Building a Cloud Service for Reproducible Simulation Management

Faical Yannick Palingwende Congo‡*

https://www.youtube.com/watch?v=euMLYw7SNdk

Abstract—The notion of capturing each execution of a script and workflow and its associated metadata is enormously appealing and should be at the heart of any attempt to make scientific simulations repeatable and reproducible.

Most of the work in the literature focus in the terminology and the approaches to acquire those metadata. Those are critical but not enough. Since one of the purposes of capturing an execution is to be able to recreate the same execution environment as in the original run, there is a great need to investigate ways to recreate a similar environment from those metadata and also to be able to make them accessible to the community for collaboration. The so popular social collaborative *pull request* mechanism in Github is a great example of how cloud infrastructures can bring another layer of public collaboration. We think reproducibility could benefit from a cloud social collaborative presence because capturing the metadata about a simulation is far from being the end game of making it reproducible, repeatable or of any use to another scientist that has difficulties to easily get them.

In this paper we define a reproducibility record atom and the cloud infrastructure to support it. We also provide a use case example with the event based simulation management tool *Sumatra* and the container system *Docker*.

Index Terms—metadata, simulations, repeatable, reproducible, Sumatra, cloud, Docker.

Introduction

Reproducibility in general is important because it is the cornerstone of scientific advancement. Either done manually or automatically; reusability, refutability and discovery are the key proprieties that make research results repeatable and reproducible.

One will find that in the literature many research have been done in defining the terminology (repeatability, reproducibility and replicability) [Slezak2011] and investigating approaches regarding the recording of simulations metadata using workflows [Oinn2006], libraries [Langer2014] or event control systems [Guo2012]). These research are critical because they focus on getting to the point where the metadata about a simulation execution have been captured in a qualitative and reliable way. Yet the use of these metadata to recreate the proper execution environment is challenging and is not only extremely valuable to the scientist that ran the simulation. It is more valuable to other scientists that share the same interest and could benefit an easy way to at least get the same results consistently. This is why we think that reproducibility can gain from a more active presence in the cloud through infrastructures that bring an easy access and collaboration around those captured metadata. The social collaborative *pull request* mechanism from Github [MacDonnell2012] is a great example about the importance of cloud infrastructures in enhancing collaboration. In fact many scientific projects from SciPy [Oliver2013] got some interest and contribution because of their exposure on Github and its ease for collaboration.

In this paper we discuss on a structure of a reproducible record atom. It is a record that we propose to ease the reconstruction of the execution environment and allow an easy assessment of its reproducibility by comparing it to others. Then we propose a cloud platform to deliver an online collaborative access around these record atoms. And finally we present an integration use case with the data driven simulation management tool Sumatra [Davidson2010].

A reproducible record atom

Defining what are the requirements that have to be recorded to better enforce the reproducibility of a simulation is of good interest in the community. From more general approaches like defining rules that have to be fulfilled [Sandve2013], to more specific approaches [Heroux2011], we can define a set of metadata that are useful to determine the reproducibility of a simulation. To do so, we have to go from the fact that the execution of a simulation involves mostly five different components: the source code or executable, the input files, the output files, the dependencies and the hosting system. The source code or executable gives all the information about what the simulation is, where it can be found (repository) and how it was run. The input files are all the files being loaded by the simulation during its execution. The output files are all the files that the simulation produced during its execution. The dependencies are all the libraries and tools that are needed by the simulation to run. The hosting system is the system in which the simulation is being ran. These components can be classified into two groups regarding repeatability as a goal. To repeat a simulation execution, the source code and the inputs are part of a group of components that are kept as the same. The dependencies and the host system on the other end are part of the components that will most likely change from the original executing system to another that is attempting a repeat. We think of them as a cause of uncertainties that lead to variations in the outputs when the source

* *Corresponding author: yannick.congo@gmail.com*
‡ *LIMOS - UMR CNRS 6158, Blaise Pascal Univerity, Campus Universitaire des Cezeaux, 2 Rue de la Chebarde, TSA 60125 - CS, 60026, 63178 Aubière CEDEX FRANCE*

code and inputs are still the same. To assess a reproducibility property on a simulation, we provide the Table 1. It defines the reproducibility properties involved (repeatable, reproducible, non-repeatable, non-reproducible or unknown) when comparing the source code, inputs and outputs of two simulations. This table is used in conjunction with the models presented later to assess the reproducibility property of any record atom in the system compared to another through a requesting mechanism that will be detailed further.

One thing is to be able to gather crucial information about a simulation yet another challenging one is to be able to recreate the same execution context as when the simulation was done the first time. It is impossible to consistently reproduce a simulation across platforms and machines if we do not have a uniform and portable way to bundle the whole simulation execution environment.

We think that container based systems [Bottomley2014] are a possible solution to ensure the consistency of the operating system and dependencies on which the simulation runs. Building and sharing a container that will deliver a runnable image in which the simulation execution is well scoped and controlled will ensure that across machines and platforms we get closer to a consistent execution environment [Melia2014].

Thus we propose here a container based recording alternative along with the captured metadata as a set of four models that combined together should be enough to deliver a reproducible record atom storage. We show here the project model in Table 2.

It describes the simulation. Its *history* field is the list of container images that have been built each time that the project source code changes. The container is setup directly from the source code of the simulation. We also propose a container model that is as simple as shown in the Table 3.

Based on the project's model in Table 2, we designed a record atom model shown in Table 4. A record is related to a project and a container in the history of the project containers. When a record atom is created, its container is the last container in the the project's history at that time. Thus, a record atom that will be done on a modified project source code has to be performed after the new container for this modified version of the project get pushed to the history field. This way we ensure that two records with different containers are from two different sources codes and also two records with the same containers are from the same source code.

A record atom reproducibility property assessment is done through a differentiation process. A differentiation process is a process that allows the resolution of a record atom reproducibility property compared to another. In this situation, the two record atoms are considered being from simulations that try to achieve the same goals. It is quite hard to know at a high level standpoint if two record atoms are the same because it will most likely be a domain related decision that proves that both records support the same claims. We focus here in an approach that provides some basic differentiation methods and allow the definition of new ones. Thus, the differentiation will most likely be based on the targeted record atom owner domain knowledge and understanding on the method used. Since the record atom is the state of a simulation execution, the inputs, outputs, dependencies and system fields have to be provided every time because from a run to another any of those may be subject to a change. Sometimes an action as simple as upgrading a library can have terrible and not easy to determine consequences on the outputs of another execution of the same simulation in the same system.

A differentiation request or shortly *diff request* is the *contract* on which the mechanism described before runs. A requesting record owner asks a targeted record atom owner to validate a record atom reproducibility proposal from him. In this mechanism, the requesting party has to define what the assessment is based on: repeated, reproduced, non-reproduced and non-repeated. This party also has to define the base differentiation method on which the assessment has been made: default, visual and custom. A default differentiation method is a Leveinstein distance[1] based differentiation on the text data. A visual one is a nobservation based knowledge assessment. And custom is left to the requester to define and propose to the targeted. It is important to point that the Table 1 is the core scheme of comparison that all differentiation request have to go through upon submission. To be accepted in the platform, the *diff request* assessment has to comply with the content of that Table. As such a *diff request* for two requests that have different inputs contents cannot be assessed as a repeat compared to one another because an input variation should lead to a reproducible assessment as pointed in the Table 1. The targeted record atom owner has to answer to the request by setting after verification on his side, the status of the request to agreed or denied. By default the status value is *proposed*. The table 5 represents the fields that a diff request contains. In fact one may say that in a model level a solved diff request is a relationship of reproducibility assessment between two records.

A project reproducibility property can be assessed from the differentiation requests on its records. All the requests that have a status to *agreed* represent a list of accepted couple of records that have been resolved as: repeated, reproduced, non-repeated and non-reproduced.

Data Driven Cloud Service Platform

To support simulation management tools metadata, we propose a cloud platform that implements the reproducible assessable record described previously. This platform has two sides. As shown in the Figure 1, an API[2] access and a Web Frontend[3] access. These two services are linked to a MongoDB[4] database that contains: the user accounts, the projects, the records, the containers and the differentiation requests. We implemented some restrictions depending on the type of access.

The API service exposes endpoints that are accessible by the Simulation management tool from the executing machine. It is a token based credential access that can be activated and renewed only from the Web Frontend access. The API allows the Simulation Management tools to push, pull and search projects and records. The API documentation will be available publicly and will present the endpoints, HTTP[5] methods and the mandatory fields in a structured JSON[6] format request content.

The Web Frontend service on the other end is controlled by the Cloud service. The Cloud service is accessible only from the Web Frontend. Thus when the user interacts with the Web Frontend, he is actually securely communicating with the Cloud service. This strongly coupled design allows a flexible deployment and upgrades but at the same time harden the security of the platform. The frontend access allows the user to manage his account and handle his API credentials which are used by the Simulation Management tools to communicate with the platform. It also allows the user to visualize his projects, records and requests. It is the only place

1. Levenshtein distance is a string metric for measuring the difference between two sequences.

Output Files	Source Code and Input Files			
	Same and Same	Same and Different	Different and Same	Different and Different
Same	Repeatable	Reproducible	Reproducible	Reproducible
Different	non-repeatable	Unknown	Unknown	Unknown

TABLE 1: *Reproducibility assessment based on source code, inputs and outputs*

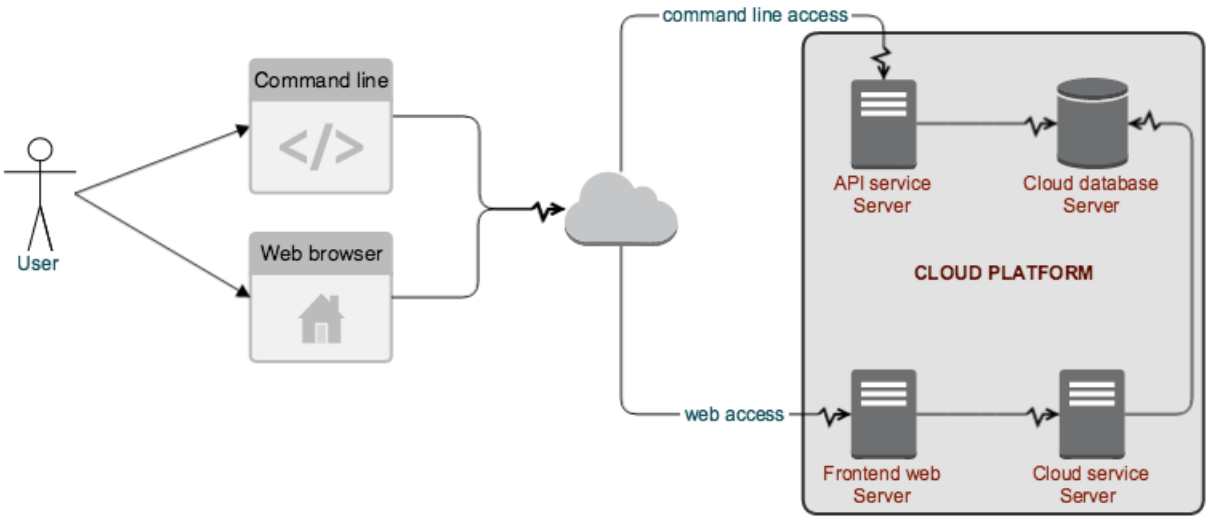

Fig. 1: *Platform Architecture.*

Fields	Descriptions
created	string: simulation creation timestamp.
private	boolean: false if project is public.
name	string: project name.
description	string: full description of the project.
goals	string: project goals.
owner	user: the creator of the project.
history	list: container images list.

TABLE 2: *Simulation metadata Project Model.*

Fields	Descriptions
created	string: simulation creation timestamp.
system	string: docker, rocket, ...
version	dict: version control source code's tag .
image	string: path to the image in the cloud.

TABLE 3: *Simulation metadata Container Model.*

where the user can update some content regarding a project, record or interact with his differentiation requests.

On the platform, the API is the only place where projects and records are automatically created. On the Web side this is still possible but it is a manual process.

A Simulation tool that needs to interact with the platform has to follow the endpoints descriptions in Tables 6 and 7.

Fields	Descriptions
created	string: execution creation timestamp.
updated	string: execution update timestamp.
program	dictionary: command, version control,...
inputs	list: input files.
outputs	list: output files.
dependencies	list: dependencies.
status	string: unknown, started, paused, ...
system	dictionary: machine and os information.
project	project: the simulation project.
image	container: reference to the container.

TABLE 4: *Simulation metadata Record Model.*

Fields	Descriptions
created	string: request creation timestamp.
sender	user: responsible of the request.
toward	record: targeted record.
from	record: requesting record.
diff	dictionary: method of differentiation.
proposition	string: repeated,reproduced,...
status	string: agreed,denied,proposed.

TABLE 5: *Simulation Record Differentiation Request Model.*

Endpoint	Content	
	Method	Envelope
$/api/v1/ < api-token > /project/pull/ < project-name >$	GET	null. Note: pull metadata about the project.
$/api/v1/ < api-token > /project/push/ < project-name >$	POST	name, description, goal... custom. Note: push project metadata.

TABLE 6: *REST Project endpoints*

Endpoint	Content	
	Method	Envelope
$/api/v1/ < api-token > /record/push/ < project-name >$	POST	program, inputs, outputs... Note: push metadata about the record.
$/api/v1/ < api-token > /record/pull/ < project-name >$	GET	null. Note: pull the container.
$/api/v1/ < api-token > /record/display/ < project-name >$	GET	null. Note: metadata of the record.

TABLE 7: *REST Record endpoints*

Integration with Sumatra and Use Case

Sumatra Integration

Sumatra is an open source event based simulation management tool. To integrate the cloud API into Sumatra we briefly investigate how Sumatra stores the metadata about a simulation execution.

To store records about executions, Sumatra implements record stores. It also has data stores that allow the storage of the simulation results. As of today, Sumatra provides three data storage options:

- FileSystemDataStore: It provides methods for accessing files stored on a local file system, under a given root directory.
- ArchivingFileSystemDataStore: It provides methods for accessing files written to a local file system then archived as .tar.gz.
- MirroredFileSystemDataStore: It provides methods for accessing files written to a local file system then mirrored to a web server.

Sumatra also provides three ways of recording the simulation metadata:

- ShelveRecordStore: It provides the Shelve based record storage.
- DjangoRecordStore: It provides the Django based record storage (if Django is installed).
- HttpRecordStore: It provides the HTTP based record storage.

Regarding the visualization of the metadata from a simulation, Sumatra provides a Django[7] tool named *smtweb*. It is a local web app that provides a web view to the project folder from where it has been ran. For a simulation management tool like Sumatra there are many advantages in integrating a cloud platform into its record storage options:

- Cloud Storage capability: When pushed to the cloud, the data is accessible from anywhere.
- Complexity reduction: There is no need for a local record viewer. The scientist can have access to his records anytime and anywhere.
- Discoverability enhancement: Everything about a simulation execution is a click away to being publicly shared.

As presented in the list of record store options, Sumatra already has an HTTP based record store available. Yet it does not suite the requirements of the cloud platform. Firstly because there is no automatic mechanism to push the data in the cloud. The MirroredFileSystemDataStore has to be fully done by the user. Secondly we think there is need for more atomicity. In fact, Sumatra gather the metadata about the execution and store it at the end of the execution, which can have many disadvantages generally when the simulation process dies or the Sumatra instance dies.

To integrate the cloud API and fully comply to the requirement cited before, we had to implement and update some parts of the Sumatra source code:

- DataStore: Currently the collect of newly created data happens at the end of the execution. This creates many issues regarding concurrent runs of the same projects because the same files are going to be manipulated. We are investigating two alternatives. The first is about running the simulation in a labeled working directory. This way, many runs can be done at the same time while having a private labeled space to write to. The second alternative consists of writing directly into the cloud. This will most likely break the already implemented data and record store paradigm in Sumatra.
- RecordStore: We make the point that the simulation management tool is the one that should comply to as many API interfaces as possible to give the user as many interoperability as possible with cloud platforms that support reproducible records. Thus, we intend to provide a total new record store that will fully integrate the API into Sumatra.
- Recording Mechanism: In Sumatra the knowledge of the final result of the execution combined with atomic state monitoring of the process will allow us to have a dynamic state of the execution. We want to make Sumatra record creation a dynamic many points recorder. In addition to an active monitoring, this feature allows the scientist to have basic informations about its runs may they crash or not.

2. Application Programming Interface.
3. Client browser access.
4. An Agile, Scalable NoSQL Database: https://www.mongodb.org/
5. HyperText Transfert Protocol.
6. A Data-Interchange format: http://json.org/
7. Python Web Framework: https://www.djangoproject.com/

Example project with Sumatra

The Sumatra repository[8] provides three test example projects. This example is based on the python one[9]. We propose here an example project as a base line to make the scientist's simulation comply with the principles described here. The platform currently supports docker as a container based system and Sumatra as a simulation management tool.

The example is the encapsulation of the execution of a python simulation code *main.py* that is simply:

```
import numpy
import sys

__version__ = "1.2.3a"

# version numbers are deliberately different
# for testing purposes
def get_version():
    return (1, 2, "3b")

def run():
    parameter_file = sys.argv[1]
    parameters = {}
    # this way of reading parameters
    execfile(parameter_file, parameters)
    # is not necessarily recommended
    numpy.random.seed(parameters["seed"])
    distr = getattr(numpy.random,
                    parameters["distr"])
    data = distr(size=parameters["n"])

    numpy.savetxt("Data/example2.dat",
                  data)


if __name__ == "__main__":
    run()
```

The input file to provide is *default.param* that contains:

```
# seed for random number generator
seed = 65785
# statistical distribution to draw values from
distr = "uniform"
# number of values to draw
n = 100
```

The instrumented project is organized as following:

- Python main: It's the simulation main source code.
- Git ignore: It contains the files that will not be versioned by git.
- Requirements: It contains all the python requirements needed by the simulation.
- Dockerfile: It contains the simulation docker container setup.
- Manage files: It's a script that allows the scientist to manage the container builds and the simulation executions.
- Sumatra integrate: It is a modified copy of Sumatra that integrates the API.

This demo example is currently working in linux and OsX systems and to run it, the scientist has to proceed as following:

- Get the source from github.
- To have an API key: Create an account on the platform and login.
- Access the user profile: In the home page, the round user floating image display two buttons that are the user profile access. Click the first one to view and the second one to edit the profile.
- Get the API key: Go to view the user profile and copy the string near the key image.
- Open the manage.sh file and replace the API key 3a8d4cc793bd3e5b85c733b523584... by this string. Update data path to be where the default.param file is located and the container path to be where the container image will be placed. By default the container image is generated in the demo-sumatra directory.
- Git global settings: Replace the git global username and email by the scientist's.
- Build the container image.
- Run the simulation: It will run main.py in the container and push the record along with the container image to the cloud space in the platform.
- Outcome: In the online dashboard, there will be a new project named demo-sumatra with a record that can be downloaded and executed with an input file like the default.param.

The following bash code, is the set of commands that will be ran by the scientist. Note that the first echo is the step described previously about replacing the API key in *manage.sh* by the scientist's one.

```
git clone github.com/faical-yannick-congo/ddsm-demo
cd ddsm-demo
git checkout setup
echo "Update the api key."
echo "To build the container image: "
./manage.sh --build --simulation demo-sumatra
echo "To run the simulation: "
./manage.sh --run-core --simulation demo-sumata
```

For a new simulation project we suggest that the scientist follow the same source structure as done in the demo example. Then to instrument his simulation, the scientist has to go through some few steps:

- Source code: The scientist may remove the script main.py and include his source code.
- Requirements: The scientist may provide the python libraries used by the simulation there.
- Dockerfile: Uncomment line 54 by removing the first character. Also the installation of non python libraries should be added here.
- Management: Here, the scientist has to update the API key and the git global settings (username and email).
- Running command: The scientist has to determine the full command that will be ran with the simulation and the input data to provide. The -v argument for docker allows file mapping from the local file system to the docker container. The -c argument allows the user to run a string command in the docker's /bin/bash terminal. More information can be found about those arguments. The scientist should update the run string to fit the simulation execution.

After performing this instrumentation on his simulation source code, the scientist has to build and run the simulation as done previously for the demo example. In addition, it is important that the scientist builds the container every time that the source modifications are ready to be tested as justified before when presenting the record model. In this case a newly exported image will be available to be ran with Sumatra. After a build, a run will

8. https://github.com/open-research/sumatra.git
9. https://github.com/faical-yannick-congo/ddsm-demo/tree/setup

execute the simulation and create the associated record that will be pushed to the cloud API. The interesting part of such a design is that the record image can be ran by any other scientist with the possibility to change the input data. This allows reproducibility at an input data level. For source code level modifications, the other scientist has to recreate an instrumented project. In the manage script, an API token is required to be able to access the cloud API. The scientist will have to put his own. A further detailed documentation will be provided. The source code of the demo can be found here[10]. It has been tested on an Ubuntu 15.04 machine and will work on any Linux or OsX machine that has docker installed.

The instrumented example presented here, has been done from a local development instance of the platform. AWS[11] server instances are being setup to host a public access to a production version of this platform. To reproduce this example demo, the url inside the *manage.sh* will have to be update accordingly to the location of the API endpoint. Further information will be delivered.

Conclusion and Perspective

Scientific computational experiments through simulation is getting more support to enhance the reproducibility of research results. Execution metadata recording systems through event control, workflows and libraries are the approaches that are investigated and quite a good number of softwares and tools implement them. Yet the aspect of having these records discoverable in a reproducible manner is still an unfulfilled need. This paper proposes a container based reproducible record atom and the cloud platform to support it. The cloud platform provides an API that can easily be integrated to the existing Data Driven Simulation Management tools and allow: reproducibility assessments, world wide web exposure and sharing. We described an integration use case with Sumatra and explained how beneficial and useful it is for Sumatra users to link the cloud API to their Sumatra tool. This platform main focus is to provide standard and generic ways for scientists to collaborate through reproducible record atoms and interact by the mean of differentiation procedures that will allow them to assess if a simulation is repeatable, reproducible, non-repeatable, non-reproducible or if its an ongoing research. A differentiation request description has been provided and can be presented as a hand shake between scientists regarding the result of simulation runs. One can request a reproducibility assessment property validation from a record against another.

We are under integration investigation for other simulation management tools used in the community. In the short term this platform will hopefully be a space where scientists could clone the entire execution environment that another scientist did. And from there be able to verify the claims of the project and investigate other execution on different input data. The container based record described here, we hope, will allow a better standard environment control across repeats and reproductions, which is a very hard battle currently for all simulation management tools. Operating systems, compilers and dependencies variations are the nightmare of reproducibility tools because the information is usually not fully accessible and recreating the appropriate environment is not an easy straight forward task.

10. https://github.com/faical-yannick-congo/ddsm-demo
11. Amazon Web Services: http://aws.amazon.com/

Finally it is important to point out that in some cases the five components (source code, inputs, hosting system, dependencies and outputs) cited before are not sufficient because the design of the simulation itself has to follow a rigorous method to better enforce reproducibility. Parallel stochastic simulations presents this requirement of determining the right techniques for generating parallel pseudorandom numbers [Hill2015].

Acknowledgments

This research paper is made possible through the help of my thesis supervisors and colleagues.

First and foremost, I would like to thank Dr. David Hill and Dr. Jonathan Guyer for their most support, encouragements and critics.

Second, I would also like to thank Dr. Daniel Wheeler for his ideas and brainstorms at the early stage of this investigation and his continuous research for better technologies for computational science.

Finally, I would like to thank Dr. Andrew Reid and Dr. Stephen Langer for their exceptional willingness to help me reshape and bring more lights in this paper. They kindly read my paper and offered invaluable detailed advices on grammar and organization of the paper.

References

[Slezak2011] P. Slezák and I. Waczulíková, *Reproducibility and Repeatability*. Physiological Research, Volume 60, Issue 1, pp. 203-205, 2011.

[Oinn2006] Tom Oinn et al, *Taverna: lessons in creating a workflow environment for the life sciences*. Concurrency and Computation: Practice and Experience, Special Issue: Workflow in Grid Systems, Volume 18, Issue 10, pages 1067–1100, 25 August 2006.

[Langer2014] Stephen Langer et al, *gtklogger: A Tool For Systematically Testing Graphical User Interfaces*. NIST Internal Publication, pp. 2-3, October 2014.

[Guo2012] Philip Guo, *CDE: A Tool for Creating Portable Experimental Software Packages*. Reproducible Research For Scientific Computing, pp. 2-3, October 2012.

[MacDonnell2012] John MacDonnell, *Git for Scientists: A Tutorial*. http://nyuccl.org/pages/gittutorial/, July 2012.

[Oliver2013] Marc Oliver, *Introduction to the Scipy Stack - Scientific Computing Tools for Python*. Jacobs University, http://math.jacobs-university.de/oliver/teaching/scipy-intro/scipy-intro.pdf, November 2013.

[Davidson2010] Andrew Davidson, *Automated tracking of computational experiments using Sumatra*. EuroSciPy 2010, http://www.andrewdavison.info/media/slides/sumatra_euroscipy2010.pdf, 2010.

[Sandve2013] Geir Kjetil Sandve et al, *Ten Simple Rules for Reproducible Computational Research.*. PLoS Comput Biol 9(10): e1003285. doi:10.1371/journal.pcbi.1003285, October 2013.

[Heroux2011] Michael A. Heroux, *Improving CSE Software through Reproducibility Requirements*. SECSE '11 Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering, pp. 28-31, ISBN: 978-1-4503-0598-3 doi:10.1145/1985782.1985787, May 2011.

[Bottomley2014] James Bottomley, *What is All the Container Hype?*. Linux Foundation, p. 2, http://www.odin.com/fileadmin/media/hcap/pcs/documents/ParCloudStorage_Mini_WP_EN_042014.pdf, April 2014.

[Melia2014] Ivan Melia et al, *Linux Containers: Why They are in Your Future and What Has to Happen First*. Cisco and RedHat, p.7, https://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf, September 2014.

[Hill2015] David Hill, *Parallel Random Numbers, Simulation, Science and reproducibility*. IEEE/AIP - Computing in Science and Engineering, Volume:17, Issue: 4, pp. 66-71.