An Analysis of the Reproducible Workflows

Wilmot Osei-Bonsu

***Abstract*—If the essence of scientific discovery could be reduced to a single term it would be reproducibility. Experiments hold merit in the scientific community only when their conclusions can be recreated by peers. The current standard for peer-reviewed submission does not carry the degree of trust that weve come to expect from other scientific disciplines. Our work will incorporate transparency into this process through a series of examples. We have created a series of reproducible workflows with a varying level of complexity aimed at guiding members of the computing community through a series of best practices. Standardizing how computational experiments are shared with the broader scientific community will bring the body of computer science computational research closer to the ideal cherished throughout the broader scientific community; namely trust.**

CONTENTS

LIST OF FIGURES

7 This is the Dockerfile for the March Madness container workflow. (1) The file begins with a base image of Ubuntu 18.04. Comment two (2) sets an environment variable in the image. This environment variable disables interactive prompts from installations during the build phase. The section beginning at comment three (3) outlines the dependencies needed to pull, compile, and run the March Madness application, as well as generate this paper. Code section number four (4) copies all relevant scripts and files into the container. These files could be generated within the container but in this implementation, the files generated before the creation of the image and passed thereafter. The final section (5) executes the script that runs the experiment. 6

8 This installation creates a Python virtual environment and installs popper. To confirm Popper installed correctly, run `popper --version`. Alternatively, `popper scaffold` followed by `popper run -f wf.yml` will verify Popper's installation. Current installation directions can be found on the official Popper documentation page. (1) Creates a directory for the virtual environment. (2) Installs a virtual environment software package. Note the command is for a Debian system. (3) Create a virtual environment for Popper. (4) Loads the virtual environment. (5) install Popper. If a new terminal window is open, re-run command four (4) before executing Popper command. 7

9 This figure depicts a directory tree layout for the March Madness simulation's Popper solution. (1) The container directory has the Dockerfile referenced by the `wf.yml` file. (2) the scripts directory contains scripts for running the simulator and validating its output. (3) The paper directory holds all information needed to construct a reproducible paper. (4) `wf.yml` is a YAML file with enumerated workflow steps. 8

10 This figure depicts `wf.yml` file for the March Madness simulator's Popper workflow. (0) `wf.yml` is a YAML file containing a `step` directory. (1) The first step runs `madness.sh` script responsible for pulling and running simulation source code from GitHub. (2) Runs result through validation script on the simulated output data. (3) This section runs a Python script to embed output in a LaTeX file. The script only requires Pythons, therefore it is run in Python 3 docker image. (4) This script generates a PDF from `paper.tex` file. (5) removes simulator source code from the host; leaves only generated paper. 8

I. Introduction

Experiments hold merit in the scientific community only when their conclusions can be recreated by peers. Due to resources limitations for researchers, tools that address the lack of reproducibility in computational experiments must be paired with resources that aggregate and educate researchers about there incorporation into existing workflows. Without these resources, researchers will remain unaware the suite of tools developed to reduce the barriers to sharing experiments.

II. Challenges to Adoption

Although a lack of incentives discourages sharing, there are a host of obstacles that make experiment transparency difficult for those who recognize its importance. Uncertainties in code dependence and the tendency for software to evolve overtime decreases the reliability of experimental results overtime.

A. Code Dependency

Environmental differences can potentially keep software from compiling or producing reliable results [2]. Even if software dependencies are well documented, a researcher cannot guarantee that an experiment will compile or run on a future system [13]. Library difference, compiler optimization, and operating variations could interfere with future efforts to reproduce experimental results. These differences are not only limited to top-level software abstraction. Variations in computer architecture, like the representation floating-point numbers and arithmetic, can make an operating system and hardware differences substantial enough to alter results.

B. Code Rot

The natural tendency for software to evolve overtime also limits experiment reproducibility. The addition of new features and removal of bugs has the potential to make older versions of software obsolete. This problem is known as code-rot [2]. This uncertainty makes building upon existing experiments problematic. Code dependency and code rot issues make reproducing computational experiments more difficult. Reducing both creates more trust in computational experiments.

C. Ad-hoc Validation

Differences in experiment descriptions make independent validation of results difficult [9]. Typical solutions include encoding a validation script along with the codebase. The script is meant to be run if a researcher wants to validate the results of the experiment in the future [9]. This method of validation differs between experiments causing confusion; especially as the code base ages. Ad-hoc validation suffers from code rot and dependency issues like that of the experimental product.

III. Conventional Solutions

In the absence of well-established reproducibility best practices, a series of common solutions have emerged. These solutions include code sharing through version control, the use of virtual machines (VM), and ad-hoc validation methods [9]. Examining these solutions will emphasize the need for well-established reproducibility workflows norms.

A. Hyper Text Markup Language

Reproducible computation research centers around the generation of reproducible papers, known as executable papers. Simple executable papers embed Hypertext Markup Language (HTML) referring to different sections of a document, source material, or external resources [13]. If the steps of the research are described accurately, future readers can reproduce the experiment and validate the results [13]. Three potential complications arise from this method.

1) Researchers use different methods to describe experiments. Without a set metric, documentation between the researcher can be complicated or difficult to understand [13].
2) Links to dependencies can break if a researcher moves between resources without adequately updating the source material [13].
3) The software and its dependencies may not work in modern or future architecture [13].

B. Version Control

The rise of version control software allows researchers to access a common code base. In some cases, this practice replaces proper reproducible workflows. Unfortunately, this solution leads to code dependency issues. Differences in Run-time environment and input data can result in varying results [9].

C. Virtual Machines

Virtual machines (VM) can be an alternative to sharing source code. A VM solution allows users to pre-program the environment parameters necessary for the program to compile and execute. This solution solves the dependency problems of version control, however, VM solutions come with reproducible problems as well. Virtual machines require a large amount of memory to execute. In experiments sensitive to system architecture, overhead costs can affect the result of performance testing experiments [9]. Additionally, computers limited in memory are unable to run a large number of VM-based experiments. VM Solutions also lack transparency. Rather than sharing information, VMs present researchers with a black box [2] making evaluating and modifying experiments difficult. Additionally, like code sharing solutions, VMs rely on ad-hoc validation for experiments. Although VM solutions solve some problems, they provide little guarantee of reproducibility to researchers.

IV. Demonstration Introduction

Solutions that integrate workflow software, virtual machines, and continuous integration exist. Despite the growth of reproducible software, researches lack incentives to implement reproducible solutions in their existing workflow [2]. This has created a need for a guide to incorporating reproducible software into experiments. The following example aims to provide this guide to researchers. A series of workflows utilizing the tools described in this paper will be used to demonstrate how an existing experiment can be made reproducible.

A. March Madness Explanation

A sports ranking simulation based on Kenneth Masseys 1998 undergraduate thesis will be used to demonstrate each software tool. The application uses historical game data to predict the ranking of teams with a variable advantage given to the home team in each game. The goal is to demonstrate how each tool can be applied to an existing project. This demonstration serves as a well-documented tutorial for creating reproducible work.

V. March Madness Simulator Scripts

The solution presented in this section utilizes a series of bash scripts to generate a reproducible paper. This is a common first solution for researchers who choose to make their work replaceable.

The March Madness simulation requires the following software packages.

- g++
- Cmake
- Installtexlive-latex-recommended
- Installtexlive-publishers
- Python 3.6

The installation of each package varies across systems. Figure 1 describes the installation process for a Debian system. Once all dependencies are installed, `run.sh` will begin the process of generating simulated team ranking and embedding them in this paper. Source code and script sharing solutions provide a simple method for enabling the future reproducibility of an experiment. However, this solution is limited by possible dependency issues as well as the potential for code rot.

```
apt-get update
apt-get install g++
apt-get install git
apt-get install cmake
apt-get installtexlive-latex-recommended
apt-get installtexlive-publishers
apt-get install python3.6
```

Fig. 1. Debian installation commands for software dependencies necessary for running March Madness simulation application

A. Generate Paper

In this example, `generate_paper.py` generates and embeds a table into `Finished_Paper.tex`. The application looks for `% March Madness Script` and replaces it with the generated table. `build.sh` compiles the LaTeX file.

B. Generating More Complex Papers

Although this implementation generates the paper using a python program and bash script there is a better solution. One method to consider is PythonTeX, Geoffrey M Poore's LaTeX package. The package creates reproducible documents using python embedded LaTeX code [10]. PythonTeX is a LaTeX package that executes Python code in a LaTeX document. This combines result analysis with the code required to perform it. Consider using this package for larger experiments.

```
#!/bin/sh
echo 'Running run.sh'

#Get path of directory containing scripts

full_path=$(realpath $0)
$dir_path=$(dirname $ full_path)

%(1) Pull and execute simulator code
echo 'Running March Madness Simulation'
$dir_path/madness.sh

%(2) Run result validation script
echo 'Running Result Validation'
$dir_path/validate.sh

%(3)Embed results in LaTeX file & create PDF
echo 'Creating LaTeX File'
$dir_path/build.sh
```

Fig. 2. This script serves as the entry point for the March Madness simulator application. It executes three scripts. (1) `run.sh` pulls source code from GitHub and executes the experiment. (2) `validate.sh` performs result validation on the results. (3) `build.sh` embeds the results in a LaTeX file and produces a PDF.

```
#!/bin/sh
echo 'Running madness.sh'
mkdir March_Madness
cd ./March_Madnes
git init
git pull
http://github.com/wbonsu/MarchMadness.git
cmake .
make
./March_Madness_Simulator
```

Fig. 3. This script pulls and compiles the March Madness ranking simulator from GitHub. A machine with git, g++, and cmake is a requirement for running this script.

VI. Containerized Workflow

As reproducibility research gains more attention, tools for managing the complexity of source code repositories, run-time environment, and experimental data processing have developed. Software containers, software that packages application dependencies in an executable file system, have increased in popularity for their ability to run experiments without run-time environment concerns [9]. These tools can be used to package the dependencies of an experiment along with any source code and data [9]. The uncertainty caused by software and environment dependencies, as well as the possibilities of code rot are mitigated with a software container workflow. Unfortunately, this solution does not address potential ad-hoc validation.

```
#!/bin/sh
echo 'Running build.sh'
pdflatex ./Finished_Paper
bibtex ./Finished_Paper
pdflatex ./Finished_Paper
pdflatex ./Finished_Paper
```

Fig. 4. This script is responsible for embedding experimental results in a LaTeX file and generating a PDF file. A machine with texlive-latex-recommended and apt-get texlive-publishers is required to run this script.

A. Terminology

The following is a set a basic Docker terminology used throughout this paper. The terms are defined by the Docker glossary.

- **Base Image** - A base image has no parent image specified in its Dockerfile. It is created using a Dockerfile with the FROM scratch directive.
- **Build** - build is the process of building Docker images using a Dockerfile. The build uses a Dockerfile and a context. The context is the set of files in the directory in which the image is built.
- **Container** - A container is a run-time instance of a docker image. A Docker container consists of a Docker image, an execution environment, and a standard set of instructions.
- **Docker Hub** - A Dockerfile is a text document that contains all the commands you would normally execute manually to build a Docker image. Docker can build images automatically by reading the instructions from a Dockerfile.
- **Image** - Docker images are the basis of containers. An Image is an ordered collection of root file system changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered file systems stacked on top of each other. An image does not have state and it never changes.
- **Parent Image** - An images parent image is the image designated in the FROM directive in the images Dockerfile. All subsequent commands are based on this parent image. A Dockerfile with the FROM scratch directive uses no parent image and creates a base image.

B. Docker

Docker has emerged as a popular open-source software container used to encapsulate environment variables [2]. Dockers simplicity relative to other reproducible workflows has made appealing to researchers exploring reproducible workflows [2]. It provides an additional layer of abstraction over operating-system-level virtualization [9].

Docker can be installed on Windows, Mac, and Linux operating systems. Docker Desktop for Windows uses Hyper-V virtualization and runs as a native Windows application [1]. It can configure and run both Windows and Linux containers. Docker Desktop for Mac uses a macOS Hypervisor framework to provide Docker functionalities for Mac operating systems [1]. Once Docker has been installed, an image can be constructed interactively or with a Dockerfile. Creating images interactively leaves little record of the software packages installed on the image and how they interact [2]. Docker provides Dockerfiles as a human-readable script for describing how to build a specific image [2]. Because Dockerfiles are small plain text files, they require fewer resources when stored and shared [2]. Additionally, users can edit Dockerfiles directly, making it a straightforward solution.

```
FROM ubuntu:18.04
RUN apt-get update
```

Fig. 5. Example of Dockerfile. FROM is the keyword necessary to select a base image. In this example the base image is Ubuntu. Any Docker image can be a base image.

C. Solution to Challenges

Docker's ability to resolve dependency and code rot issues makes it a good candidate for reducing the technical issues associated with reproducibility. Its simplicity reduces social barriers to implementing a reproducible workflow.

1) Code Dependency: Docker workflow solutions solve code dependency issues by providing researchers with a binary file complete with all dependencies preconfigured [2]. Researchers are provided with a binary image with all software installed, configured, and tested [2]. Because of its similarity with virtual machines, Docker addresses code dependency issues. However, unlike traditional virtual machines, Docker images share a core Linux kernel with the host machine. This allows them to run with a low overhead cost. Computers that can run only a few VMs, can manage 100 docker containers [2]. This simplicity has aided Docker in becoming a tool of choice for reproducible workflows [2].

2) Code Rot: Docker mitigates code rot by limiting software environments to a particular operating system and suite of libraries. These distributions use a staggered release model with testing to catch potential problems and regular security updates [2]. Because limiting software environments does not completely avoid the challenges of code rot, Docker also provides a utility to save an image as a portable compressed file that can be loaded by another docker user [2]. This provides a robust way to run a particular. Additionally, when constructing a Dockerfile, checks and tests can be added following each command [2]. This verifies the setup process and reduces the likelihood of code rot.

VII. March Madness Simulator Container Workflow

This section will provide an example of applying docker to an existing experiment. The goal is to reduce the social barriers preventing researchers from applying containerized solutions to their workflows. This section will begin with instructions for downloading docker then transition to creating a general docker workflow. It will conclude with the specific docker workflow that generated this paper.

A. Workflow Set up

Although docker images can be created using Windows and Mac OS desktop applications, the following example will use a virtual machine. This example creates a Linux virtual machine, using VM Virtual Box and Ubuntu 18.04, before installing before starting the Docker installation process. Before installing Docker, ensure the virtual machine is up to date. Docker provides functionality for creating groups after installation but for this example will omit that step. Directions for specific machines can be found on the Docker homepage.

```
$ yes| apt-get install docker.io
$ vim Dockerfile
{Dockerfile contense found in later figure}
$ docker build .
$ docker run -d <Image_id>
$ docker cp <ImageId>:\Finished_Paper.pdf .
```

Fig. 6. This figure shows the steps required to create the March Madness simulator's Docker workflow.

Once Docker is running on the virtual machine, the next step is creating the Dockerfile. The file will contain a parent image followed by commands that describe the experiment. For this example, the bash and python scripts that will run the march madness simulation application and generate the article are copied into the Dockerfile from the host machine.

The command `docker build .` is run to build a docker image from the completed Dockerfile. Then, `docker images` will display the host machine's docker images. The information is formatted as follows: `REPOSITORY SIZE TAG IMAGE TIME`.

`$ docker run (ImageId)` creates and runs a container from a specified docker image. By default, a container created with `$ docker run (ImageId)` will terminate after it completes its instructions. `$ docker run -d (ImageId)` keeps the container open after it has completed its instructions. `$ docker cp (CONTAINERID) : (FILE PATH IN CONTAINER) (HOST PATH TARGET) -d (ImageId)` will copy specified files from the docker container to the host machine. This example will use this to copy the PDF created by the container back in to the host machines file system.

Applying the above instructions to an existing workflow will enable others to execute the experiment on their machines without considering software dependencies and environment variables. Docker provides functionality for exporting images as a compressed file with the command, `$ docker save (ImageId) > tarfilename.tar`, The compressed file can be shared privately, through GitHub or BitBucket. Additionally, an image stored as a compressed file can be loaded with `$ docker load < tarfilename.tar.gz`.
Researchers who choose to utilize a container solution can also push or pull an image to the Docker Hub, a resource for sharing Docker images and components.

```
% (1) Pull parent image.
From ubuntu:18.04

% (2) Set environment variables.
ENV DEBIAN_FRONTEND=noninteractive

% (3) Download software dependencies.
RUN yes| apt-get update &&
    yes| apt-get install g++ &&
    yes | apt-get install git &&
    yes | apt-get install
    texlive-latex-recommended &&
    yes | apt-get install
    texlive-publishers &&
    yes | apt-get install python3.6

% (4) Copy scripts into image
COPY ./scripts/run.sh /
COPY ./scripts/madness.sh /
COPY ./scripts/build.sh /
COPY ./scripts/validate.sh /
COPY ./scripts/generate_paper.py /
COPY ./scripts/paper.tex /
COPY ./scripts/Finish_Paper.bib

% (5) Run entrypoint script.
CMD ./run.sh
```

Fig. 7. This is the Dockerfile for the March Madness container workflow. (1) The file begins with a base image of Ubuntu 18.04. Comment two (2) sets an environment variable in the image. This environment variable disables interactive prompts from installations during the build phase. The section beginning at comment three (3) outlines the dependencies needed to pull, compile, and run the March Madness application, as well as generate this paper. Code section number four (4) copies all relevant scripts and files into the container. These files could be generated within the container but in this implementation, the files generated before the creation of the image and passed thereafter. The final section (5) executes the script that runs the experiment.

B. Display results

When the docker image associated with this Dockerfile is executed, build.sh generates the table that follows. The table shows the top 8 finishers in the 2019 March Madness tournament alongside the simulated results without any home-field advantage.

Docker Simulated Results			
2019 Results	Home-Field Advantage 0	Home-Field Advantage 2	Home-Field Advantage 4
Virginia	Gonzaga	Gonzaga	Gonzaga
Texas Tech	Duke	Duke	Duke
Auburn	Virginia	Virginia	Virginia
Michigan St	North Carolina	North Carolina	North Carolina
Purdue	Texas Tech	Texas Tech	Texas Tech
Duke	Michigan St	Michigan St	Michigan St
Gonzaga	Houston	Houston	Houston
Kentucky	Kentucky	Kentucky	Tennessee
Oregon	Tennessee	Tennessee	Kentucky
Tennessee	Virginia Tech	Virginia Tech	Virginia Tech
Virginia Tech	Auburn	Auburn	Michigan
LSU	Purdue	Michigan	Auburn

VIII. Verification and Validation

In computer science, validation and verification, abbreviated V&V, forms the foundation of reproducible software and computational experiments. Verification refers to internal characteristics of software, those confirmed by unit testing, while validation focuses on external characteristics [**?**]. The presence of both in a computation experiment form the cornerstone of reproducibility in computational experiments and trust in experimental results.

A. Popper

Popper has emerged as a tool capable of adding verification and validation to computational experiments. Popper provides researchers with a common framework for describing all dependencies and artifacts for a given experiment [9]. Artifacts in this context refer to any component of an experiment. The software package provides a protocol for generating self-contained experiments while allowing researchers to specify validation criteria [9] According to the developers of the software tool, Popper allows researchers to automate the execution and validation of computational and data-intensive experimentation workflows [11]. Popper has become a leading software for validating and verifying artifacts.

B. Popperized

An experiment is considered popperized, or popper-compliant, if it contains the following [9]:

- Experimental code
- Experiment orchestration code
- References to data dependencies
- Experiment parameters
- Validation criteria and results

Popper workflows are defined in an Ain't Markup Language (YAML) file entitled `wf.yml` [11]. The YAML file describes the location of source information and the process for executing and validating the experiments and displaying its results. Source code and data sets are generally excluded from the Popper repository. That information is stored in an adjacent repository and extracted by Popper as input for the experiment. Other files relevant to the experiment are stored in the popperized directory. After an experiment runs, the results and images are either consumed by a post-processing or paper generating script. This workflow allows writers to present their work transparently [9]. Future readers can explore results and re-run experiments.

C. Automated Validation

Experiment verification and validation can be classified into two categories: logic or result.

Popper handles experimental logic verification using continuous-integration (CI) services, such as TravisCI. Integrating TravisCI in a Popper workflow requires a `.travis.yml` file, containing a list of tests, in the root directory [9]. These tests can verify that the paper is in a state where it can be built [9].

Result testing is related to the integrity of the experimental results [9]. Validation that is not hardware dependent can be executed on any CI platform, like TravisCI. It is advisable for tests dependent on measuring underlying hardware to be conducted as part of the post-processing routine [9]. For hardware-dependent validation that requires corroboration of a baseline model, that step can be executed before the experiment runs [9].

IX. March Madness Simulator Popper Workflow

This section will provide an example of applying Popper to an existing experiment. This serves as a guide for researchers who want to apply Popper to their workflows. This section will describe the installation process for Popper then its implementation in the March Madness simulation application.

A. Download & Configure Popper

The following installation directions can be found on the Popper official documentation site. Popper provides a pip package for installation. Depending on your Python distribution pip may not work. In this case, it is recommended that you use a virtual environment, directions for which can be found here or in Figure 8. Once Popper has been installed, `popper scaffold` will download an example Popper workflow. To execute run `popper run -f wf.yml`.

```
(1) mkdir ./virtualenvs
(2) apt-get install virtualenvs
(3) virtualenv ./virtualenvs/popper
(4) source ./virtualenvs/popper/bin/activate
(5) pip install popper
```

Fig. 8. This installation creates a Python virtual environment and installs popper. To confirm Popper installed correctly, run `popper --version`. Alternatively, `popper scaffold` followed by `popper run -f wf.yml` will verify Popper's installation. Current installation directions can be found on the official Popper documentation page. (1) Creates a directory for the virtual environment. (2) Installs a virtual environment software package. Note the command is for a Debian system. (3) Create a virtual environment for Popper. (4) Loads the virtual environment. (5) install Popper. If a new terminal window is open, re-run command four (4) before executing Popper command.

B. March Madness Simulation Popper Workflow

The `wf.yml` file describes a popper-compliant experiment. The steps are enumerated in a YAML dictionary named `steps`. Each item in the list is a dictionary term and has at least a `uses` attribute that describes the docker image executed for that step.

A step can reference a container image in the directory, GitHub repository, or Docker Hub. Popper documentation provides examples for each source. The March Madness simulation's Popper workflow references a Dockerfile in the directory and contains the `uses`, `runs` and `args` attributes. A complete table of attributes can be found here.

```
.
|__ (1) container
|   |_ Dockerfile
|__ (2) scripts
|   |_ generate_paper.py
|   |_ madness.sh
|   |_ validate.sh
|__ (3) Paper
|   |_ paper.tex
|   |_ paper.bib
|   |_ build.sh
|   |_ popper.bib
|__ (4) wf.yml
```

Fig. 9. This figure depicts a directory tree layout for the March Madness simulation's Popper solution. (1) The container directory has the Dockerfile referenced by the `wf.yml` file. (2) the scripts directory contains scripts for running the simulator and validating its output. (3) The paper directory holds all information needed to construct a reproducible paper. (4) `wf.yml` is a YAML file with enumerated workflow steps.

```
# (0) Start of file
steps:

# (1) Run March Madness Application
- uses: './container'
  runs: './scripts/madness.sh'

# (2) Run Validation
- uses: './container'
  runs: './scripts/validate.sh'

# (3) Create LaTeX Files
- uses: docker://python:3
  runs: ['./scripts/generate_paper.py',
         './Paper/paper.tex',
         './March_Madness/output.txt',
         './March_Madness/output_field.txt',
         '12']
# (4) Generate Paper
- uses: './container'
  runs: './Paper/build.sh'

# (5) Clean directory
- uses: './container'
  args: ['rm', '-r', 'March_Madness']
```

Fig. 10. This figure depicts `wf.yml` file for the March Madness simulator's Popper workflow. (0) `wf.yml` is a YAML file containing a `step` directory. (1) The first step runs `madness.sh` script responsible for pulling and running simulation source code from GitHub. (2) Runs result through validation script on the simulated output data. (3) This section runs a Python script to embed output in a LaTeX file. The script only requires Pythons, therefore it is run in Python 3 docker image. (4) This script generates a PDF from `paper.tex` file. (5) removes simulator source code from the host; leaves only generated paper.

C. Generate Results

The following step in the YAML file embeds the March Madness simulation workflow's results table into this document.

```
- uses: docker://python:3
  args: ['./scripts/generate_paper.py',
         './Paper/paper.tex',
         './March_Madness/output.txt',
         './March_Madness/output_field.txt',
         '12']
```

The table lists the top twelve teams in the 2019 March Madness Championship followed by three sets of simulated results. Each simulated result adds a set number of points to the home team's score.

Docker Simulated Results			
2019 Results	Home-Field Advantage 0	Home-Field Advantage 2	Home-Field Advantage 4
Virginia	Gonzaga	Gonzaga	Gonzaga
Texas Tech	Duke	Duke	Duke
Auburn	Virginia	Virginia	Virginia
Michigan St	North Carolina	North Carolina	North Carolina
Purdue	Texas Tech	Texas Tech	Texas Tech
Duke	Michigan St	Michigan St	Michigan St
Gonzaga	Houston	Houston	Houston
Kentucky	Kentucky	Kentucky	Tennessee
Oregon	Tennessee	Tennessee	Kentucky
Tennessee	Virginia Tech	Virginia Tech	Virginia Tech
Virginia Tech	Auburn	Auburn	Michigan
LSU	Purdue	Michigan	Auburn
North Carolina	Michigan	Purdue	Purdue
Houston	Youngstown St	Nevada	Nevada
Florida St	Nevada	Buffalo	Florida St
Michigan	Buffalo	Florida St	Buffalo

D. Popper Tutorial Reflection

Popper has a steep learning curve for those not familiar with the software package. The resources and examples outlined in the section provide an introduction for those interested in applying the software to their work. A researcher interested in but not informed about the topic of creating reproducible workflows with Popper should have an entry-point for applying this tool.

X. Collective Knowledge

Several tools have emerged to address reproducibility, most notably Docker [2] and Popper. Docker and other virtual environment software have gained popularity due to their ability to take a snapshot of the environment where a software experiment was conducted. Unfortunately, environment snapshots limit researchers' need to validate and build upon previous experiments [3]. Collective Knowledge (CK) is a framework that provides a standard application program interface (API) that enables researchers to share their projects and artifacts in a common format [4]. Researchers using CK can share their entire workflow or components, such as source code and data sets, through repositories like GitHub and Bitbucket [3]. The software service abstracts away access to hardware allowing another researcher to reproduce an experiment under similar conditions [3]. If a researcher is unable to reproduce an experiment due to an incomplete description of software

dependencies, they can debug the workflow and share the fixed repository with the Collective Knowledge community. CK's long term goal is to enable a collaborative, reproducible, and sustainable research environment based on DevOps principles [4].

A. Features

Entities, repositories, actions, and modules form the basic vocabulary of collective CK [3]. Entries refer to individual components of a workflow, like source code, data sets, and scripts [12]. CK facilitates sharing and organization by assigning unique identifiers (UUIDs) to every entry within the project [12]. Each entry is stored in a sperate directory with its information stored in a JavaScript Object Notation (JSON) file located in a subdirectory entitled .cm [4]. Modules are a specific type of entry that implements the functionality of CK. Modules act as a collection of entries and the actions that operate on them [12]. This creates a two-level directory structure where the top-level directories represent CK modules [12]. The second-level directory store program source code, datasets, and experiments. Actions are CK functionalities offered by modules that operate on lower level entries [12].

B. Code Dependency

CK reduces code dependency by automatically detecting and rebuilding the environment of a workflow and installing missing software packages [5]. All software, data, and models are represented by packages serialized by automatically generated UUID, semantic tags, and information about versioning and supported platforms [5]. By cataloging this information, CK uses the JSON API to automatically detect already installed software and install missing software and other source-specific packages automatically [7].

C. Code Rot

Members of the Collective knowledge community who download a solution and have trouble running the experiment due to missing environment specification can debug the workflow and post a corrected version to the CK database [12]. This feature reduces the likelihood of out-of-date repositories making code rot less likely over time.

D. Demonstration

The first step in creating and running a CK repository is downloading the CK source code. This can be done on Linux, Windows, or MacOS [6]. For a Linux system `pip install ck` will download and install CK. After installing CK, `ck pull repo` will download a CK repository.

If the repository has a compile action the following command will trigger the compilation of the source code and CKs automated software dependency detection. At t his point, extra plugins, modules, and packages can be added to the existing project. Once the source code has been compiled `ck run [module_name]: [entity(optional)]` will run the experiment.

E. CodeReef

CodeReef is an open platform for sharing components for machine learning experiments across different systems. The service provides a way to package and share models as customizable files. Like Collective Knowledge it is an open-source Python-based library. The major challenge for a researcher trying to make use of machine learning models is figuring out how to integrate the models in their complex system efficiently. This challenge is compounded by the difference is software dependencies, hardware specifications, and data formats. CodeReef was developed to address a need for sharing machine learning models efficiently. CodeReefs long term goal is to provide researchers with a simple, standardized, and widely adopted resource for producing research papers in a collaborative and reproducible way.

F. Collective Knowledge Drawbacks

The creators of CodeReef chose to use the open Collective Knowledge format to share components and workflows because of its continued use in academia and industrial projects. However, CodeReef addresses two major limitations of Collective Knowledge when applied to machine learning workflow [8]. CK technology is a distributed system without a centralized location for components. This makes keeping track of all community contributions difficult [8]. Additionally, the distributed nature of CK makes adding new components, assembling workflows, and automatic testing across different platforms difficult [8]. CK lacks repository and entity versioning, making it challenging to maintain stable workflows. A bug in one CK component can break dependent workflows in adjacent project [8]. Improving on these allows issues allows CodeReef to tackle code dependency and code rot in a similar way to Collective Knowledge while lowering the barrier of entry through a simplification of the software service.a

XI. Appendix

Throughout my four years at Saint John's University, I have taken several courses that have contributed to the skills used to complete this capstone project. From researching and writing skills to a knowledge of programming and problem-solving skills, my time at SJU has made the completion of this project possible.

The first skill necessary for this project was the ability to aggregate sources and conduct research. Courses like First-Year Seminar (FYS 111), taught by Dr. Kyle McClure, and Problem Solving Seminar (CHEM 215), taught by Dr. Henrey Jakubowski, allowed me to cultivate new researching skills while developing my critical reading and writing skills. The ability to communicate my ideas in writing was also supported by writing-intensive courses, like Introduction to Peace and Conflict Studies (PCST 111), Great Issues in Philosophy (PHIL 121), Fiction & Poetry (ENGL 122A) and Ethical Issues In Computing (CSCI 369).

The other necessary for the completion of this project was critical thinking and problem-solving. These skills were developed in the computer science and mathematics courses I have taken at Saint John's University. Every computer

science course I completed helped me conduct my research, but Software Development (CSCI 230), Agile & Efficient Software Development (CSCI 317), and Programming Contest Team (CSCI 217A) were among the most useful. I was introduced to GitHub and the basic tenants of software development needed to write the March Madness simulator in CSCI 230. The application was written in CSCI 317 where I also cultivated software development skills necessary to complete the examples presented in my project. Lastly, the problem-solving skills I acquired from CSCI 217A was instrumental in completing the various implementations of reproducibility tools. Without the rigorous practice in problem-solving and debugging, provided by CSCI 217, I would not have been able to complete these examples. I am thankful for these, and all the other courses I have taken in my computer science education.

The mathematics courses I completed at CSB/SJU challenged me to think critically and communicate efficiently. Both of these skills proved useful in this capstone research project. Foundations of Mathematics (Math 241) and Combinatorics/Graph Theory (MATH 322) improved my ability to communicate technical material in writing. I am thankful for every mathematics course I completed at CSB/SJU and the facility that supported me through them.

I would like to thank Dr. Mike Heroux of the CSB/SJU Computer Science faculty. I have learned invaluable skills from Research Seminar (CSCI 373) and Agile/Efficient Software Development (CSCI 317). CSCI 373 developed my writing and public speaking skill and CSCI 317 prepared me for the challenges I have encountered in my internships. I am confident these skills will aid me in my professional career as I move forward from Saint John's University. I would also like to thank Dr. Heroux for the opportunity to research reproducible software in my final year at CSB/SJU. It was a privilege to work with him and I am grateful for the skills I have acquired.

I would like to thank the computer science department at CSB/SJU. This includes John Miller, Dr. Imad Rahal, Dr. Jeremy Iverson, Dr. Noreen Herzfeld, Dr. Andrew Holey, Dr. Peter Ohmann, and Dr. Jim Schenpf. I am thankful for the support they have provided me throughout my time at CSB/SJU.

XII. Concluding Remarks

The problem of reproducibility in the computer science community can be broken down into two sub-problems. The first is a need for the technical tools necessary for replicating experiments. Without these tools, there is no guarantee that experiments can be trusted in the future. The second is a lack of incentives for creating replicable work.

Researchers face significant barriers to entry in learning new tools. They lack incentives to cultivate an understanding of tools outside of those needed to conduct their experiments. Without being easy to use and adapt to existing workflows, no reproducible solution will be successful [2]. To gain widespread adoption, reproducible solutions must make it easier for researchers to perform tasks [2]. Unfortunately, tools that provide more functionality have increased in complexity. To combat this, resources must exist to educate researchers about the tools available to them. Ultimately, Researchers have options, at times more than they have time to invest in. A central location for cataloging these tools and demonstrating how they can be applied to existing workflows seamlessly is crucial for the adoption of reproducibility in the computational science community.

References

[1] Docker glossary, May 2020.
[2] Carl Boettiger. An introduction to docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014.
[3] G. Fursin, A. Lokhmotov, and E. Plowman. Collective knowledge: Towards r d sustainability. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 864–869, 2016.
[4] Grigori Fursin. Collective knowledge framework (ck). 2017.
[5] Grigori Fursin. Ck features. 2018.
[6] Grigori Fursin. Demonstrating portable and customizable ck benchmarking workflows. 2019.
[7] Grigori Fursin. Portable workflows. 2019.
[8] Grigori Fursin, Herve Guillou, and Nicolas Essayan. Codereef: an open platform for portable mlops, reusable automation actions and reproducible benchmarking, 2020.
[9] Noah Watkins Carlos Maltzahn Jay Lofstead Kathryn Mohror Andrea Arpaci-Dusseau Remzi Apraci-Dusseau Ivo Jimenez, Michael Sevilla. The popper convention: Making reproducible systems evaluation practical. 2017.
[10] Geoffrey M. Poore. Reproducible documents with pythontex. 2013.
[11] Popper. Popper. *Popper Practical Falsifiable Research*.
[12] Michel Steuwer. Ck: Collective knowledge. 2017.
[13] R. Strijkers. Toward executable scientic publications. *Procedia Computer Science*, 4, 2011.