Design of a Scientific Data Analysis Support Platform

Nathan Martindale[‡*], Jason Hite[‡], Scott Stewart[‡], Mark Adams[‡]

Abstract—Software data analytic workflows are a critical aspect of modern scientific research and play a crucial role in testing scientific hypotheses. A typical scientific data analysis life cycle in a research project must include several steps that may not be fundamental to testing the hypothesis, but are essential for reproducibility. This includes tasks that have analogs to software engineering practices such as versioning code, sharing code among research team members, maintaining a structured codebase, and tracking associated resources such as software environments. Tasks unique to scientific research include designing, implementing, and modifying code that tests a hypothesis. This work refers to this code as *an experiment*, which is defined as a software analog to physical experiments.

A software experiment manager should support tracking and reproducing individual experiment runs, organizing and presenting results, and storing and reloading intermediate data on long-running computations. A software experiment manager with these features would reduce the time a researcher spends on tedious busywork and would enable more effective collaboration. This work discusses the necessary design features in more depth, some of the existing software packages that support this workflow, and a custom developed open-source solution to address these needs.

Index Terms—reproducible research, experiment life cycle, data analysis support

Introduction

Modern science increasingly uses software as a tool for conducting research and scientific data analyses. The growing number of libraries and frameworks facilitating this work has greatly lowered the barrier to usage, allowing more researchers to benefit from this paradigm. However, as a result of the dependence on software, there is a need for more thorough integration of sound software engineering practices with the scientific process. The fragility of complex environments containing heavily interconnected packages coupled with a lack of provenance of the artifacts generated throughout the development of an experiment increases the potential for long-term problems, undetected bugs, and failure to reproduce previous analyses.

* *Corresponding author: martindalena@ornl.gov*
‡ *Oak Ridge National Laboratory*

Fundamentally, science revolves around the ability for others to repeat and reproduce prior published works, and this has become a difficult task with many computation-based studies. Often, scientists outside of a computer science field may not have training in software engineering best practices, or they may simply disregard them because the focus of a researcher is on scientific publications rather than the analysis software itself. Lack of documentation and provenance of research artifacts and frequent failure to publish repositories for data and source code has led to a crisis in reproducibility in artificial intelligence (AI) and other fields that rely heavily on computation [SBB13], [DMR+09], [Hut18]. One study showed that quantifiably few machine learning (ML) papers document specifics in how they ran their experiments [GGA18]. This gap between established practices from the software engineering field and how computational research is conducted has been studied for some time, and the problems that can stem from it are discussed at length in [Sto18].

To mitigate these issues, computation-based research requires better infrastructure and tooling [Pen11] as well as applying relevant software engineering principles [Sto18], [Dub05] to allow data scientists to ensure their work is effective, correct, and reproducible. In this paper we focus on the ability to manage reproducible workflows for scientific experiments and data analyses. We discuss the features that software to support this might require, compare some of the existing tools that address them, and finally present the open-source tool Curifactory which incorporates the proposed design elements.

Related Work

Reproducibility of AI experiments has been separated into three different degrees [GK18]: *Experiment reproduciblity*, or repeatability, refers to using the same code implementation with the same data to obtain the same results. *Data reproducibility*, or replicability, is when a different implementation with the same data outputs the same results. Finally, *method reproducibility* describes when a different implementation with different data is able to achieve consistent results. These degrees are discussed in [GGA18], comparing the implications and trade-offs on the amount of work for the original researcher versus an external researcher, and the degree of generality afforded by a reproduced implementation. A repeatable experiment places the greatest burden on the original researcher, requiring the full codebase and experiment to be sufficiently documented and published so that a peer is able to correctly repeat it. At the other end of the spectrum, method reproducibility demands the greatest burden on the external researcher, as they must implement and run the experiment from scratch. For the remainder of this paper, we refer to "reproducibility" as experiment reproducibility (repeatability). Tooling that is able to assist with documentation and organization of a published experiment reduces the amount of work for the original researcher and still allows for the lowest level of burden to external researchers to verify and extend previous work.

In an effort to encourage better reproducibility based on datasets, the Findable, Accessible, Interoperable, and Reusable (FAIR) data principles [WDA+16] were established. These principles recommend that data should have unique and persistent identifiers, use common standards, and provide rich metadata description and provenance, allowing both humans and machines to effectively parse them. These principles have been extended more broadly to software [LGK+20], computational workflows [GCS+20], and to entire data pipelines [MLC+21].

Various works have surveyed software engineering practices and identified practices that provide value in scientific computing contexts, including various forms of unit and regression testing, proper source control usage, formal verification, bug tracking, and agile development methods [Sto18], [Dub05]. In particular, [Sto18] described many concepts from agile development as being well suited to an experimental context, where the current knowledge and goals may be fairly dynamic throughout the project. They noted that although many of these techniques could be directly applied, some required adaptation to make sense in the scientific software domain.

Similar to this paper, two other works [DGST09], [WWG21] discuss sets of design aspects and features that a workflow manager would need. Deelman et al. describe the life cycle of a workflow as composition, mapping, execution, and provenance capture [DGST09]. A workflow manager must then support each of these aspects. Composition is how the workflow is constructed, such as through a graphical interface or with a text configuration file. Mapping and execution are determining the resources to be used for a workflow and then utilizing those resources to run it, including distributing to cloud compute and external representational state transfer (REST) services. This also refers to scheduling subworkflows/tasks to reuse intermediate artifacts as available. Provenance, which is crucial for enabling repeatability, is how all artifacts, library versions, and other relevant metadata are tracked during the execution of a workflow.

Wratten, Wilm, and Göke surveyed many bioinformatics pipline and workflow management tools, listing the challenges that tooling should address: data provenance, portability, scalability, and re-entrancy [WWG21]. Provenance is defined the same way as in [DGST09], and further states the need for generating reports that include the tracking information and metadata for the associated experiment run. Portability—allowing set up and execution of an experiment in a different environment—can be a challenge because of the dependency requirements of a given system and the ease with which the environment can be specified and reinitialized on a different machine or operating system. Scalability is important especially when large scale data, many compute-heavy steps, or both are involved throughout the workflow. Scalability in a manager involves allowing execution on a high-performance computing (HPC) system or with some form of parallel compute. Finally they mention re-entrancy, or the ability to resume execution of a compute step from where it last stopped, preventing unnecessary recomputation of prior steps.

One area of the literature that needs further discussion is the design of automated provenance tracking systems. Existing workflow management tools generally require source code modifications to take full advantage of all features. This can entail a significant learning curve and places additional burden on the researcher. To address this, some sources propose automatic documentation of experiments and code through static source code analysis [NFP+20], [Red19].

Beyond the preexisting body of knowledge about software engineering principles, other works [SNTH13], [KHS09] describe recommended rules and practices to follow when conducting computation-based research. These include avoiding manual data manipulation in favor of scripted changes, keeping detailed records of how results are produced (manual provenance), tracking the versions of libraries and programs used, and tracking random seeds. Many of these ideas can be assisted or encapsulated through appropriate infrastructure decisions, which is the premise on which this work bases its software reviews.

Although this paper focuses on the scientific workflow, a growing related field tackles many of the same issues from an industry standpoint: machine learning operations (MLOps) [Goy20]. MLOps, an ML-oriented version of DevOps, is concerned with supporting an entire data science life cycle, from data acquisition to deployment of a production model. Many of the same challenges are present, reproducibility and provenance are crucial in both production and research workflows [RMRO21]. Infrastructure, tools, and practices developed for MLOps may also hold value in the scientific community.

A taxonomy for ML tools that we reference throughout this work is from [QCL21], which describes a characterization of tools consisting of three primary categories: general, analysis support, and reproducibility support, each of which is further subdivided into aspects to describe a tool. For example, these subaspects include data visualization, web dashboard capabilities, experiment logging, and the interaction modes the tool supports, such as a command line interface (CLI) or application programming interface (API).

Design Features

We combine the two sets of capabilities from [DGST09] and [WWG21] with the taxonomy from [QCL21] to propose a set of six design features that are important for an experiment manager. These include orchestration, parameterization, caching, reproducibility, reporting, and scalability. The crossover between these proposed feature sets are shown in Table 1. We expand on each of these in more depth in the subsections below.

Orchestration

Orchestration of an experiment refers to the mechanisms used to chain and compose a sequence of smaller logical steps into an overarching pipeline. This provides a higher-level view of an experiment and helps abstract away some of the implementation details. Operation of most workflow managers is based on a directed acyclic graph (DAG), which specifies the stages/steps as nodes and the edges connecting them as their respective inputs and outputs. The intent with orchestration is to encourage designing distinct, reusable steps that can easily be composed in different ways to support testing different hypotheses or overarching experiment runs. This allows greater focus on the design of the experiments than the implementation of the underlying functions that the experiments consist of. As discussed in the taxonomy [QCL21], pipeline creation can consist of a combination of scripts, configuration files, or a visual tool. This aspect falls within the composition capability discussed in [DGST09].

This work	**[DGST09]**	**[WWG21]**	**Taxonomy [QCL21]**
Orchestration	Composition	—	Reproducibility/pipeline creation
Parameterization	—	—	—
Caching	—	Re-entrancy	—
Reproducibility	Provenance	Provenance, portability	Reproducibility
Reporting	—	—	Analysis/visualization, web dashboard
Scalability	Mapping, execution	Scalability	Analysis/computational resources

TABLE 1: *Comparing design features listed in various works.*

Parameterization

Parameterization specifies how a compute pipeline is customized for a particular run by passing in configuration values to change aspects of the experiment. The ability to customize analysis code is crucial to conducting a compute-based experiment, providing a mechanism to manipulate a variable under test to verify or reject a hypothesis.

Conventionally, parameterization is done either through specifying parameters in a CLI call or by passing configuration files in a format like JSON or YAML. As discussed in [DGST09], parameterization sometimes consists of more complicated needs, such as conducting parameter sweeps or grid searches. There are libraries dedicated to managing parameter searches like this, such as hyperopt [BYC13] used in [RMRO21].

Although not provided as a design capability in the other works, we claim the mechanisms provided for parameterization are important, as these mechanisms are the primary way to configure, modify, and vary experiment execution without explicitly changing the code itself or modifying hard-coded values. This means that a recorded parameter set can better "describe" an experiment run, increasing provenance and making it easier for another researcher to understand what pieces of an experiment can be readily changed and explored.

Some support is provided for this in [DGST09], stating that the necessity of running many slight variations on workflows sometimes leads to the creation of ad hoc scripts to generate the variants, which leads to increased complexity in the organization of the codebase. Improved mechanisms to parameterize the same workflow for many variants helps to manage this complexity.

Caching

Refining experiment code and finding bugs is often a lengthy iterative process, and removing the friction of constantly rerunning all intermediate steps every time an experiment is wrong can improve efficiency. Caching values between each step of an experiment allows execution to resume at a certain spot in the pipeline, rather than starting from scratch every time. This is defined as *re-entrancy* in [WWG21].

In addition to increasing the speed of rerunning experiments and running new experiments that combine old results for analysis, caching is useful to help find and debug mistakes throughout an experiment. Cached outputs from each step allow manual interrogation outside of the experiment. For example, if a cleaning step was implemented incorrectly and a user noticed an invalid value in an output data table, they could use a notebook to load and manipulate the intermediate artifact tables for that data to determine what stage introduced the error and what code should be used to correctly fix it.

Reproducibility

Mechanisms for reproducibility are one of the most important features for a successful data analysis support platform. Reproducibility is challenging because of the complexity of constantly evolving codebases, complicated and changing dependency graphs, and inconsistent hardware and environments. Reproducibility entails two subcomponents: provenance and portability. This falls under the provenance aspect from [DGST09], both data provenance and portability from [WWG21], and the entire reproducibility support section of the taxonomy [QCL21].

Data provenance is about tracking the history, configuration, and steps taken to produce an intermediate or final data artifact. In ML this would include the cleaning/munging steps used and the intermediate tables created in the process, but provenance can apply more broadly to any type of artifact an experiment may produce, such as ML models themselves, or "model provenance" [SH18]. Applying provenance beyond just data is critical, as models may be sensitive to the specific sets of training data and conditions used to produce them [Hut18]. This means that everything required to directly and exactly reproduce a given artifact is recorded, such as the manipulations applied to its predecessors and all hyperparameters used within those manipulations.

Portability refers to the ability to take an experiment and execute it outside of the initial computing environment it was created in [WWG21]. This can be a challenge if all software dependency versions are not strictly defined, or when some dependencies may not be available in all environments. Minimally, allowing portability requires keeping explicit track of all packages and the versions used. A 2017 study [OBA17] found that even this minimal step is rarely taken. Another mechanism to support portability is the use of containerization, such as with Docker or Podman [SH18].

Reporting

Reporting is an important step for analyzing the results of an experiment, through visualizations, summaries, comparisons of results, or combinations thereof. As a design capability, *reporting* refers to the mechanisms available for the system to export or retrieve these results for human analysis. Although data visualization and analysis can be done manually by the scientist, tools to assist with making these steps easier and to keep results organized are valuable from a project management standpoint. Mechanisms for this might include a web interface for exploring individual or multiple runs. Under the taxonomy [QCL21], this falls primarily within analysis support, such as data visualization or a web dashboard.

Scalability

Many data analytic problems require large amounts of space and compute resources, often beyond what can be handled on an individual machine. To efficiently support running a large experiment, mechanisms for scaling execution are important and could include anything from supporting parallel computation on an experiment or stage level, to allowing the execution of jobs on remote machines or within an HPC context. This falls within both mapping and execution from [DGST09], the scalability aspect from [WWG21], and the computational resources category within the taxonomy [QCL21].

Existing Tools

A wide range of pipeline and workflow tools have been developed to support many of these design features, and some of the more common examples include DVC [KPP+22] and MLFlow [MLf22]. We briefly survey and analyze a small sample of these tools to demonstrate the diversity of ideas and their applicability in different situations. Table 2 compares the support of each design feature by each tool.

DVC

DVC [KPP+22] is a Git-like version control tool for datasets. Orchestration is done by specifying *stages*, or runnable script commands, either in YAML or directly on the CLI. A stage is specified with output file paths and input file paths as dependencies, allowing an implicit pipeline or DAG to form, representing all the processing steps. Parameterization is done by defining within a YAML file what the possible parameters are, along with the default values. When running the DAG, parameters can be customized on the CLI. Since inputs and outputs are file paths, caching and re-entrancy come for free, and DVC will intelligently determine if certain stages do not need to be re-computed.

A saved experiment or state is frozen into each commit, so all parameters and artifacts are available at any point. No explicit tracking of the environment (e.g., software versions and hardware info) is present, but this could be manually included by tracking it in a separate file. Reporting can be done by specifying per-stage metrics to track in the YAML configuration. The CLI includes a way to generate HTML files on the fly to render requested plots. There is also an external "Iterative Studio" project, which provides a live web dashboard to view continually updating HTML reports from DVC. For scalability, parallel runs can be achieved by queuing an experiment multiple times in the CLI.

MLFlow

MLFlow [MLf22] is a framework for managing the entire life cycle of an ML project, with an emphasis on scalability and deployment. It has no specific mechanisms for orchestration, instead allowing the user to intersperse MLFlow API calls in an existing codebase. Runnable scripts can be provided as entry points into a configuration YAML, along with the parameters that can be provided to it. Parameters are changed through the CLI. Although MLFlow has extensive capabilities for tracking artifacts, there are no automatic re-entrancy methods. Reproducibility is a strong feature, and provenance and portability are well supported. The tracking module provides provenance by recording metadata such as the Git commit, parameters, metrics, and any user-specified artifacts in the code. Portability is done by allowing the environment for an entry point to be specified as a Conda environment or Docker container. MLFlow then ensures that the environment is set up and active before running. The CLI even allows directly specifying a GitHub link to an mlflow-enabled project to download, set up, and then run the associated experiment. For reporting, the MLFlow tracking UI lets the user view and compare various runs and their associated artifacts through a web dashboard. For scalability, both distributed storage for saving/loading artifacts as well as execution of runs on distributed clusters is supported.

Sacred

Sacred [GKC+17] is a Python library and CLI tool to help organize and reproduce experiments. Orchestration is managed through the use of Python decorators, a "main" for experiment entry point functions and "capture" for parameterizable functions, where function arguments are automatically populated from the active configuration when called. Parameterization is done directly in Python through applying a config decorator to a function that assigns variables. Configurations can also be written to or read from JSON and YAML files, so parameters must be simple types. Different observers can be specified to automatically track much of the metadata, environment information, and current parameters, and within the code the user can specify additional artifacts and resources to track during the run. Each run will store the requested outputs, although there is no re-entrant use of these cached values. Portability is supported through the ability to print the versions of libraries needed to run a particular experiment. Reporting can be done through a specific type of observer, and the user can provide custom templated reports that are generated at the end of each run.

Kedro

Kedro [ABC+22] is another Python library/CLI tool for managing reproducible and modular experiments. Orchestration is particularly well done with "node" and "pipeline" abstractions, a node referring to a single compute step with defined inputs and outputs, and a pipeline implemented as an ordered list of nodes. Pipelines can be composed and joined to create an overarching workflow. Possible parameters are defined in a YAML file and either set in other parameter files or configured on the CLI. Similar to MLFlow, while tracking outputs are cached, there's no automatic mechanism for re-entrancy. Provenance is achieved by storing user-specified metrics and tracked datasets for each run, and it has a few different mechanisms for portability. This includes the ability to export an entire project into a Docker container. A separate Kedro-Viz tool provides a web dashboard to show a map of experiments, as well as showing each tracked experiment run and allowing comparison of metrics and outputs between them. Projects can be deployed into several different cloud providers, such as Databricks and Dask clusters, allowing for several options for scalability.

Curifactory

Curifactory [MHSA22] is a Python API and CLI tool for organizing, tracking, reproducing, and exporting computational research experiments and data analysis workflows. It is intended primarily for smaller teams conducting research, rather than production-level or large-scale ML projects. Curifactory is available on GitHub[1] with an open-source BSD-3-Clause license. Below, we describe the mechanisms within Curifactory to support each of the six capabilities, and compare it with the tools discussed above.

1. https://github.com/ORNL/curifactory

	Orchestration	Parameterization	Caching	Provenance	Portability	Reporting	Scalability
DVC	+	+	++	+	+	+	+
MLFlow		+	*	++	++	++	++
Sacred	+	++	*	++	+	+	
Kedro	+	+	*	+	++	++	++
Curifactory	+	++	++	++	++	+	+

TABLE 2: *Supported design features in each tool. Note, + indicates that a feature is supported, ++ indicates very strong support, and * indicates tooling that supports caching artifacts as a provenance tool but does not provide a mechanism for automatically reloading cached values as a form of re-entrancy.*

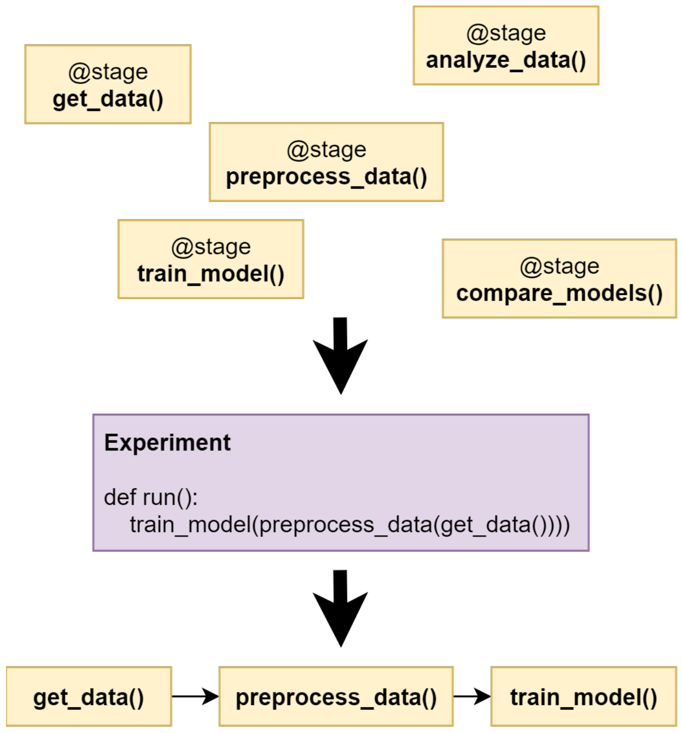


Fig. 1: *Stages are composed into an experiment.*

Orchestration

Curifactory provides several abstractions, the lowest level of which is a *stage*. A stage is a function that takes a defined set of input variable names, a defined set of output variable names, and an optional set of caching strategies for the outputs. Stages are similar to Kedro's nodes but implemented with `@stage()` decorators on the target function rather than passing the target function to a `node()` call. One level up from a stage is an *experiment*: an experiment describes the orchestration of these stages as shown in Figure 1, functionally chaining them together without needing to explicitly manage what variables are passed between the stages.

```python
@stage(inputs=None, outputs=["data"])
def load_data(record):
    # every stage has the currently active record
    # passed to it, which contains the "state", or
    # all previous output values associated with
    # the current argset, as defined in the
    # Parameterization section
    # ...

@stage(inputs=["data"], outputs=["model", "stats"])
def train_model(record, data):
    # ...
```

```python
@stage(inputs=["model"], outputs=["results"])
def test_model(record, model):
    # ...

def run(argsets, manager):
    """An example experiment definition.

    The primary intent of an experiment is to run
    each set of arguments through the desired
    stages, in order to compare results at the end.
    """
    for argset in argsets:
        # A record is the "pipeline state"
        # associated with each set of arguments.
        # Stages take and return a record,
        # automatically handling pushing and
        # pulling inputs and outputs from the
        # record state.
        record = Record(manager, argsets)
        test_model(train_model(load_data(record)))
```

Parameterization

Parameterization in Curifactory is done directly in Python scripts. The user defines a dataclass with the parameters they need throughout their various stages in order to customize the experiment, and they can then define parameter files that each return one or more instances of this arguments class. All stages in an experiment are automatically given access to the current argument set in use while an experiment is running.

While configuration can also be done directly in Python in Sacred, Curifactory makes a different trade-off: A parameter file or `get_params()` function in Curifactory returns an array of one or more argument sets, and arguments can directly include complex Python objects. Unlike Sacred, this means Curifactory cannot directly translate back and forth from static configuration files, but in exchange allows for grid searches to be defined directly and easily in a single parameter file, as well as allowing argument sets to be composed or even inherit from other argument set instances. Importantly, Curifactory can still encode representations of arguments into JSON for provenance, but this is a one directional transformation.

This approach allows a great deal of flexibility, and is valuable in experiments where a large range of parameters need to be tested or there is significant repetition among parameter sets. For example, in an experiment testing different effects of model training hyperparameters, there may be several parameter files meant to vary only the arguments needed for model training while using the same base set of data cleaning arguments. Composing these parameter sets from a common imported set means that any subsequent changes to the data cleaning arguments only need to be modified in one place, rather than each individual parameter file.

```python
@dataclass
class MyArgs(curifactory.ExperimentArgs):
    """Define the possible arguments needed in the
    stages."""
    random_seed: int = 42
    train_test_ratio: float = 0.8
    layers: tuple = (100,)
    activation: str = "relu"

def get_params():
    """Define a simple grid search: return
    many arguments instances for testing."""
    args = []
    layer_sizes = [10, 20, 50, 100]
    for size in layer_sizes:
        args.append(MyArgs(name=f"network_{size}",
            layers=(size,)))
    return args
```

Caching

Curifactory supports per-stage caching, similar to memoization, through a set of easy-to-use caching strategies. When a stage executes, it uses the specified cache mechanism to store the stage outputs to disk, with a filename based on the experiment, stage, and a hash of the arguments. When the experiment is re-executed, if it finds an existing output on disk based on this name, it short-circuits the stage computation and simply reloads the previously cached files, allowing a form of re-entrancy. Adding this caching ability to a stage is done through simply providing the list of caching strategies to the stage decorator, one for each output:

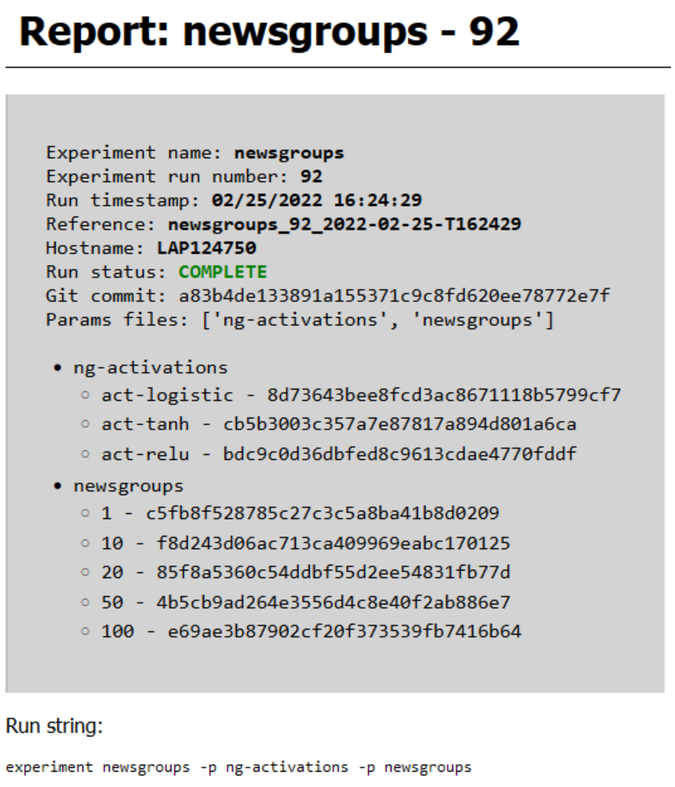
```python
@stage(
    inputs=["data"],
    outputs=["training_set", "testing_set"],
    cachers=[PandasCSVCacher]*2
):
def split_data(record, data):
    # stage definition
```

Reproducibility

As mentioned before, reproducibility consists of tracking provenance and metadata of artifacts as well as providing a means to set up and repeat an experiment in a different compute environment. To handle provenance, Curifactory automatically records metadata for every experiment run executed, including a logfile of the console output, current Git commit hash, argument sets used and the rendered versions of those arguments, and the CLI command used to start the run. The final reports from each run also include a graphical representation of the stage DAG, and shows each output artifact and what its cache file location is.

Curifactory has two mechanisms to fully track and export an experiment run. The first is to execute a "full store" run, which creates a single exported folder containing all metadata mentioned above, along with a copy of every cache file created, the output run report (mentioned below), as well as a Python requirements.txt and Conda environment dump, containing a list of all packages in the environment and their respective versions. This run folder can then be distributed. Reproducing from the folder consists of setting up an environment based on the Conda/Python dependencies as needed, and running the experiment command using the exported folder as the cache directory.

The second mechanism is a command to create a Docker container that includes the environment, entire codebase, and artifact cache for a specific experiment run. Curifactory comes with a default Dockerfile for this purpose, and running the experiment with the Docker flag creates an image that exposes a Jupyter notebook to repeat the run and keep the artifacts in memory, as well as a file server pointing to the appropriate cache for manual exploration and inspection. Directly reproducing the experiment can be done either through the exposed notebook or by running the Curifactory experiment command inside of the image.



Report: newsgroups - 92

```
Experiment name: newsgroups
Experiment run number: 92
Run timestamp: 02/25/2022 16:24:29
Reference: newsgroups_92_2022-02-25-T162429
Hostname: LAP124750
Run status: COMPLETE
Git commit: a83b4de133891a155371c9c8fd620ee78772e7f
Params files: ['ng-activations', 'newsgroups']

 • ng-activations
    ◦ act-logistic - 8d73643bee8fcd3ac8671118b5799cf7
    ◦ act-tanh - cb5b3003c357a7e87817a894d801a6ca
    ◦ act-relu - bdc9c0d36dbfed8c9613cdae4770fddf
 • newsgroups
    ◦ 1 - c5fb8f528785c27c3c5a8ba41b8d0209
    ◦ 10 - f8d243d06ac713ca409969eabc170125
    ◦ 20 - 85f8a5360c54ddbf55d2ee54831fb77d
    ◦ 50 - 4b5cb9ad264e3556d4c8e40f2ab886e7
    ◦ 100 - e69ae3b87902cf20f373539fb7416b64
```

Run string:

```
experiment newsgroups -p ng-activations -p newsgroups
```

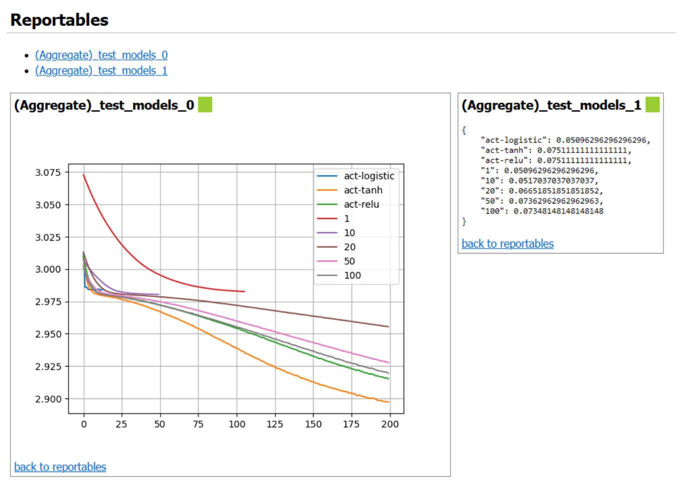
Fig. 2: *Metadata block at the top of a report.*

Reporting

While Curifactory does not run a live web dashboard like MLFlow, DVC's Iterative Studio, and Kedro-viz, every experiment run outputs an HTML experiment report and updates a top-level index HTML page linking to the new report, which can be browsed from a file manager or statically served if running from an external compute resource. Although simplistic, this reduces the dependencies and infrastructure needed to achieve a basic level of reporting, and produces stand-alone folders for consumption outside of the original environment if needed.

Every report from Curifactory includes all relevant metadata mentioned above, including the machine host name, experiment sequential run number, Git commit hash, parameters, and command line string. Stage code can add user-defined objects to output in each report, such as tables, figures, and so on. Curifactory comes with a default set of helpers for several basic types of output visualizations, including basic line plots, entire Matplotlib figures, and dataframes.

The output report also contains a graphical representation of the DAG for the experiment, rendered using Graphviz, and shows the artifacts produced by each stage and the file path where they are cached. An example of some of the components of this report are rendered in figures 2, 3, 4, and 5.



```
{
    "act-logistic": 0.05096296296296296,
    "act-tanh": 0.07511111111111111,
    "act-relu": 0.07511111111111111,
    "1": 0.05096296296296296,
    "10": 0.05170370370370370 37,
    "20": 0.06651851851851852,
    "50": 0.07362962962962963,
    "100": 0.07348148148148148
}
```

Fig. 3: User-defined objects to report ("reportables").

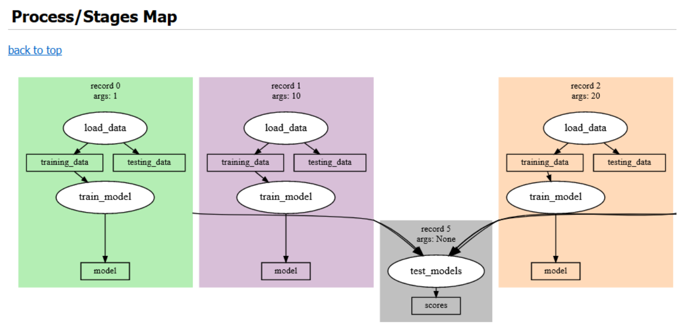


Fig. 4: Graphviz rendering of experiment DAG. Each large colored area represents a single record associated with a specific argset. White ellipses are stages, and the blocks in between them are the input and output artifacts.

Scalability

Curifactory has no integrated method of executing portions of jobs on external compute resources like Kedro and MLFlow, but it does allow local multi-process parallelization of parameter sets. When an experiment run would entail executing a series of stages for each argument set in series, Curifactory can divide the collection of argument sets into one subcollection per process, and runs the experiment in parallel on each subcollection. By taking advantage of the caching mechanism, when all parallel runs complete, the experiment reruns in a single process to aggregate all of the precached values into a single report.

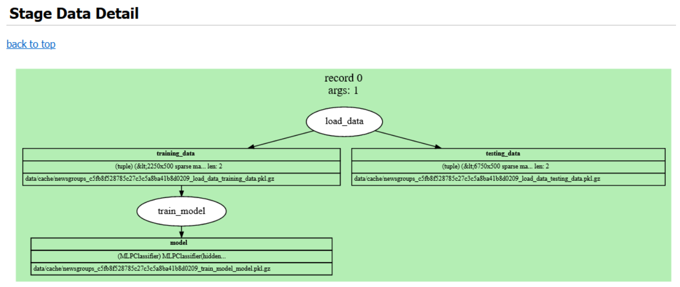

Stage Data Detail
back to top
record 0
args: 1
load_data
training_data
(tuple) (<2250x500 sparse ma... len: 2
data/cache/newsgroups_c5fb8f528785c27c3c5a8ba41b8d0209_load_data_training_data.pkl.gz
testing_data
(tuple) (<6750x500 sparse ma... len: 2
data/cache/newsgroups_c5fb8f528785c27c3c5a8ba41b8d0209_load_data_testing_data.pkl.gz
train_model
model
(MLPClassifier) MLPClassifier(hidden...
data/cache/newsgroups_c5fb8f528785c27c3c5a8ba41b8d0209_train_model_model.pkl.gz

Fig. 5: Graphviz rendering of each record in more depth, showing cache file paths and artifact data types.

Conclusion

The complexity in modern software, environments, and data analytic approaches threaten the reproducibility and effectiveness of computation-based studies. This has been compounded by the lack of standardization in infrastructure tools and software engineering principles applied within scientific research domains. While many novel tools and systems are in development to address these shortcomings, several design critieria must be met, including the ability to easily compose and orchestrate experiments, parameterize them to manipulate variables under test, cache intermediate artifacts, record provenance of all artifacts and allow the software to port to other systems, produce output visualizations and reports for analysis, and scale execution to the resource requirements of the experiment. We developed Curifactory to address these criteria specifically for small research teams running Python based experiments.

Acknowledgements

The authors would like to acknowledge the US Department of Energy, National Nuclear Security Administration's Office of Defense Nuclear Nonproliferation Research and Development (NA-22) for supporting this work.

References

[ABC+22] Sajid Alam, Lorena Bălan, Gabriel Comym, Yetunde Dada, Ivan Danov, Lim Hoang, Rashida Kanchwala, Jiri Klein, Antony Milne, Joel Schwarzmann, Merel Theisen, and Susanna Wong. Kedro. https://kedro.org/, March 2022.

[BYC13] James Bergstra, Daniel Yamins, and David Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on Machine Learning*, pages 115–123. PMLR, February 2013.

[DGST09] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25:524–540, May 2009. doi:10.1016/j.future.2008.06.012.

[DMR+09] David L. Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. Reproducible Research in Computational Harmonic Analysis. *Computing in Science Engineering*, 11(1):8–18, January 2009. doi:10.1109/MCSE.2009.15.

[Dub05] P.F. Dubois. Maintaining correctness in scientific programs. *Computing in Science Engineering*, 7(3):80–85, May 2005. doi:10.1109/MCSE.2005.54.

[GCS+20] Carole Goble, Sarah Cohen-Boulakia, Stian Soiland-Reyes, Daniel Garijo, Yolanda Gil, Michael R. Crusoe, Kristian Peters, and Daniel Schober. FAIR Computational Workflows. *Data Intelligence*, 2(1-2):108–121, January 2020. doi:10.1162/dint_a_00033.

[GGA18] Odd Erik Gundersen, Yolanda Gil, and David W. Aha. On Reproducible AI: Towards Reproducible Research, Open Science, and Digital Scholarship in AI Publications. *AI Magazine*, 39(3):56–68, September 2018. doi:10.1609/aimag.v39i3.2816.

[GK18] Odd Erik Gundersen and Sigbjørn Kjensmo. State of the Art: Reproducibility in Artificial Intelligence. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018. doi:10.1609/aaai.v32i1.11503.

[GKC+17] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In *Proceedings of the 16th Python in Science Conference*, pages 49–56, Austin, Texas, 2017. SciPy. doi:10.25080/shinma-7f4c6e7-008.

[Goy20] A. Goyal. Machine learning operations, 2020.

[Hut18] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, February 2018. doi:10.1126/science.359.6377.725.

[KHS09] Diane Kelly, Daniel Hook, and Rebecca Sanders. Five Recommended Practices for Computational Scientists Who Write Software. *Computing in Science Engineering*, 11(5):48–53, September 2009. doi:10.1109/MCSE.2009.139.

[KPP+22] Ruslan Kuprieiev, Saugat Pachhai, Dmitry Petrov, Paweł Redzyński, Casper da Costa-Luis, Peter Rowlands, Alexander Schepanovski, Ivan Shcheklein, Batuhan Taskaya, Jorge Orpinel, Gao, Fábio Santos, David de la Iglesia Castro, Aman Sharma, Zhanibek, Dani Hodovic, Nikita Kodenko, Andrew Grigorev, Earl, Nabanita Dash, George Vyshnya, maykulkarni, Max Hora, Vera, Sanidhya Mangal, Wojciech Baranowski, Clemens Wolff, and Kurian Benoy. DVC: Data Version Control - Git for Data & Models. Zenodo, April 2022. doi:10.5281/zenodo.6417224.

[LGK+20] Anna-Lena Lamprecht, Leyla Garcia, Mateusz Kuzak, Carlos Martinez, Ricardo Arcila, Eva Martin Del Pico, Victoria Dominguez Del Angel, Stephanie van de Sandt, Jon Ison, Paula Andrea Martinez, Peter McQuilton, Alfonso Valencia, Jennifer Harrow, Fotis Psomopoulos, Josep Ll Gelpi, Neil Chue Hong, Carole Goble, and Salvador Capella-Gutierrez. Towards FAIR principles for research software. *Data Science*, 3(1):37–59, January 2020. doi:10.3233/DS-190026.

[MHSA22] Nathan Martindale, Jason Hite, Scott L. Stewart, and Mark Adams. Curifactory. https://github.com/ORNL/curifactory, March 2022.

[MLC+21] Sonia Natalie Mitchell, Andrew Lahiff, Nathan Cummings, Jonathan Hollocombe, Bram Boskamp, Dennis Reddyhoff, Ryan Field, Kristian Zarebski, Antony Wilson, Martin Burke, Blair Archibald, Paul Bessell, Richard Blackwell, Lisa A. Boden, Alys Brett, Sam Brett, Ruth Dundas, Jessica Enright, Alejandra N. Gonzalez-Beltran, Claire Harris, Ian Hinder, Christopher David Hughes, Martin Knight, Vino Mano, Ciaran McMonagle, Dominic Mellor, Sibylle Mohr, Glenn Marion, Louise Matthews, Iain J. McKendrick, Christopher Mark Pooley, Thibaud Porphyre, Aaron Reeves, Edward Townsend, Robert Turner, Jeremy Walton, and Richard Reeve. FAIR Data Pipeline: Provenance-driven data management for traceable scientific workflows. *arXiv:2110.07117 [cs, q-bio]*, October 2021. arXiv:2110.07117.

[MLf22] MLflow: A Machine Learning Lifecycle Platform. https://mlflow.org/, April 2022.

[NFP+20] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 1542–1551, New York, NY, USA, August 2020. Association for Computing Machinery. doi:10.1145/3394486.3403205.

[OBA17] Babatunde K. Olorisade, Pearl Brereton, and Peter Andras. Reproducibility in Machine Learning-Based Studies: An Example of Text Mining. In *Reproducibility in ML Workshop, 34th International Conference on Machine Learning*, ICML 2017, June 2017.

[Pen11] Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, December 2011. doi:10.1126/science.1213847.

[QCL21] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. A Taxonomy of Tools for Reproducible Machine Learning Experiments. In *AIxIA 2021 Discussion Papers, 20th International Conference of the Italian Association for Artificial Intelligence*, pages 65–76, 2021.

[Red19] Sergey Redyuk. Automated Documentation of End-to-End Experiments in Data Science. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 2076–2080, April 2019. doi:10.1109/ICDE.2019.00243.

[RMRO21] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools. *Applied Sciences*, 11(19):8861, January 2021. doi:10.3390/app11198861.

[SBB13] Victoria Stodden, Jonathan Borwein, and David H. Bailey. Publishing Standards for Computational Science: "Setting the Default to Reproducible". Pennsylvania State University, 2013.

[SH18] Peter Sugimura and Florian Hartl. Building a Reproducible Machine Learning Pipeline. *arXiv:1810.04570 [cs, stat]*, October 2018. arXiv:1810.04570.

[SNTH13] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology*, 9(10):e1003285, October 2013. doi:10.1371/journal.pcbi.1003285.

[Sto18] Tim Storer. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Computing Surveys*, 50(4):1–32, July 2018. doi:10.1145/3084225.

[WDA+16] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C. 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3(1):160018, March 2016. doi:10.1038/sdata.2016.18.

[WWG21] Laura Wratten, Andreas Wilm, and Jonathan Göke. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature Methods*, 18(10):1161–1168, October 2021. doi:10.1038/s41592-021-01254-9.