Combating Workflow Failures with Integrity-based Checkpoints and Blockchain

1st Omkar Bhide, *Indiana University*, Bloomington, IN, USA, okbhide@iu.edu
2nd Raquel Hill, *Indiana University*, Bloomington, IN, USA, ralhill@indiana.edu
3rd Karan Vahi, *USC ISI*, CA, USA, vahi@isi.edu
3rd Mats Rynge, *USC ISI*, CA, USA, rynge@isi.edu
5th Von Welch, *Indiana University*, Bloomington, IN, USA, vwelch@iu.edu

***Abstract*—Workflow management systems are subject to failures, including: processor, network congestion, and machine reboot. Various fault tolerance techniques have been proposed to address these failures. Data integrity errors also cause workflows to fail, but little or no attention has been given to integrity faults. The Scientific Workflow Integrity with Pegasas (SWIP) project has shown data integrity errors do occur in the wild. These errors occur when transferring and storing experiment data. The inability of today's validation mechanisms such as TCP checksums and Layer 2 checksums, motivated the SWIP project to add an extra layer of application layer data integrity verification using cryptographic hashes. Currently, the SWIP project takes a checkpoint all approach for integrity data; moving all integrity data for a task to stable storage. In this paper, we characterize nodes in workflow graphs based on the graph structure and propose several integrity-based checkpointing strategies. These strategies use a node's properties to determine which nodes to checkpoint. When failures occur the proposed integrity-based checkpointing strategies allow us to validate the integrity of data of preceding workflow tasks and re-use data during workflow retries. This paper focuses on characterizing workflow graphs to identify key nodes, thereby reducing the overheads of workflow retries. We also explore and evaluate the blockchain technology to securely preserve the integrity meta-data.**

***Index Terms*—eScience, workflows, data integrity, checkpointing, error recovery, blockchain**

I. Introduction

eScience involves the application of computer technology to undertake modern scientific investigation, including the preparation, experimentation, data collection, results dissemination, and long-term storage and accessibility of all materials generated through the scientific process. Workflow Management Systems (WMS) help researchers to seamlessly use distributed resources, like XSEDE [1] and Open Science Grid [2], without worrying about their integration. WMS also enable researchers to automate workflow execution, monitor workflows, and maintain provenance information. As with any computing infrastructure, the distributed environments that support WMS are subject to failures, including: processor failure, numerical exceptions, machine reboot, network congestion, etc [3]. In addition, while executing workflows across distributed resources, it is difficult to ensure data integrity, i.e. data was not corrupted in transit or at rest. TCP checksums have been shown to be too small for modern data sizes [4]. Studies [5] [6] show that between 1 in 16 million and 1 in 10 billion TCP segments will have corrupt data and a correct TCP checksum. The checksum cannot detect certain errors, like reordering of 2 bytes, inserting or deleting zero valued bytes, and multiple errors which sum to zero. Though the chances of such corruptions that are not detected by TCP checksums are approximately 0.001 percent, on a Gigabit Ethernet network that could be as many as 15 packets per second. Moreover there exist gaps between integrity protections in individual implementations/technologies within an application (e.g. between a data transfer and data at rest). Data integrity is crucial for scientific experiments, since use of corrupt data can lead to erroneous results. Large scientific experiments are usually difficult to reproduce and verify. Data integrity assurance for scientific experiments thus plays a crucial role in the workflow execution.

Various approaches have been proposed to address fault tolerance issues in distributed systems [7], [8]. Hwang et al. [3] categorize these approaches as either workflow-level or task-level approaches. Workflow-level approaches specify failure recovery procedures as part of an application's structure, and include the use of alternative tasks to handle failures. Task-level use several techniques, including: retrying, checkpointing, and replication to address task failures. Current checkpoint strategies do not consider validating the integrity of the data that is checkpointed. Their notion of check-pointing moves output data of all or some completed tasks to stable storage [9], thus allowing failure recovery or retries to begin by reading the output of prior tasks that the failed task depends on. These checkpoint strategies do not consider integrity errors that may occur in stable storage or during the transmission of data between storage devices and compute nodes.

In this paper, we studied workflow failures in the Pegasus WMS [10]. The Pegasus project encompasses a set of technologies that help workflow-based applications execute in a number of different environments including desktops, campus clusters, grids, and cloud. Pegasus bridges the scientific domain and the execution environment by automatically mapping high-level workflow descriptions onto distributed resources. Pegasus has a number of features that contribute to its usability and effectiveness, like portability, performance, scalability, provenance, data management, reliability and error recovery. Error handling in Pegasus tries to recover by retrying tasks, retrying the entire workflow, providing workflow-level check-pointing, re-mapping portions of the workflow, trying alternative data sources for staging data, and, when all else fails, providing a rescue workflow containing a description of only the work that remains to be done. In this paper, we take a check-pointing and retry approach to address failures. We assume that all workflow task output is check-pointed; stored in stable storage. We focus specifically on check-pointing task-level integrity data to facilitate task retries. Such integrity data includes hashes or checksums of a tasks input and output files. We analyze the characteristics of the Workflow task graph to determine how best to reduce the overhead of our integrity-based check-pointing scheme. Integrity meta-data like, checksums and hashes, are subject to similar data corruption risk while in transfer or stored. To avoid this chicken and egg problem, we propose the use of blockchain technology to securely preserve the integrity meta-data.

Rest of the paper is organized as follows. Section II underlines the related work. Section III provides the taxonomy of workflow nodes with respect to workflow retries, along with structural analysis of five different real world workflows. Section IV, describes the methodology used to conduct the experiment. Section V, describes the basics of Blockchain technology and our implementation of a blockchain smart contract. Section VI, provides the results and analysis of our experiment, followed by conclusions and future work in Section VII.

II. Related Work

Checkpointing has been studied a great deal in distributed, parallel systems as an efficient fault tolerance technique, especially for long-running applications, like scientific workflows. Checkpointing approach reduce the workflow retry overheads. Hwang et al. [3] provide a failure handling framework for workflows executing within a grid environment. They use checkpointing as well as other failure recovery techniques, and measure the expected execution time of workflow tasks under failure conditions. Li Han et al. [11] propose a checkpointsome strategy for Minimal Series-Parallel Graphs (M-SPGs), which are relevant to many real-world workflow applications, including the Pegasus WMS. They propose a task scheduling algorithm that leverages the M-SPG structure to assign sub-graphs to individual processors, and they use dynamic programming to determine how to checkpoint the subgraphs. Their approach prevents cross over dependencies, which occur where Task T1 fails, but it is not checkpointed, and T2 depends on T1. Our work complements these failure recovery schemes by proposing strategies for creating, checkpointing, and maintaining integrity data for workflow tasks.

Albert et al. [12], discuss error tolerance of complex networks within the context of two network graph structures: exponential and scale-free. The exponential network is homogeneous, where most nodes have approximately the same number of links. The scale-free network is inhomogeneous, which means that the majority of the nodes have one or two links, and only a few nodes have a large number of links. They show that these scale-free graphs are robust against node failures. We present workflow graphs that have this same inhomogeneous property, but inhomogeneity does not translate to robustness for these workflow graphs. More specifically, workflow nodes that have a high in-degree depend on the successful completion of the tasks that precede them. If any preceding tasks fail, these nodes with high in-degree may be delayed indefinitely.

Though prior checkpointing techniques [3], [9] reduce workflow retry overheads, they don't necessarily consider integrity validation of preserved checkpoint data. The checkpoint data is subject to data corruption in storage and during transfer, and therefore need integrity protections. Moreover, the integrity meta-data itself should be preserved securely. Edoardo Gaetani et al. [13] present a blockchain based database for cloud environments which assures data integrity. They discuss the research challenges and benefits of using the latest blockchain technology. The authors mention that using the blockchain to face data integrity threats seems to be a natural choice, but its current limitations of low throughput, high latency, and weak stability hinder the practical feasibility of any blockchain-based solutions. Azaria et al. [14] propose a framework for using blockchain to store medical records, and discuss the guarantees for data integrity privacy. Liang et al. [15] harness blockchain 's data integrity and immutable properties, and propose to use of blockchain for preserving provenance information. Similar to prior work, we propose to use blockchain technology to store our integrity meta-data because it provides essential security properties that support the validation of such data.

III. Taxonomy of Workflow Nodes

A scientific workflow, is a high-level specification of a set of tasks and the dependencies between them that must be satisfied in order to accomplish a specific goal. Users can specify the steps and dependencies within a workflow, either abstractly or concretely. This can be done in various ways like, DAGMan [16], DAX. The abstract workflow can be represented using Directed Acyclic Graph in Extensible Markup Language (XML) format i.e. DAX, which researchers can generate using any type of scripting language. This abstract workflow is then transformed into a concrete workflow representation for underlying execution by the workflow management systems. The resulting graph is most commonly represented as a Directed Acyclic Graph (DAG).

A graph is a collection of vertices and edges, where the vertices are connected in pairs by edges. In the case of a directed graph, each edge has an orientation, from one vertex to another vertex. A path in a directed graph can be described by a sequence of edges having the property that the ending vertex of each edge in the sequence is the same as the starting vertex of the next edge in the sequence. A path in a directed graph essentially forms a cycle if the starting vertex of its first edge equals the ending vertex of its last edge. A directed acyclic graph is a directed graph that has no cycles, and has orientation for each edge. Fig. 1 depicts an example DAG graph.

Our workflow graphs may be characterized as inhomogeneous, which means that the majority of the nodes have one or two edges, and only a few nodes have a large number of edges [12]. More specifically, only a few nodes within the workflow graph have a large in-degree. While Albert et al. [12] show that this inhomogeneous property creates network communications graphs that have redundant paths and are resilient to failure, this same property creates failure vulnerabilities in our directed workflow graphs. Therefore, in this section, we define a taxonomy of nodes that characterizes the failure vulnerabilities in our workflow graph.

Not all workflow graphs are identical, similarly some nodes in a graph are structurally more crucial than others. We analyze the characteristics of some real world workflow task graphs, and create a taxonomy of nodes. We use this taxonomy to set priorities and to develop strategies for checkpointing integrity data. A detailed description of these strategies is provided in Section IV. The objective of the proposed strategies is to reduce the overall overhead of workflow failure recovery by maintaining integrity data for key nodes within the workflow graph.

Fig. 1 depicts a sample workflow DAG or dependency graph. Using this example DAG, we explain the taxonomy of nodes with respect to integrity based workflow retries. This taxonomy will be later used to evaluate actual DAGs using different strategies, discussed in next section.

- Impactful Nodes:
  A node is impactful in terms of workflow retries if its out-degree is greater than one, which suggests that its output files are being used by two or more children. Securely preserving integrity meta-data is crucial for an impactful node. For example, in Fig 1. A is an impactful node, if any of its children (B,C,D) fail they can securely re-use the stored output data by validating it using the preserved integrity meta-data. This demands that A's fine-grained integrity data be preserved, enabling its children to validate their subset of A's output, for e.g. B uses subset of files (f3, f4) from A's output files (f1, f2, f3, f4, f5, f6).

- Vulnerable Nodes:
  A node is vulnerable to workflow failures if its in-degree is greater than one, which suggests that its depends on multiple parents for their output. Since a vulnerable node takes data from multiple parents, it can be argued that it processes large amounts of data, thus requiring considerable computing resources, including large data transfers. Given the integrity failures that are associated with transferring large data files, at a minimum, coarse-grained integrity meta-data should be preserved for vulnerable nodes. For example, node E is a vulnerable node, and requires preserving integrity meta-data for all of its output files.

- Intermediate Nodes:
  Intermediate nodes are the predecessors of a vulnerable node. Failure of any of these intermediate nodes delays or prevents the execution of its vulnerable child. For example, nodes B,C,D are intermediate nodes and node E is a vulnerable node. Failure of any of the intermediate nodes affect the vulnerable child node. To enable workflow retries, we recommend securely preserving integrity meta-data of each intermediate node.

- Normal Nodes: A normal node is any node which is not impactful, vulnerable or intermediate. These nodes are mostly serialized with in-degree and out-degree of one.

Having defined the taxonomy of nodes, its important to note that workflow nodes can have overlapping classifications. In other words, a node can be impactful, vulnerable and/or intermediate at the same time. In such cases we define a precedence, where impactful nodes have precedence over all other nodes and intermediate nodes have precedence over vulnerable nodes and normal nodes. This precedence is used in the strategies described in Section IV.

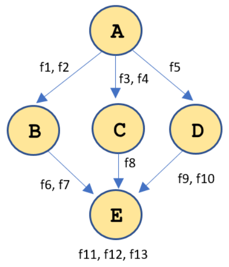


Fig. 1: Example workflow dependency Graph with output files for each node

Pseudo-code 0: Pre-processing

```
$DG \leftarrow$ Dependency graph of the input DAG
for i in range (0, n) do
    if DG[i].out-degree $> 1$ then
        DG[i].label('impactful')
    else if DG[i].in-degree $> 1$ then
        DG[i].label('vulnerable')
        pred $\leftarrow$ DG[i].predecessors()
        for j in pred do
            DG[j].label('intermediate')
    else
        DG[i].label('normal')
```

We apply this taxonomy to some real world synthesized workflows running on Pegasus, namely Montage [17], LIGO [18], CyberShake [19], Sipht [20], Epigenomics [21]. We analyze the DAGs of these scientific experiments, and evaluate the structure of the graph with respect to node failures and workflow retries. We use the Pseudo-code 0 depicted below, to pre-process the DAG file and label the nodes based on the taxonomy described earlier. The pre-processing step takes a workflow dependency graph as input, which is parsed from the actual workflow DAG file. It then traverses all the nodes, and for each node, it checks if the node is impactful, vulnerable and/or intermediate. If a node has out-degree > 1 then it's marked impactful, if it has in-degree > 1 then it's marked as vulnerable, and all the predecessors of a vulnerable node are marked intermediate. Fig. 2 provides a color legend for the workflow nodes. Subsequent text discusses the structure of each workflow DAG, along with the description of each application.

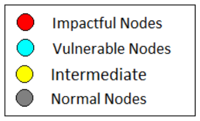


Fig. 2: Legend for workflow node colors

A. *Montage Workflow*

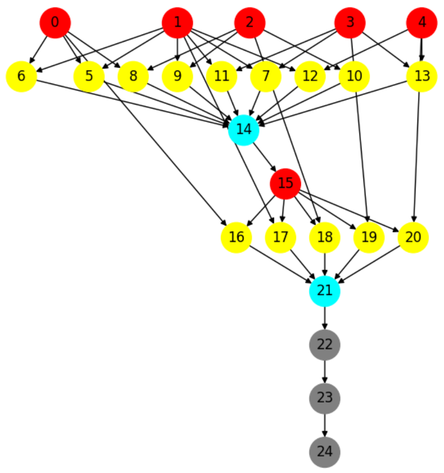


Fig. 3: Montage Workflow (25 Nodes)

The Montage application is created by NASA/IPAC which stitches together multiple input images to create custom mosaics of the sky. Fig. 3 shows the synthesized workflow dependency graph, which resembles the actual workflow graph used for the Montage scientific experiment. Referring to the legend in Fig. 3, we see that the nodes in the workflow have been identified as being impactful, intermediate, vulnerable or normal nodes. The workflow starts with nodes (0-4), which feed in to nodes 6-13, as well as nodes 16-20. Node 14 collects all the output data from previous nodes and passes it node 15. Nodes 0-4 and 15 are marked impactful since they have out-degree greater than one. Nodes (6-13) are marked intermediate since they are predecessors of a vulnerable node 14. Similarly, nodes (16-20) are marked intermediate since they are predecessors of a vulnerable node 21. Nodes 22, 23 and 24 are linearly executed and marked as normal nodes.

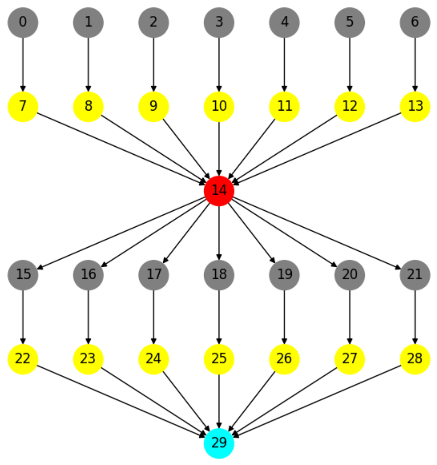


Fig. 4: Inspiral Workflow (30 Nodes)

B. *LIGO Inspiral Workflow*

LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational wave-forms from data collected during the coalescing of compact binary systems. Depicted in Fig. 4, the workflow starts with six different nodes (0-6) which feed data to nodes (7-13) respectively. Node 14 then collects outputs of nodes (7-13). Since node 14 has in-degree of greater than 1, it is vulnerable, which makes it's predecessors (nodes 7-13) intermediate. However, node 14 is also impactful, since it's out-degree is greater than 1. Node 14 feeds data to nodes (15-21), which further flow to nodes (22-28) respectively. Node 29 collects all the data from nodes (22-28), thus making it vulnerable and nodes (22-28) intermediate. You might notice that node 14 is impactful as well as vulnerable, but since impactful nodes take precedence over vulnerable nodes, it is marked red. Nodes (0-6) and (15-21) are marked normal nodes.

C. *CyberShake Workflow*

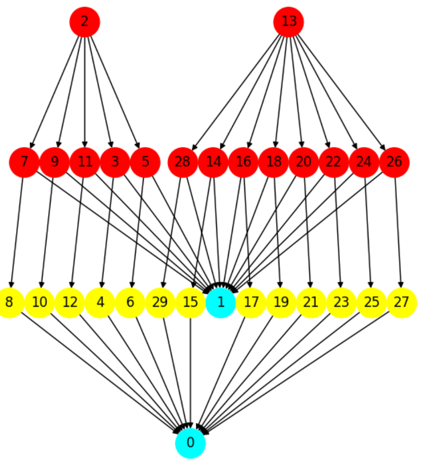


Fig. 5: CyberShake Workflow (30 Nodes)

The CyberShake workflow is used by the Southern California Earthquake Center to characterize earthquake hazards in a region. Depicted in Fig. 5, this workflow graph begins with two nodes 2 and 13, which are impactful with out-degree greater than 1. Node 2 feeds into nodes (3,5,7,9,11), lets call this set A. Node 13 feeds into nodes (14,16,18,20,22,24,26,28), lets call this set B. Node 1, collects subset of data from each of nodes in set A and B. With in-degree greater than 1, node 1 is treated as vulnerable node. Each node from set A and B also feed data to a node in (4,6,8,10,12,15,17,19,21,23,25,27,29), lets call it set C. Since each node in set A and B has an out-degree greater than 1, they are all treated as impactful. Node 0, collects data from all nodes in set C, making it a vulnerable node. Note that there are no normal nodes in this workflow.

D. Epigenomics Workflow

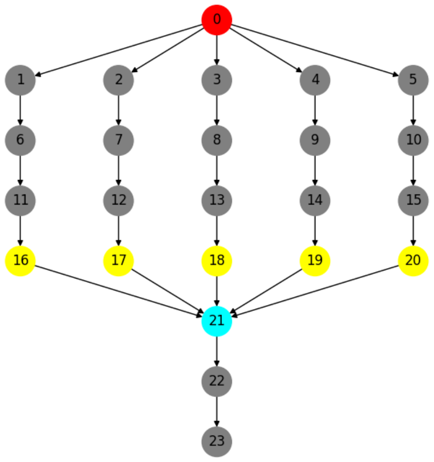


Fig. 6: Epigenomics (24 Nodes)

The epigenomics workflow created by the USC Epigenome Center and the Pegasus Team is used to automate various operations in genome sequence processing. This is yet another unique graph, depicted in Fig. 6, which begins with node 0, which feeds data to node (1-5), making it a impactful node. Each of the nodes in (1-5) continue serial execution marked as normal nodes e.g. 1-6-11-16. Node 21 collects data from nodes (16-20), making it a vulnerable node, treating nodes (16-20) as intermediate. Nodes 22 and 23 further execute serially.

E. Sipht Workflow

The Sipht workflow, from the bio-informatics project at Harvard, is used to automate the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database. The workflow graph for this experiment, depicted in Fig. 7, starts with nodes (18-21), which feed data to node 22. Since node 22 has in-degree greater than 1, it is a vulnerable node. This implies node (18-21) are intermediate. However, node 22 feeds data to multiple nodes, thus making it impactful as well. Node 26, depends upon nodes 22 and 23, thus making it vulnerable and node 23 an intermediate node. Similarly, node

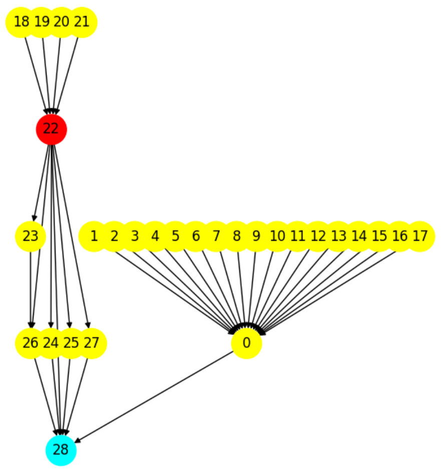
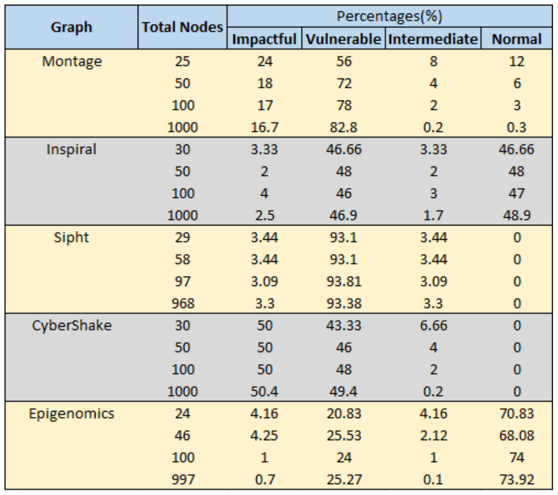


Fig. 7: Sipht Workflow (29 Nodes)

28 depends upon nodes (24-27) and nodes (0-17) making it a vulnerable node, and nodes (24-27) intermediate nodes. Notice that node 26 is vulnerable as well as intermediate, but is marked yellow, since intermediate nodes take precedence over vulnerable nodes. Similarly, node 0 is vulnerable as well as intermediate node.

F. Structural Analysis Overview



Graph	Total Nodes	Percentages(%)			
		Impactful	Vulnerable	Intermediate	Normal
Montage	25	24	56	8	12
	50	18	72	4	6
	100	17	78	2	3
	1000	16.7	82.8	0.2	0.3
Inspiral	30	3.33	46.66	3.33	46.66
	50	2	48	2	48
	100	4	46	3	47
	1000	2.5	46.9	1.7	48.9
Sipht	29	3.44	93.1	3.44	0
	58	3.44	93.1	3.44	0
	97	3.09	93.81	3.09	0
	968	3.3	93.38	3.3	0
CyberShake	30	50	43.33	6.66	0
	50	50	46	4	0
	100	50	48	2	0
	1000	50.4	49.4	0.2	0
Epigenomics	24	4.16	20.83	4.16	70.83
	46	4.25	25.53	2.12	68.08
	100	1	24	1	74
	997	0.7	25.27	0.1	73.92

Fig. 8: Graph Characterizations

Along with above mentioned graphs, we performed structural analysis of larger versions of these workflow graphs, which includes an increase in the number of nodes. Fig. 8 depicts a table, which provides the percentage of different nodes in each of the analyzed graphs. We can see that, with the increase in number of nodes, the node taxonomy percentages more or less remain the same. For example, in all four versions of Montage workflow, we observe that a large percentage of the graph nodes are vulnerable, followed by impactful nodes, followed by normal nodes, with least number of intermediate nodes. This suggests that, the workflow graph structure of the scientific experiments more or less, remains the same even with increase in number of nodes and data processed.

IV. Methodology for preserving integrity meta-data

A. Need to securely preserve integrity meta-data

We know that data gets corrupted at rest and during transfers [22] [23] [24]. These integrity failures motivated the community to introduce integrity validation checks using cryptographic hashes. However, this integrity meta-data is subject to same data corruption issues at rest and during transmission. Keeping in mind the use-case for error recovery during workflow retries, it's of paramount importance to assure integrity of the integrity meta-data itself. Traditionally, such meta-data is stored on the users/researcher's local storage or on cloud storage. There has been data corruption cases for data stored in cloud [25], which raises the question: what is the best way to securely preserve the integrity meta-data? We evaluate the use of blockchain technology as a possible solution for this problem, later in Section V.

B. What and how much to preserve?

We use the workflow graph structure characteristics to determine which node's integrity meta-data is to be preserved, and its granularity. We intend to reduce three different costs associated with workflow retries - blockchain storage costs, blockchain processing costs and validation costs. Storage cost is the cost in United States Dollar (USD) for securely preserving the integrity meta-data. Processing cost, is the time (in seconds) required to preserve the integrity meta-data. Validation cost, refers to the overhead of network data transfers required for validating a node's input data. In this section, we present four strategies for check-pointing integrity meta-data. The first two strategies are naive because they treat all nodes the same and do not leverage the node taxonomy to distinguish the treatment of nodes. Strategy one, stores granular integrity information for all the nodes, as depicted in pseudo-code 1. This strategy enables validating each and every output file in the workflow individually. However, this increases the storage and processing costs of preserving the integrity meta-data.

Pseudo-code 1: Strategy 1 - Workflow Execution

```
Workflow execution starts
for node n in DAG do
    input ← input data file(s)
    if Validate(input) then
        Execute the node tasks
        Preserve(PerHash(output))
    else
        Retry node n for  times
        if (If all retries fail) then
            Generate a rescue DAG
            Re-start workflow
```

Another basic strategy would be to preserve coarse grained integrity meta-data for all the nodes, as depicted in pseudo-code 2. This method reduces the storage and processing costs of preserving integrity meta-data since, only one hash per node would be preserved. However, it increases the validation costs drastically, since unnecessary files transfer would be required to validate the composite hashes.

Pseudo-code 2: Strategy 2 - Workflow Execution

```
Workflow execution starts
for node n in DAG do
    input ← all input data file(s)
    if Validate(input) then
        Execute the node tasks
        Preserve(CompositeHash(output))
    else
        Retry node n for  times
        if (If all retries fail) then
            Generate a rescue DAG
            Re-start workflow
```

Pseudo-code 3: Strategy 3 - Workflow Execution

```
Workflow execution starts
for node n in DAG do
    input ← input data file(s)
    if Validate(input) then
        Execute the node tasks
        if DG[n].label == impactful then
            Preserve(PerFileHash(output))
        else
            Preserve(CompositeHash(output))
    else
        Retry node n for  times
        if (If all retries fail) then
            Generate a rescue DAG
            Re-start workflow
```

We try to devise a non-trivial strategy to tackle the trade off between efficient error recovery and overhead costs. Pseudo-code 3, presents our third strategy. Once the workflow DAG is pre-processed using pseudo-code 0, the workflow starts execution within the Workflow Management System. Each node n follows the steps within the loop to preserve and validate integrity meta-data. The node collects its input data, either from parent nodes or from the user (if its the root node). It then checks if the input data has not been corrupted while being stored or transmitted. In case of the root node, the user provides the initial inputs along with it's integrity meta-data i.e. cryptographic hashes. For all other nodes, when a parent generates outputs, its integrity meta-data is securely preserved. The function, $Validate(input)$ will retrieve the appropriate integrity meta-data and validate the inputs. If it is successful, the node proceeds with its task execution, and generates its output files. The node's label determines "what integrity meta-data is preserved?". If the current node is labelled as a 'impactful' node during pre-processing, we generate separate hashes for each of the output files, using $PerFileHash(output)$. These individual hashes are then securely preserved. If the current node is anything other than a impactful node, i.e. if it's a non-impactful node, we generate a single composite hash of the output. This is achieved by hashing together the separate file hashes. This single composite hash of the output is then securely preserved. Let's take a simple example of a graph from Fig. 1 and walk-through the workflow execution process, using pseudo-code 3:

1) Node A from Fig. 1, is a impactful node, and thus all its five output files (f1, f2, f3, f4, f5) are hashed (hash(f1), hash(f2), hash(f3), hash(f4), hash(f5)) and preserved separately.
2) Node B takes its inputs (f1, f2) and performs integrity validation using the preserved hashes for (f1, f2). Lets assume the validation is successful, node executes its tasks, and generates output files (f6, f7). Since node B is non-impactful, it generates a composite hash of its output files, $hash(hash(f6), hash(f7))$.
3) Node C validates its inputs (f3, f4) and upon success, executes its tasks and generates output file (f8). Since node C is non-impactful it generates a single composite hash of its output and preserves it.
4) Node D takes its input file (f5), and runs integrity validation. Lets say, due to data corruption during file transfer or some unknown reasons the validation fails. In this case, the WMS retries the validation for several times. Upon failure of each retry, the workflow is aborted and a rescue DAG is generated which depicts the state of the workflow (Success: A, B, C; Failure: D; Pending: E).
5) When the workflow is re-started using the rescue DAG, it begins with node D. This time around, let's say the validation of (f5) is success-full and node D executes its tasks. It generates outputs (f9, f10) and generates a composite hash of the outputs being a non-impactful node.
6) Node E takes the inputs from B (f6, f7), C (f8) and D (f9, f10). It validates the inputs one-by-one, and upon success, executes its tasks and generates outputs (f11, 12, f13). Since its a non-impactful node, it generates a single composite hash of its output files, $(hash(hash(f11), hash(12), hash(13)))$ and securely preserves it.

Strategy 3, takes a conservative approach towards node failures, and thus preserves coarse-grained integrity meta-data at a minimum for all the nodes, which saves on the storage and processing costs of securely preserving integrity meta-data. Impactful nodes are treated differently by preserving fine-grained integrity meta-data, which saves on the validation costs.

Pseudo-code 4, illustrates our fourth strategy, which preserves a hash of each output file for impactful nodes. It preserves a single composite hash of output file hash(es) for vulnerable and intermediate nodes. It preserves no integrity meta-data for normal nodes, which implies that, if a normal node fails, workflow execution must re-start from the nearest impactful, vulnerable or intermediate ancestor. Thus, this strategy saves on the storage and processing costs by not preserving any integrity meta-data for the normal node output data. However it also increases the workflow failure recovery overhead, if any of the normal nodes fail.

Pseudo-code 4: Strategy 4 - Workflow Execution

```
Workflow execution starts
for node n in DAG do
    input ← input data file(s)
    if (Validate(input)) then
        Execute the node tasks
        if (DG[n].label == 'impactful') then
            Preserve(PerFileHash(output))
        else if (DG[n].label == 'vulnerable') then
            Preserve(CompositeHash(output))
        else if (DG[n].label == 'intermediate') then
            Preserve(CompositeHash(output))
        else
            Preserve none
    else
        Retry node n for  times
        if (If all retries fail) then
            Generate a rescue DAG
            Re-start workflow
```

C. Where to preserve?

As described earlier, the integrity meta-data itself can be subject to data corruption during transfer and at rest. This is important since, if integrity meta-data is corrupted, there is no way to determine whether the meta-data or actual data is corrupt. Without the needed integrity assurance, the failed tasks would require re-running. In the worst case, if both the data and meta-data are corrupted and validate successfully, the data corruption would go unnoticed, as discussed in the case of TCP checksums. Possible solution to this chicken-and-egg problem is to store the hashes in a decentralized environment which is immutable and easily accessible. Third party cloud storage providers like Google Drive and Amazon S3 are available, which can be used along with access permissions to store the hash meta-data, however this involves trusting the third party. Moreover, studies have shown that data on the cloud is also subject to data corruption [25]. Alternative solutions include, using the decentralized immutable ledger called Blockchain, which eliminates the use of trusted third party. Blockchain is the newest of technologies which provides a public ledger which is immutable and provides data integrity by design. Though blockchain smart contract storage is immutable, it can be openly accessed by the public. This also provides the general public, a well documented and easy way to access and verify scientific experiment data for re-use, verifying provenance and experiment reproducibility.

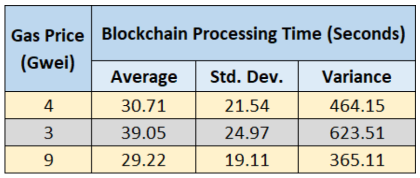
V. Blockchain Basics and Implementation

Blockchain is a publicly shared database, consisting of a ledger of transactions. The ledger keeps track of data and its ownership. This data can be of any type e.g. currency, inventory, intellectual property, media, records etc. Each machine running the blockchain protocol is called a node, and it maintains a copy of the ledger. Blockchains eliminate the problem of trust that affect other databases, since it provides full decentralization, extreme fault tolerance and independent verification. In an Ethereum network, blockchain transactions refer to the interactions between accounts in a blockchain network. Accounts identify entities such as human personas, mining nodes and software agents/smart contracts. Accounts use public key cryptography to sign and authenticate a transactions origin. There are two types of Ethereum accounts, externally owned accounts (EOA) and smart contract accounts which store programs and provides services.

These account entities have state; accounts have a balance, while contracts have both balance and storage. The state of all accounts is updated with each block, and provides the state of the Ethereum network. Smart contracts specify the code in a contract account. Contract accounts only perform an operation when instructed to do so by an EOA. Transactions are stored in blocks on the blockchain, and once a block is placed on the blockchain, it cannot be changed and is thus immutable. On a public blockchain, anyone can read or write data. Transaction history of these reads/writes becomes a part of the blockchain and thus readily available for verification. Reading data is free but writing to the public blockchain is not. This cost, known as gas and priced in ether, helps to discourage spam and pays to secure the network. Ether is measured in Gwei, where 1 Ether of gas is 1,000,000,000 Gwei. Smart contracts can accept and store Ether, data, or a combination of both. Using the logic programmed into the contract, it can distribute that ether to other accounts or even other smart contracts. These Smart contracts are written in a language called Solidity. Solidity is statically typed, and supports inheritance, libraries, and complex user-defined types, among much else. Solidity syntax is like JavaScript. In the next sub-section, we describe our initial approach of using Ethereum blockchain to securely preserve integrity meta-data.

A. *Pegasus Integrity Smart Contract*

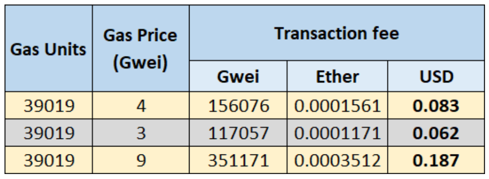
We implemented a Ethereum blockchain smart contract, to evaluate the storage and processing costs of using blockchain to securely preserve integrity meta-data. The overall objective of this experiment is to better understand the storage and processing costs of preserving workflow integrity meta-data on the blockchain. We opted to use the Ethereum test network called Ropsten [26] for our development and experimentation. It closely resembles the Ethereum main network, without the need to buy real cryptocurrency. We deployed a simple smart contract called Swipeth on the Ropsten test network with the address: 0x921cc22021e5caec11322348059abf2efa3233b0, which simply stores a key-value pair, each 32 bytes long. The key contains the Pegasus workflow id and the value represents a single cryptographic hash. This contract exposes two functions, Get and Set which are used to set the value and retrieve the values from the blockchain. We offer three increasing levels of financial incentives to the blockchain miners: Safe Low (low price - slow speed), Standard (average price - normal speed), and Fast (high price - fast speed). The Gas prices are considered based on the current Ethereum network status [27]. We evaluate 30 transactions to store integrity meta-data on the blockchain using three different gas prices, and provide average the timings in Fig. 9. As shown in the figure, using highest gas price of 9 Gwei, the average time to store a hash on the blockchain is 29.22 seconds for the ropsten test network.



Gas Price (Gwei)	Blockchain Processing Time (Seconds)		
	Average	Std. Dev.	Variance
4	30.71	21.54	464.15
3	39.05	24.97	623.51
9	29.22	19.11	365.11

Fig. 9: Blockchain - Processing Cost (one hash)

Fig. 10 shows the cost analysis of using blockchain to store integrity meta-data. Transaction fee is the Gas units spend per transaction multiplied by Gas price specified. The Gas prices are considered based on the current Ethereum network status [27]. As mentioned earlier, Gas price affects how fast the transaction is mined, and thus affects the processing cost overhead. Gas units estimates are provided by the blockchain API, based on the smart contract complexity. Pegasus smart contract requires 39019 gas units. Our experiment shows that using higher Gas price of 9 Gwei, the average transaction fee to store a composite hash on the blockchain is 0.187 USD, which is minuscule.

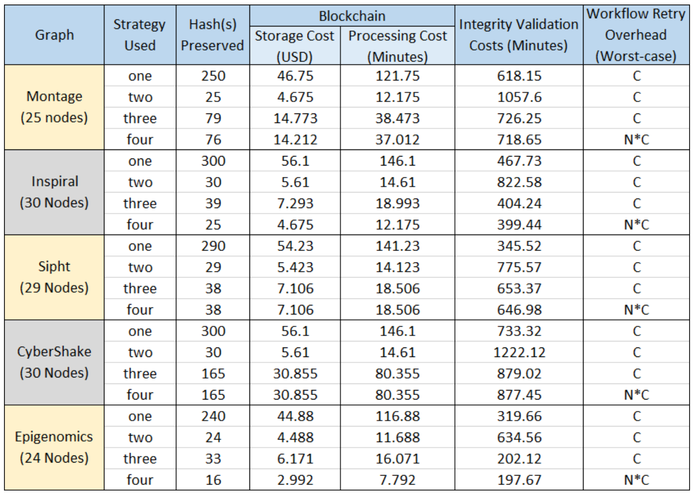


Gas Units	Gas Price (Gwei)	Transaction fee		
		Gwei	Ether	USD
39019	4	156076	0.0001561	**0.083**
39019	3	117057	0.0001171	**0.062**
39019	9	351171	0.0003512	**0.187**

Fig. 10: Blockchain - storage cost (one hash)

VI. Results and Analysis

Using the structural analysis of the five scientific workflows from Section III, and the storage and processing costs of preserving integrity meta-data on blockchain from Section IV, we compare the total overhead costs the four different strategies, with respect to the five example graphs. Fig. 11 provides the comparison of overhead costs. Below we also discuss the validation costs for each strategy, however quantifying the actual validation costs is part of our future work, which involves simulating the workflows, integrity validations and network transfers.



Graph	Strategy Used	Hash(s) Preserved	Blockchain		Integrity Validation Costs (Minutes)	Workflow Retry Overhead (Worst-case)
			Storage Cost (USD)	Processing Cost (Minutes)		
Montage (25 nodes)	one	250	46.75	121.75	618.15	C
	two	25	4.675	12.175	1057.6	C
	three	79	14.773	38.473	726.25	C
	four	76	14.212	37.012	718.65	N*C
Inspiral (30 Nodes)	one	300	56.1	146.1	467.73	C
	two	30	5.61	14.61	822.58	C
	three	39	7.293	18.993	404.24	C
	four	25	4.675	12.175	399.44	N*C
Sipht (29 Nodes)	one	290	54.23	141.23	345.52	C
	two	29	5.423	14.123	775.57	C
	three	38	7.106	18.506	653.37	C
	four	38	7.106	18.506	646.98	N*C
CyberShake (30 Nodes)	one	300	56.1	146.1	733.32	C
	two	30	5.61	14.61	1222.12	C
	three	165	30.855	80.355	879.02	C
	four	165	30.855	80.355	877.45	N*C
Epigenomics (24 Nodes)	one	240	44.88	116.88	319.66	C
	two	24	4.488	11.688	634.56	C
	three	33	6.171	16.071	202.12	C
	four	16	2.992	7.792	197.67	N*C

Fig. 11: Blockchain storage and processing overheads

Fig. 11 provides the storage and processing costs for preserving integrity meta-data on the blockchain, as well as validation costs for each of the different strategies. We assume that each node generates 10 output files. This is certainly not a realistic assumption, and the number of files generated would vary drastically in real scenarios, however this experiment provides valuable insights of how different strategies compare. Referring to Fig. 9, we consider paying 9 Gwei and 29.22 seconds to store a single hash to calculate the overhead. Also, referring to Fig. 10, we consider the cost of storing to be 0.187 USD based on gas price of 9 Gwei. Thus, we know that we can store a single hash on the blockchain with 0.187 USD and within 29.22 seconds. We use this data to compute total overheads for each strategy. Strategy used, determines the number of hashes we are preserving. Strategy one dictates that we preserve a hash for each file of all nodes. Strategy two dictates that we preserve one composite hash for each node. Strategy three dictates that we preserve a hash per file for impactful nodes, and a composite hash for each non-impactful node. Strategy four, dictates that we preserve a hash per file for impactful nodes, a composite has for each vulnerable or intermediate node, and preserving nothing for the normal nodes. The total number of nodes preserved for different strategies are then multiplied by 0.187 to get the total cost overhead, and multiplied by 29.22 to get the total processing overhead. Let us walk through the process for the montage workflow. The Montage workflow with 25 nodes, has 6 impactful nodes, 14 intermediate nodes, 2 vulnerable nodes and 3 normal nodes. Reiterating the assumption that each node generates 10 output files. Using strategy 1, we will preserve a hash for all files of all nodes i.e. 250 hashes (25 * 10). Using strategy 2, we preserve a composite hash for each node i.e. 25 hashes (25 * 1). Using strategy 3, we get 79 hashes [(6*10) + (19*1)]. Using strategy 4, we get 76 hashes [(6 * 10) + (14 * 1) + (2 * 1) + (3 * 0)].

Validation costs, as described earlier, refers to the overhead of network transfers required to validate the node's input data. Based on the strategy used, the validation cost is largely affected. For example, referring to Fig. 1, if node A preserves a composite hash of its output files, node B would need to obtain files (f3, f4, f5) in addition to (f1, f2) in order to validate the composite hash. On the contrary, if node A preserves a hash per output file, node B would just obtain the required (f1, f2) files and validate them individually. These unnecessary file transfers increase the overall workflow run-time, especially if there are large number of non-impactful nodes in the workflow. While calculating the validation costs, we assumed each file to be 1 Gb, and network bandwidth of 1 Gbps for file transfers. We use a random number generator function in Python called $randint(0, 10)$ [28] to randomly assign number of input files to each of the nodes in the workflow graph, in the range 1 to 10. Based on the assumptions and randomly assigned input files, we then calculate the time required for each node to collect the required input files and validate them. Fig. 11 depicts the total validation costs for different workflows using different strategies. In case of strategy four, children of normal nodes don't validate their input files.

In case of failures, the node has to be retried. We try to depict the workflow retry overhead for failures using two variables. We use the variable N to denote longest chain of normal nodes. For e.g. N is 3 for workflow in Fig. 6. We use the variable C to denote the average computation costs of the node's tasks. Workflow retry overhead column in Fig. 11 depicts the maximum retry overhead per failure for different strategies. For strategies one to three, the maximum retry overhead per failure is the cost to retry the failing node's tasks i.e. C. In case of strategy four, we assume worst case scenario where the longest chain of normal nodes fail i.e. N. Thus maximum workflow retry overhead would be the computation cost of each of the normal nodes in the chain ie. $N * C$.

Looking at the table, it is clear that strategy one has the highest storage and processing costs, and the lowest validation costs for all the five workflows. Though the figures appear small, note that these workflows are synthesized smaller version of the actual larger workflows with several thousand nodes and larger data-set. Strategy 1 won't be a scalable solution, as the blockchain storage and processing costs increase linearly with data files. On the contrary, Strategy 2 has the least storage and performance overhead costs of all the strategies, which makes it scalable in terms of storage and processing costs, however it drastically increases the validation costs. Strategies 3 and 4 try to balance the pros and cons of strategies 1 and 2. Strategy 3, considerably reduces the storage and processing costs compared to strategy 1 and reduces the validation costs compared to strategy 2. Moreover, since every node at the least preserves a composite hash, in case of failure, the workflow can be re-started from the failing node, by validating the data using the composite hash. Thus the workflow retry overhead is minimal i.e. C. Strategy 4, saves a bit more on the storage and processing costs, by taking a less conservative approach towards node failures, compared to strategy 3. It doesn't preserve any integrity meta-data for normal nodes. Validation costs are comparable to strategy

3. However, failure of any normal node, would require restarting the workflow from the nearest impactful, vulnerable or intermediate node. This affects the potential workflow retries overhead costs i.e. $N * C$. Thus, strategies 3 and 4 balance the trade-off between storage-processing costs and validation costs, however, strategy 4 introduces additional workflow retry costs compared to other strategies. These four experimental strategies provide a good basis to explore new parameters in taxonomy and new strategies. We note various research topics for future work in the next section, and conclude the paper.

VII. Conclusion and Future Work

This paper shows how the structure of the workflow graph affects error recovery, enabling reduction in overhead costs of securely preserving integrity meta-data for workflow retries. We evaluate five different scientific experiments and studied their graph structure. We also experiment the use of blockchain technology as a way to securely preserve integrity meta-data, by implementing a blockchain smart contract and evaluating the associated costs. Finally we provide analysis of how the graph structure can be used to reduce the overhead costs of preserving integrity meta-data and validation costs.

As part of our future work, we would like to generate probabilistic models for node failures and then quantify the overheads by simulating a workflow and its failures. Also, the existing taxonomy can be expanded to include other factors like, the type of tasks in the workflow node, computation time of tasks and periodic check-pointing. Another research direction is to study if the structure of these workflow DAGs is unique to the scientific application domain. In addition, it would be useful to evaluate whether it is possible to re-structure the workflow graphs to support efficient error recovery, without affecting the core functionality.

Acknowledgment

This project is supported by the National Science Foundation under Grant 1642070, 1642053, and 1642090. The views expressed do not necessarily reflect the views of the National Science Foundation or any other organization.

References

[1] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, *et al.*, "Xsede: accelerating scientific discovery," *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62–74, 2014.

[2] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, *et al.*, "The open science grid," in *Journal of Physics: Conference Series*, vol. 78, p. 012057, IOP Publishing, 2007.

[3] S. Hwang and C. Kesselman, "Gridworkflow: A flexible failure handling framework for the grid," in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, HPDC '03, p. 12, IEEE, 2003.

[4] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and crcs over real data," *IEEE/ACM Transactions on Networking*, vol. 6, pp. 529–543, Oct 1998.

[5] C. Partridge, J. Hughes, and J. Stone, "Performance of checksums and crcs over real data," in *ACM SIGCOMM Computer Communication Review*, vol. 25, pp. 68–76, ACM, 1995.

[6] J. Stone and C. Partridge, "When the crc and tcp checksum disagree," in *ACM SIGCOMM computer communication review*, vol. 30, pp. 309–319, ACM, 2000.

[7] F. C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," *ACM Comput. Surv.*, vol. 31, pp. 1–26, Mar. 1999.

[8] M. Elder, *Fault Tolerance in Critical Information Systems*. PhD thesis, University of Virginia, 2001.

[9] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," *IEEE Transactions on Computers*, vol. 67, pp. 1105–1120, 2017.

[10] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[11] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pp. 487–497, IEEE, 2017.

[12] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *nature*, vol. 406, no. 6794, p. 378, 2000.

[13] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Blockchain-based database to ensure data integrity in cloud computing environments," 2017.

[14] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "Medrec: Using blockchain for medical data access and permission management," in *Open and Big Data (OBD), International Conference on*, pp. 25–30, IEEE, 2016.

[15] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla, "Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 468–477, IEEE Press, 2017.

[16] "Dagman (directed acyclic graph manager)." http://www.cs.wisc.edu/condor/dagman/.

[17] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, and M.-H. Su, "Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand," in *Optimizing Scientific Return for Astronomy through Information Technologies*, vol. 5493, pp. 221–233, International Society for Optics and Photonics, 2004.

[18] A. Abramovici, W. E. Althouse, R. W. Drever, Y. Gürsel, S. Kawamura, F. J. Raab, D. Shoemaker, L. Sievers, R. E. Spero, K. S. Thorne, *et al.*, "Ligo: The laser interferometer gravitational-wave observatory," *Science*, vol. 256, no. 5055, pp. 325–333, 1992.

[19] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya, *et al.*, "Scec cybershake workflowsautomating probabilistic seismic hazard analysis calculations," in *Workflows for e-Science*, pp. 143–163, Springer, 2007.

[20] J. Livny, H. Teonadi, M. Livny, and M. K. Waldor, "High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas," *PloS one*, vol. 3, no. 9, p. e3197, 2008.

[21] H. Li, J. Ruan, and R. Durbin, "Mapping short dna sequencing reads and calling variants using mapping quality scores," *Genome research*, pp. gr–078212, 2008.

[22] B. Panzer-Steindel, "Data integrity," *CERN/IT*, 2007.

[23] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. Rao, and P. Zhou, "Evaluating the impact of undetected disk errors in raid systems.," in *DSN*, pp. 83–92, 2009.

[24] S. Liu, E.-S. Jung, R. Kettimuthu, X.-H. Sun, and M. Papka, "Towards optimizing large-scale data transfers with end-to-end integrity verification," in *Big Data (Big Data), 2016 IEEE International Conference on*, pp. 3002–3007, IEEE, 2016.

[25] C. Wang, Q. Wang, K. Ren, and W. Lou, "Ensuring data storage security in cloud computing," in *Quality of Service, 2009. IWQoS. 17th International Workshop on*, pp. 1–9, Ieee, 2009.

[26] "Ropsten: Ethereun test network." https://ropsten.etherscan.io/.

[27] "ethgas." https://ethgasstation.info/.

[28] "random.randint." https://docs.python.org/2/library/random.html.