Sciunits: Reusable Research Objects

Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, Tanu Malik
School of Computing
DePaul University, Chicago, IL, USA
Email: {*dtonthat, gfils1, zhihao.yuan, tmalik1*}*@depaul.edu*

***Abstract*—Science is conducted collaboratively, often requiring knowledge sharing about computational experiments. When experiments include only datasets, they can be shared using Uniform Resource Identifiers (URIs) or Digital Object Identifiers (DOIs). An experiment, however, seldom includes only datasets, but more often includes software, its past execution, provenance, and associated documentation. The Research Object has recently emerged as a comprehensive and systematic method for aggregation and identification of diverse elements of computational experiments. While a necessary method, mere aggregation is not sufficient for the sharing of computational experiments. Other users must be able to easily recompute on these shared research objects. In this paper, we present the *sciunit*, a reusable research object in which aggregated content is recomputable. We describe a Git-like client that efficiently creates, stores, and repeats sciunits. We show through analysis that sciunits repeat computational experiments with minimal storage and processing overhead. Finally, we provide an overview of sharing and reproducible cyberinfrastructure based on sciunits gaining adoption in the domain of geosciences.**

I. Introduction

Research objects—aggregations of digital artifacts such as code, data, scripts, and temporary experiment results—provide a means to share knowledge about computational experiments. In recent times, sharing computational experiments has become vital; scientific claims, inevitably asserted via computational experiments, remain poorly verified in text-based research papers. Research objects, together with the paper, provide an authoritative and far more complete record of a piece of research.

Several tools now exist to help authors create research objects from a variety of digital artifacts (see [1] for several tools and [2] for a variety of research objects). The tools enable research objects to be shared on websites that disseminate scholarly information, such as Figshare [3]. Despite their advantages, shared research objects do not permit easy reuse of their contents to verify their computations, or easy adaptation of their contents for reuse in new experiments. Often, the extent of reuse is subject to the amount of accompanying documentation, which may be limited to compilation and installation instructions. If documentation is scanty, research objects will remain unused.

The minimum use case for sharing a computational experiment (in the form of a shared research object) involves repeating its original execution and verifying its results. To truly exploit its potential, however, it must support modified reuse. Therefore, the research object must be created and stored not as a simple aggregation of digital content, as previously advocated [4], [5], but in a readily-computable form: as a *reusable* research object. We demonstrate the distinction in two ways.

Consider a typical research paper with an analysis based on large amounts of code and data, and assume that the researcher authoring the paper has used the code and data to conduct a number of experiments that produce the paper's target figures and results. The example paper's digital artifacts relating to its experiments may be bundled together in a medium such as a file archive (.tar), compressed file format (.gz), virtual image, or container. A shared research object is free to use any of these mediums. A reusable research object, however, must use a virtual image or container, since it must produce a "computational research object" that, when downloaded and shared, will guarantee an instantly-executable unit of computation.

Also consider the example paper's metadata, which, similar to the metadata in most papers, is interspersed throughout the project's written analysis, and throughout its code and data. The metadata can take many forms, including annotations, version information, and provenance. A shared research object's metadata usually serves a purely informational purpose, and is seldom used literally in the paper's experiments. A reusable research object, however, utilizes literal metadata by directly linking it to the code and data of the experiments. In particular, provenance, if collected in standard form, can guide different forms of reusable analysis – exact, partial, or modified reuse. Keywords and annotations can provide reference to additional datasets for modified reuse. In other words, a reusable research object can execute conditionally based on its embedded metadata, instead of simply including it as a stand-alone digital artifact that requires more interpretive labor to reason about and reuse.

In this paper, we describe the *sciunit*, a reusable research object that has a lifetime beyond being shared on scholarly exchange websites. The sciunit does not simply bundle digital artifacts, but uses application virtualization (AV) to automatically create a container of an executable application. In AV, operating system calls during application execution are interrupted to enable the copying of all binaries, data, and software dependencies into a container. The resulting container is portable and instantly reusable: it can be run on any compatible machinewithout installation, configuration, or root permissions.

Similar to shared research objects, users can attach additional annotations to reusable research objects to describe containers. Each container also incorporates associated provenance, and users can use the included provenance to create repurposed containers These containers enable exact or partial repeatability of the sciunit.

While AV facilitates the creation of reusable research objects, in its traditional form [6], [7] it is inherently inefficient when used to create multiple containers that are each based on slight modifications of an application. On repeated container creation, traditional AV methods behave the same regardless of the amount of similarity that exists between the original and modified applications. Traditional methods always create an entirely new container, which will contain wholesale duplication of digital artifacts, such as system dependencies, common binaries, or even common data files. Additionally, a large digital artifact present in two application versions, but that changes only slightly in content, will still consume its full amount of space in each corresponding container. Thus space consumption grows substantially, which is particularly of issue when a user shares different versions of an analysis or pipeline. We show how, when using our own AV method, multiple containers can be stored efficiently in one sciunit using a common block-based storage based on content deduplication techniques [8].

We further increase reusability of sciunits by using their embedded metadata to help guide in their comprehension and modification. In particular, we use included provenance to provide an overview of the overall workflow of a container. When AV techniques are used to create a container, traditionally the collected provenance information is at the file and process level, which is too fine-grained to show the overall workflow.

This paper makes the following contributions: (i) We present Sciunit-CLI[9], a Python/C-based Git-like client that creates sciunits, shares them on scholarly exchange websites such as Figshare and Hydroshare, and repeats shared sciunits either locally or remotely; (ii) We describe the AV method used in Sciunit-CLI to build a container for reuse; (iii) We describe versioned storage based on content deduplication methods to efficiently store multiple containers in a single sciunit; and (iv) We describe the interactive provenance visualization that summarizes embedded provenance in a container and simplifies repeating the container partially or modifying it.

The rest of the paper is organized as follows:

- Section II: overall architecture of our work.
- Section III: creating a sciunit using application virtualization that builds a container with embedded provenance.
- Section IV: storing multiple containers in a single sciunit.
- Section V: utilizing provenance within a sciunit for repeating and reuse.
- Section VI: optimizing the embedded provenance for visualization in summarized graphs.
- Section VII: detailed experimental analysis.
- Section VIII: evolution of research objects, and their creation and use in related applications.
- Section IX: conclusions.

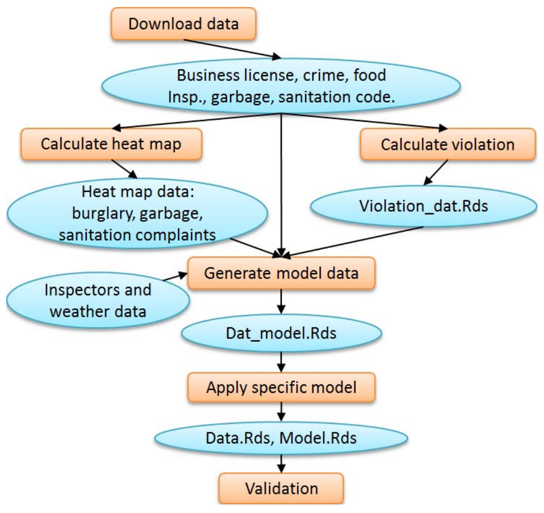


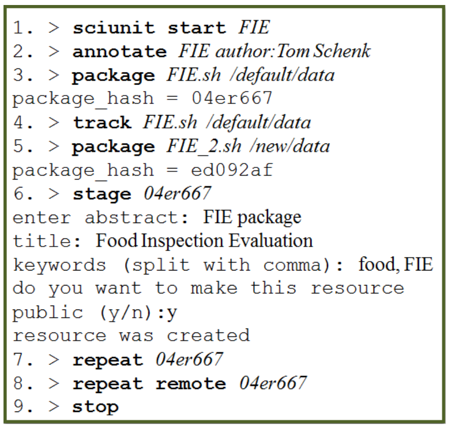
Fig. 1. Conceptual view of the steps required to run the Food Inspection Evaluation [10] predictive model

II. The Sciunit-CLI: Architecture and use

Our reference implementation is a client program, the Sciunit-CLI[9] that creates, stores, and executes reusable research objects. We use a real-world example to highlight the primary commands and salient features of the client. Figure 1 shows an example of a predictive model used for forecasting critical violations during sanitation inspection [10]. The software consists of scripts written in different languages (R and Shell) that operate on input datasets acquired from the City of Chicago Socrata data portal [11]. The output of the predictive model is continually tested using a double-blind retrodiction; the Department of Public Health conducts inspections via its normal operational procedure, which are compared with the output of the model. The pre-processing code is shared on GitHub [12], the data is available via public repositories [11], and the predictive model analysis is also published [13].

Bundling these artifacts into a shared research object would likely be inefficient given data from nine different sources, which changes periodically, making analysis conducted within a certain time range obsolete. The Sciunit-CLI can be used to build a reusable research object consisting of identifiers of one or more re-executable containers, along with other listed digital artifacts.

The Sciunit-CLI is a Git-like Python/C command-line interface (CLI) client used to build sciunits. Figure 2 shows a sample user interaction with this client. The user instantiates a namespaced sciunit titled *FIE* (Line 1), and can associate annotations with the sciunit (Line 2). To create a container within the sciunit, the user runs the application with the *package* command (Line 3). Packaging an application also incorporates provenance information of the application run; provenance can also be audited without creating a container (Line 4). Many containers can be created within the same



```
1. > sciunit start FIE
2. > annotate FIE author:Tom Schenk
3. > package FIE.sh /default/data
package_hash = 04er667
4. > track FIE.sh /default/data
5. > package FIE_2.sh /new/data
package_hash = ed092af
6. > stage 04er667
enter abstract: FIE package
title: Food Inspection Evaluation
keywords (split with comma): food, FIE
do you want to make this resource
public (y/n):y
resource was created
7. > repeat 04er667
8. > repeat remote 04er667
9. > stop
```

Fig. 2. User interaction with sciunit client

sciunit by using the *package* command again (Line 5).

The *package* command makes minimal assumptions regarding the nature of the application. In particular, the user application can be written in any combination of programming languages, e.g. C, C++, Fortran, Shell, Java, R, Python, Julia, etc, or be used as part of a workflow system such as Galaxy [14], Swift [15], Kepler [16] etc. While our description assumes local execution, in practice, an application's execution can be either local or distributed. We choose an example with local execution since the description of the underlying AV method (Section III) for distributed and parallel applications, such as database applications [17] and HPC programs [18] is beyond the scope of the current paper.

The sciunit is stored locally unless explicitly shared with a remote repositoryusing the *stage* command, which instructs the client to upload the container to a Web-based repository (Line 6). The Sciunit-CLI uses Hydroshare [19] for geoscience applications and Figshare [3] otherwise as its Web-based repository.

A container within the sciunit can be re-run directly on the local machine with the *repeat* command, or on a remote execution server with the *repeat remote* command (Lines 7 and 8). In our reference implementation, remote execution refers to execution via Hydroshare. On remote execution, the target container is automatically downloaded to a remote execution server, and, if the container is compatible with the execution server's architecture, the execution server runs it and sends the results back to the user. The user can also modify a container by downloading it, modifying its code or data and running it locally, and then uploading the modified container, at which point a new version of the container can be staged.

Further improvements of the *repeat remote* command such as connecting with the remote server via *ssh* or enabling partial remote executions is part of our ongoing work. The client, and accompanying server-side infrastructure that stores and manages sciunits, form a reproducible infrastructure, currently in use within the geosciences domain in the United States (http://geotrusthub.org). The site provides full technical documentation and examples from domain sciences using the client.

III. Creating sciunits

Tools based on application virtualization method typically run in two modes: an audit mode to create a container, and an execution mode to re-run a container [6], [7]. In AV audit mode, a container of a user application is created as the user executes the application (in the context of auditing, such an execution is termed a *reference execution*). We describe the audit process assuming that the application is running on a Linux machine. During execution, the Linux *strace* utility is used to monitor the running application process. *Strace* internally attaches itself to the process using the *ptrace* system call to monitor all the system calls of the running process. It intercepts each system call[1] to determine the running process' state and the arguments to the system call. For example, when a process accesses a file or a library using the system call *fopen()*, the *fopen()* call is intercepted. The intercepted system call is "paused" to examine input arguments and the process control block. For instance, in *fopen()*, the file path parameter is extracted. By intercepting all calls, AV auditing determines all[2] program binaries, libraries, scripts, and environment variables that a user program is dependent on. Inclusion of data files is optional, which the user may or may not want to package based on the size of the dataset. The audit process is similar for Windows and macOS, except that different OS-specific monitoring utilities are used.

The system call pause time is brief, requiring only two lightweight context switches added to the normal system call flow; experiments show that the overhead of intercepting system calls is minimal. During the pause, the identified dependencies are used in two ways: first, to create a "sandbox" application container that includes all identified dependencies, and second, to create an interaction log of the reference execution. The sandbox container is named with a package hash and placed in a special "root path" (as described in Section IV), and contains all the dependencies that were identified during the reference execution audit. The dependencies are placed at the same path within the special root path as they were identified in the original system. Figure 3 shows the contents of a container.

The interaction log generated during the AV audit phase contains interactions between processes when they are forked or execed, or between processes and files when files are opened or closed. The log also stores the logical range of times that processes interacted with other processes or with files. A provenance graph is obtained by toplogically sorting the interaction log.

In AV execution mode, the application is executed from the container itself by monitoring its processes with *strace*, interrupting application system calls and extracting their path

[1]There are approximately 50 such calls defined in the POSIX standard

[2]Not all program dependencies can be detected through this method. But a program's static dependencies are much simpler to gather using programs such as *file, ldd, strings, and objdump*. Our client provides commands for users to find additional dependencies, and include them, if necessary.

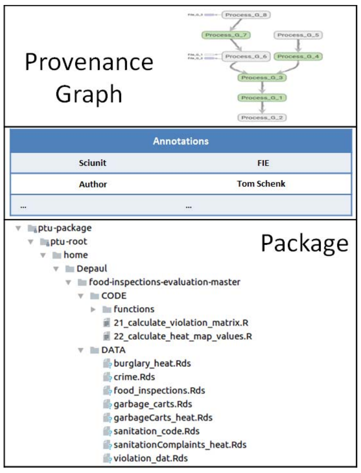


Fig. 3. Example sciunit container

arguments, and redirecting all system call paths to paths within the special root path of the sandbox container. By redirecting all file requests into the container, the AV execution method fools the application program into believing that it is executing on the original audit-time machine with original file paths [7].

The advantages of using the AV method are the ease with which a reusable research object can be created, and the machine-agnostic reuse that such an object provides. The disadvantages of the method are that the generated provenance is too fine-grained (at the file and process level) for ready analysis, and that repeated containerization can lead to many redundant files in the same research object. We address these two concerns in the next two sections.

IV. Storing sciunits

A reusable research object may include many containers. If the AV audit method described in Section III is used on an application to create a container for a sciunit, then each time that same application is audited all the same file dependencies of the application will be copied into a new container. This copying takes place even if the same dependencies were present in other previously-created containers based on the same audited application. One way to eliminate such dependencies is to check for duplicate dependencies as the container is created during the AV audit phase. However, this slows the audit phase down delaying the construction of the container. The Sciunit-CLI de-duplicates on completion of the *package* command as a background process.

Sciunit-CLI uses content-defined chunking to divide the container's content into small chunks identified by a hash value. New chunks are compared to stored chunks, and whenever matches occur, redundant chunks are replaced with small references that point to stored chunks.

To identify chunks in a file, we do not use fixed-size chunking, which is simple and fast but faces the problem of low de-duplication ratio that stems from the boundary-shift problem [8]. For example, if one or several bytes are inserted at the beginning of a file, all current chunk boundaries declared by FSC will be shifted and no duplicate chunks will be detected. Instead we use content-defined chunking (CDC)[8], that uses a sliding window technique on the content of files and computes a hash value (e.g., Rabin fingerprint [20]) of the window. In Rabin CDC, the Rabin hash for a window containing a n byte sequence $B_1, B_2, \ldots, B_n$ is defined as a polynomial $RH(X_{(i,n)}) =:$

$$RH(B_1, B_2, ..., B_n) = \{\sum_{x=1}^{n} B_x p^{n-x}\} \bmod D \qquad (1)$$

in which D is the average chunk size. Rabin hash is a rolling hash algorithm since it is able to compute the hash in an iterative fashion, i.e., the current hash can be incrementally computed from the previous value using a recurrence relation defined as:

$$RH(X_{(i,n)}) \leftarrow (RH(X_{(i-1,n)}) + X_i - X_{(i-n)}) \bmod M \qquad (2)$$

in which n is the window size, $X_{(i,n)}$ represents the window bytes at byte position 'i', and M is the total length of the file. Using the recurrence relation, the hash value at any byte position i can be cheaply computed from the hash at byte position $i-1$. A chunk boundary is declared if the hash value satisfies some pre-defined condition, such as if the lowest k bits of the Rabin hash value match a threshold value.

Content-defined de-duplication is used in the popular Linux utility *rsync* and we use it in a similar way in our work. However, unlike *rsync* we search for hashes differently. In particular, instead of using a combination of fixed-size and rolling hashes, as used in *rsync*, we simply iterate over all calculated hashes, speeding up computation. This is justifiable since we expect each research object to be fairly modest in size, unlike large-scale storage and backup systems where *rsync* is commonly used.

Once rolling hashes have been computed from a file, and a different block is detected, the difference itself can be be stored either as a delta or as a distinct block. The delta method is typically used when the predominant use case is to efficiently obtain a specific version of a file. In our case, we needed to strike a balance between storing multiple overlapping containers and storing versions of a single container. Thus we chose the distinct block method, as shown in Figure 4, in which all unique blocks across all containers, versioned or not, are stored.

Given this optimization, a container then is just a symbolic view over deduplicated storage, as shown in Figure 4. However, for the user this use of optimized storage is opaque. The Sciunit-CLI uses a manifest to store multiple containers, and to select a specific container to run. To run, the client first materializes the selected container by enumerating and simply concatenating the blocks corresponding to the selected container. The materialization requires negligible processing. This procedure is fundamentally different from a delta-based mechanism, in which the blocks corresponding to a selected container will have to first reconstructed by applying the deltas.

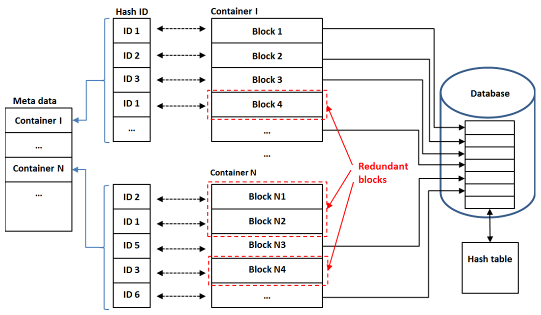


Fig. 4. Block-based deduplication of containers

V. Reusing sciunits

When a sciunit is published, the server distinguishes between the computational part (i.e. the application container) and the non-computational part (i.e. the informational digital artifacts) of the sciunit. The computational part is associated with a cloud instance that can remotely execute the container on user request. A new user can reuse a published sciunit in one of three ways: (i) exact repeat-execution, (ii) partial repeat execution, or (iii) modified repeat execution. To exactly repeat, a container is simply downloaded and then run locally with the *repeat* command, or run remotely with the *repeat remote* command: the container will execute exactly as it did when it was created with the *package* command. To partially repeat or run a modified repeat, a container is downloaded, processed for partial or modified execution, and then either run locally or published to run remotely. We now describe the processing required for partial and modified repeat executions in detail.

A. Partial Repeat Execution

To partially repeat, a user selects one or multiple processes within a container. These processes are identified by their short pathname or PID, and the user can also use the provenance graph to aid in identification. While the provenance graph can be quite detailed for a user to choose specific processes, in Section VI we describe how a user can see a summarized application workflow akin to the workflow presented in Figure 1 from the provenance graph. Thus, for example, using the container from Figure 1, a user selects the processes "Calculate violation" and "Generate model data" as the group of processes to be partially repeated. Since this user-selected group of processes may not include all related processes needed for re-execution, we must determine these related processes, along with the data files they reference. The determined processes and files will constitute the new "partial repeat" container or "sub-container". Algorithm 1 shows the procedure for building the sub-container. It starts with the list of user-selected processes (*selectedProcs*), and progresses to include all relevant processes and files by traversing the lineage of the graph (Lines 10-16). The *getDeps* function assumes that any intermediate data files, if included as dependencies, still exist as generated from previous execution runs. The execution of this algorithm ensures that the data file "Heat map data",

Algorithm 1: Build sub-container for partial execution

```
1  BuildSubContainer(selectedProcs, container):
2      subContainer = initialize(container)
3      allProcs = getAllProcs(container)
4      requiredProcs = getProcs(selectedProcs,
         allProcs)
5      reqProcDeps = getDeps(requiredProcs)
6      foreach dep in {reqProcDeps} do
7          /* add dep to correct location in subContainer */
8          add(dep, container, subContainer)
9      return subContainer
10 getProcs(selectedProcs, allProcs):
11     result = {selectedProcs}
12     foreach proc in {allProcs} do
13         foreach selProc in {selectedProcs} do
14             if isDescendant(proc, selProc) then
15                 result = result ∪ proc
16                 break
17     return result
18 getDeps(requiredProcs):
19     result = ∅
20     foreach reqProc in {requiredProcs} do
21         /* retrieve all related files and dependencies */
22         deps = relevantResources(reqProc)
23         result = result ∪ deps
24     return result
```

generated from the previous run of the process "Calculate heat map", is included in the sub-container, even though in the new partial repeat execution the process "Calculate heat map" will not be re-executed.

B. Modified Repeat Execution

To run a modified repeat of a sciunit container, a user examines a downloaded container and determines how particular computations within it should be modified (e.g. by modifying certain sections of code or input data). The sciunit's included provenance graph aids this modification task greatly. Next the user runs the modified container. To share the modification, the user would simply run it with the *package* command, and then publish it with the *stage* command. Enabling modification through a visualization mode, in which users can specify alternate processes or input data files assisted by a GUI, is part of future work.

VI. Provenance graph visualization

Provenance information generated by AV audit methods is fine-grained. A graph created from a complete set of generated provenance, using normal visualization structures such as tree or list representations, would be far too replete to be of real practical value. When viewed, this graph would present significant system-level detail that would inhibit a basic comprehension of the overall application workflow. For example, the intuitive workflow of Figure 1, consisting of 12 nodes and 13 edges, would be represented fully as a dense graph of 146 nodes and 321 edges (Figure 5(a) shows a part of this replete graph). Thus, to create a more intuitive graph, we use a graph summarization method that condenses the low-level details of the full generated provenance information. The graph summarization method is explained in detail in [**?**], and is briefly described in this section. We further describe how we extend the summarization method to create a graph that presents dynamic workflow cross-sections in a responsive visual interface.

Given a directed graph $G = (V, E)$, where V is the set of vertices[3] and E is the set of edges, we denote $Input(u)$ and $Output(u)$ as the sets of input and output edges of vertex u. Respectively, $Input(u) = \{e | \exists v \in V,\ e = (v, u) \in E\}$, and $Output(u) = \{e | \exists v \in V,\ e = (u, v) \in E\}$. The direction of an edge characterizes the dependency of its vertices. For example, a process u spawned by process v is represented by the edge (u, v), and a file u read by process v is represented by the edge (v, u). The graph G is summarized based on the following two rules:

Definition 1 (Similarity): Two vertices u and v are called *similar* if and only if they share the same type and have the same input and output connection sets: $Type(u) = Type(v)$, $input(u) = input(v)$ and $output(u) = output(v)$.

The similarity rule groups multiple vertices into a single vertex if the vertices have same type and are connected by the same number and type of edges. Additionally, edges of similar vertices will be grouped into a single corresponding edge. When applied to our provenance graph, this rule groups different files in the same directory.

Definition 2 (Packability): A vertex u belongs to v's $generalization\ set$ if and only if vertex u connects to v and satisfies one of following conditions:

- Vertex u is a file that has only one connection to process v: $Type(u) = file$ and $\{\exists! e \mid e \in E \wedge (e = (u, v) \vee e = (v, u))\}$.
- Vertex u is a process that has only one output connection to process v: $Type(u) = process$ and $\{\exists! e \mid e \in E \wedge e = (u, v)\}$.
- Vertex u is a file that has only two connections – an output connection to process v and an input connection another process x: $Type(u) = file$ and $\{\exists!(e_1, e_2) \mid (\exists x \in V,\ v \neq x)\ \wedge (e_1 = (u, v) \in E, e_2 = (x, u) \in E)\}$.

The packability rule identifies hubs in the provenance graph by packing files or processes that are connected by single edges into their parent nodes. It also packs files that are generated by a single process and consumed by a single process into their parent processes by producing a process-to-process edge.

When applied in sequence, the similarity and packability rules condense the detail-level of a graph while preserving its core workflow elements. Figure 5 illustrates how applying these two rules to a replete graph produces a graph summary that shows the primary processes in a workflow. Figure 5(a) presents the original replete provenance graph of one subtask of the FIE workflow (the data processing steps "Calculate Violation" and "Calculate Heat Map" of Figure 1). Applying the two summarization rules produces the final graph in Figure 5(c), which is similar to the conceptual workflow (except it is upside-down, due to the nature of provenance data flow).

To lay out the summarized graph, we adopt two visualization techniques: scoping and annotation. In scoping, nodes similar to each other or packed together are represented as single nodes, which can be expanded on user action to reveal the details they conceal. For example, in Figure 7, similarity and packability rules group the nodes within the box into the single node "P_R_27070" (process 27070 runs a subprocess using file "21_calulate_violation_matrix.R" and writes data to file "violation_data.Rds"). The expanded view within the box was obtained by clicking on the concealing node "P_R_27070." Here "Process_G_5" is another concealing node hiding all the dependencies of the R process calculating the violation matrix.

To further improve the layout of the graph, we use an annotation method that assigns higher visualization precedence to process nodes, but annotates them with corresponding file nodes. Figure 5(d) shows how the annotation "File_G_2," which is a library dependency used both by "P_R_27070" and "P_R_27091," is attached to the two process nodes that generated it. Thus, given a file with n edges ($n \geq 2$), we replace this file with n annotations. A user can always toggle the expanded view to see how the file and process nodes were originally connected. We choose to annotate files – instead of processes – since an application workflow is typically defined by the primary processes that it runs.

VII. Experiments

The true usefulness of sciunits can only be measured by their adoption. Efficiency of creating sciunits can be a driving force in adopting the use of sciunits over traditional shared research objects. When an efficiently-versioned, easily-created sciunit is shared, along with an embedded, self-describing application workflow, we believe the probability for reuse will greatly increase. In this section, through two complex real-world workflows, we quantify the performance of packaging and repeating sciunits, the time and space overheads of storing them, and the efficiency of reusing them utilizing integrated provenance visualizations.

A. Use cases

We consider two real-world use cases for experimental evaluation: (i) the Food Inspection Evaluation (**FIE**) [10] workflow, a computationally-intense use case which has been the running example in our paper, and (ii) the Variable Infiltration Capacity (**VIC**) [21] model, an I/O-intensive data pre-processing pipeline for a hydrology model taken from geotrusthub.org. The first use case is notable for its transparency in its rigorous inspection audits, owing to the influence of the Open Data movement within the City of Chicago. The

[3]in our graph, a vertex is of type "file" or of type "process"

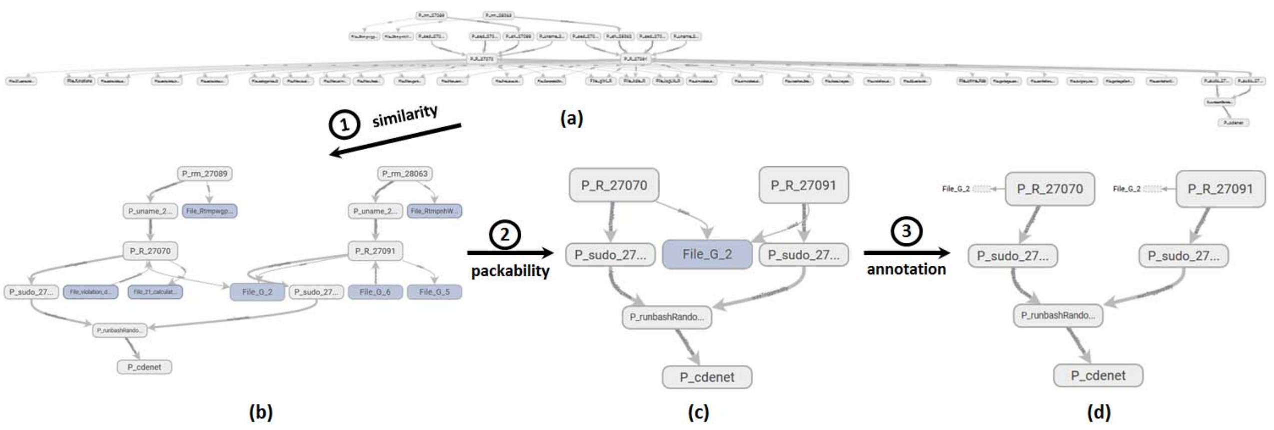


Fig. 5. Graph summarization of a replete graph

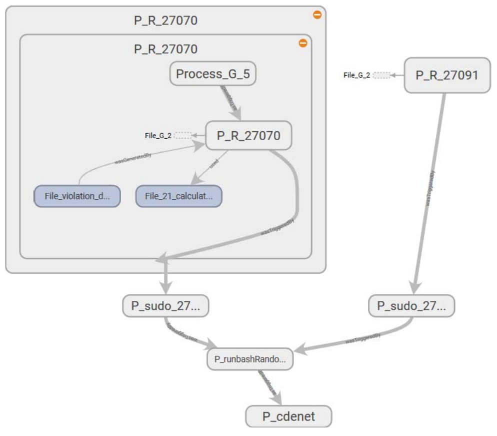
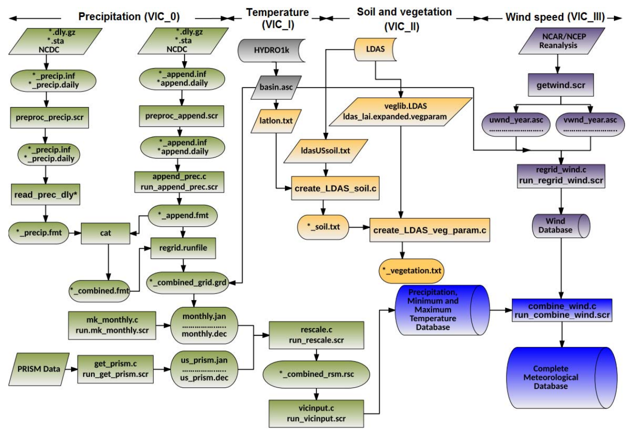


Fig. 6. Expanded view of concealing node ”P_R_27070”



Precipitation (VIC_0)
Temperature (VIC_I)
Soil and vegetation (VIC_II)
Wind speed (VIC_III)
*.dly.gz *.sta NCDC
*_precip.inf *_precip.daily
preproc_precip.scr
read_prec_dly*
*_precip.fmt
cat
*_combined.fmt
*_append.inf *append.daily
preproc_append.scr
append_prec.c run_append_prec.scr
*_append.fmt
regrid.runfile
*_combined_grid.grd
mk_monthly.c run.mk_monthly.scr
monthly.jan monthly.dec
PRISM Data
get_prism.c run_get_prism.scr
us_prism.jan us_prism.dec
rescale.c run_rescale.scr
*_combined_rsm.rsc
vicinput.c run_vicinput.scr
HYDRO1k
basin.asc
latlon.txt
LDAS
veglib.LDAS ldas_lai.expanded.vegparam
ldasUSsoil.txt
create_LDAS_soil.c
*_soil.txt
create_LDAS_veg_param.c
*_vegetation.txt
Precipitation, Minimum and Maximum Temperature Database
NCAR/NCEP Reanalysis
getwind.scr
uwnd_year.asc
vwnd_year.asc
regrid_wind.c run_regrid_wind.scr
Wind Database
combine_wind.c run_combine_wind.scr
Complete Meteorological Database

Fig. 7. Conceptual view of the VIC workflow [21]

second use case is a highly-relevant test bed for sciunits: the VIC model is very popular in the hydrology community, and its data preprocessing pipeline, which relies heavily on legacy code, is notoriously difficult to reassemble [21].

Tables I and II describe the details of FIE and VIC in terms of source code file programming languages, number of source code and data files, number of program files required as dependencies, and total application sizes (both FIE and VIC have four sub-tasks, labeled 0, I, II, and III, that are described below). Figures 1 and 7 show conceptual views of the application workflows for the two use cases.

Each use case is characterized by a shareable model, in which each step is conducted independently by one user, and subsequently shared with another user who builds upon or forks the shared workflow in the following step. Thus the FIE workflow, for example, is broken down into the following sub-tasks, each encapsulated in a single application: (i) FIE_0, which calculates a heat map from downloaded inspection records; (ii) FIE_I, which processes the heat map to generate data model inputs; (iii) FIE_II, which applies a specific model and validates it; (iv) FIE_III, which downloads the original inspection records and applies an end-to-end validation routine to the previous three sub-tasks. The download process of subtask iv is often the most time-consuming step.

The main sciunit client was implemented in Python and C. Sciunit's versioning tool was written in C++, using the block-based deduplication techniques proposed in [8] and [20]. Sciunit's provenance graph visualization was written in Python, using libraries from TensorBoard [22]. All sciunit client *package* and *repeat* experiments, along with their baseline normal application runs, were conducted on a laptop with an Intel Core i7-4750HQ 2.0 GHz CPU, 16 GB of main memory, and a 1 TB SATA SSD, running the Arch Linux 64-bit OS.

B. Creating Sciunits

Tables I and II present the baseline normal execution times for the sub-tasks of the two use cases. We note that each application encompasses substantial resources (in the form of code and data), has many external dependencies, and is also characterized by lengthy CPU-and-memory-intensive

TABLE I
FOOD INSPECTION EVALUATION SUB-TASK APPLICATIONS

	FIE_0	FIE_I	FIE_II	FIE_III
Source Languages	R, Bash	R, Bash	R, Bash	R, Bash
Source Files	19	20	24	29
Data Files	2	8	14	14
Dependency Files	255	255	411	659
Size of All Files	133.2MB	178.4MB	289.7MB	306.6MB
Normal Run Time	52.046s	238.833s	295.785s	7200s

TABLE II
VARIABLE INFILTRATION CAPACITY SUB-TASK APPLICATIONS

	VIC_0	VIC_I	VIC_II	VIC_III
Source Languages	C, C++, Python, C shell, Fortran			
Source files	35	61	77	97
Data files	3689	6313	11460	11481
Dependency Files	247	260	314	357
Size of All Files	1.2GB	1.3GB	2.2GB	2.3GB
Normal Run Time	158.734s	306.069s	363.147s	377.29s

tasks. Additionally, the nature of FIE's processing tasks differ significantly from those of VIC. FIE front-loads its input data sets into memory, and then utilizes machine-learning logic to process its data. VIC also runs many intricate calculations, but differs from FIE in that it interlaces file input and output operations regularly throughout its code. This difference will be key in understanding that sciunits have minimal performance impact on most – but not all – types of applications.

Figure 8 compares the baseline normal execution time of each subtask[4] with the time consumed by packaging the sub-task with the sciunit *package* command, and with the time consumed by repeating the sub-task with the sciunit *repeat* command. We note that the performance impact of auditing and repeating on FIE's run times was negligible: auditing FIE with *package* resulted in only a 3.6% time increase, and executing FIE with *repeat* added only a 1.3% increase to run time. Conversely, both packaging and repeating VIC with sciunit each nearly doubled the original application run times: as noted in the preceding paragraph, it was evident that using sciunit with IO-intensive applications affected application performance significantly.

We obtain one further observation from these experiments by comparing each application *package* time with its corresponding *repeat* time. Compared to application repeat increases, auditing increases were slightly higher. This difference can be understood by examining sciunit's behavior during AV audit-time: auditing entails copying an application's code and data into a sciunit container, but running the sciunit container with *repeat*, however, only redirects to these copied files, and therefore precludes the file copy time.

C. Storing sciunits

Figure 9 presents the space saved by storing multiple sciunit containers in deduplicated storage. The total space consumed

[4]Test results for the FIE_III and VIC_III sub-tasks were omitted due to significant amounts of network-dependent downloading operations.

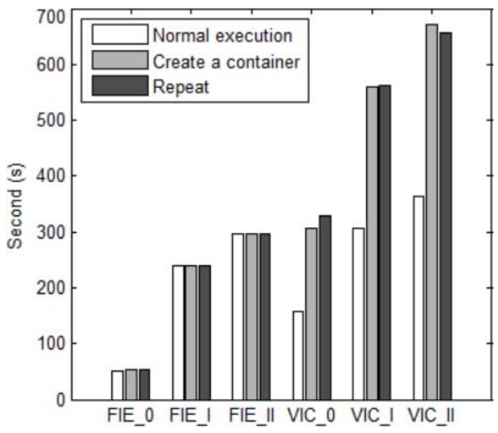


Fig. 8. Execution times for normal runs, creating containers, and repeating

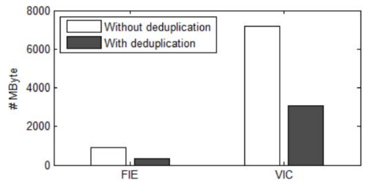


Fig. 9. Saved space with content deduplication

by FIE versions 0-III, storing each version separately, was 907MB, compared with a deduplicated total of 333MB. Similarly, VIC versions 0-III consumed 7GB in total separate storage, but when deduplicated consumed a total of 3GB.

We also measured the computational complexity of committing and reconstructing a version to and from deduplicated storage. Committing a package involves taking an input container, constructing a single-file archive from it, and then performing deduplication on the archive against stored blocks. Consequently, commit times are a function of the size of the container. The reconstruction process only requires extracting the relevant blocks from storage and creating a package. Even though reconstruction is merely a block-concatenation process, it also entails recreating the original file entries from the block, and therefore can have a measurable time overhead.

We measured both commit and reconstruction times that were far less than the normal baseline application execution times, and which would likely be imperceptible to users. Figure 10 shows the time in seconds for committing and reconstructing each sub-task[5]. Commit times were always greater than reconstruction times, due to the computation of rolling hashes during commit[6]. Reconstruction times were dominated by the process of un-archiving individual files.

[5]We depicted only packages of large size in order to clearly compare differences in commit and reconstruction times.

[6]The time for deduplication itself during commits was negligible, since it consists of single hash table lookups.

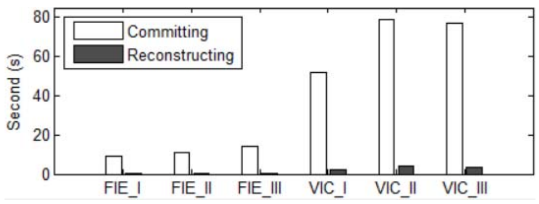


Fig. 10. Execution times for committing and reconstructing versions

D. Reusing sciunits with Provenance Visualizations

Application virtualization has traditionally led to fine-grained provenance graphs that are often difficult to decipher. In this sub-section we determine if our summarization rules produce a usable provenance graph that is closer to a theoretical, intuitive user application workflow. We focus this discussion on experiments for the FIE sub-tasks, but note that experiment results for the VIC sub-tasks were similar.

To evaluate the effectiveness of summarization, we first considered three traditional, replete (i.e. fine-grained) provenance graphs generated by the legacy PTU application on auditing FIE_I, FIE_II, FIE_III[7]. We calculated the number of nodes (each a process or a file) and edges present in each replete graph. Next we calculated the number of nodes present in the corresponding sciunit container provenance graphs (these latter, summarized graphs were produced using the similarity and packability rules). Figure 11 depicts a comparison of the two graphs. It should be noted that the same provenance log (produced by PTU on audit) was used in the generation of both sets of graphs.

Graph summarization reduced the number of file nodes, process nodes, and edges by averages of 90%, 46%, and 86%. On closer examination, since the annotation technique only applies to files and their associated edges, we observed a larger decrease in the number of files and edges than the decrease in the number of processes. Of crucial – but less measurable – importance, we noted that the much smaller number of nodes and edges of the summarized graphs also carried more meaningful, intuitive labels, similar to those in Figure 5.

We also measured the number of clicks needed to expand summarized graphs to replete graphs. For FIE_III, which had the largest graph, expanding any summarized node required a maximum of four user clicks to reach its replete view. Expanding all the nodes in this large graph took 45 clicks. This observation showed that graphs were summarized very well spatially and intuitively, yet still capable of allowing fully-detailed provenance examination with a modest amount of user interaction.

VIII. Evolution of Research Objects

In this section we trace the evolution of the concept of research objects and their use toward advancing scholarly communication. Research objects are increasingly seen as the new social object for advancing science [23]. They can be used for dissemination of scholarly work, measuring research impact, and assessing credit and attribution [24], which is the past was mostly done through research papers. The Research Object Model [4], [25] is a comprehensive standard defining the concept of a research object as a bundle of artifacts that provides a complete digital record of a piece of research. Implementations of the standard have primarily focused on structured workflow objects [26] [27] [28], and have not yet encompassed general applications (i.e. applications executed without a formal workflow system). In this paper, we describe the sciunit client, a tool for creating a research object that includes containers created during run-time execution of an application, within a workflow system [14], [15], [16] and outside [29].

To create a research object, digital artifacts must be placed within it, either manually with explicit commands such as those used in RO-Manager [30] (a tool that uses the RO-Bundle specification [5]), or automatically by using an AV tool such as Code, Data, and Environment (CDE) [6] [31] that containerizes an application as it executes. In this paper, we have chosen the AV tool Provenance-To-Use (PTU) [7] [32], which is built on top of CDE, to automatically capture provenance while creating containers, and have extended it for versioning and summarizing its provenance. We exclude more recent methods (such as [33]) that require users to learn new languages, and instead focus on the integration of DevOps tools in research objects.

Recorded provenance can be made more conducive to new analyses by summarizing it using statistical [34] and non-statistical [35] [36] methods. Our sciunit client uses non-statistical methods to summarize a research object's provenance, and extends the methods to visualize the summarized provenance spatially.

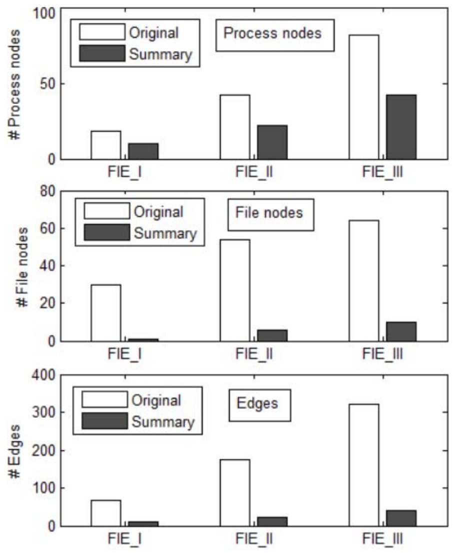


Fig. 11. Number of nodes and edges in original and summarized graphs

[7]We did not consider FIE_0 in this analysis since its original replete graph was too small and simple to benefit measurably from summarization.

Other methods to build and reuse containers, such as Topology and Orchestration Specification for Cloud Applications [37], still rely on user action to create the topology, relationship, and node specifications that are eventually translated to Dockerfiles [38]. In our case, Docker is merely a wrapper for standardization, since application virtualization creates a self-contained container, and the translation to Dockerfiles from the collected provenance is fairly straightforward.

IX. Conclusion

Computational reproducibility [39] is a formidable goal requiring advancements in policy [40], user perception [41], and reproducible practices and tools [1]. As we embrace this goal within the geosciences [42], we have encountered that general tools advocated for computational reproducibility must be enhanced in various ways. In this paper, we have challenged simple aggregation and advocated for containers, storing multiple of them with a relatively low storage cost, in logical sciunits, and their reuse in exact, partial, or modifiable forms using intuitive description of the reference execution. We demonstrated an easy-to-use Git-like client, the Sciunit-CLI that enables reproducibility for a wide variety of use cases. Yet, there are emerging requirements to address reproducibility within Jupyter notebooks, Matlab, distributed data-intensive programs, parallel HPC applications, which we hope to address as part of future work.

Acknowledgment

The authors would like to thank Tom Schenk and Gene Leynes for the FIE use case and Jonathan Goodall and Bakinam Essawy for the VIC use case. The authors would also like to acknowledge support for this work from the National Science Foundation under grants NSF ICER-1639759, ICER-1661918, ICER-1440327, and ICER-1343816. Sciunit-CLI is downloadable from [9].

References

[1] V. Stodden, F. Leisch, and R. D. Peng, Eds., *Implementing Reproducible Research*. CRC Press, 2014.
[2] T. Malik, Q. Pham, and I. T. Foster, “SOLE: Towards Descriptive and Interactive Publications,” *Implementing Reproducible Research*, vol. 33, 2014. [Online]. Available: \url{https://osf.io/w6fp4/files/}
[3] Figshare.com, “Figshare,” https://figshare.com/, 2017, [Online; accessed 2-May-2017].
[4] K. Belhajjame, J. Zhao, D. Garijo *et al.*, “Using a suite of ontologies for preserving workflow-centric research objects,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 32, pp. 16–42, 2015.
[5] S. Soiland-Reyes, M. Gamble, and R. Haines, “Research object bundle 1.0,” https://researchobject.github.io/specifications/bundle/, 2014, [Online; accessed 2-May-2017].
[6] P. J. Guo and D. Engler, “CDE: Using system call interposition to automatically create portable software packages,” in *USENIX*, 2011.
[7] Q. Pham, T. Malik, and I. Foster, “Using Provenance for Repeatability,” in *TaPP*, 2013.
[8] A. Muthitacharoen, B. Chen, and D. Mazières, “A low-bandwidth network file system,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.
[9] “Sciunit-CLI,” https://bitbucket.org/geotrust/sciunit-cli.git, 2017, [Online; accessed 10-Sep-2017].
[10] City of Chicago, “Food Inspection Evaluation,” https://chicago.github.io/food-inspections-evaluation/, 2017, [Online; accessed 05-2017].
[11] ——, “Chicago data portal,” https://data.cityofchicago.org/, 2017, [Online; accessed 7-May-2017].
[12] ——, “Food Inspection Evaluation predictions-source code,” https://github.com/Chicago/food-inspections-evaluation, 2016, [Online; accessed 7-May-2017].
[13] ——, “Food Inspection Evaluation,” https://chicago.github.io/food-inspections-evaluation/predictions/, 2017, [Accessed 05-2017].
[14] J. Goecks, A. Nekrutenko, and J. Taylor, “Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences,” *Genome Biology*, vol. 11, 2010.
[15] Y. Zhao, M. Hategan *et al.*, “Swift: Fast, reliable, loosely coupled parallel computation,” in *IEEE Congress on Services*, 2007.
[16] I. Altintas, O. Barney, and E. Jaeger-Frank, “Provenance collection support in the kepler scientific workflow system,” in *IPAW*, 2006.
[17] Q. Pham, T. Malik, B. Glavic, and I. Foster, “LDV: Light-weight database virtualization,” in *ICDE*, 2015.
[18] Q. Pham, “A framework for reproducible computational research,” Ph.D. dissertation, Dept. of Computer Science, University of Chicago, 2014.
[19] Hydroshare.com, “Hydroshare,” https://www.hydroshare.org/, 2017, [Online; accessed 2-May-2017].
[20] M. O. Rabin *et al.*, *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Lab, Univ., 1981.
[21] M. M. Billah, J. L. Goodall *et al.*, “Using a data grid to automate data preparation pipelines required for regional-scale hydrologic modeling,” *Environmental Modelling & Software*, vol. 78, 2016.
[22] Tensorflow.org, “Tensorflow,” https://www.tensorflow.org/, 2017, [Online; accessed 2-May-2017].
[23] D. De Roure, “Towards Computational Research Objects,” in *ACM Workshop on Digital Preservation of Research Methods and Artefacts*, 2013.
[24] Yale Roundtable, “Reproducible Research,” vol. 12, pp. 8–13, 2010.
[25] S. Bechhofer, I. Buchan, D. De Roure, P. Missier *et al.*, “Why linked data is not enough for scientists,” *Future Generation Computer Systems*, vol. 29, 2013.
[26] O. Corcho, D. Garijo Verdejo, K. Belhajjame *et al.*, “Workflow-centric research objects: First class citizens in scholarly discourse.” 2012.
[27] D. De Roure, K. Belhajjame, P. Missier, J. Gmez-Prez *et al.*, “Towards the preservation of scientific workflows,” in *8th International Conference on Preservation of Digital Objects (iPRES)*, 2011.
[28] I. Santana-Perez and M. S. Pérez-Hernández, “Towards reproducibility in scientific workflows: An infrastructure-based approach,” *Scientific Programming*, 2015.
[29] Q. Pham, T. Malik, I. Foster *et al.*, “SOLE: Linking research papers with science objects,” in *IPAW*, 2012.
[30] https://github.com/wf4ever/ro-manager, 2016, [Accessed 02-May-2017].
[31] P. J. Guo, “CDE: Run any Linux application on-demand without installation,” in *LISA*, 2011.
[32] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, “An invariant framework for conducting reproducible computational science,” *Journal of Computational Science*, vol. 9, 2015.
[33] P. Ivie and D. Thain, “Prune: A preserving run environment for reproducible scientific computing,” in *IEEE e-Science*, 2016.
[34] P. Macko, D. Margo, and M. Seltzer, “Local clustering in provenance graphs,” in *ACM CIKM*, 2013.
[35] Y. Tian, R. A. Hankins, and J. M. Patel, “Efficient aggregation for graph summarization,” in *ACM SIGMOD*, 2008.
[36] S. Cohen, S. Cohen-Boulakia, and S. Davidson, “Towards a model of provenance and user views in scientific workflows,” in *Data Integration in the Life Sciences*, 2006.
[37] Standard OASIS, “Topology and orchestration specification for cloud applications version 1.0,” 2013.
[38] R. Qasha, J. Cała, and P. Watson, “A framework for scientific workflow reproducibility in the cloud,” in *IEEE e-Science*, 2016.
[39] J. Freire, P. Bonnet, and D. Shasha, “Computational reproducibility: State-of-the-art, challenges, and database research opportunities,” in *ACM SIGMOD*, 2012.
[40] V. Stodden, P. Guo, and Z. Ma, “Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals,” in *PloS one*, vol. 8, 2013.
[41] D. Penny, “Nature Reproducibility survey,” May 2016. [Online]. Available: https://figshare.com/articles/Nature_Reproducibility_survey/3394951
[42] T. Malik, “GeotrustHub,” https://geotrusthub.org/, 2017, [Online; accessed 10-Sep-2017].