bar graph, and a bar graph with three crucial bars highlighted in yellow. The leftmost column shows the respective diffs in the script file that led to each change in the output graph file (i.e., the code change responsible for turning the line graph into a bar, and then the change for highlighting the three bars in yellow).

Use case: Using this graphical interface, researchers can answer the question, *“Which version of my source code produced the graph that looked like this?”* The inline diff view allows users to focus on significant changes without needing to learn to use a version control system or maintaining multiple copies of files. Once they inspect the source code edits that led to each output, they might wonder what influenced them


Diffs in input (source code) files
Executed commands
Output file versions
Output file provenance viewer
Inputs
Command
Output: ~/Desktop/elaine-frontline/ivins-extra-hours-2001.png
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Number of minutes
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

```
~/Desktop/elaine-frontline/frontline3.py
~/Desktop/elaine-frontline/K8USAMRIIDIvinsSwipeRecords200127173-2723013.txt
~/Desktop/elaine-frontline/door-5pm-7am.tsv

--- ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:44.7780
+++ ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:55.8160
@@ -127,8 +127,7 @@
   ind = np.arange(len(z))
   pylab.clf()
   pylab.bar(ind, z['total_minutes'], color='b')
-
-  #pylab.bar(ind[6:9], z['total_minutes'][6:9], color='r')
```

```
4 minutes ago
~/Desktop/elaine-frontline
python frontline3.py
```

```
~/Desktop/elaine-frontline/frontline3.py
~/Desktop/elaine-frontline/K8USAMRIIDIvinsSwipeRecords200127173-2723013.txt
~/Desktop/elaine-frontline/door-5pm-7am.tsv

--- ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:26.4350
+++ ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:44.7780
@@ -126,7 +126,7 @@
   z.sort(order=['month'])
   ind = np.arange(len(z))
   pylab.clf()
-  pylab.plot(ind, z['total_minutes'], 'bo-', color='b')
+  pylab.bar(ind, z['total_minutes'], color='b')
```

```
4 minutes ago
~/Desktop/elaine-frontline
python frontline3.py
```

```
~/Desktop/elaine-frontline/frontline3.py
~/Desktop/elaine-frontline/K8USAMRIIDIvinsSwipeRecords200127173-2723013.txt
~/Desktop/elaine-frontline/door-5pm-7am.tsv

--- ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:08.3750
+++ ~/Desktop/elaine-frontline/frontline3.py    2011-12-29 16:25:26.4350
@@ -128,8 +128,6 @@
   pylab.clf()
   pylab.plot(ind, z['total_minutes'], 'bo-', color='b')

-  #pylab.bar(ind, z['total_minutes'], color='b')
-
```

```
4 minutes ago
~/Desktop/elaine-frontline
python frontline3.py
```

Figure 7.7: The Computational Context Viewer shows how diffs in input source code files and command-line parameters affect each version of a given output file.

to make these edits. To answer such questions, they can click on a source file and launch the Activity Context Viewer (see next section).

7.4.3 Activity Context Viewer

The Activity Context Viewer allows researchers to answer the following question: *"What actions influenced me to make these edits to my source code?"* Researchers rarely edit code in isolation; while they work, they often consult documentation web pages, related research papers in the form of PDF or Word documents, and other source code. They sometimes write code comments, notes, or hand-drawn sketches to explain the rationale for their edits, but doing so is a tedious manual process.

The design of the Activity Context Viewer was directly inspired by a research prototype called Passages [80], which logs all textual documents read and written by the user and enables contextual searches such as, *"Which papers and web pages did I read when writing the 'related work' section of this paper?"*

The Burrito system automatically captures all of this context surrounding code edits and makes it visible to the user via the Activity Context Viewer. The user launches this application either from the command-line or by selecting a file in the Activity Feed or the Computational Context Viewer. The Activity Context Viewer looks similar to Figure 7.7 where each row represents a version of the selected file. Each row displays four fields:

- **Diffs** of the given source file between the previous and current versions.
- **Resources read** while working on edits to the current version, including web pages visited, on-disk documents viewed, and other source code files read. The user can click on any web page link to open it in a browser and can click on any filename to open the version that existed at the given time.
- **Resources written** while working on this version, including other edited source code files, checkpoints, status updates, and digital sketches drawn.
- **Annotations** that the user can write for this version.

Use case: Researchers can use this tool to re-create their prior work context. For example, *"When I left work last week, I was editing this part of my script and had a collection of reference materials open ... what were they?"*

Determining what resources were read

The Activity Context Viewer determines what resources the user was reading during a given time period by querying the GUI trace for the windows that were active (i.e., in the foreground). If the user switches to a web browser window in the midst of editing the source file (or between browser windows/tabs), then the system adds the currently-displayed web pages to the *resources read* list. If the user switches to another GUI application such as a PDF viewer, then the system looks up the process ID associated with the active window and queries the OS-level provenance graph for the files within `/home` that the associated process has read. Applications open dozens of library files, but usually the only file they open within `/home` is the target resource file (and some easily-ignorable dotfiles). However, if the user opens, say, ten text files in the Vim editor, then the Burrito core does not know which file a user was viewing at a particular time. Our Vim plugin provides the required precision, though, because it logs an event whenever the user switches active documents within Vim.

Determining what constitutes a file version

The Activity Context Viewer subsumes the history view of a version control system, but what constitutes a file version? In a version control system, the user explicitly marks versions using commits. However, since Burrito automatically records all file writes, creating a new version on every write will likely overwhelm the user. Thus, the system only creates a new version when

- another process reads the given source file (*causality-based versioning* [120]),
- the text editor closes the source file,
- the user adds an annotation to a file write event, or
- the user creates a happy/sad checkpoint event.

The Activity Context Viewer improves upon version control system histories by adding edit-level context. For example, users can see what documentation, tutorial, or discussion forum web pages they were consulting when making specific code edits. Thanks to NILFS, users can also click on any file that was read or edited at the same time as the target version to view its contents at the indicated time, even if that file no longer exists.

7.4.4 Lab Notebook Generator

The Lab Notebook Generator creates an HTML file summarizing the user's activities and notes in a given time period. It currently provides the following functionality:

- Feed events are grouped into *phases* separated by user login sessions, checkpoints, and status updates.
- Within a phase, all files read/written are displayed in a directory tree along with any annotations. All Bash commands executed and websites visited are displayed along with any annotations.
- Hashtags within annotations are rendered as hyperlinks that jump to events to which they refer.
- All digital sketches are rendered as inline HTML images, and the user can specify what output files (e.g., data visualizations) to also render as images.

Use cases: Lab notebook HTML files can be archived, shared with colleagues, and used as the basis for writing status reports and preparing notes for student/advisor meetings. Researchers can customize their notebook format by altering the Lab Notebook Generator script.

7.5 Performance Evaluation

While the main contributions of this work are the capabilities offered by the Burrito platform and applications, we wanted to verify that these capabilities do not cause excessive disk space usage or run-time slowdowns.

7.5.1 Disk Space Usage

A machine running Burrito needs to use extra hard disk space to store the master Burrito database and old versions of files in NILFS. Note that raw log files can be deleted as soon as their data are integrated into the master database. To give a

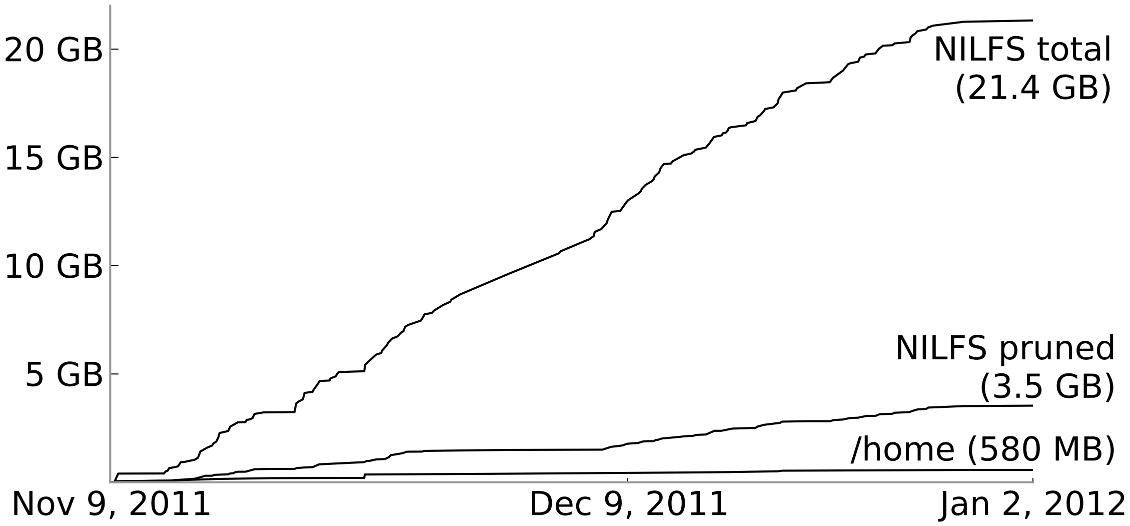


Figure 7.8: Disk space usage over our past two months of development work. "NILFS pruned" is the result of eliminating all NILFS snapshots that changed only dotfiles.

sense of disk space usage growth, we present statistics from the first two months of me developing Burrito while simultaneously running it ("dogfooding") on a Fedora 14 machine. During this time period — Nov. 9, 2011 to Jan. 2, 2012 — I wrote and revised thousands of lines of code and executed thousands of commands to test my prototype ideas.

Our master MongoDB database grew to 799 MB in two months, with the majority of space used to store OS-level provenance (702 MB) and GUI traces (88 MB).

NILFS is responsible for most of the extra disk space usage, since it stores deltas of all old versions of modified files. In this two-month development period, NILFS created 159475 total snapshots, approximately once every time a file was created, deleted, or modified. Figure 7.8 shows the growth of our `/home` directory over time (grows to 580 MB, 7494 inodes) and the extra space taken by the NILFS snapshot deltas (grows to 21.4 GB). Note that NILFS disk space usage is proportional to the rate of file editing activity, so if we had stored some large but never-changing data files in `/home`, then the extra NILFS space usage would remain identical.

When we investigated what kinds of files were being modified in each snapshot, we discovered that the majority of snapshot deltas consisted only of dotfiles and files in dot-directories such as `~/.config/` and `~/.cache/`. These are usually temporary or system configuration files that users do not care about versioning. Thus, we deleted all snapshots that only affected such files and ended up with 31047 snapshots (19% of the total). A daemon could periodically detect and prune these inconsequential snapshots. The "NILFS pruned" line in Figure 7.8 shows that the remaining snapshots only used 3.5 GB of extra space.

Estimating based on these two months of development activity, Burrito uses ∼2 GB of extra disk space per month when dotfile-only snapshots are pruned. More aggressive snapshot pruning [120] can further reduce space usage.

7.5.2 Run-Time Slowdowns

In the first two months that I have been developing and running Burrito full-time on my machine, I have not felt any perceptible lag in my interactive Linux desktop experience. To more objectively quantify the performance impacts of Burrito, though, we ran five diverse benchmarks on a 32-bit PC with four Intel Core i3 2.93GHz processors, 3GB of RAM, and a clean installation of Fedora 14.

We ran each benchmark in the user's home directory under four conditions:

- **Base**line: home directory formatted as the default ext4 filesystem,
- **NILFS**: home directory formatted as NILFS,
- NILFS+**OSprov**: also turns on OS-level provenance collection,
- NILFS+OSprov+**GUI**: also turns on GUI interaction tracing.

Table 7.3 summarizes our results. We now describe details about each benchmark:

- **PDF and image viewing**: We measured the time it took for the Evince PDF reader to open 37 academic paper PDFs from our literature search (63 MB total), and for the GIMP image editor to open 50 JPEG images (5 MB total), respectively. Each opened PDF and JPEG pops up in a new GUI window, which triggers GUI tracer activity. NILFS is slightly faster than the ext4 baseline,

Benchmark	Base	NILFS	+OSprov	+GUI
PDF viewing	10.9s	−1%	5%	12%
Image viewing	35.1s	−8%	−7%	1%
Scientific	65.3s	∼ 0%	∼ 0%	⋆
Unzip/untar	22.1s	20%	22%	⋆
Kernel compile	2810s	2%	12%	⋆

Table 7.3: Run-time slowdowns of Burrito core components relative to baseline, averaged over three executions. ⋆We skip the GUI condition for non-GUI benchmarks.

possibly because these files were all saved to disk at approximately the same time, and the log-structured nature of NILFS leads to slightly better disk layout for reads. Also, since each document loads in fractions of a second, the extra OSprov and GUI overheads are not perceptible.

- **Scientific computing**: We ran a widely-used performance benchmark for the R numerical analysis environment, which exercises matrix and linear algebra operations [19]. Since this benchmark was CPU-bound, running times were nearly identical across conditions.

- **Unzip/untar, Kernel compile**: We measured the time it took to unzip and untar the Linux 3.1.8 kernel source tarball using `bunzip2` and `tar`, respectively. We then compiled the kernel after configuring with default settings. Both of these workloads are CPU- and I/O-heavy.

In sum, most slowdowns were $\leq$ 12%, with the exception of unzip/untar, where NILFS had greater slowdowns from creating new snapshots while writing to 37083 new files.

7.6 Discussion and Future Work

7.6.1 The Burrito Platform

Portability: The Burrito platform consists of open-source software that can be deployed on any modern Linux distribution without needing to compile a custom kernel. In our experiences with the PASS provenance-collecting kernel [119], we found that researchers were reluctant to install a custom kernel on their machines, which made deployment difficult. Also, user-level code is much easier to maintain and debug than kernel code. The hardest Burrito component to install was the NILFS filesystem, but this task should become easier in the future as mainstream Linux distributions include NILFS (or a similar versioning filesystem) as an install-time option. We have installed the entire Burrito platform on Fedora 14 and uploaded a demo virtual machine image online [3].

Security and privacy: The platform core and plugins run with the same permissions as ordinary user processes. All collected data remains on the user's own machine (assuming plugins are not malicious). Designing policies for securing and sharing traces is beyond the scope of this work, although it is interesting for future work.

Limitations: Burrito is not designed to support full reproducibility of previous computations. Although it can roll back file versions to re-execute old experiments, it does not capture environment variables or external sources of non-determinism. Its main goal is to provide experimental documentation, not full reproducibility. Some related projects aim to provide full reproducibility: DejaView [102] augments Linux with a virtual display driver and virtual execution environment to efficiently capture a rich history of user activity and enable execution to restore to any time in the past. Others have augmented virtual machines with fine-grained checkpoints to enable a similar form of "time travel" [100, 150]. Unlike these systems, Burrito requires no virtualization and also integrates OS-level provenance, plugins, and annotations.

Also, Burrito assumes that all components are running on a single machine with one system clock. Adapting Burrito for a distributed (e.g., cluster or multi-user) environment is an interesting area for future research. One could imagine extending Burrito to support real-time sharing of traces in a collaborative work environment.

Possible extensions: Our colleagues have suggested many different types of plugins to the Burrito core. At one extreme, someone even suggested a "big brother" plugin that attaches an electronic eye tracker to the user's monitor, so that Burrito knows exactly which windows the user is viewing at all times. Another possible plugin adds cryptographic signing and uploading of traces to a trusted server, so that it can serve as an official audit trail for, say, intellectual property disputes over novel algorithms or scientific discoveries. Finally, although Burrito can subsume traditional version control systems, to become even more useful for programmers already using version control, Burrito could be extended to exploit the metadata present in version control repositories.

Broader applications: Burrito is useful even for non-programmers. People who engage in "information foraging" tasks, such as journalists, marketing analysts, administrative staff, and humanities researchers, can benefit from its workflow tracking and in-context notetaking features.

For example, if a government intelligence analyst is consulting dozens of web pages, downloading hundreds of PDF documents, and copying and pasting text snippets together into a PowerPoint presentation, he/she can use Burrito to answer provenance queries such as "Which PDF files was I reading when writing up this specific part of my presentation?" or "Which PDFs did I pay particularly close attention to last week, as indicated by how much time I spent reading them?"

7.6.2 Burrito Applications

In this work, we built initial prototypes of four applications to show the potential of the Burrito platform (Section 7.4), but we are not claiming that these are the optimal interfaces for documenting or querying computational experiments.

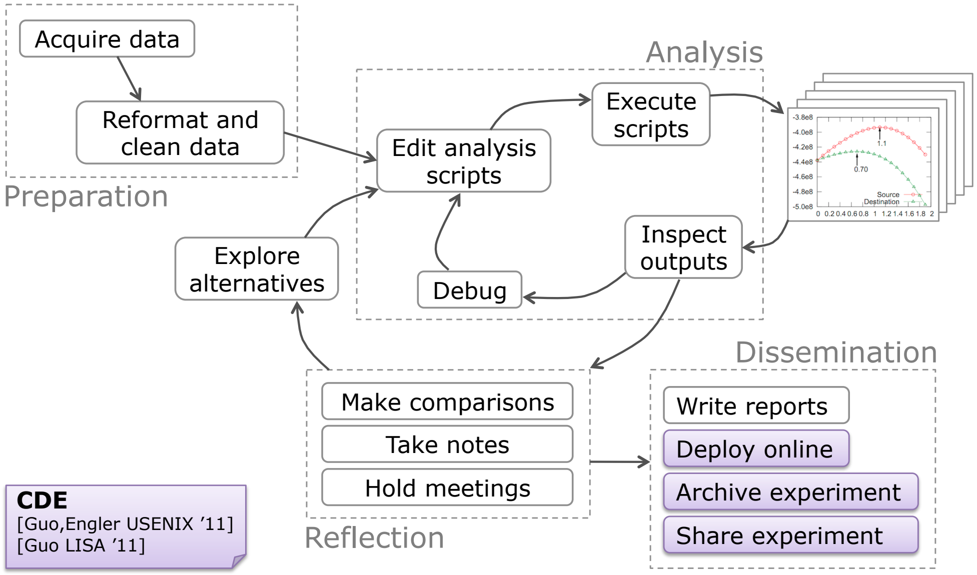
Our colleagues to whom we demonstrated Burrito were mainly concerned about possible information overload arising from recording too much data and automatically maintaining too many file versions. Oftentimes, a user is working on several distinct activities concurrently (e.g., a mix of personal emailing/web browsing and work-related programming), so it is important to tease those apart into parallel streams. Thus, they suggested improvements such as inferring/clustering activity patterns, filtering/ranking relevant context, and even incorporating data from other plugins (e.g., audio, clipboard) to bring more structure to the Burrito timestream.

Our colleagues also suggested building other Burrito applications such as a personal productivity monitor [20] and a contextual file search tool [145]. Some power users wanted a command-line interface where they could make queries such as, *"Find me all old versions of script X that generated an output file containing $\geq$ 50 data points with an error value $\leq$ 0.05."* An Emacs enthusiast wanted to write an Emacs extension that provides the notification and annotation features of the Activity Feed (Section 7.4.1) completely within Emacs.

The Burrito platform is complete enough to support building any of the aforementioned applications. The enthusiasm expressed by our early users and their ability to quickly suggest additional useful applications give some indication that the platform captures the appropriate amount of context and exposes the proper interfaces. However, these insights are purely anecdotal. We have not dealt with HCI or design-related issues in this work, which has only focused on the systems and platform aspects of Burrito. We leave a full usability evaluation of Burrito for future work.

Chapter 8

CDE: Deploy and Reproduce Experiments



This chapter presents a software packaging tool called CDE, which helps researchers deploy, archive, and share their experiments. Its contents were adapted from a 2011 conference paper that I co-authored with Dawson Engler [74] and a follow-up paper that describes several new features [70]. All uses of “we”, “our”, and “us” in this chapter refer to both authors.

8.1 Motivation

The simple-sounding task of taking software that runs on one person's machine and getting it to run on another machine can be painfully difficult in practice, even if both machines have the same operating system. Since no two machines are identically configured, it is hard for developers to predict the exact versions of software and libraries already installed on potential users' machines and whether those conflict with the requirements of their own software. Thus, software companies devote considerable resources to creating and testing one-click installers for products such as Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers must carefully specify the proper dependencies in order to integrate their software into package management systems [11] (e.g., RPM on Linux, MacPorts on Mac OS X). Despite these efforts, online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring software and dependencies.

In particular, research programmers are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their job is not to release production-quality software. Instead, they usually "release" their software by uploading their source code and data files to a server and maybe writing up some informal installation instructions. There is a slim chance that their colleagues will be able to run their research code "out-of-the-box" without some technical support.

8.2 CDE System Overview

In this chapter, we present a tool called CDE [4] that makes it easy for people such as research programmers to get their software running on other machines without the hassle of manually creating a robust installer or dealing with user complaints about dependencies. CDE automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines without any installation (see Figure 8.1). To use CDE, the user simply:

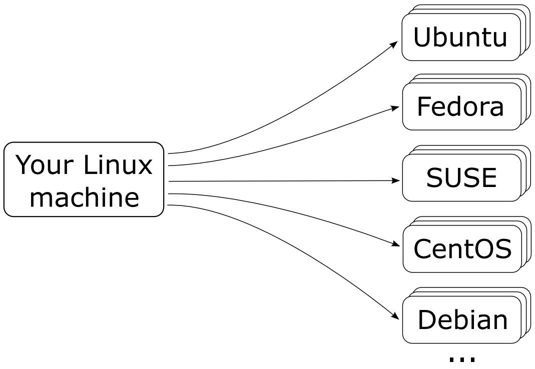


Figure 8.1: CDE enables users to package up any Linux application and deploy it to all modern Linux distros.

1. Prepends any set of Linux commands with the `cde` executable. `cde` executes the commands and uses `ptrace` system call interposition to collect all code, data, and environment variables used during execution into a self-contained package.

2. Copies the resulting CDE package to an x86-Linux machine running any distro from the past ∼5 years.

3. Prepends the original packaged commands with the `cde-exec` executable to run them on the target machine. `cde-exec` uses `ptrace` to redirect file-related system calls so that executables can load the required dependencies from within the package. Execution can range from ∼0% to ∼30% slower.

The main benefits of CDE are that creating a package is as easy as executing the target program under its supervision, and that running a program within a package requires no installation, configuration, or root permissions.

In addition, CDE offers an *application streaming mode*, described in Section 8.6. Figure 8.2 shows its high-level architecture: The system administrator first installs multiple versions of many popular Linux distros in a “distro farm” in the cloud (or an internal compute cluster). The user connects to that distro farm via an ssh-based protocol from any x86-Linux machine. The user can now run *any* application available within the package managers of any of the distros in the farm. CDE’s streaming mode fetches the required files on-demand, caches them locally on the user’s machine,

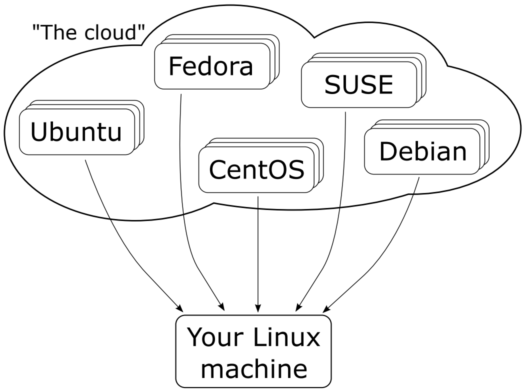


Figure 8.2: CDE's streaming mode enables users to run any Linux application on-demand by fetching the required files from a farm of pre-installed distros in the cloud.

and creates a portable distro-independent execution environment. Thus, Linux users can instantly run the hundreds of thousands of applications already available in the package managers of all distros without being forced to use one specific release of one specific distro[1].

We will use an example to introduce the core features of CDE. Suppose that Alice is a climate scientist whose experiment involves running a Python weather simulation script on a Tokyo dataset using this Linux command:

```
python weather_sim.py tokyo.dat
```

Alice's script (`weather_sim.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ numerical analysis code compiled into shared libraries. If Alice wants her colleague Bob to run and build upon her experiment, then it is not sufficient to just send her script and `tokyo.dat` data file to him. Even if Bob has a compatible version of Python on his machine, he will not be able to run her script until he compiles, installs, and configures the extension modules that she used (and all of their transitive dependencies).

[1] The package managers included in different releases of the same Linux distro often contain incompatible versions of many applications!

8.2.1 Creating a New Package With cde

To create a self-contained package with all dependencies required to run her experiment on another machine, Alice prepends her command with the cde executable:

```
cde python weather_sim.py tokyo.dat
```

cde runs her command normally and uses the Linux ptrace mechanism to monitor all files it accesses throughout execution. cde creates a new sub-directory called cde-package/cde-root/ and copies all of those accessed files into there, mirroring the original directory structure. For example, if her script dynamically loads an extension module (shared library) named /usr/lib/weather.so, then cde will copy it to cde-package/cde-root/usr/lib/weather.so (see Figure 8.3). cde also saves the values of environment variables in a file within cde-package/.

When execution terminates, the cde-package/ sub-directory (which we call a "CDE package") contains all of the files required to run Alice's original command.

8.2.2 Executing a Package With cde-exec

Alice zips up the cde-package/ directory and transfers it to Bob's Linux machine. Now Bob can run Alice's experiment without installing anything on his machine. He unzips the package, changes into the sub-directory containing the script, and prepends the original command with the cde-exec executable (also in the package):

```
cde-exec python weather_sim.py tokyo.dat
```

cde-exec sets up the environment variables saved from Alice's machine and executes the version of python and its extension modules from within the package. cde-exec uses ptrace to monitor all system calls that access files and rewrites their path arguments to the corresponding paths within the cde-package/cde-root/ sub-directory. For example, when her script requests to load the /usr/lib/weather.so extension library using an open system call, cde-exec rewrites the path argument of the open call to cde-package/cde-root/usr/lib/weather.so (see Figure 8.3). This

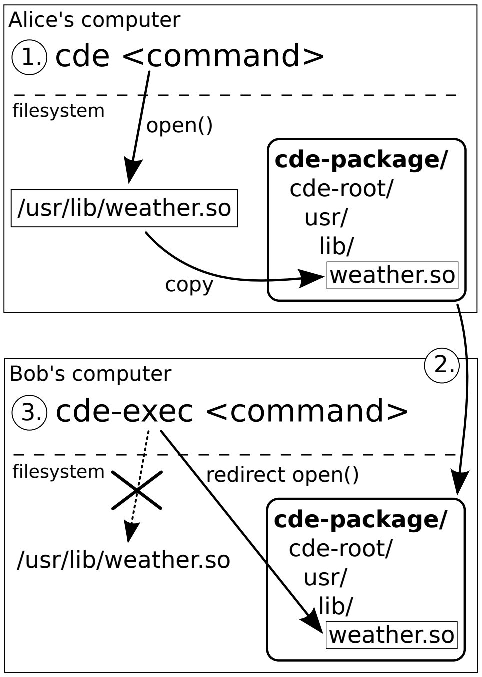


Figure 8.3: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends her package to Bob's computer, 3.) Bob runs that same command with `cde-exec`, which redirects file accesses into the package.

path redirection is essential, because `/usr/lib/weather.so` probably does not exist on Bob's machine.

Not only can Bob reproduce Alice's exact experiment, but he can also edit her script and dataset and then re-run to explore variations and alternative hypotheses, as long as he does not cause the script to import new Python extension modules that are not in the package. Also, since a CDE package is a directory tree, Bob can add additional dataset files into the package to run related experiments.

8.2.3 CDE Package Portability

Alice's CDE package can execute on any Linux machine with an architecture and kernel version that are compatible with its constituent binaries. CDE currently works on 32-bit and 64-bit variants of the x86 architecture (i386 and x86-64, respectively).

In general, a 32-bit `cde-exec` can execute 32-bit packaged applications on 32- and 64-bit machines. A 64-bit `cde-exec` can execute both 32-bit and 64-bit packaged applications on a 64-bit machine. Extending CDE to other architectures (e.g., ARM) is straightforward because the `strace` tool that CDE is built upon already works on many architectures. However, CDE packages cannot be transported *across* architectures without using a CPU emulator.

Our portability experiments (Section 8.8.1) show that packages are portable across Linux distros released within 5 years of the distro where the package originated. Besides sharing with colleagues such as Bob, Alice can also deploy her package to run on a cluster for more computational power or to a public-facing server machine for real-time online monitoring. Since she does not need to install anything as root, she does not risk perturbing existing software on those machines. Also, having her script and all of its dependencies (including the Python interpreter and extension modules) encapsulated within a CDE package makes it somewhat "future-proof" and likely to continue working on her machine even when its version of Python and associated extensions are upgraded in the future.

8.3 Design and Implementation

CDE uses the Linux `ptrace` system call to monitor the target program's processes and threads, read/write to its memory, and modify its system call arguments, all without requiring root permission. System call interposition using `ptrace` is a well-known technique that researchers have used for implementing tools such as secure sandboxes [67, 94], record-replay systems [103], and user-level filesystems [146].

We implemented CDE by adding 3000 lines of C code to the `strace` system call monitoring tool. CDE only works on x86-based Linux machines (32-bit and 64-bit) but should be easy to extend to other hardware architectures. Although implementation details are Linux-specific, these ideas could be used to implement CDE for another OS such as Mac OS X or Windows.

Category	Linux syscalls	`cde` action	`cde-exec` action
File path access	`open[at],mknod[at],fstatat64,access,faccessat,readlink[at],truncate[64],stat[64],creat,lstat[64],oldstat,oldlstat,chown[32],lchown[32],fchownat,chmod,fchmodat,utime,utimes,futimesat`	Copy file into package	Redirect path into package
Local sockets	`bind,connect`	None	Redirect path into package[†]
Mutate filesystem	`link[at],symlink[at],rename[at],unlink[at],mkdir[at],rmdir`	Repeat in package	Redirect path into package
Get current dir.	`getcwd`	Update current directory	Spoof current directory
Change directory	`chdir,fchdir`	Update current directory	
Spawn child	`fork,vfork,clone`	Track child process or thread	
Execute program	`execve`	Copy binary into package	Maybe run dynamic linker

Table 8.1: The 48 Linux system calls intercepted by `cde` and `cde-exec`, and actions taken for each category of syscalls. Syscalls with suffixes in [brackets] include variants both with and without the suffix: e.g., `open[at]` means `open` and `openat`. [†]For `bind` and `connect`, `cde-exec` only redirects the path if it is used to access a file-based socket for local IPC.

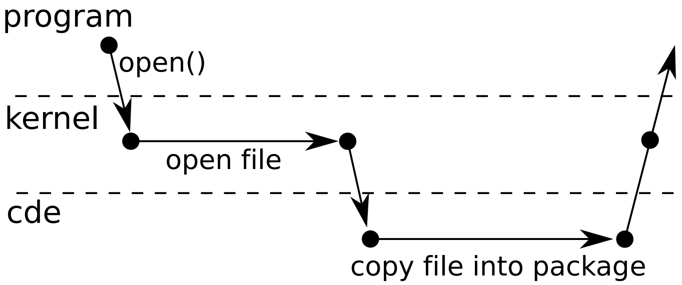


Figure 8.4: Timeline of control flow between the target program, kernel, and `cde` process during an `open` syscall.

8.3.1 Creating a New Package With `cde`

Primary action: The main job of `cde` is to use `ptrace` to monitor the target program's system calls and copy all of its accessed files into a self-contained package. `cde` only cares about the subset of syscalls that take a file path string as an argument, which are listed in the "File path access" category in Table 8.1 (and also `execve`). After the kernel finishes executing one of these syscalls and is about to return to the target program, `cde` wakes and observes the return value. If the return value signifies that the indicated file exists, then `cde` copies that file into the package (Figure 8.4).

Note that many syscalls operate on files but take a file descriptor as an argument rather than a file path (e.g., `mmap`); `cde` does not need to track those, since it already tracks the calls that create file descriptors from file paths.

Copying files into package: Prior to copying a file into the package, `cde` creates all necessary sub-directories and symbolic links to mirror the original file's location. In our example, `cde` will copy `/usr/lib/weather.so` into the package as `cde-package/cde-root/usr/lib/weather.so` (Figure 8.3). For efficiency, copies are done via hard links if possible.

If a file is a symlink, then both it and its target must be copied into the package. Multiple levels of symlinks, to both files and directories, must be properly handled. More subtly, *any component* of a path may be a symlink to a directory, so the exact directory structure must be replicated within the package. For example, we once encountered a path `/usr/lib/gcc/4.1.2/libgcc.a`, where `4.1.2` is a symlink to a directory named `4.1.1`. We observed that some programs are sensitive to exact filesystem layout, so `cde` must faithfully replicate symlinks within the package, or else those programs will fail with cryptic errors when run from within the package. See Section 8.4.1 for more details about the mechanics of deep copying.

Finally, if the file being copied is an ELF binary (executable or library code), then `cde` searches through the binary's contents for constant strings that are filenames and then copies those files into the package. Although this hack is simplistic, it works well in practice to partially overcome CDE's limitation of only being able to gather dependencies on executed paths. It works because many binaries dynamically load libraries whose filenames are constant strings. For example, we encountered a Python extension library that dynamically loads one of a few versions of the Intel Math Kernel Library based on the current CPU's capabilities. Without this hack, any given execution will only copy *one* version of the Intel library into the package, so packaged execution will fail when running on another machine with different CPU capabilities. Finding and copying all versions of the Intel library into the package makes the program more likely to run on machines with different hardware.

Here is how `cde` handles the other syscalls in Table 8.1:

Mutate filesystem: After each call that mutates the filesystem, `cde` repeats the same action on the corresponding copies of files in the package. For example, if a program renames a file from `foo` to `bar`, then `cde` also renames the copy of `foo` in the package to `bar`. This way, at the end of execution, the package's contents mirror the "post-state" of the original filesystem's contents, not the "pre-state" before execution.

Updating current working directory: At the completion of `getcwd`, `chdir`, and `fchdir`, `cde` updates its record of the monitored process's current working directory, which is necessary for resolving relative paths.

Tracking sub-processes and threads: If the target program spawns sub-processes, `cde` also attaches onto those children with `ptrace` (it attaches onto spawned threads in the same way). `cde` keeps track of each monitored process's current working directory and shared memory segment address (needed for Section 8.3.2). `cde` remains single-threaded and responds to events queued by `ptrace`.

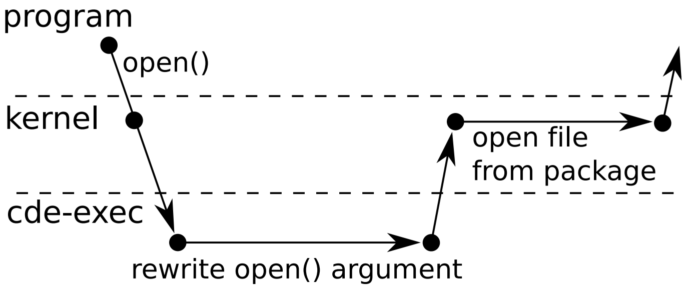


Figure 8.5: Timeline of control flow between the target program, kernel, and `cde-exec` process during an `open` syscall.

This feature is useful for packaging up workflows consisting of multiple program invocations, such as a compilation job. For example, running "`cde make`" will track all sub-processes that the Makefile spawns and package up the source files and compiler toolchain. Now you can edit and compile the given project on another Linux machine by simply running "`cde-exec make`", without needing to install any compilation tools or header files on that machine.

execve: `cde` copies the executable's binary into the package. For a script, `cde` finds the name of its interpreter binary from the shebang (`#!`) line. If the binary is dynamically-linked, `cde` also finds its dynamic linker (e.g., `ld-linux.so.2`) and copies it into the package. The dynamic linker is responsible for loading the shared libraries that a program needs at start-up time.

8.3.2 Executing a Package With `cde-exec`

Primary action: The main job of `cde-exec` is to use `ptrace` to redirect file paths that the target program requests into the package. Before the kernel executes most syscalls listed in Table 8.1, `cde-exec` rewrites their path argument(s) to refer to the corresponding path within `cde-package/cde-root/` (Figure 8.5). By doing so, `cde-exec` creates a chroot-like sandbox that fools the target program into "believing" that it is executing on the original machine. Unlike chroot, this sandbox does not require root access to set up, and it is user-customizable (see Section 8.3.3).

In our example, suppose that Alice runs her experiment within the `/expt` directory on her computer:

```
cd /expt
cde python weather_sim.py tokyo.dat
```

She then sends the package to Bob's computer. If Bob unzips it into his home directory (/home/bob), then he can run these commands to execute her Python script:

```
cd /home/bob/cde-package/cde-root/expt
cde-exec python weather_sim.py tokyo.dat
```

Note that Bob needs to first change into the /expt sub-directory within the package, since that is where Alice's scripts and data files reside. When cde-exec starts, it finds Alice's python executable within the package (with the help of $PATH) and launches it. Now if her program requests to open, say, /usr/lib/weather.so, cde-exec rewrites the path argument of the open call to /home/bob/cde-package/cde-root/usr/lib/weather.so, so that the kernel opens the library version within the package.

Implementing syscall rewriting: Since ptrace allows cde-exec to directly read and write into the target program's memory, the easiest way to rewrite a syscall's argument is to simply override its buffer with a new string. However, this approach does not work because the new path string is always longer than the original, so it might overflow the buffer. Also, if the program makes a system call with a constant string, the buffer would be read-only.

Instead, what cde-exec does is redirect the *pointer* to the buffer. When the target program (or one of its sub-processes) first makes a syscall, cde-exec forces it to make another syscall to attach a 16KB shared memory segment (a trick from [146]). Now cde-exec can write data into that shared segment and have it be visible in the target program's address space. The two large rectangles in Figure 8.6 show the address spaces of the target program and cde-exec, respectively. Figure 8.6 illustrates the three steps involved in syscall argument rewriting:

1. cde-exec uses ptrace to read the original argument from the traced program's address space.

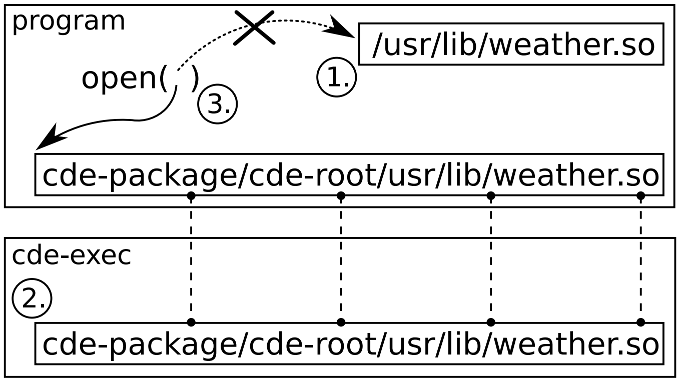

program
open( )
1.
3.
/usr/lib/weather.so
cde-package/cde-root/usr/lib/weather.so
cde-exec
2.
cde-package/cde-root/usr/lib/weather.so

Figure 8.6: Example address spaces of target program and `cde-exec` when rewriting path argument of `open`. The two boxes connected by dotted lines are shared memory.

2. `cde-exec` creates a new string representing the path redirected inside of the package and writes it into the shared memory buffer. This value is immediately visible in the target program's address space.

3. `cde-exec` uses `ptrace` to mutate the syscall's argument to point to the start of the shared memory buffer (in the target program's address space). x86-Linux syscall arguments are stored in registers, so `ptrace` mutates the target program's registers prior to executing the call. Most syscalls only take one filename argument, which is stored in `%ebx` on i386 and `%rdi` on x86-64. Syscalls such as `link`, `symlink`, and `rename` take two filename arguments; their second argument is stored in `%ecx` on i386 and `%rsi` on x86-64.

Spoofing current working directory: At the completion of the `getcwd` syscall, `cde-exec` mutates the return value string to eliminate all path components up to `cde-root/`. For example, when Bob runs Alice's script:

```
cd /home/bob/cde-package/cde-root/expt
cde-exec python weather_sim.py tokyo.dat
```

If her Python script requests its current working directory using `getcwd`, the kernel will return the true full path: `/home/bob/cde-package/cde-root/expt`. Then `cde-exec` will truncate that string so that it becomes `/expt`, which is the value it would have returned if it were running on Alice's machine. We have encountered many programs that break when `getcwd` is not spoofed.

There is no danger of buffer overflow here since the new string is always shorter, and the buffer cannot be read-only, since the kernel must be able to mutate it. Some programs call `readlink("/proc/self/cwd")` to get the current working directory, so we also spoof the return value for that particular syscall instance.

execve: When the target program executes a dynamically-linked binary, `cde-exec` rewrites the `execve` syscall arguments to execute the dynamic linker stored in the package (with the binary as its first argument) rather than directly executing the binary. For example, if Bob invokes "`cde-exec python weather_sim.py tokyo.dat`", `cde-exec` will prepend the dynamic linker filename to the `argv` array argument of the `execve` syscall:

```
argv[0]: cde-package/cde-root/lib/ld-linux.so.2
argv[1]: /usr/bin/python
argv[2]: weather_sim.py
argv[3]: tokyo.dat
```

(Also note that although `argv[1]` is `/usr/bin/python`, that path will get redirected into the version of the binary file within the package during a subsequent `open` syscall.)

Here is why `cde-exec` needs to explicitly execute the dynamic linker: When a user executes a dynamically-linked binary, Linux first executes the system's default dynamic linker to resolve and load its shared libraries. However, we have found that the dynamic linker on one Linux distro might not be compatible with binaries created on another distro, due to minor differences in ELF binary formats. Therefore, to maximize portability across machines, `cde` copies the dynamic linker into the package, and `cde-exec` executes the dynamic linker from the package rather than having Linux execute the system's version. Without this hack, we have noticed that even a trivial "hello world" binary compiled on one distro (e.g., Ubuntu with Linux 2.6.35) will not run on an older distro (e.g., Knoppix with Linux 2.6.17)[2].

[2] It actually crashes with a cryptic "Floating point exception" error message!

A side-effect of rewriting `execve` to call the dynamic linker is that when a target program inspects its own executable name, the kernel will return the name of the dynamic linker, which is incorrect. Thus, `cde-exec` spoofs the return values of calls to `readlink("/proc/self/exe")` and `readlink("/proc/<$PID>/exe")` to return the original executable's name. This spoofing is necessary because some narcissistic programs crash with cryptic errors if their own names are not properly identified.

8.3.3 Ignoring Files and Environment Variables

By convention, Linux directories such as `/dev`, `/proc`, and `/sys` contain pseudo-files (e.g., device files) that do not make sense to include in a CDE package. Also, environment variables such as `$XAUTHORITY` and the corresponding `.Xauthority` file (for X Window authorization) are machine-specific. Informed by our debugging experiences and user feedback, we have manually created a blacklist of directories, files, and environment variables for CDE to ignore, so that packages can be portable across machines. By "ignore" we mean that `cde` will not copy those files (or variables) into a package, and `cde-exec` will not redirect their paths and instead access the real versions on the machine. This user-customizable blacklist is implemented as a plain-text options file. Figure 8.7 shows this file's default contents.

CDE also allows users to customize which paths it should ignore (leave alone) and which it should redirect into the package, thereby making its sandbox "semi-permeable". For example, one user chose to have CDE ignore a directory that mounts an NFS share containing huge data files, because he knew that the machine on which he was going to execute the package also mounts that NFS share at the same path. Therefore, there was no point in bloating up the package with those data files.

8.3.4 Non-Goals

CDE only intercepts 14% of all Linux 2.6 syscalls (48 out of 338), but those are sufficient for creating portable self-contained Linux packages. CDE does not need to intercept more syscalls because it is not designed to perform:

```
# These directories often contain pseudo-files that shouldn't be tracked
ignore_prefix=/dev/
ignore_exact=/dev
ignore_prefix=/proc/
ignore_exact=/proc
ignore_prefix=/sys/
ignore_exact=/sys
ignore_prefix=/var/cache/
ignore_prefix=/var/lock/
ignore_prefix=/var/log/
ignore_prefix=/var/run/
ignore_prefix=/var/tmp/
ignore_prefix=/tmp/
ignore_exact=/tmp

ignore_substr=.Xauthority     # Ignore to allow X Window programs to work

ignore_exact=/etc/resolv.conf # Ignore so networking can work properly

# Access the target machine's password files:
# (some programs like texmacs need these lines to be commented-out,
#  since they try to use home directory paths within the passwd file,
#  and those paths might not exist within the package.)
ignore_prefix=/etc/passwd
ignore_prefix=/etc/shadow

# These environment vars might lead to 'overfitting' and hinder portability
ignore_environment_var=DBUS_SESSION_BUS_ADDRESS
ignore_environment_var=ORBIT_SOCKETDIR
ignore_environment_var=SESSION_MANAGER
ignore_environment_var=XAUTHORITY
ignore_environment_var=DISPLAY
```

Figure 8.7: The default CDE options file, which specifies the file paths and environment variables that CDE should ignore. `ignore_exact` matches an exact file path, `ignore_prefix` matches a path's prefix string (e.g., directory name), and `ignore_substr` matches a substring within a path. Users can customize this file to tune CDE's sandboxing policies (see Section 8.3.3).

- **Deterministic replay**: CDE does not try to replay exact execution paths like record-replay tools [37, 103, 138] do. Thus, CDE does not need to capture sources of randomness, thread scheduling, and other non-determinism. It also does not need to create snapshots of filesystem state for rollback/recovery.
- **OS/hardware emulation**: CDE does not spoof the OS or hardware. Thus, programs that require specialized hardware or device drivers will not be portable across machines. Also, CDE cannot capture remote network dependencies.
- **Security**: Although CDE isolates target programs in a chroot-like sandbox, it does not guard against attacks to circumvent such sandboxes [66]. Users should only run CDE packages from trusted sources[3].
- **Licensing**: CDE does not attempt to "crack" software licenses, nor does it enforce licensing or distribution restrictions. It is ultimately the package creator's responsibility to make sure that he/she is both willing and able to distribute the files within a package, abiding by the proper software and data set licenses.

8.4 Semi-Automated Package Completion

CDE's primary limitation is that it can only package up files accessed on executed program paths. Thus, programs run from within a CDE package will fail when executing paths that access new files (e.g., libraries, configuration files) that the original execution(s) did not access.

Unfortunately, *no automatic tool* (static or dynamic) can find and package up all the files required to successfully execute all possible program paths, since that problem is undecidable in general. Similarly, it is also impossible to automatically quantify how "complete" a CDE package is or determine what files are missing, since every file-related system call instruction could be invoked with complex or non-deterministic arguments. For example, the Python interpreter executable has only one `dlopen` call site for dynamically loading extension modules, but that `dlopen` could

[3] Of course, the same warning applies to *all* downloaded software.

be called many times with different dynamically-generated string arguments derived from script variables or configuration files.

There are two ways to cope with this package incompleteness problem. First, if the user executes additional program paths, then CDE will add new files into the same `cde-package/` directory. However, making repeated executions can get tedious, and it is unclear how many or which paths are necessary to complete the package[4].

Another way to make CDE packages more complete is by manually copying additional files and sub-directories into `cde-package/cde-root/`. For example, while executing a Python script, CDE might automatically copy the few Python standard library files it accesses into, say, `cde-package/cde-root/usr/lib/python/`. To complete the package, the user could copy the entire `/usr/lib/python/` directory into `cde-package/cde-root/` so that *all* Python libraries are present. A user can usually make his/her package complete by copying only a few crucial directories into the package, since programs store all of their files in several top-level directories (see Section 8.4.3).

However, programs also depend on shared libraries that reside in system-wide directories such as `/lib` and `/usr/lib`. Copying all the contents of those directories into a package results in lots of wasted disk space. In Section 8.4.2, we present an automatic heuristic technique that finds nearly all shared libraries that a program requires and copies them into the package.

8.4.1 The OKAPI Utility for Deep File Copying

Before describing our heuristics for completing CDE packages, we first introduce a utility library we built called OKAPI (pronounced "*oh-copy*"), which performs detailed copying of files, directories, and symlinks. OKAPI does one seemingly-simple task that turns out to be tricky in practice: copying a filesystem entity (i.e., a file, directory, or symlink) from one directory to another while fully preserving its original sub-directory and symlink structure (a process that we call *deep-copying*). CDE uses OKAPI to copy files into the `cde-root/` sub-directory when creating a new package,

[4] similar to trying to achieve 100% coverage during software testing

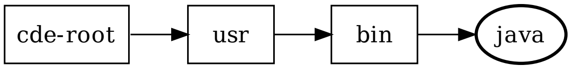


Figure 8.8: The result of copying a file named `/usr/bin/java` into `cde-root/`.

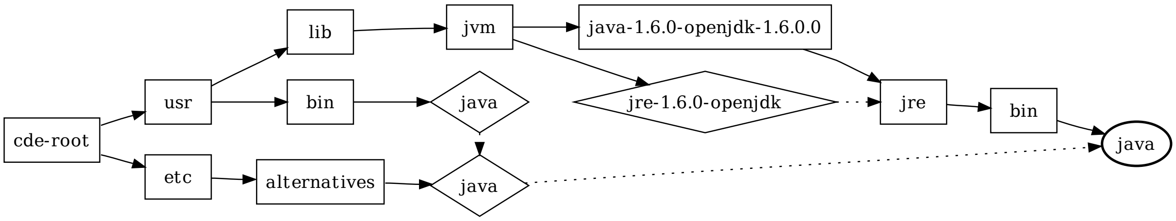


Figure 8.9: The result of using OKAPI to deep-copy a single `/usr/bin/java` file into `cde-root/`, preserving the exact symlink structure from the original directory tree. Boxes are directories (solid arrows point to their contents), diamonds are symlinks (dashed arrows point to their targets), and the bold ellipse is the real `java` executable.

and the support scripts of Sections 8.4.2 and 8.4.3 also use OKAPI.

For example, suppose that CDE needs to copy the `/usr/bin/java` executable file into `cde-root/` when it is packaging a Java application. The straightforward way to do this is to use the standard `mkdir` and `cp` utilities. Figure 8.8 shows the resulting sub-directory structure within `cde-root/`, with the boxes representing directories and the bold ellipse representing the copy of the `java` executable file located at `cde-root/usr/bin/java`. However, it turns out that if CDE were to use this straightforward copying method, the Java application would *fail to run* from within the CDE package! This failure occurs because the `java` executable introspects its own path and uses it as the search path for finding the Java standard libraries. On our Fedora Core 9 machine, the Java standard libraries are actually installed in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0`, so when `java` reads its own path as `/usr/bin/java`, it cannot possibly use that path to find its standard libraries.

In order for Java applications to properly run from within CDE packages, all of their constituent files must be "deep-copied" into the package while replicating their original sub-directory and symlink structures. Figure 8.9 illustrates the complexity of deep-copying a single file, `/usr/bin/java`, into `cde-root/`. The diamond-shaped nodes represent symlinks, and the dashed arrows point to their targets. Notice how `/usr/bin/java` is a symlink to `/etc/alternatives/java`, which is itself a symlink to `/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`. Another complicating factor is that `/usr/lib/jvm/jre-1.6.0-openjdk` is itself a symlink to the `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/` directory, so the actual `java` executable resides in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/`. Java can only find its standard libraries when these paths are all faithfully replicated within the CDE package.

The okapi utility library automatically performs the deep-copying required to generate the filesystem structure of Figure 8.9. Its interface is as simple as ordinary `cp`: The caller simply requests for a path to be copied into a target directory, and okapi faithfully replicates the sub-directory and symlink structure.

okapi performs one additional task: rewriting the contents of symlinks to transform absolute path targets into relative path targets within the destination directory (e.g., `cde-root/`). In our example, `/usr/bin/java` is a symlink to `/etc/alternatives/java`. However, okapi cannot simply create the `cde-root/usr/bin/java` symlink to also point to `/etc/alternatives/java`, since that target path is outside of `cde-root/`. Instead, okapi must rewrite the symlink target so that it actually refers to `../../etc/alternatives/java`, which is a relative path that points to `cde-root/etc/alternatives/java`.

The details of this particular example are not important, but the high-level message that Figure 8.9 conveys is that deep-copying even a single file can lead to the creation of over a dozen sub-directories and (possibly-rewritten) symlinks. The problem that okapi solves is not Java-specific; we have observed that many real-world Linux applications fail to run from within CDE packages unless their files are deep-copied in this detailed way.

okapi is also available as a free standalone command-line tool [4]. To our knowledge, no other Linux file copying tool (e.g., `cp`, `rsync`) can perform the deep-copying and symlink rewriting that okapi does.

8.4.2 Heuristics for Copying Shared Libraries

When Linux starts executing a dynamically-linked executable, the dynamic linker (e.g., `ld-linux*.so*`) finds and loads all shared libraries that are listed in a special `.dynamic` section within the executable file. Running the `ldd` command on the executable shows these start-up library dependencies. When CDE is executing a target program to create a package, CDE finds all of these dependencies as well because they are loaded at start-up time via `open` system calls.

However, programs sometimes load shared libraries in the middle of execution using, say, the `dlopen` function. This run-time loading occurs mostly in GUI programs with a plug-in or extension architecture. For example, when the user instructs Firefox to visit a web page with a Flash animation, Firefox will use `dlopen` to load the Adobe Flash Player shared library. `ldd` will not find that dependency since it is not hard-coded in the `.dynamic` section of the Firefox executable, and CDE will only find that dependency if the user actually visits a Flash-enabled web page while creating a package for Firefox.

We have created a simple heuristic-based script that finds most or all shared libraries that a program requires[5]. The user first creates a base CDE package by executing the target program once (or a few times) and then runs our script, which works as follows:

1. Find all ELF binaries (executables and shared libraries) within the package using the Linux `find` and `file` utilities.

2. For each binary, find all constant strings using the `strings` utility, and look for strings containing "`.so`" since those are likely to be shared libraries.

3. Call the `locate` utility on each candidate shared library string, which returns the *full absolute paths* of all installed shared libraries that match each string.

4. Use OKAPI to copy each library into the package.

5. Repeat this process until no new libraries are found.

[5] always a superset of the shared libraries that `ldd` finds

This heuristic technique works well in practice because programs often list all of their dependent shared libraries in string *constants* within their binaries. The main exception occurs in dynamic languages such as Python or MATLAB, whose programs often dynamically generate shared library paths based on the contents of scripts and configuration files. Of course, our technique provides no completeness guarantees since the package completeness problem is undecidable in general.

Another limitation of this technique is that it is overly conservative and can create larger-than-needed packages, since the `locate` utility can find more libraries than the target program actually needs.

8.4.3 OKAPI-Based Directory Copying Script

In general, running an application once under CDE monitoring only packages up a subset of all required files. In our experience, the easiest way to make CDE packages complete is to copy entire sub-directories into the package. To facilitate this process, we created a script that repeatedly calls OKAPI to copy an entire directory at a time into `cde-root/`, automatically following symlinks to other directories and recursively copying as needed.

Although this approach might seem primitive, it is effective in practice because applications often store all of their files in a few top-level directories. When a user inspects the directory structure within `cde-root/`, it is usually obvious where the application's files reside. Thus, the user can run our OKAPI-based script to copy the entirety of those directories into the package.

Evaluation: To demonstrate the efficacy of this approach, we have created complete self-contained CDE packages for six of the largest and most popular Linux applications. For each application, we made an initial packaging run with `cde`, inspected the package contents, and copied at most three directories into the package. The entire packaging process took several minutes of human effort per application. Here are our full results:

- **AbiWord** is a free alternative to Microsoft Word. After an initial packaging run, we saw that some plugins were included in the `cde-root/usr/lib/`

abiword-2.8/plugins and cde-root/usr/lib/goffice/0.8.1/plugins directories. Thus, we copied the entirety of those two original directories into cde-root/ to complete its package, thereby including all AbiWord plugins.

- **Eclipse** is a sophisticated IDE and software development platform. We completed its package by copying the /usr/lib/eclipse and /usr/share/eclipse directories into cde-root/.

- **Firefox** is a popular web browser. We completed its package by copying /usr/lib/firefox-3.6.18 and /usr/lib/firefox-addons into cde-root/ (plus another directory for the third-party Adobe Flash player plug-in).

- **GIMP** is a sophisticated graphics editing tool. We completed its package by copying /usr/lib/gimp/2.0 and /usr/share/gimp/2.0 into cde-root/.

- **Google Earth** is an interactive 3D mapping application. We completed its package by copying /opt/google/earth into cde-root/.

- **OpenOffice.org** is a free alternative to the Microsoft Office suite. We completed its package by copying the /usr/lib/openoffice directory into cde-root/.

8.5 Seamless Execution Mode

When executing a program from within a package, cde-exec redirects all file accesses into the package by default, thereby creating a chroot-like sandbox with cde-package/cde-root/ as the pseudo-root directory (see Figure 8.3, Step 3). However, unlike chroot, CDE does not require root access to run, and its sandbox policies are flexible and user-customizable (see Section 8.3.3).

This default chroot-like execution mode is fine for running self-contained GUI applications such as games or web browsers, but it is a somewhat awkward way to run most types of UNIX-style command-line programs that research programmers often prefer. If users are running, say, a compiler or command-line image processing utility from within a CDE package, they would need to first move their input data files into the package, run the target program using `cde-exec`, and then move the resulting output data files back out of the package, which is a cumbersome process.

Let's consider a modified version of the Alice-and-Bob example from Section 8.2. Suppose Alice is a researcher who is developing a Python script to detect anomalies in network log files. She normally runs her script using this Linux command:

```
python detect_anomalies.py net.log
```

Let's say she packages up her command with CDE and sends that package to Bob. He can now re-run her original analysis on the `net.log` file from within the package. However, if Bob wants to run Alice's script on his own log data (e.g., `bob.log`), then he needs to first move his data file inside of `cde-package/cde-root/`, change into the appropriate sub-directory deep within the package, and run:

```
cde-exec python detect_anomalies.py bob.log
```

In contrast, if Bob had actually installed the proper version of Python and its required extension modules on his machine, then he could run Alice's script from *anywhere* on his filesystem with no restrictions. Some CDE users wanted CDE-packaged programs to behave just like regularly-installed programs rather than requiring input files to be moved inside of a `cde-package/cde-root/` sandbox, so we implemented a *seamless execution mode* that largely achieves this goal.

Seamless execution mode works using a simple heuristic: If `cde-exec` is being invoked from a directory *not* in the CDE package (i.e., from somewhere else on the user's filesystem), then only redirect a path into `cde-package/cde-root/` if the file that the path refers to actually exists within the package. Otherwise simply leave the path unmodified so that the program can access the file normally. No user intervention is needed in the common case.

The intuition behind why this heuristic works is that when programs request to load libraries and other mandatory components, those files must exist within the package, so their paths are redirected. On the other hand, when programs request to load an input file passed via, say, a command-line argument, that file does not exist within the package, so the original path is used to load it from the native filesystem.

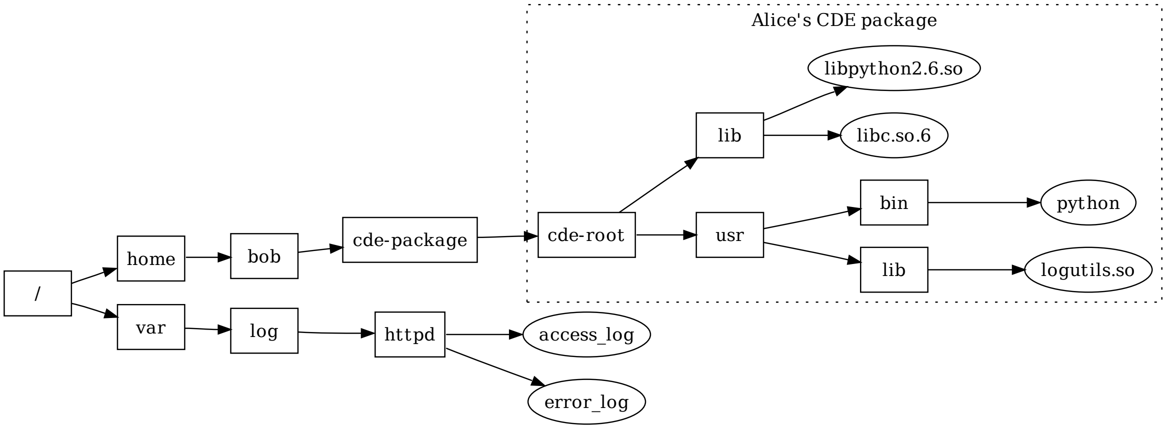


Figure 8.10: Example filesystem layout on Bob's machine after he receives a CDE package from Alice (boxes are directories, ellipses are files). CDE's seamless execution mode enables Bob to run Alice's packaged script on the log files in `/var/log/httpd/` without first moving those files inside of `cde-root/`.

In the example shown in Figure 8.10, if Bob ran Alice's script to analyze an arbitrary log file on his machine (e.g., his web server log, `/var/log/httpd/access_log`), then `cde-exec` will redirect Python's request for libraries (e.g., `/lib/libpython2.6.so` and `/usr/lib/logutils.so`) inside of `cde-root/` since those files exist within the package, but `cde-exec` will *not* redirect `/var/log/httpd/access_log` and instead load the real file from its original location.

Seamless execution mode fails when the user wants the packaged program to access a file from the native filesystem, but an identically-named file actually exists within the package. In the above example, if `cde-package/cde-root/var/log/httpd/access_log` existed, then that file would be processed by the Python script instead of `/var/log/httpd/access_log`. There is no automated way to resolve such name conflicts, but `cde-exec` provides a "verbose mode" where it prints out a log of what paths were redirected into the package. The user can inspect that log and then manually write redirection/ignore rules in a configuration file to control which paths `cde-exec` redirects into `cde-root/`. For instance, the user could tell `cde-exec` to *not* redirect any paths starting with `/var/log/httpd/*`.

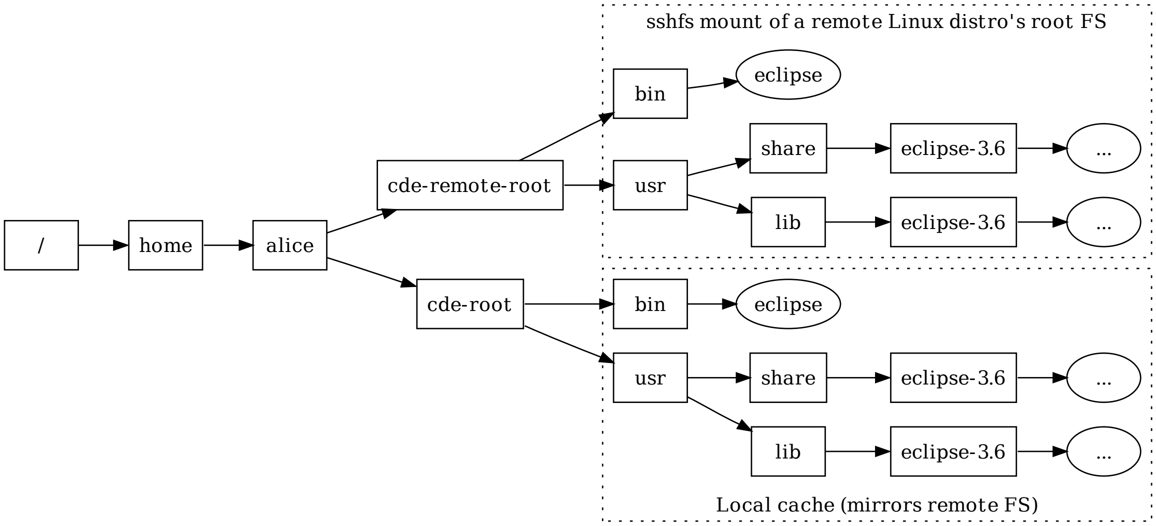


Figure 8.11: An example use of CDE's streaming mode to run Eclipse 3.6 on any Linux machine without installation. `cde-exec` fetches all dependencies on-demand from a remote Linux distro and stores them in a local cache.

8.6 On-Demand Application Streaming

We now introduce an application streaming mode where CDE users can instantly run any Linux application on-demand without having to create, transfer, or install any packages. Figure 8.2 shows a high-level architectural overview. The basic idea is that a system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). When a user wants to run some application that is available on a particular distro, they use sshfs (an ssh-based network filesystem [24]) to mount the root directory of that distro into a special `cde-remote-root/` mount point. Then the user can use CDE's streaming mode to run any application from that distro locally on their own machine.

8.6.1 Implementation and Example

Figure 8.11 shows an example of streaming mode. Let's say that Alice wants to run the Eclipse 3.6 IDE on her Linux machine, but the particular distro she is using makes it difficult to obtain all the dependencies required to install Eclipse 3.6. Rather than suffering through dependency hell, Alice can simply connect to a distro in the farm that contains Eclipse 3.6 and then use CDE's streaming mode to "harvest" the required dependencies on-demand.

Alice first mounts the root directory of the remote distro at `cde-remote-root/`. Then she runs "`cde-exec -s eclipse`" (`-s` activates streaming mode). `cde-exec` finds and executes `cde-remote-root/bin/eclipse`. When that executable requests shared libraries, plug-ins, or any other files, `cde-exec` will redirect the respective paths into `cde-remote-root/`, thereby executing the version of Eclipse 3.6 that resides in the cloud distro. However, note that the application is running locally on Alice's machine, not in the cloud.

An astute reader will immediately realize that running applications in this manner can be slow, since files are being accessed from a remote server. While sshfs performs some caching, we have found that it does not work well enough in practice. Thus, we have implemented our own caching layer within CDE: When a remote file is accessed from `cde-remote-root/`, `cde-exec` uses OKAPI to make a deep-copy into a local `cde-root/` directory and then redirects that file's path into `cde-root/`. In streaming mode, `cde-root/` initially starts out empty and then fills up with a subset of files from `cde-remote-root/` that the target program has accessed.

To avoid unnecessary filesystem accesses, CDE's cache also keeps a list of file paths that the target program tried to access from the remote server, even keeping paths for *non-existent files*. On subsequent runs, when the program tries to access one of those paths, `cde-exec` will redirect the path into the local `cde-root/` cache. It is vital to track non-existent files since programs often try to access non-existent files at start-up while doing, say, a search for shared libraries by probing a list of directories in a search path. If CDE did not track non-existent files, then the program would still access the directory entries on the remote server before discovering that those files still do not exist, thus slowing down performance.

With this cache in place, the first time an application is run, all of its dependencies must be downloaded, which could take several seconds to minutes. This one-time delay is unavoidable. However, subsequent runs simply use the files already in the local cache, so they execute at regular `cde-exec` speeds. Even running a *different* application for the first time might still result in some cache hits for, say, generic libraries such as `libc`, so the entire application does not need to be downloaded.

Finally, the package incompleteness problem faced by regular CDE (see Section 8.4) no longer exists in streaming mode. When the target application needs to access new files that do not yet exist in the local cache (e.g., Alice loads a new Eclipse plug-in for the first time), those files are transparently fetched from the remote server and cached.

8.6.2 Synergy With Package Managers

Nearly all Linux users are currently running one particular distro with one default package manager that they use to install software. For instance, Ubuntu users must use APT, Fedora users must use YUM, SUSE users must use Zypper, Gentoo users must use Portage, etc. Moreover, different releases of the *same* distro contain different software package versions, since distro maintainers add, upgrade, and delete packages in each new release[6].

As long as a piece of software and all of its dependencies are present within the package manager of the exact distro release that a user happens to be using, then installation is trivial. However, as soon as even one dependency cannot be found within the package manager, then users must revert to the arduous task of compiling from source (or configuring a custom package manager).

CDE's streaming mode frees Linux users from this single-distro restriction and allows them to run software that is available within the package manager of any distro in the cloud distro farm. The system administrator is responsible for setting up the farm and provisioning access rights (e.g., ssh keys) to users. Then users can directly install packages in any cloud distro and stream the desired applications to run locally on their own machines.

Philosophically, CDE's streaming mode maximizes user freedom since users are

[6] We once tried installing a machine learning application that depended on the `libcv` computer vision library. The required `libcv` version was found in the APT repository on Ubuntu 10.04, but it was not found in the repositories on the two neighboring Ubuntu releases: 9.10 and 10.10.

now free to run any application in any package manager from the comfort of their own machines, regardless of which distro they choose to use. CDE complements traditional package managers by leveraging all of the work that the maintainers of each distro have already done and opening up access to users of all other distros. This synergy can potentially eliminate quasi-religious squabbles and flame-wars over the virtues of competing distros or package management systems. Such fighting is unnecessary since CDE allows users to freely choose from amongst all of them.

8.7 Real-World Use Cases

Since we released the first version of CDE on November 9, 2010, it has been downloaded at least 5,000 times as of March 2012 [4]. We cannot track how many people have directly checked out its source code from GitHub, though. We have exchanged hundreds of emails with CDE users and discovered six salient real-world use cases as a result of these discussions. Table 8.2 shows that we used 16 CDE packages, mostly sent in by our users, as benchmarks in the experiments reported in Section 8.8. They contain software written in diverse programming languages and frameworks. We now summarize the six use case categories and benchmarks within each category.

Distributing research software: The creators of two research tools found CDE online and used it to create portable packages that they uploaded to their websites:

The website for `graph-tool`, a Python/C++ module for analyzing graphs, lists these (direct) dependencies: "GCC 4.2 or above, Boost libraries, Python 2.5 or above, expat library, NumPy and SciPy Python modules, GCAL C++ geometry library, and Graphviz with Python bindings enabled" [30]. Unsurprisingly, lots of people had trouble compiling it: 47% of all messages on its mailing list (137 out of 289) were questions related to compilation problems. The author of `graph-tool` used CDE to automatically create a portable package (containing 149 shared libraries and 1909 total files) and uploaded it to his website so that users no longer needed to suffer through the pain of manually compiling it.

`arachni`, a Ruby-based tool that audits web application security [29], requires six hard-to-compile Ruby extension modules, some of which depend on versions of Ruby and libraries that are not available in the package managers of most modern Linux distributions. Its creator, a security researcher, created and uploaded CDE packages and then sent us a grateful email describing how much effort CDE saved him: *"My guess is that it would take me half the time of the development process to create a self-contained package by hand; which would be an unacceptable and truly scary scenario."*

We used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as tarballs of source code: `pads` [59] and `saturn` [35]. 44% of the messages on the `pads` mailing list (38 / 87) were questions related to troubles with compiling it (22% for `saturn`). Once we successfully compiled these projects (after a few hours of improvising our own hacks since the instructions were outdated), we created CDE packages by running their regression test suites, so that others do not need to suffer through the compilation process.

Even the `saturn` team leader admitted in a public email, "As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release" [21]. In contrast, our CDE packages are largely immune to "bit rot" (until the user-kernel ABI changes) because they contain all required dependencies.

Running production software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions, especially when system upgrades are infeasible. For example, an engineer at Cisco wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port, and then ran the tools from within the packages on his work machines. He sent us one of the packages, which we used as a benchmark: the `meld` visual diff tool.

Package name	Description	Dependencies	Creator
Distributing research software			
`arachni`	Web app. security scanner framework [29]	Ruby (+ extensions)	security researcher
`graph-tool`	Lib. for manipulation & analysis of graphs [30]	Python, C++, Boost	math researcher
`pads`	Language for processing ad-hoc data [59]	Perl, ML, Lex, Yacc	self
`saturn`	Static program analysis framework [35]	Perl, ML, Berkeley DB	self
Running production software on incompatible distros			
`meld`	Interactive visual diff and merge tool for text	Python, GTK+	software engineer
`bio-menace`	Classic video game within a MS-DOS emulator	DOSBox, SDL	game enthusiast
`google-earth`	3D interactive map application by Google	shell scripts, OpenGL	self
Creating reproducible computational experiments			
`kpiece`	Robot motion planning algorithm [149]	C++, OpenGL	robotics researcher
`gadm`	Genetic algorithm for social networks [104]	C++, make, R	self
Deploying computations to cluster or cloud			
`ztopo`	Batch processing of topological map images	C++, Qt	graduate student
`klee`	Automatic bug finder & test case generator [41]	C++, LLVM, μClibc	self
Submitting executable bug reports			
`coq-bug-2443`	Incorrect output by Coq proof assistant [5]	ML, Coq	bug reporter
`gcc-bug-46651`	Causes GCC compiler to segfault [7]	gcc	bug reporter
`llvm-bug-8679`	Runs LLVM compiler out of memory [12]	C++, LLVM	bug reporter
Collaborating on class programming projects			
`email-search`	Natural language semantic email search	Python, NLTK, Octave	college student
`vr-osg`	3D virtual reality modeling of home appliances	C++, OpenSceneGraph	college student

Table 8.2: CDE packages used as benchmarks in our experiments, grouped by use cases. A label of “self” in the “Creator” column means that I created the package. All other packages were created by CDE users, most of whom we have never met.

Hobbyists applied CDE in a similar way: A game enthusiast could only run a classic game (`bio-menace`) within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the game on his other machines. We also helped a user create a portable package for the Google Earth 3D map application (`google-earth`), so he can now run it on older distros whose libraries are incompatible with Google Earth.

Creating reproducible computational experiments: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. In the past few years, science journals and CS conferences (e.g., SIGMOD, FSE) have encouraged authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard for people to set up all of the (often-undocumented) dependencies required to re-run experiments. In fact, it can even be difficult to re-run *one's own experiments* in the future, due to inevitable OS and library upgrades. To ensure that he could later re-run and adjust experiments in response to reviewer critiques for a paper submission [41], our group-mate Cristian took the hard drive out of his computer at paper submission time and archived it in his drawer!

In our experience, the results of many computational science experiments can be reproduced within CDE packages since the programs are output-deterministic [37], always producing the same outputs (e.g., statistics, graphs) for a given input. A robotics researcher used CDE to make the experiments for his motion planning paper (`kpiece`) [149] fully reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (`gadm`) [104].

Deploying computations to cluster or cloud: People working on computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism. However, before they can deploy their computations to a cluster or cloud computing (e.g., Amazon EC2), they must first install all of the required executables and dependent libraries on the cluster machines. At best, this process is tedious and time-consuming; at worst, it can be impossible, since regular users often do not have root access on cluster machines.

A user can create a self-contained package using CDE on their desktop machine and then execute that package on the cluster or cloud (possibly many instances in parallel), without needing to install any dependencies or to get root access on the remote machines. Our colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps (`ztopo`). However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive `klee` [41] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines. Klee can be hard to compile since it depends on LLVM, which is very picky about specific versions of GCC and other build tools being present before it will compile.

Submitting executable bug reports: Bug reporting is a tedious manual process: Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries (e.g., *"I'm running Java version 1.6.0_13, Eclipse SDK Version 3.6.1, ..."*), and maybe attaching an input file that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier and more reliable solution: The bug reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. The developer can also modify the input file and command-line parameters and then re-execute, in order to investigate the bug's root cause.

To show that this technique has some potential of working, we asked people who recently reported bugs to popular open-source projects to use CDE to create executable bug reports. Three volunteers sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (`coq-bug-2443`) [5], one that segfaults the GCC compiler (`gcc-bug-46651`) [7], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (`llvm-bug-8679`) [12].

Since CDE is not a record-replay tool, it is not guaranteed to reproduce non-deterministic bugs. However, at least it allows the developer to run the exact versions of the faulting executables and dependent libraries.

Collaborating on class programming projects: Two users sent us CDE packages they created for collaborating on class assignments. Rahul, a Stanford grad student, was using NLTK [113], a Python module for natural language processing, to build a semantic email search engine (`email-search`) for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their Linux machines due to conflicting library versions and dependency hell. This meant that they could only run one instance of the project at a time on Rahul's laptop for query testing and debugging. When Rahul discovered CDE, he created a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Joshua, an undergrad from Mexico, emailed us a similar story about how he used CDE to collaborate on and demo his virtual reality class project (`vr-osg`).

8.8 Evaluation

Our evaluation aims to address these four questions:

- How portable are CDE packages across different Linux distros (Section 8.8.1)?
- How does CDE fare against an example one-click installer (Section 8.8.2)?
- How much benefit does dynamic dependency tracking provide (Section 8.8.3)?
- How much of a run-time slowdown does CDE impose (Section 8.8.4)?

8.8.1 Evaluating CDE Package Portability

To show that CDE packages can successfully execute on a wide range of Linux distros and kernel versions, we tested our benchmark packages from Table 8.2 on popular distros from the past 5 years. We installed fresh copies of these distros (listed with the versions and release dates of their kernels) on a 3GHz Intel Xeon x86-64 machine:

- Sep 2006 — CentOS 5.5 (Linux 2.6.18)
- Oct 2007 — Fedora Core 8 (Linux 2.6.23)
- Oct 2008 — openSUSE 11.1 (Linux 2.6.27)
- Sep 2009 — Ubuntu 9.10 (Linux 2.6.31)
- Feb 2010 — Mandriva Free Spring (Linux 2.6.33)
- Aug 2010 — Linux Mint 10 (Linux 2.6.35)

We installed 32-bit and 64-bit versions of each distro and executed our 32-bit benchmark packages (those created on 32-bit distros) on the 32-bit versions, and our 64-bit packages on the 64-bit versions. Although all of these distros reside on one physical machine, none of our benchmark packages were created on that machine: CDE users created most of the packages, and we made sure to create our own packages on other machines.

Results: Out of the 96 unique configurations we tested (16 CDE packages each run on 6 distros), all executions succeeded except for one[7]. By "succeeded", we mean that the programs ran correctly, as far as we could observe: Batch programs generated identical outputs across distros, regression tests passed, we could interact normally with GUI programs, and we could reproduce the symptoms of executable bug reports.

In addition, we were able to successfully execute all of our 32-bit packages on the 64-bit versions of CentOS, Mandriva, and openSUSE (the other 64-bit distros did not support executing 32-bit binaries).

[7] `vr-osg` failed on Fedora Core 8 with a known error related to graphics drivers.

In sum, we were able to use CDE to successfully execute a diverse set of programs (from Table 8.2) "out-of-the-box" on a variety of Linux distributions from the past 5 years, without performing any installation or configuration.

8.8.2 Comparing Against a One-Click Installer

To show that the level of portability that CDE enables is substantive, we compare CDE against a representative one-click installer for a commercial application. We installed and ran Google Earth (Version 5.2.1, Sep 2010) on our 6 test distros using the official 32-bit installer from Google. Here is what happened on each distro:

- CentOS (Linux 2.6.18) — installs fine but Google Earth crashes upon start-up with variants of this error message repeated several times, because the GNU Standard C++ Library on this OS is too old:

  ```
  /usr/lib/libstdc++.so.6:
  version ‘GLIBCXX_3.4.9’ not found
  (required by ./libgoogleearth_free.so)
  ```

- Fedora (Linux 2.6.23) — same error as CentOS

- openSUSE (Linux 2.6.27) — installs and runs fine

- Ubuntu (Linux 2.6.31) — installs and runs fine

- Mandriva (Linux 2.6.33) — installs fine but Google Earth crashes upon start-up with this error message because a required graphics library is missing:

  ```
  error while loading shared libraries:
  libGL.so.1: cannot open shared object
  file: No such file or directory
  ```

- Linux Mint (Linux 2.6.35) — installer program crashes with this cryptic error message because the XML processing library on this OS is *too new* and thus incompatible with the installer:

```
setup.data/setup.xml:1: parser error :
  Document is empty
setup.data/setup.xml:1: parser error :
  Start tag expected, '<' not found
Couldn't load 'setup.data/setup.xml'
```

In summary, on 4 out of our 6 test distros, a binary installer for the fifth major release of Google Earth (v5.2.1), a popular commercial application developed by a well-known software company, failed in its *sole goal* of allowing the user to run the application, despite advertising that it should work on any Linux 2.6 machine.

If a team of professional Linux developers had this much trouble getting a widely-used commercial application to be portable across distros, then it is unreasonable to expect researchers to be able to create portable Linux packages for their prototypes.

In contrast, once we were able to install Google Earth on just *one machine* (Dell desktop running Ubuntu 8.04), we ran it under CDE supervision to create a self-contained package, copied the package to all 6 test distros, and successfully ran Google Earth on all of them without any installation or configuration.

8.8.3 Evaluating the Importance of Dynamic Tracking

To show the importance of dynamic (run-time) dependency tracking, we compare CDE against a simple but representative static analysis. We wrote a script that runs the Linux `ldd` and `strings` utilities on an executable file to find all string constants representing shared libraries on which it depends, and then recursively runs `ldd` and `strings` on those libraries and their dependencies until the set of files converges. Although this basic static technique only finds libraries named by constant strings[8], it represents what people actually do in practice, since it automates the tedious manual process of "chasing down and copying over dependent libraries" that folk wisdom (e.g., blog posts and forums) suggests as the way to transport Linux binaries across machines [28]. For instance, a senior software engineer at a European internet security

[8] It is difficult in general for a static analysis to model dynamically-generated strings; we know of no static dependency gathering tool that works in this way.

company told us via email: *"For years I use [sic] a shellscript to figure out library dependencies with ldd but always wanted something that handles all dependencies of an application, and especially of the combination of binaries and shell scripts. Yesterday, I found CDE on slashdot, it is exactly what I need."*

In contrast, since CDE executes the target executable in addition to statically searching for constant strings, CDE can find dependencies on shared libraries (and all other files) named by dynamically-generated strings, in addition to those that a static technique finds.

Results: Table 8.3 shows that in all but four benchmarks, the static technique found fewer libraries than CDE. Thus, it cannot be used to create a portable package since the program will fail if *even one library is missing.* (For similar reasons, static linking when compiling will not work either.) Even on the four benchmarks where the static technique found all required libraries, a user would still have to manually insert all input, configuration, and other data files into the package. The "# total files" column in Table 8.3 shows that packages contain dozens to thousands of files, often scattered across many directories, so this process can be tedious and error-prone.

Table 8.3 also shows why it is necessary for CDE to dynamically track dependencies, since most benchmarks load libraries that are not named by constant strings. At one extreme, the four benchmarks where the static technique performed worst (`google-earth`, `graph-tool`, `meld`, `arachni`) consist of scripts written in interpreted languages. The interpreter dynamically loads libraries and invokes other executables based on the contents of those scripts. A static analysis of the interpreter's executable (e.g., Python) can only find the libraries needed to start up the interpreter; however, the majority of libraries that each script requires are indirectly specified within the script itself. For example, executing a simple line of Python code "`import numpy`" in `graph-tool` causes the Python interpreter to import the NumPy numerical analysis module, which consists of 23 shared libraries scattered across 7 sub-directories. The interpreter dynamically generates the pathnames of those 23 libraries by processing strings read from environment variables and config files, so it is unlikely that any static analysis could ever generate those 23 pathnames and find the corresponding libraries. CDE easily finds those libraries since it monitors actual execution.

	# shared library files			# total
Package name	Total	Statically found		files
`google-earth`	82	3	(4%)	243
`graph-tool`	149	9	(6%)	1909
`meld`	93	8	(9%)	507
`arachni`	48	6	(13%)	381
`gcc-bug-46651`	13	2	(15%)	114
`email-search`	138	28	(20%)	3052
`gadm`	18	4	(22%)	268
`saturn`	16	8	(50%)	455
`pads`	9	5	(56%)	150
`ztopo`	59	35	(59%)	164
`vr-osg`	39	28	(72%)	57
`bio-menace`	27	26	(96%)	107
`coq-bug-2443`	3	3	(100%)	29
`klee`	6	6	(100%)	18
`llvm-bug-8679`	8	8	(100%)	14
`kpiece`	30	30	(100%)	45

Table 8.3: The number of total shared library (`*.so*`) files in each CDE package, and the number (and percent) of those found by a simple static analysis of binaries and their dependent libraries. The rightmost column shows the number of total files in each package.

8.8.4 Evaluating CDE Run-Time Slowdown

The primary drawback of executing a CDE-packaged application is the run-time slowdown due to extra user-kernel context switches. Every time the target application issues a system call, the kernel makes two extra context switches to enter and then exit the `cde-exec` monitoring process, respectively. `cde-exec` performs some computations to calculate path redirections, but its run-time overhead is dominated by context switching. Disabling path redirection still results in similar overheads.

We informally evaluated the run-time slowdown of `cde` and `cde-exec` on 34 diverse Linux applications. In summary, for CPU-bound applications, CDE causes almost no slowdown, but for I/O-bound applications, CDE causes a slowdown of up to ∼30%.

SPEC CPU benchmarks: We first ran CDE on the entire SPEC CPU2006 benchmark suite (both integer and floating-point benchmarks) [23]. We chose this suite because it contains CPU-bound applications that are representative of the types of programs that computational scientists and other researchers are likely to run with CDE. For instance, SPEC CPU2006 contains benchmarks for video compression, molecular dynamics simulation, image ray-tracing, combinatorial optimization, and speech recognition.

We ran these experiments on a Dell machine with a 2.67GHz Intel Xeon CPU running a 64-bit Ubuntu 10.04 distro (Linux 2.6.32). Each trial was run three times, but the variances in running times were negligible.

Table 8.4 shows the percentage slowdowns incurred by using `cde` to create each package (the 'pack' column) and by using `cde-exec` to execute each package (the 'exec' column). The 'exec' column slowdowns are shown in **bold** since they are more important for our users: A package is only created once but executed multiple times. In sum, slowdowns ranged from non-existent to ∼4%, which is unsurprising since the SPEC CPU2006 benchmarks were designed to be CPU-bound and not make much use of system calls.

Real-world benchmarks: To test more realistic I/O-bound applications, we measured running times for executing the following commands in the five CDE packages that we created (those labeled with "self" in the "Creator" column of Table 8.2):

	Native	CDE slowdown	
Benchmark	run time	pack	exec
`400.perlbench`	23.7s	3.0%	**2.5%**
`401.bzip2`	47.3s	0.2%	**0.1%**
`403.gcc`	0.93s	2.7%	**2.2%**
`410.bwaves`	185.7s	0.2%	**0.3%**
`416.gamess`	129.9s	0.1%	**0%**
`429.mcf`	16.2s	2.7%	**0%**
`433.milc`	15.1s	2%	**0.6%**
`434.zeusmp`	36.3s	0%	**0%**
`435.gromacs`	133.9s	0.3%	**0.1%**
`436.cactusADM`	26.1s	0%	**0%**
`437.leslie3d`	136.0s	0.1%	**0%**
`444.namd`	13.9s	3%	**0.3%**
`445.gobmk`	97.5s	0.4%	**0.2%**
`447.dealII`	28.7s	0.5%	**0.2%**
`450.soplex`	5.7s	2.2%	**1.8%**
`453.povray`	7.8s	2.2%	**1.9%**
`454.calculix`	1.4s	5%	**4%**
`456.hmmer`	48.2s	0.2%	**0.1%**
`458.sjeng`	121.4s	0%	**0.2%**
`459.GemsFDTD`	55.2s	0.2%	**1.6%**
`462.libquantum`	1.8s	2%	**0.6%**
`464.h264ref`	87.2s	0%	**0%**
`465.tonto`	229.9s	0.8%	**0.4%**
`470.lbm`	31.9s	0%	**0%**
`471.omnetpp`	51.0s	0.7%	**0.6%**
`473.astar`	103.7s	0.2%	**0%**
`481.wrf`	161.6s	0.2%	**0%**
`482.sphinx3`	8.8s	3%	**0%**
`483.xalancbmk`	58.0s	1.2%	**1.8%**

Table 8.4: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package on the SPEC CPU2006 benchmarks, using the “train” datasets.

	Native	CDE slowdown		Syscalls
Command	time	pack	exec	per sec
`gadm` (algorithm)	4187s	0%†	**0%**†	19
`pads` (inferencer)	18.6s	3%†	**1%**†	478
`klee`	7.9s	31%	**2%**†	260
`gadm` (make plots)	7.2s	8%	**2%**†	544
`gadm` (C++ comp)	8.5s	17%	**5%**	1459
`saturn`	222.7s	18%	**18%**	6477
`google-earth`	12.5s	65%	**19%**	7938
`pads` (compiler)	1.7s	59%	**28%**	6969

Table 8.5: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package. Each entry reports the mean taken over 5 runs; standard deviations are negligible. Slowdowns marked with † are *not* statistically significant at $p < 0.01$ according to a t-test.

- `pads` — Compile a PADS [59] specification into C code (the '`pads` (compiler)' row in Table 8.5), and then infer a specification from a data file (the '`pads` (inferencer)' row in Table 8.5).

- `gadm` — Reproduce the GADM experiment [104]: Compile its C++ source code ('C++ comp'), run genetic algorithm ('algorithm'), and use the R statistics software to visualize output data ('make plots').

- `google-earth` — Measure startup time by launching it and then quitting as soon as the initial Earth image finishes rendering and stabilizes.

- `klee` — Use Klee [41] to symbolically execute a C target program (a STUN server) for 100,000 instructions, which generates 21 test cases.

- `saturn` — Run the regression test suite, which contains 69 tests (each is a static program analysis).

We measured the following on a Dell desktop (2GHz Intel x86, 32-bit) running Ubuntu 8.04 (Linux 2.6.24): number of seconds it took to run the original command ('Native time'), percent slowdown vs. native when running a command with `cde` to create a package ('pack'), and percent slowdown when executing the command from within a CDE package with `cde-exec` ('exec'). We ran each benchmark five times under each condition and report mean running times. We used an *independent two-group t-test* [43] to determine whether each slowdown was statistically significant (i.e., whether the means of two sets of runs differed by a non-trivial amount).

Table 8.5 shows that the more system calls a program issues per second, the more CDE causes it to slow down due to all of the additional context switches. Also, creating a CDE package ('pack' column) is slower than executing a program within a package ('exec' column) because CDE must create new sub-directories and copy files into the package.

CDE execution slowdowns ranged from negligible (not statistically significant) to ~30%, depending on system call frequency. As expected, CPU-bound workloads such as the `gadm` genetic algorithm and the `pads` inferencer machine learning algorithm had almost no slowdown, while those that were more I/O- and network-intensive (e.g., `google-earth`) had the largest slowdowns.

When using CDE to run GUI applications, we did not notice any loss in interactivity due to the slowdowns. When we navigated around the 3D maps within the `google-earth` GUI, we felt that the CDE-packaged version was just as responsive as the native version. When we ran GUI programs from CDE packages that users sent to us (the `bio-menace` game, `meld` visual diff tool, and `vr-osg`), we also did not perceive any visible lag.

The main caveat of these experiments is that they are informal and meant to characterize "typical-case" behavior rather than being stress tests of worst-case behavior. One could imagine developing adversarial I/O intensive benchmarks that issue tens or hundreds of thousands of system calls per second, which would lead to greater slowdowns. We have not run such experiments yet.

Finally, we also ran some informal performance tests of `cde-exec`'s seamless execution mode. As expected, there were no noticeable differences in running times versus regular `cde-exec`, since the context-switching overhead dominates `cde-exec` computation overhead.

8.9 Discussion

Our design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to run on other Linux machines with as little effort as possible. However, we are not proposing CDE as a replacement for traditional software installation. CDE packages have a number of limitations. Most notably,

- They are not guaranteed to be complete.
- Their constituent shared libraries are "frozen" and do not receive regular security updates. (Static linking also shares this limitation.)
- They run slower than native applications due to `ptrace` overhead. We measured slowdowns of up to 28% in our experiments (Section 8.8.4), but slowdowns can be worse for I/O-heavy programs.

Software engineers who are releasing production-quality software should obviously take the time to create and test one-click installers or integrate with package managers. But for the millions of research scientists, prototype designers, system administrators, programming course students and teachers, and hobby hackers who just want to deploy their *ad-hoc* software as quickly as possible, CDE can emulate many of the benefits of traditional software distribution with much less required labor: In just minutes, users can create a base CDE package by running their program under CDE supervision, use our *semi-automated heuristic tools* to make the package complete, deploy to the target Linux machine, and then execute it in *seamless execution mode* to make the target program behave like it was installed normally.

Lessons learned: We would like to conclude by sharing some generalizable lessons that we learned throughout the past two years of developing CDE and supporting thousands of users from diverse fields.

- First and foremost, start with a conceptually-clear core idea, make it work for basic non-trivial cases, document the still-unimplemented tricky cases, launch your system, and then get feedback from real users. User feedback is by far the easiest way for you to discover what bugs are important to fix and what new features to add next.

- A simple and appealing quick-start webpage guide and screencast video demo are essential for attracting new users. No potential user is going to read through dozens of pages of an academic research paper before deciding to try your system. In short, even hackers need to learn to be great salespeople.

- To maximize your system's usefulness, you must design it to be easy-to-use for beginners but also to give advanced users the ability to customize it to their liking. One way to accomplish this goal is to have well-designed default settings, which can be adjusted via command-line options or configuration files. The defaults must work effectively "out-of-the-box" without any tuning, or else new users will get frustrated.

- Resist the urge to add new features just because they're "interesting", "cool", or "potentially useful". Only add new features when there are compelling real users who demand it. Instead, focus your development efforts on fixing bugs, writing more test cases, improving your documentation, and, most importantly, attracting new users.

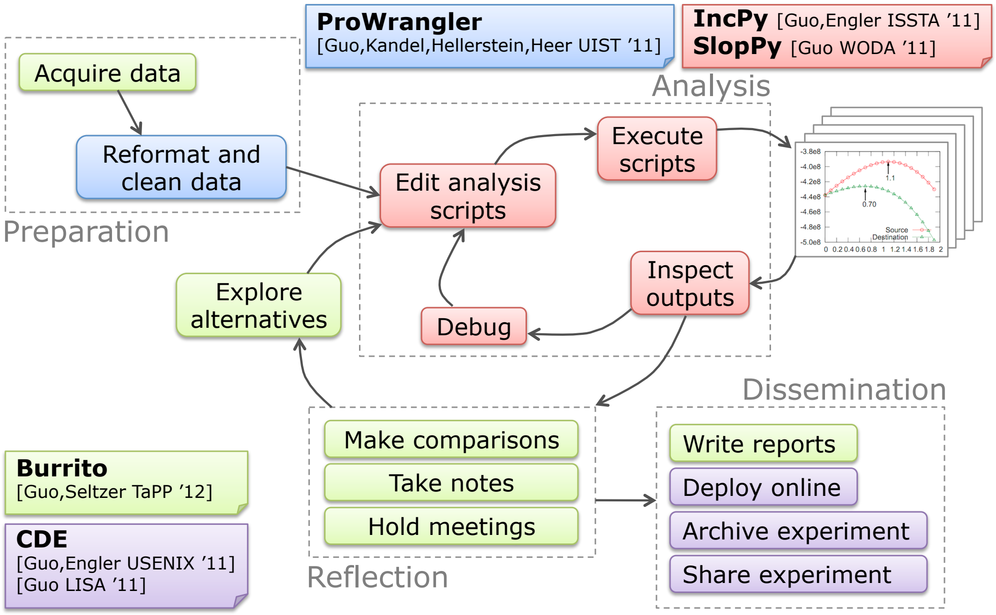
- Users are by far the best sources of bug reports, since they often stress your system in ways that you could have never imagined. Whenever a user reports a bug, send them a sincere *thank you* note, try to create a representative minimal test case, and add it to your regression test suite.

- If a user has a conceptual misunderstanding of how your system works, then think hard about how you can improve your documentation or default settings to eliminate this misunderstanding.

In sum, make something useful, get real users, keep them happy, and have fun!

Chapter 9

Discussion



This final chapter discusses how the tools I have built for my dissertation can be integrated together to improve the process of research programming, what challenges still remain open, proposed directions for future research, and broader implications of my work.

9.1 A Better Research Programming Workflow

Suppose that a research programmer named Alice had access to production-quality versions of all five tools that I presented in this dissertation. In what ways would her workflow be better than the status quo described in Chapter 2 ? In sum, these tools allow Alice to iterate and potentially discover insights faster by offloading the burdens of data management and provenance to the computer.

9.1.1 Preparation Phase

Burrito tracks the provenance of all data that Alice acquires. She can write Burrito plugins to periodically ping online data sources to check whether her data is still up-to-date. Also, she does not need to worry about being overly meticulous regarding file and directory naming conventions, since Burrito tracks metadata and enables context-aware file search.

Alice imports raw data sets into Proactive Wrangler to do data cleaning and reformatting. A Proactive Wrangler plugin for Burrito can enable Alice to trace provenance between raw and cleaned data sets at finer granularity than file-level (e.g., table-level, row-level, or even tuple-level granularities). Burrito also enables her to take notes throughout the data cleaning process and also restore old versions of data files if she makes mistakes. Proactive Wrangler can then export its transformations as Python scripts, so that she can re-use those scripts to clean additional similarly-formatted data.

9.1.2 Analysis Phase

Alice can write her analysis scripts in any combination of languages or libraries of her liking, and Burrito will track provenance, notes, and auxiliary reference sources (e.g., web pages and PDF documents) throughout her workflow. Even though my implementation of Burrito is for Linux, a production-quality version could be made for any major operating system.

If Alice uses Python to write her scripts, then she can use a production-quality interpreter that combines IncPy and SlopPy features to provide automatic memoization, error tolerance, and incremental error recovery. (And if she prefers to use another language, the ideas from IncPy and SlopPy could be ported to create an enhanced run-time system for that language.) A Burrito plugin for such an interpreter could track provenance for Python functions and memoized results in addition to the regular file-level provenance provided by Burrito.

The automatic file versioning provided by Burrito protects Alice against software bugs and enables her to take more risks with her experiments, because she can easily revert to old working versions of her code and data sets.

9.1.3 Reflection Phase

The combination of IncPy and Burrito helps Alice efficiently explore alternatives. IncPy speeds up her iterations and manages intermediate (cached) data sets, while Burrito allows her to compare and take notes on the outputs of her trials. She no longer needs to manually organize dozens of disparate notes files throughout her filesystem. She can also step away from her computer, make sketches using a digital pen on a notepad or electronic whiteboard, and then integrate those freehand sketches with her typed notes in the Burrito timestream. Prior to meeting with her colleagues or supervisor, Alice can run a Burrito application to print out a status report of her progress. After her meetings, she can annotate parts of her workflow (e.g., scripts, output graphs) with new to-do list action items. All of her notes are linked with their original context (in both “space” and time), and she can use whichever combination of software tools and programming languages she likes.

9.1.4 Dissemination Phase

When Alice is ready to write up a paper on her research findings, she can use the Burrito Lab Notebook Generator to aggregate her results and notes to use as the basis for writing. She can then run CDE to archive her experiments so that she can make modifications in the future if reviewers require additional evidence. She can also distribute her CDE packages to colleagues who want to play with her experiments in addition to reading her paper.

She could create a CDE plugin for Burrito that packages up not only the files accessed by her experimental scripts, but also their associated notes and provenance. These in-context notes and provenance collected throughout her research process could serve as documentation for how to interpret the contents of her CDE packages.

9.2 Remaining Challenges and Future Directions

I have identified three main areas of challenges in research programming that my dissertation has not yet addressed: scaling up to multiple machines, scaling up to multiple people, and better dissemination of insights gained throughout the research process. These areas provide ample opportunities for future research agendas.

9.2.1 Cloud-Scale Rapid Iteration

This dissertation has only focused on the use case of one researcher working on his/her desktop machine. However, it is inevitable that more research programming will be done on clusters of machines and on cloud computing services due to the need to process massive amounts of data (e.g., terabytes to petabytes). There is currently a tension between latency and scalability: Software tools already exist to process data at cluster/cloud scales [1, 52, 128, 151], but they are cumbersome to administer and are not designed to support the sorts of seamless rapid iteration that are vital for research programming.

Thus, it seems natural to propose the following future direction: How can we get all of the convenient desktop workflow benefits of Section 9.1 when processing cloud-scale data? The high-level challenge here is how to maintain the user experience of working on one's personal desktop machine while scaling up to handle terabytes to petabytes of data processing. Aggressive memoization (using IncPy-like techniques), error tolerance, and incremental reporting of partial results (using SlopPy-like techniques) could help accomplish this goal. Similar problems have been studied in detail in systems such as MapReduce Online [48], and, more generally, aggregation, sampling, and approximation methods from the database community. But to my knowledge, previous work has not focused on HCI issues related to integrating into the research programming workflow.

9.2.2 The Future of Collaborative Research

This dissertation has only addressed challenges faced by a lone researcher, but research programming is often done in a group with collaborators analyzing data together toward a common goal. An interesting direction for future work is how to scale up the techniques presented in this dissertation not just to multiple machines, but to multiple people simultaneously collaborating on a project. Combining knowledge of the research programming workflow with the rich literature on Computer Supported Cooperative Work [93] could lead to novel research collaboration tools. For instance, there is a rich body of prior work related to collaborating on data analysis and visualization, embodied by systems such as *sense.us* [86]. Those ideas could be ported over to the research programming domain, which is more focused on writing imperative programs in heterogeneous environments rather than working solely within a specially-designed data analysis and visualization system. One ideal vision is to aspire to do for research programming what Google Docs has done for collaborative text editing or what Git [112] has done for team software engineering.

For example, how could Burrito-like activity traces be shared in real-time to inform teammates or supervisors about what each person is working on? This type of system has obvious benefits for team coordination. But real-time trace sharing might also enable senior team members to quickly catch “newbie mistakes” made by their younger colleagues, thereby saving them the trouble of going down certain common dead-ends. For example, current Ph.D. students often spend days or even weeks exploring a dead-end path in their research, since their advisors are not aware of the details of what they are working on. Of course, hitting dead-ends is a natural part of doing exploratory research, but some of those inefficiencies could be prevented with better guidance from senior colleagues. Improved monitoring tools could be a key pre-requisite for enabling mentors to provide such timely and detailed guidance.

Technical challenges include privacy, security, real-time synchronization, and user interface design for maintaining user attention on his/her own trace while simultaneously being able to monitor several other people's traces. However, even larger challenges include the potential social and cultural issues that must be resolved before people are comfortable sharing the details of their day-to-day research programming activities (i.e., revealing "how the sausage is made"), especially with supervisors.

Finally, sharing computational activities could speed up everyone's analyses and eliminate redundant computations. Imagine a shared IncPy-like persistent cache for memoized analysis results, annotated with Burrito-like notes about which experimental trials created each piece of data. Then people would know which trials their teammates have tried, and if they want to explore variants of those trials, their scripts will run faster due to cache hits. Making the analogy to version control systems, it is as though people can fork a "branch" off of their teammates' analyses, and every long-running computation gets cached into a master database for everyone's benefit.

9.2.3 Ph.D.-In-A-Box

Despite the fact that so much modern research is being done on computers, papers are still the primary means of disseminating new knowledge. Since dead trees are an impoverished communications medium, the ideas in this dissertation could be used to build a richer electronic medium where one's colleagues, apprentices, and intellectual adversaries can interactively explore the results of one's research experiments. It is straightforward to imagine how CDE packages or Burrito VM images can provide such functionality, perhaps hosted on a cloud service where readers can visit a web site to re-run and tweak those experiments. However, a far more exciting challenge is how to capture and present a rich history of project progression over time, so that readers can learn from the *entire research process*, not merely explore the final results.

To convey the potential benefits of learning from research *processes* rather than just end results, I will make an analogy to mathematics. Mathematics research papers are written in a concise manner presenting a minimal set of proofs of lemmas and theorems. Readers unfamiliar with the process of discovery in mathematics might mistakenly assume that some lofty genius must have dreamt up the perfect proof fully-formed and scribbled it down on paper. The truth is far messier: Much like research programmers, mathematicians explore numerous hypotheses, go down dead-ends, backtrack repeatedly, talk through ideas with colleagues, and gradually cobble together a finished proof. Then they painstakingly whittle down that proof until it can be presented as elegantly as possible, and only then do they write up the final paper. The vast majority of intellectual wisdom lies in the *process* of working through the problems, and such knowledge is not represented anywhere in the published paper. Mathematicians learn their craft not just by reading papers, but by actually watching their colleagues and mentors at work. Imagine if there was a way to capture and present the entire months-long process of a famous mathematician coming up with a particular proof. Aspiring mathematicians could learn far more from such an interactive presentation than from simply reading the final polished paper.

I believe that such a goal of "total recall" is easier to accomplish in the context of computational research. Since most of the work is being done on the computer, it is easier to trace the entire workflow history and provide Burrito-like interfaces for in-context annotations. First-class support for branching and backtracking are vital needs in such a system, since much of the wisdom gained from a research apprenticeship is learning from what did not work and what dead-ends to avoid. In this vision, all Ph.D. students would maintain a hard disk image containing the complete trials and tribulations of their 5–6 years' worth of experiments. This "Ph.D.-In-A-Box" could be used to train new students and to pass down all of the implicit knowledge, experiences, tricks, and wisdom that are often lost in a dead-tree paper dissertation.

Extending this analogy further, imagine an online library filled with the collected electronic histories of all research projects, not just their final results in published form. It now becomes possible to perform pattern recognition and aggregation across multiple projects to discover common "tricks-of-the-trade". Someone new to a field, say machine learning, can now immersively learn from the collective wisdom of thousands of expert machine learning researchers rather than simply reading their papers. One could argue that, in the limit, such a system would be like "indexing" all of those researchers' brains and making that knowledge accessible. I actually believe that such a system can be more effective than “brain indexing”, since people subconsciously apply tricks from their intuitions and often forget the details of what they were working on (especially failed trials). In this vision of the future, a paper is merely a facade for the real contributions of the full research process.

Such a dream system can benefit multiple parties: 1.) Researchers can revisit and reflect upon their old experiments. 2.) Colleagues and students can learn from and scrutinize the work of fellow researchers. 3.) “Meta-researchers” can perform meta-analyses of work processes, methodologies, and findings within a field at massive scales. 4.) HCI researchers have a large corpus of real user interaction data to use for designing next-generation recommendation and mixed-initiative interaction systems.

9.3 Conclusion: Broader Implications

At its core, this dissertation is about decomposing a ubiquitous activity — research programming — into parts that are better suited for humans and those better suited for machines. How can we optimally exploit machines for what they do well (e.g., tracking experimental processes), and how do we get them to stay out of our way for the parts where humans can be more effective? More simply, how can we maximize human productivity by offloading all of the unsavory work to machines? However, one danger of automation is lack of transparency: How can we also keep humans “in the loop” so that they can gain the proper insights and debug when experiments do not go according to plan? The five tools I have presented in this dissertation are examples of design along this spectrum of mixed-initiative interfaces [90].

In the past few decades, advances in optimizing compilers, high-level programming languages, integrated development environments, debuggers, automated software bug finders, and testing frameworks have made the lives of professional software engineers far more pleasant and productive than during the early days of computing. The five tools I presented are small steps toward ensuring that similar improvements occur for the large and fast-growing population of research programmers. In closing, the most important contribution of this dissertation might not necessarily be the details of my prototype solutions, but rather raising awareness for the significance of the problems.

Bibliography

[1] Apache Hadoop home page http://hadoop.apache.org/.

[2] AT-SPI: http://projects.gnome.org/accessibility/.

[3] Burrito project home page: http://www.pgbovine.net/burrito.html.

[4] CDE public source code repository, https://github.com/pgbovine/CDE.

[5] Coq proof assistant: Bug 2443, http://coq.inria.fr/bugs/show_bug.cgi?id=2443.

[6] Dropbox - Simplify your life, http://www.dropbox.com/.

[7] GCC compiler: Bug 46651, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651.

[8] Hashtag: http://en.wikipedia.org/wiki/Hashtag.

[9] IncPy home page: http://www.pgbovine.net/incpy.html.

[10] JSON data format: http://www.json.org/.

[11] List of software package management systems, http://en.wikipedia.org/wiki/List_of_software_package_management_systems.

[12] LLVM compiler: Bug 8679, http://llvm.org/bugs/show_bug.cgi?id=8679.

[13] Mac OS X Bundle Programming Guide: Introduction, http://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html.

[14] Mathematica notebook technology, `http://www.wolfram.com/technology/nb/`.

[15] MongoDB: `http://www.mongodb.org/`.

[16] Official Python wiki: Large Python Projects `http://wiki.python.org/moin/LargePythonProjects`.

[17] Parallel Python `http://www.parallelpython.com/`.

[18] Python home page: PEP 255 — Simple Generators `http://www.python.org/dev/peps/pep-0255/`.

[19] R benchmark 2.5: `http://r.research.att.com/benchmarks/R-benchmark-25.R`.

[20] RescueTime: `https://www.rescuetime.com/`.

[21] Saturn online discussion thread, `https://mailman.stanford.edu/pipermail/saturn-discuss/2009-August/000174.html`.

[22] SlopPy home page: `http://www.pgbovine.net/SlopPy.html`.

[23] Spec cpu2006 benchmarks, `http://www.spec.org/cpu2006/`.

[24] SSH Filesystem, `http://fuse.sourceforge.net/sshfs.html`.

[25] Supercomputer Event Logs `http://www.cs.sandia.gov/~jrstear/logs/`.

[26] SystemTap: `http://sourceware.org/systemtap/`.

[27] The Phoenix System for MapReduce Programming `http://mapreduce.stanford.edu/`.

[28] Tutorial: Static, Shared Dynamic and Loadable Linux Libraries, `http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html`.

[29] `arachni` project home page, `https://github.com/Zapotek/arachni`.

[30] `graph-tool` project home page, `http://projects.skewed.de/graph-tool/`.

[31] Viterbi algorithm `http://en.wikipedia.org/wiki/Viterbi_algorithm`.

[32] VMware ThinApp User's Guide, `http://www.vmware.com/pdf/thinapp46_manual.pdf`.

[33] Stopping silent errors with exceptions `http://perltraining.com.au/tips/2005-04-12.html`. *Perl training Australia*, 2005.

[34] Umut A. Acar. Self-Adjusting Computation (An Overview). In *Plenary Talk at ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[35] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. PASTE '07, pages 43–48. ACM, 2007.

[36] Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott. PDS: A virtual execution environment for software deployment. VEE '05, pages 175–185, New York, NY, USA, 2005. ACM.

[37] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. SOSP '09, pages 193–206. ACM, 2009.

[38] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[39] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, CHI '09. ACM, 2009.

[40] M. Burnett. Visual programming. In J. G. Webster, editor, *Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons Inc., 1999.

[41] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, pages 209–224. USENIX Association, 2008.

[42] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society.

[43] John M. Chambers. *Statistical Models in S.* CRC Press, Inc., Boca Raton, FL, USA, 1991.

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 205–218. USENIX Association, 2006.

[45] Ewen Cheslack-Postava, Tahir Azim, Behram F.T. Mistree, Daniel Reiter Horn, Jeff Terrace, Philip Levis, and Michael J. Freedman. A scalable server for 3D metaverses. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX '12. USENIX Association, 2012.

[46] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion.*

[47] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *NSDI '10*. USENIX Association, 2010.

[48] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21. USENIX Association, 2010.

[49] A. Cypher. *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

[50] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc., New York, NY, 2003.

[51] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc., New York, NY, 2003.

[52] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10. USENIX Association, 2004.

[53] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[54] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *LISA '04, the 18th USENIX conference on system administration*, 2004.

[55] Mira Dontcheva, Steven M. Drucker, David Salesin, and Michael F. Cohen. Relations, cards, and search templates: user-guided web data integration and layout. In *ACM UIST*, pages 61–70, 2007.

[56] Per Cederqvist et al. *Version Management with CVS.* Network Theory Ltd., 2006.

[57] Stuart I. Feldman. Make — a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[58] Scott Fertig, Eric Freeman, and David Gelernter. Lifestreams: An alternative to the desktop metaphor. CHI '96, 1996.

[59] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05*, pages 295–304. ACM, 2005.

[60] Kathleen Fisher and David Walker. The Pads project: An overview. In *International Conference on Database Theory*, March 2011.

[61] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10:11–21, May 2008.

[62] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485–496, April 2008.

[63] Iddo Friedberg. Email: Skipping a bad record read in SeqIO — `http://portal.open-bio.org/pipermail/biopython-dev/2009-June/006189.html`. *Biopython dev. mailing list*, 2009.

[64] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09*. ACM, 2009.

[65] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: an extensible data cleaning tool. In *ACM SIGMOD*, page 590, 2000.

[66] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. NDSS '03, 2003.

[67] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. NDSS '04, 2004.

[68] N Gehani. Exceptional C or C with exceptions. *Software: Practice and Experience*, 22(10):827–848, 1992.

[69] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM POPL*, pages 317–330, 2011.

[70] Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 2011 USENIX Large Installation System Administration Conference*, LISA '11. USENIX Association, 2011.

[71] Philip J. Guo. Sloppy Python: Using dynamic analysis to automatically add error tolerance to ad-hoc data processing scripts. In *Proceedings of the 2011 International Workshop on Dynamic Analysis*, WODA'11. ACM, 2011.

[72] Philip J. Guo and Dawson Engler. Linux kernel developer responses to static analysis bug reports. In *Proceedings of the 2009 USENIX Annual Technical Conference (short paper)*, USENIX '09. USENIX Association, 2009.

[73] Philip J. Guo and Dawson Engler. Towards practical incremental recomputation for scientists: An implementation for the Python language. In *TaPP '10: Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance*, 2010.

[74] Philip J. Guo and Dawson Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference (short paper)*, USENIX '11. USENIX Association, 2011.

[75] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11. ACM, 2011.

[76] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive wrangling: mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th symposium on User Interface Software and Technology*, UIST '11. ACM, 2011.

[77] Philip J. Guo and Margo Seltzer. Burrito: Wrapping your lab notebook in computational infrastructure. In *TaPP '12: Proceedings of the 4th USENIX Workshop on the Theory and Practice of Provenance*, 2012.

[78] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10. ACM, 2010.

[79] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "Not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11. ACM, 2011.

[80] Karl Gyllstrom. Passages through time: chronicling users' information interaction history by recording when and what they read. IUI '09, pages 147–156, New York, NY, USA, 2009. ACM.

[81] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 9–16. VLDB Endowment, 2006.

[82] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *PLDI '09*, pages 25–37. ACM, 2009.

[83] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[84] William Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *ACM PLDI*, 2011.

[85] Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. d.note: revising user interfaces through change tracking, annotations, and alternatives. CHI '10, pages 493–502, New York, NY, USA, 2010. ACM.

[86] Jeffrey Heer, Fernanda Viegas, and Martin Wattenberg. Voyagers and voyeurs: Supporting asynchronous collaborative information visualization. In *ACM Human Factors in Computing Systems (CHI)*, pages 1029–1038, 2007.

[87] Kurtis Heimerl, Janani Vasudev, Kelly G. Buchanan, Tapan Parikh, and Eric Brewer. Metamouse: Improving multi-user sharing of existing educational applications. In *International Conference on Information and Communication Technologies and Development*, 2010.

[88] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *PLDI '00*, pages 311–320. ACM, 2000.

[89] Paul Heymann and Hector Garcia-Molina. Contrasting controlled vocabulary and tagging: Do experts choose the right names to label the wrong things? In *ACM International Conference on Web Search and Data Mining, Late Breaking Results Session*, February 2009.

[90] Eric Horvitz. Principles of mixed-initiative user interfaces. In *ACM CHI*, pages 159–166, 1999.

[91] David Huynh and Stefano Mazzocchi. Google Refine. `http://code.google.com/p/google-refine/`.

[92] David F. Huynh, Robert C. Miller, and David R. Karger. Potluck: semi-ontology alignment for casual users. In *ISWC*, pages 903–910, 2007.

[93] Michal Jacovi, Vladimir Soroka, Gail Gilboa-Freedman, Sigalit Ur, Elad Shahar, and Natalia Marmasse. The chasms of cscw: a citation graph analysis of the

cscw conference. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 289–298, New York, NY, USA, 2006. ACM.

[94] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. NDSS '00, 2000.

[95] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization Journal*, 10:271–288, 2011.

[96] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM CHI*, 2011.

[97] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Advanced Visual Interfaces*, 2012.

[98] Diane Kelly, Daniel Hook, and Rebecca Sanders. Five recommended practices for computational scientists who write software. *Computing in Science and Engineering*, 11:48–53, 2009.

[99] Mats Kihln. Electronic lab notebooks – do they work in reality? *Drug Discovery Today*, 10(18):1205 – 1207, 2005.

[100] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. USENIX '05. USENIX Association, 2005.

[101] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40, July 2006.

[102] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. DejaView: a personal virtual computer recorder. SOSP '07, pages 279–292, New York, NY, USA, 2007. ACM.

[103] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pages 155–166, 2010.

[104] Mayank Lahiri and Manuel Cebrian. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, 2010.

[105] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[106] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM TODS*, 26(4):476–519, 2001.

[107] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[108] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter: automating & sharing how-to knowledge in the enterprise. In *ACM CHI*, pages 1719–1728, 2008.

[109] Jure Leskovec. *Dynamics of Large Networks*. PhD thesis, Carnegie Mellon University, 2008.

[110] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. End-user programming of mashups with Vegemite. In *IUI*, pages 97–106, 2009.

[111] Greg Little and Robert C. Miller. Translating keyword commands into executable code. In *ACM UIST*, pages 135–144, 2006.

[112] Jon Loeliger. *Version Control with Git*. O'Reilly Media, 2009.

[113] Edward Loper and Steven Bird. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics*, 2002.

[114] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[115] Diana MacLean. Provenance, PASS & People: a Research Report. Technical report, Harvard University, 2007.

[116] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[117] Donald Michie. ”Memo” Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[118] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Tech. Conf.*, pages 161–174, 2001.

[119] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana MacLean, Daniel Margo, Margo Seltzer, and Robin Smogor. Layering in provenance systems. USENIX ’09. USENIX Association, 2009.

[120] Kiran-Kumar Muniswamy-Reddy and David A. Holland. Causality-based versioning. FAST ’09, pages 15–28, 2009.

[121] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, USENIX ’06. USENIX Association, 2006.

[122] Brad A. Myers, Andrew J. Ko, and Margaret M. Burnett. Invited research overview: end-user programming. In *ACM CHI Extended Abstracts*, pages 75–80, 2006.

[123] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Virtual execution environments*, VEE '07. ACM, 2007.

[124] William Stafford Noble. A quick guide to organizing computational biology projects. *PLoS Computational Bio.*, 5(7), 07 2009.

[125] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100, 2006.

[126] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *DSN '07*, Washington, DC, USA, 2007. IEEE Computer Society.

[127] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference*, USENIX '99. USENIX Association, 1999.

[128] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[129] Bryan O'Sullivan. *Mercurial: The Definitive Guide.* O'Reilly Media, 2009.

[130] Kayur Patel, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. Gestalt: integrated support for implementation and analysis in machine learning. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 37–46, New York, NY, USA, 2010. ACM.

[131] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[132] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[133] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A survey of the practice of computational science. In *State of the Practice Reports*, SC '11. ACM, 2011.

[134] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proc. VLDB*, 2001.

[135] Jun Rekimoto. Time-machine computing: a time-centric approach for the information environment. UIST '99, New York, NY, USA, 1999. ACM.

[136] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI '04*. USENIX Association, 2004.

[137] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[138] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *AADEBUG*, pages 69–76. ACM Press, 2005.

[139] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*. Springer, 2005.

[140] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, pages 110–123, New York, NY, USA, 1999. ACM.

[141] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03, the 17th USENIX conference on system administration*, 2003.

[142] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2005.

[143] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and re-using workflows with VisTrails. In *SIGMOD '08*. ACM, 2008.

[144] Jiwon Seo. personal email communication, discussing sequential vs. parallel scripts, 2009.

[145] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using provenance to aid in personal file search. USENIX '07, pages 13:1–13:14. USENIX Association, 2007.

[146] Richard P. Spillane, Charles P. Wright, Gopalan Sivathanu, and Erez Zadok. Rapid file system development using ptrace. In *Experimental Computer Science*. USENIX Association, 2007.

[147] Carl Staelin. mkpkg: A software packaging tool. In *LISA '98, the 12th USENIX conference on system administration*, 1998.

[148] Christina Strong, Stephanie Jones, Aleatha Parker-Wood, Alexandra Holloway, and Darrell D. E. Long. Los Alamos National Laboratory Interviews. Technical Report UCSC-SSRC-11-06, UC Santa Cruz, 2011.

[149] Ioan A. Sucan and Lydia E. Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics*, pages 449–464, 2008.

[150] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42, January 2008.

[151] A. Thusoo, J.S. Sarma, N. Jain, Zheng Shao, P. Chakka, Ning Zhang, S. Antony, Hao Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE)*, pages 996 –1005, March 2010.

[152] Walter F. Tichy. RCS — a system for version control. *Software: Practice and Experience*, 15(7):637–654, July 1985.

[153] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. Building mashups by example. In *ACM IUI*, pages 139–148, 2008.

[154] Hal Varian. Hal Varian on how the Web challenges managers. In *McKinsey Quarterly*, Jan 2009.

[155] Tao Yue. personal email communication, July 11, 2010.

[156] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (Software Engineering in Practice track)*, ICSE '12. ACM, 2012.

Philip Jia Guo

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dawson Engler) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Jeffrey Heer)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Margo Seltzer)

Approved for the University Committee on Graduate Studies