A Framework to capture and reproduce the Absolute State of Jupyter Notebooks

Dimuthu Wannipurage
dwannipu@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

Suresh Marru
smarru@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

Marlon Pierce
marpierc@iu.edu
Cyberinfrastructure Integration Research Center, Pervasive Technology Institute, Indiana University
Bloomington, IN, USA

ABSTRACT

Jupyter Notebooks are an enormously popular tool for creating and narrating computational research projects. They also have enormous potential for creating reproducible scientific research artifacts. Capturing the complete state of a notebook has additional benefits; for instance, the notebook execution may be split between local and remote resources, where the latter may have more powerful processing capabilities or store large or access-limited data. There are several challenges for making notebooks fully reproducible when examined in detail. The notebook code must be replicated entirely, and the underlying Python runtime environments must be identical. More subtle problems arise in replicating referenced data, external library dependencies, and runtime variable states. This paper presents solutions to these problems using Juptyer's standard extension mechanisms to create an archivable system state for a running notebook. We show that the overhead for these additional mechanisms, which involve interacting with the underlying Linux kernel, does not introduce substantial execution time overheads, demonstrating the approach's feasibility.

CCS CONCEPTS

• **Software and its engineering** → **Software design engineering**; **Open source model**; • **Applied computing** → **Enterprise data management**; • **Information systems** → **Data warehouses**.

KEYWORDS

Jupyter Notebooks, Apache Airavata, Reproducible Science

ACM Reference Format:
Dimuthu Wannipurage, Suresh Marru, and Marlon Pierce. 2022. A Framework to capture and reproduce the Absolute State of Jupyter Notebooks. In *Practice and Experience in Advanced Research Computing (PEARC '22), July 10–14, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3491418.3530296

PEARC '22, July 10–14, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9161-0/22/07…$15.00
https://doi.org/10.1145/3491418.3530296

1 INTRODUCTION

Project Jupyter's notebook format [14] has revolutionized interactive computing and has become ubiquitous among researchers using machine learning techniques and in scientific computing communities. Jupyter's simple-to-use user interfaces, ease of deployment, rich visualization support, and support for multiple programming languages have nurtured a large user community with a diverse set of use cases, from simple Python code execution to complicated neural network simulations in high-performance computing environments. When code execution and visualization are combined with embedded textual descriptions, the Jupyter ecosystem for computing and data analysis can be viewed as an infrastructure for providing narration for the computing and data life cycle as stories [7, 19]. This storytelling is essential for reproducible science [17] or, even better or re-creatable, or tweakable science [2].

The high user adoption of Jupyter has resulted in a large ecosystem supporting multiple programming languages through kernel extensions and integration into users' development environments. Commercial cloud platform integrations such as Google Colaboratory [6] and a GitHub-based notebook repository MyBinder have fostered user sharing and collaboration of notebooks. Community support for popular data platforms has also enabled notebook reuse [13].

Reproducibility is an essential aspect in any science discipline and sharing someone's work through a notebook may be meaningful only if others can reproduce their work. While there has been a significant amount of work done in making Jupyter Notebooks configurable and accessible to resonate with the user's requirements, there has been significantly less work to make notebooks reproducible across various platforms. We have identified several areas where reproducing a notebook execution is critical based on these considerations.

2 USE CASES FOR PORTABLE AND REPRODUCIBLE NOTEBOOKS

2.1 Reproducing Code from Scientific Publications

Software is a critical component of science, and reproducibility [10, 17, 18], and scholarly articles are increasingly referencing Jupyter Notebooks with code segments and visualizations. However, it is very cumbersome if not impossible to reproduce the published bundle of notebooks and associated code and data. Few challenges include recreating implicit data and identical runtime environments,

including a cascade of runtime library dependencies. Notebook kernels such as IPython rely on local Python virtual environments. Notebooks also interact with accessible external computational and data resources that are not easily translatable to shared collaborators. Out-of-band communication with authors of notebooks is necessary to fully reproduce the combination of data, dependencies, and runtime environment of the published work. Conducting these steps is a non-trivial and time-consuming process.

2.2 Sharing Classroom Learning Material

Jupyter Notebooks are a popular mechanism for providing classroom learning materials. Examples such as [3] allow instructors to share notebooks, and students can execute them on their laptops during a classroom session. There are several cases when this process becomes significantly complicated. Students' laptops will have various operating system types and versions. Some may have different Python runtime versions, which may be incompatible with the instructor's notebook version. In some cases, instructors might need to use external data files as inputs for the code fragments in the notebook, and these also need to be shipped with the notebook code.

2.3 Transferring Local Computations to Between Local and High-Performance Computing Environments

Researchers have access to a scale of computing power from laptops to high-end supercomputers and computational clouds. With the profound usage of Jupyter, it is essential to support notebook portability across these resources without fragmenting data and code too much. As an illustrative use case, a researcher might start coding a notebook on a workstation, but training a higher-order neural network may necessitate a computer with a powerful GPU, large memory, and access to large data sets. Potentially, post-processing analysis may be bought back to a local laptop. In other cases, portions of the notebook may need to be run on a computer containing data that cannot be easily transferred to another place due to higher data volume or low network bandwidth. The data may need to remain under specific security controls in some cases.

These mix and match executions within a single Jupyter Notebook are not supported in the standard distribution. Some work has been done in this area [4] to customize the vanilla iPython kernel and export the code execution to remotely hosted iPython kernel environments. However, these approaches require from the user a significant amount of effort and expertise in distributed systems. In addition, these approaches focus on keeping the Python runtime context synced between external kernel environments but do not address any data-level dependency with the code fragments in the notebook. For example, when developing a neural network in a notebook, we use the training and testing datasets available in the local disk. Supposing others need to replicate the same environment, relevant input files must be manually selected. Then the notebook code must be updated with relative file paths and distributed with the notebook.

2.4 Running on On-Demand or Time-Bound Computing Infrastructure

Preemptable infrastructures like Amazon Spot Instances [1] or HPC shared resources with bounded maximum wall time do not guarantee the complete end-to-end execution of a long-running notebook. Users may also need to stop the notebook execution and create a snapshot to resume it from the last checkpoint at another time by reloading all the data and local context of the Jupyter session. In such scenarios, having a feature to create a comprehensive snapshot of the runtime saves time and computing resource utilization. When Colab Notebooks [6] expire after the threshold, the platform kills the notebook, and all the local states are lost unless the user manually serializes and uploads it to Google Drive. It would be beneficial to run the notebook from the last stopped checkpoint even after the original session has expired in all of these cases.

3 REPRODUCIBILITY OF NOTEBOOKS

Jupyter Notebooks inherently are sharable, but several missing components hinder the process of reproducing results in a different environment. Restarting a running notebook in a different environment can be made possible by five essential requirements:

(1) The code within the notebook is portable.
(2) Python runtimes should be identical.
(3) Replicate referenced local data.
(4) External library dependencies should be replicated or installed if missing.
(5) Python runtime variables should be re-initialized.

Requirements 1 and 2 are external states which do not depend on the execution of the notebook. Requirements 3, 4, and 5 are internal to the Python runtime and the order of cell execution of the notebook (Figure 1). If any of the above five requirements are not met, the replicated environment is not identical to the source environment. In most sharing solutions [23, 24] we can see that only requirements 1 and 2 are satisfied but not requirements 3, 4, and 5.

In this paper, we propose and implement a framework for Jupyter Notebooks to satisfy all five requirements when replicating the state of a running notebook. In addition to the functional requirements, our solution is designed to prioritize the ease of use for users so that they do not have to put any more effort into making this work than running a vanilla notebook environment.

4 A NOVEL APPROACH TO CAPTURE THE ABSOLUTE STATE OF NOTEBOOKS

To achieve the absolute state capture of a notebook, we primarily utilized the Jupyter Magic Extension support provided by the IPython kernel. In addition, we made modifications to the iPython kernel to integrate with external services to capture information that cannot be directly captured from the kernel itself. We did not focus on steps 1 and 2 in the framework development as they are tightly coupled with the deployment. Current framework implementation, which is relevant to the scope of this paper, is available in the GitHub repository [25].

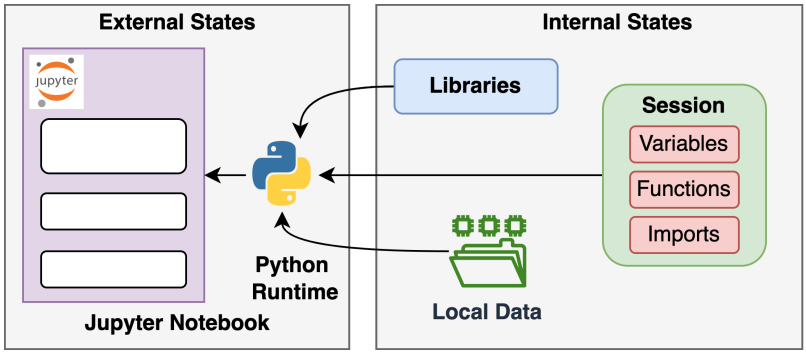

Figure 1: Components associated with the full state of a Jupyter Notebook. External states are not associated with the execution, and internal states depend on the code and execution order of the notebook.

4.1 Capturing Data Dependencies

Suppose a Python code in a Jupyter Notebook uses a file in the local file system for reading or writing purposes. In this case, this event should be captured if we want to replicate the data dependencies. However, this is a highly complicated task. There may be many Jupyter session-to-file system interactions either directly through the code the user has written or implicit libraries and function calls of the Python runtime. For example, a simple "pip install" command might access hundreds of files in the pip cache, and we should not consider them as data dependencies for the replication. Assuming that we have a way to filter out unwanted file system events, we still have the issue of capturing these file system events. Python and the iPython kernel do not have any native API to monitor file operations, so we must look for alternate ways.

4.1.1 Monitoring Linux Kernel System Calls by the IPython Kernel: All file operations are forwarded to the operating system as system calls regardless of the language-level file system operation implementation. To be precise, in Linux, all of these language bindings invoke the "openat" [15] system call to get the file descriptor of the file. We decided to capture the events from the Jupyter kernel process to the Linux kernel and log the invocations targeting "openat" system calls. We used Linux's "strace" [16] tool to filter and log the "openat" system calls coming through the IPython kernel process, which is responsible for the target Jupyter Notebook. There is a one-to-one relationship with a running Jupyter Notebook session and an IPython kernel process by design, so it is easy to derive file operations done by a notebook by capturing the procedures done by the designated IPython kernel process. Through this method, we managed to capture all the file system events coming from the Jupyter kernel runtime process irrespective of the language bindings used to access data at the programming language level.

The integration of the IPython kernel tracing framework is performed in two separate steps. In the first step, we capture system calls sent by each IPython kernel and log them into individual process-specific log files. In the second step, we analyze these logs files and identify relevant file system invocations.

Fig.2 illustrates the first stage processes. We developed a process-tracing server, which can start Linux "strace" subprocesses for monitoring given process ids. These "strace" sub-processes capture the "openat" system calls and log them in a specified log file. When a user creates a Jupyter Notebook, the Jupyter Notebook Server initializes a dedicated IPython kernel process to handle requests. We updated the start method of the IPython kernel to invoke the process tracing server with its process id as a parameter. The communication between these components is performed through a Unix socket to minimize the communication overhead. Once the process tracing server receives the kernel start message, it starts a "strace" process to trace the IPython kernel process. Dashed arrows in Fig. 2 show the tracing connection created from the "strace" job. Once the events are captured, the "strace" job writes only the "openat" system call log to a designated log file. Dotted arrows in Fig.2 depict that connection. The "strace" Linux command needs to be executed as the superuser to access kernel system calls. For this reason, we separate the IPython kernel and process tracing servers into two separated entities; IPython kernels run as the regular user and the tracing service runs as the superuser.

The second major step analyzes these log files and captures relevant file system operations to create a list of files that the Jupyter Notebook accessed. To achieve this, we developed a Jupyter Magic [11] extension to parse the traced log file and filter out relevant file operations for further processing (Fig. 3). At any point in the notebook, users can run the Jupyter Magic line command to invoke this workflow and get a list of files that they directly or indirectly used in the notebook up to that point. We use a rule-based filter to ignore all operations not required to reproduce the notebook state. Discounted file operations include those associated with Python cache files, Jupyter Notebook checkpoint files, library files created and accessed as byproducts of notebook autosaves, and "pip install" commands for filtering relevant file operations.

These filtered files are used as the reference to create the export of data dependencies for the notebook execution, as explained in the next section of the paper.

4.2 Determining Dependencies on Optional Libraries

Python's runtime ships with a default set of libraries like math, io, and sys to perform common operations for mathematical, system, and file operations. In addition, many out-of-the-box libraries can be installed from the pipi.org repository [22]. Typically users use these

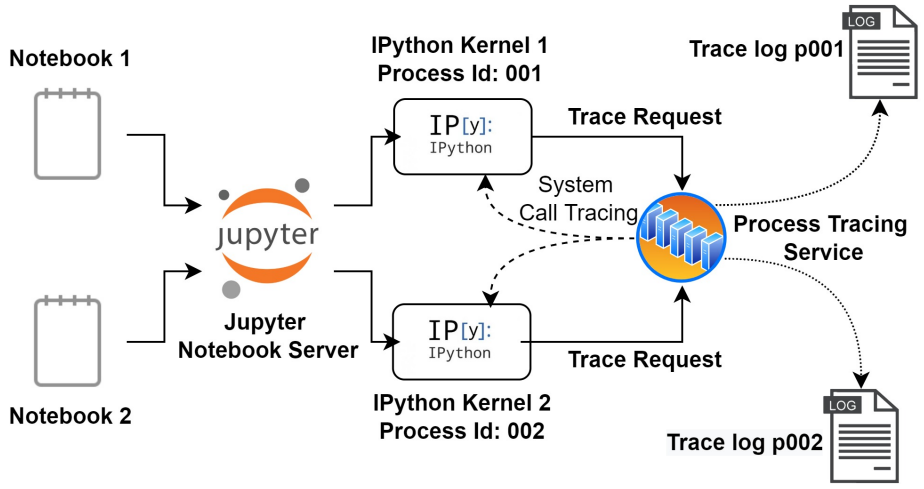

Figure 2: The workflow for capturing the filesystem calls from a Jupyter Notebook process to the operating system kernel. When an IPython process is started for a notebook, the process tracing service receives a notification to trace the process and then logs output into a file. Each notebook process has a different trace log file.

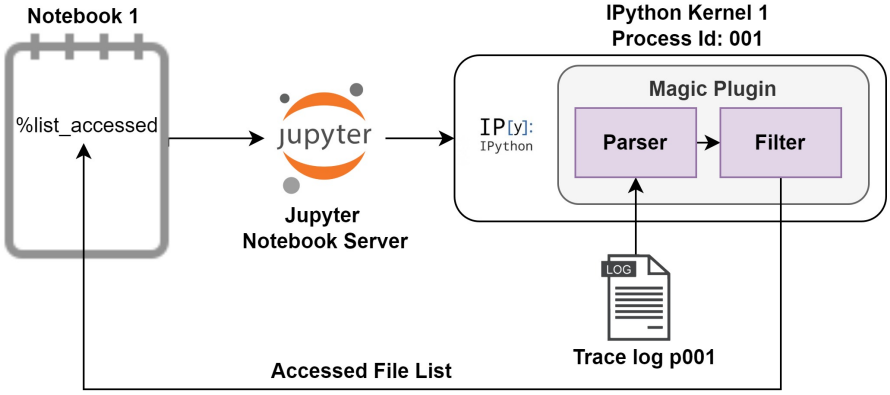

Figure 3: Workflow of processing a trace log file inside a Jupyter Magic plugin to shortlist the valid list of files accessed by the Jupyter Notebook. Trace log p001 is the log file associated with Notebook 1 that is created as depicted in Fig. 2.

libraries in notebooks either by installing them inside the notebook using the "pip install" command or using already installed libraries from the Python library cache of the running machine. However, each library may have multiple versions that contain some API-level and performance-related changes, so the code written using a particular library is somewhat coupled with the version of the library being used. When we share a notebook containing these library imports, there is no specific way to guide the users on which libraries and versions need to be installed before running them. Typically, the approach is to run the shared notebook, and if there is a missing library in the Python environment, install the missing one using the "pip install" command. However, it is still not guaranteed to work properly as we do not know the exact version of the library that was installed in the author's environment. For this reason, syntax errors and performance issues are possible even though the relevant library is installed in the second system.

Considering the above concerns, deriving the optionally installed libraries and their versions loaded in a Jupyter notebook session is critical to reproducing the library dependencies. However, this needs to be carried out carefully. In Python, the library version loaded into the notebook's IPython session depends on the way it is being bootstrapped. If the notebook was installed in a Python virtual environment, dependencies are loaded from the virtual environment's context. If, instead, the notebook is executed in the user's default environment, libraries are loaded from Python's global library cache. The best way to derive the correct library version is to connect to the specific notebook session and get the relevant library version.

To solve this problem, we developed another Python Magic extension that can be bundled into the running IPython kernel of the notebook. According to the IPython Magic framework, all the extensions receive the local namespace of the running IPython

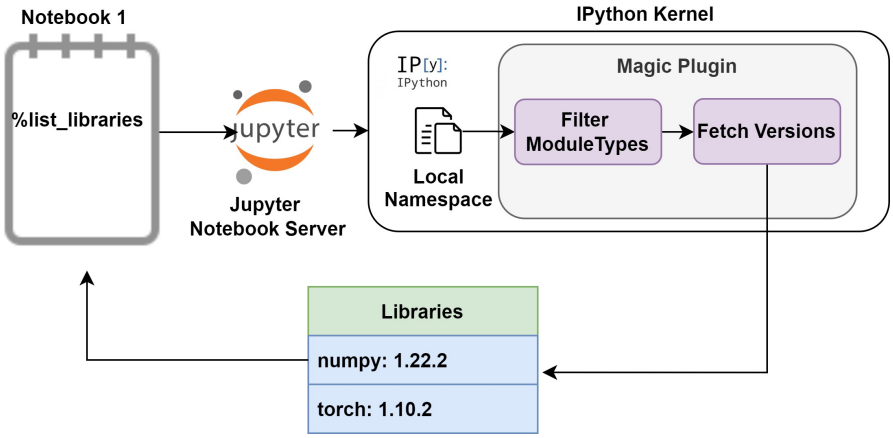

Figure 4: Magic plugin extensions capture the libraries' list and their versions imported inside a notebook session.

process. We can find all the imported libraries and versions inside this local namespace using a type-based filter below.

```
def imports():
    for name, val in local_ns.items():
        if isinstance(val, types.ModuleType):
            yield val.__name__
```

Next, we filtered out default libraries shipped with the Python runtime and captured the externally installed libraries. Once we get the library names, versions of the loaded library can be derived using the importlib [21] library.

Once all the relevant library names and versions are fetched, we save that metadata in a Python dictionary object where keys are the library name and values are versions (Fig. 4). This dictionary is used to export the library dependencies, as described in Section 5.

4.3 Capturing the Notebook's Runtime Session

The last and most challenging step to achieve complete reproducibility of a Jupyter Notebook is to capture the notebook's runtime session. This is an essential feature for checkpointing a notebook and restarting it in the same or different environment to continue from the last executed position. This requires serializing the execution session and reloading it back when it is restarted. A typical running Jupyter session may include library imports, initialized variables, and function declarations. If we want to create a snapshot of a session, we need a way to serialize all these types of entities when storing the state and deserialize them when restoring to a running session. In addition, we need to find a way to programmatically list the minimal set of entities that is sufficient to replicate the entire session.

A local namespace is a dictionary of objects where keys are entity names and values are entities created throughout the notebook session. These keys consist of variable names, function names, imports, and previous cells' execution ids. We developed a Jupyter Magic extension (Fig. 5) to capture this session information using the local namespace information that the extension receives from the Jupyter framework. Here we are interested only in the global variables, function definitions, and library imports required to recreate a session.

The challenge is to derive the subset of these entities from the local namespace as it represents all the entities as generic key-value pairs. To find this set of entity names, we use the "who" [9] built-in Jupyter Magic command, which prints the signatures of imports, functions, and global variables. Once those entities are identified, we create a minimal version of the local namespace dictionary by filtering derived entity names. To export this sub-namespace object, we serialize it to a binary file using the "dill" [20] library, which is capable of serializing most of Python objects into binary format. However, there may be scenarios where some objects in the sub-namespace can not be serialized. To address this problem, we do a binary search on the sub-namespace and perform serialization tests on each subtree to determine the availability of any possibly incompatible entity. We remove any incompatible entities from the sub-namespace object and warn the user if there are any.

Once the sub-namespace object is serialized into byte format, we use it to build the notebook state bundle as mentioned in Section 5.

5 STRATEGY FOR REPRODUCING STATEFUL NOTEBOOKS

The previous section discussed how we capture the information to reproduce the whole notebook state. This section outlines how we package and export that information to be reproduced in a different environment. We also discuss how to bootstrap an environment when we import a notebook state from an exported bundle as mentioned above.

5.1 Packaging Strategy for Captured States

We use an archiving approach when exporting a notebook state that uses all of the above-mentioned state capturing techniques. First, all the files identified in the data dependency capturing stage are copied into the root level of the archive directory. To avoid conflicts, each file is duplicated with a unique UUID name rather than the original name. We also create a JSON file containing UUID-to-absolute-file-path mappings; this JSON file is also copied to the root level of the archive directory and the data. Library dependencies that we

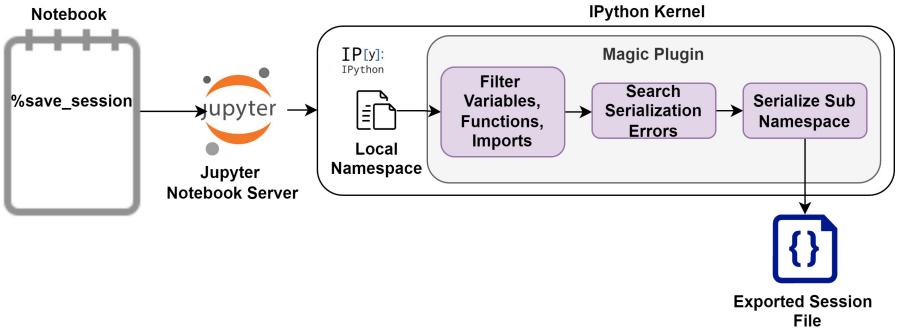

Figure 5: Magic Plugin implementation workflow to capture the variables, imports, and functions in the local namespace of the Jupyter Notebook session and serialize into a file.

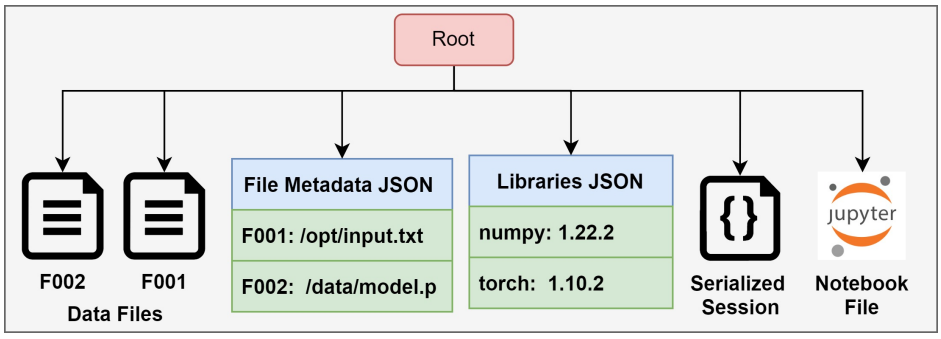

Figure 6: The structure of the archive file that contains a serialized stateful Jupyter Notebook

capture (as described in the previous section) are converted into a JSON format that contains library names and versions; this JSON file is also placed in the root level of the archive directory. In the notebook session capturing stage, we serialize the sub-namespace dictionary object to a file; it is also copied into the root level of the archive directory. Finally, we copied the notebook file as the last stage of archive creation. Figure 7 illustrates the contents and file structure of the archive.

To simplify the process, all of the operations mentioned above are integrated into a single Jupyter Magic command so that the user can invoke it inside the notebook file. Upon invocation, the custom Magic code internally invokes the state capturing magic commands described earlier, creates the archiving directory based on the data retrieved from those invocations, creates the archive file using the zip tool, and provides a link for the user to download it through the browser. This archived zip file can be later used to recreate the same state of the notebook in a different environment.

5.2 Recreating a Stateful Notebook from an Exported Archive

To recreate a notebook environment from an exported archive, we need to execute the following steps.

(1) Unarchive the exported archive file.
(2) Parse the File Metadata JSON file and copy data files into designated locations provided in the values of the JSON.
(3) Parse the Libraries JSON file and install those dependencies with versions using "pip install" commands.
(4) Start the IPython kernel and initialize a session to the exported notebook files.
(5) Run a Jupyter magic command to parse the serialized session and merge it with the local namespace of the notebook.

All of these steps can be performed in any notebook environment, including bare metal, cloud, and containerized platforms. We have developed a single script to accept the archive file with the notebook runtime to reproduce the archived environment. However, there may be issues when placing data files in Step 2 if the notebook is running in a different operating system that does not follow the file system structure of the source environment. For example, a notebook archive created on a Linux computer may fail at Step 2 on a computer running a Windows operating system.

Considering these scenarios, we recommend using a containerized environment when reproducing a notebook to make the initialization process independent of the host's file system. Both state-recording and state-replicating notebook environments can be bootstrapped from a one-line command to make the process as simple as possible for the user. To facilitate state recording and recreate a notebook, we have developed a container image that bundles a Jupyter runtime [5], a tracing server, and a python installation.

To start a notebook environment in state recording mode, users can run the command:

```
docker run --cap-add=SYS_PTRACE -it -p 8888:8888
    dimuthuupe/ipykernel:1.0
```

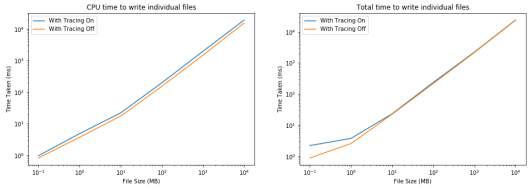

Figure 7: CPU time and total time to write a single file in various sizes using the notebook's Python runtime in tracing on and off scenarios

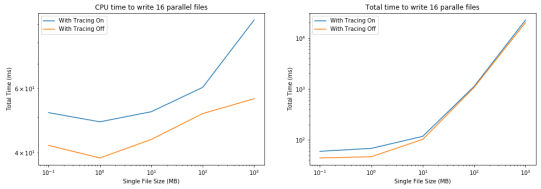

Figure 8: CPU time and total time to write 16 parallel files (1 per CPU core) in various sizes using the notebook's Python runtime in tracing on and off scenarios

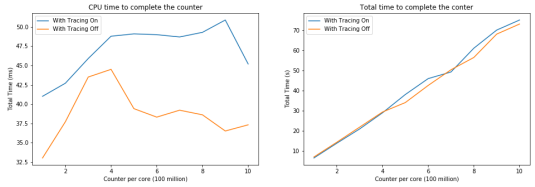

Figure 9: CPU time and total time run parallel counters (1 per CPU core) in various sizes using the notebook's Python runtime in tracing on and off scenarios

To recreate an environment from a state archive:

```
docker run --cap-add=SYS_PTRACE -it -p 8888:8888
-v <ARCHIVE_FILE>:/opt/ARCHIVE.zip dimuthuupe/ipykernel:1.0
```

6 PERFORMANCE ANALYSIS

We use an external Linux "strace" process to capture file operations performed by the notebook process to export the file list required to reproduce an identical environment. However, running an "strace" process along with the notebook to capture system calls may have performance implications because recording the operating system's system calls will cause them to block or wait. We record only "openat" system calls at the design level to mitigate the impact. However, there may still be a performance hit as everything from a mounted disk to a network socket in Linux is represented as a file. We may be capturing unwanted events that will eventually affect the notebook's overall performance.

We performed various stress tests on strace-enabled and disabled environments to evaluate this impact. We ran IO intensive, CPU intensive, and mixed workloads to identify areas affected by a bottleneck. All the performance tests were carried out on a workstation with 32 GB memory, 16 cores and disk write speed of 1GB/s. To keep both environments consistent, we ran both notebooks as Docker containers.

To enable and disable stracing, the container distribution of the kernel accepts the environment variable `ENABLE_TRACE` as a runtime configuration.

6.1 Running IO-intensive workloads

6.1.1 Single file write performance: In this experiment, we carried out a test to measure the performance impact from tracing system calls for a single file write. To simulate this scenario, we ran the following Python code fragment with various file sizes to measure the time it took to write a file into the disk with random characters. We captured CPU time and overall wall time for each run as metrics. CPU time is the time the code spends in the CPU, including the Linux kernel, and wall time is the total time it takes to run, including the CPU time and system call overheads (Fig. 7).

```
with open("random.dat", "wb") as output:
    output.write(np.random.bytes(FILE_SIZE))
```

6.1.2 Multiple file write performance: We then ran the same logic of writing files parallelly across 16 cores to measure the performance of parallel file writes (Fig. 8). This experiment fully utilizes CPU, memory, and disk IO, including multi-core executions, in-memory buffer creation, and parallel disk writes.

6.2 Running CPU Intensive workloads

To measure how the system performed when the CPU is fully utilized, we ran static counters on each CPU core parallelly using the following code fragment and measured the time taken for all the counters to finish (Fig. 9).

```
def f(x):
  counter = 1
  while True:
    counter = counter + 1
    if counter > COUNTER_SIZE:
      break
Pool(psutil.cpu_count()).map(f, range(processes))
```

6.3 Observations

We have analyzed three test scenarios: IO intensive, mixed (both IO and CPU), and CPU intensive operations to measure the impact of tracing the Jupyter notebook kernel process using the "strace" program to capture file operations. Among the three scenarios, we observed that IO intensive operations are the least affected by strace, while CPU intensive operations are the most affected in terms of the CPU time spent. However, measuring the total time to complete these operations, which is the real time between start and end, shows that significantly less cost is paid by the tracing enabled setup than a non-tracing environment. It is very clear that tracing affects CPU time, but because the proportion of CPU time to the total time is very low (the majority of total time is governed by the system call overhead), the impact of the CPU time does not significantly affect the final overall performance.

7 CONCLUSION & FUTURE WORK

This paper presents the framework-level concepts needed to implement a fully reproducible Jupyter Notebook environment and the results of a performance analysis of possible bottleneck scenarios. Currently, the proof of concept implementation of the framework supports Docker-based executions in single-user mode. Even with this limitation, our approach is potentially beneficial for long-running notebooks, especially on HPC resources: Users can pause running notebooks at some point and resume in a different environment with zero configuration changes. In the future, we plan to integrate this framework as a secure multi-tenanted service on the Jetstream2 [8] cloud computing system with the scaling support of JupyterHub [12]. Integrating with Apache Airavata components, we plan to facilitate seamless remote execution of machine learning models on Jetstream2's GPU resources coupled with analysis on local resources all within a single notebook session.

REFERENCES

[1] Amazon. 2022. Amazon Spot Instances. https://aws.amazon.com/ec2/spot

[2] ReScience C. 2022. Reproducible science is good. replicated science is better. https://rescience.github.io/

[3] Julien Chastang, Rich Signell, and Jeremy Fischer. 2018. A unidata jupyterhub server: An online pyaos resource for students and educators. In *98th American Meteorological Society Annual Meeting, AMS.*

[4] Renato LF Cunha, Lucas C Villa Real, Renan Souza, Bruno Silva, and Marco AS Netto. 2021. Context-aware Execution Migration Tool for Data Science Jupyter Notebooks on Hybrid Clouds. In *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 30–39.

[5] Dimuthu. 2022. Customized IPython Kernel Docker Image to capture Notebook state. https://hub.docker.com/r/dimuthuupe/ipykernel

[6] Google. 2022. Colaboratory. https://research.google.com/colaboratory/

[7] Brian Granger and Fernando Pérez. 2021. Jupyter: Thinking and storytelling with code and data. *Authorea Preprints* (2021).

[8] David Y Hancock, Jeremy Fischer, John Michael Lowe, Winona Snapp-Childs, Marlon Pierce, Suresh Marru, J Eric Coulter, Matthew Vaughn, Brian Beck, Nirav Merchant, et al. 2021. Jetstream2: Accelerating cloud computing via Jetstream. *Practice and Experience in Advanced Research Computing* (2021), 1–8.

[9] IPython. 2022. IPython "who" Magic Command. https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-who

[10] Caroline Jay, Robert Haines, and Daniel S Katz. 2020. Software must be recognised as an important output of scholarly research. *arXiv preprint arXiv:2011.07571* (2020).

[11] Jupyter. 2022. Developing Custom Magic Extensions for Jupyter Notebooks. https://ipython.readthedocs.io/en/stable/config/custommagics.html

[12] Jupyter. 2022. JupyterHub. https://jupyter.org/hub

[13] Jupyter. 2022. Kernels. https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

[14] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. *Jupyter Notebooks-a publishing format for reproducible computational workflows.* Vol. 2016.

[15] Linux. 2022. Openat System Call. https://linux.die.net/man/2/openat

[16] Linux. 2022. Strace - Process tracing command. https://man7.org/linux/man-pages/man1/strace.1.html

[17] National Academies of Sciences Engineering, Medicine, et al. 2019. Reproducibility and replicability in science. (2019).

[18] Roger D Peng. 2011. Reproducible research in computational science. *Science* 334, 6060 (2011), 1226–1227.

[19] Fernando Perez and Brian E Granger. 2015. Project Jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September* 11, 207 (2015), 108.

[20] Python. 2022. Dill: Python Object Serialization Library. https://pypi.org/project/dill

[21] Python. 2022. Importlib Library. https://docs.python.org/3/library/importlib.html

[22] Python. 2022. Pypi Repository. https://pypi.org

[23] David M. Rosenberg and Charles C. Horn. 2016. Neurophysiological analytics for all! Free open-source software tools for documenting, analyzing, visualizing, and sharing using electronic notebooks. (2016).

[24] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. , e1007007 pages.

[25] SciGaP. 2022. Framework to reproduce Jupyter Notebook Environments. https://github.com/SciGaP/iPython-Kerner-Changes