ElasticNotebook: Enabling Live Migration for Computational Notebooks

Zhaoheng Li
Univ. of Illinois Urbana-Champaign
zl20@illinois.edu

Pranav Gor
Univ. of Illinois Urbana-Champaign
gor2@illinois.edu

Rahul Prabhu
Univ. of Illinois Urbana-Champaign
rprabhu5@illinois.edu

Hui Yu
Univ. of Illinois Urbana-Champaign
huiy3@illinois.edu

Yuzhou Mao
University of Michigan - Ann Arbor
yuzhom@umich.edu

Yongjoo Park
Univ. of Illinois Urbana-Champaign
yongjoo@illinois.edu

ABSTRACT

Computational notebooks (e.g., Jupyter, Google Colab) are widely used for interactive data science and machine learning. In those frameworks, users can start a *session*, then execute *cells* (i.e., a set of statements) to create variables, train models, visualize results, etc. Unfortunately, existing notebook systems do not offer live migration: when a notebook launches on a new machine, it loses its *state*, preventing users from continuing their tasks from where they had left off. This is because, unlike DBMS, the sessions directly rely on underlying kernels (e.g., Python/R interpreters) without an additional data management layer. Existing techniques for preserving states, such as copying all variables or OS-level checkpointing, are unreliable (often fail), inefficient, and platform-dependent. Also, re-running code from scratch can be highly time-consuming.

In this paper, we introduce a new notebook system, Elastic-Notebook, that offers live migration via checkpointing/restoration using a novel mechanism that is reliable, efficient, and platform-independent. Specifically, by observing all cell executions via transparent, lightweight monitoring, ElasticNotebook can find a reliable and efficient way (i.e., *replication plan*) for reconstructing the original session state, considering variable-cell dependencies, observed runtime, variable sizes, etc. To this end, our new graph-based optimization problem finds how to reconstruct all variables (efficiently) from a subset of variables that can be transferred across machines. We show that ElasticNotebook reduces end-to-end migration and restoration times by 85%-98% and 94%-99%, respectively, on a variety (i.e., Kaggle, JWST, and Tutorial) of notebooks with negligible runtime and memory overheads of <2.5% and <10%.
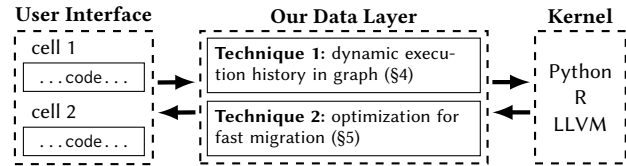
Figure 1: Our transparent data layer (in the middle) enables robust, efficient, and platform-independent live migration.

1 INTRODUCTION

Computational notebooks[1] (e.g., Jupyter [61, 99], Rstudio [85]) are widely used in data science and machine learning for interactive tutorials [60], data exploration [18, 25, 119], visualization [27], model tuning and selection [9, 112], etc. Cloud providers offer Software-as-a-Services (e.g., AWS hub [90], Azure ML studio [4], Google Colab [46], IBM Watson studio [54]) with commonly used libraries (e.g., Pandas, PyTorch). A notebook workflow begins with a user starting a *computing session*. Then, the user can execute a *cell* (i.e., a set of statements), one by one, to load datasets, create variables, train models, visualize results, etc. The session can be terminated manually or automatically to save resources and costs.

Limitation: No Live Replication. Unfortunately, existing notebooks do not offer transparent infrastructure scaling (independent of applications), which are becoming increasingly popular in the cloud for instant scalability and cost reduction (e.g., auto-scaling DBMS [84, 110], micro-service orchestration [24, 66]). That is, if we copy a notebook file to a new VM (e.g., for larger memory) or suspend a session to save costs, the resumed notebook loses its *state* (i.e., a set of variables), having only code and outputs. In other words, the user cannot resume their task from where they had previously left off. This is because the notebooks directly rely on underlying kernels (e.g., Python/R interpreters, C++ REPL) without an additional data management layer. Accordingly, the variables residing in processes are erased as they terminate with sessions. To address this, we can potentially save those variables and restore them on a new environment. However, existing techniques such as serializing all variables [36, 37, 39] and checkpointing OS processes [3, 17, 42, 59] may fail, are inefficient, and platform-dependent (discussed shortly). Finally, re-running code from scratch can be time-consuming.

---

[1]In this work, we use the term a "notebook" to mean either a system serving the notebook or the contents of the notebook, depending on the context.

Table 1: Comparison between our ElasticNotebook and other possible approaches to saving/restoring session states

Approach	Mechanism
Serialization-based tools [34, 37–39, 102]	Serializes and stores variables during computing session (fails with unserializable variables)
System-level checkpointing [3, 17, 42, 59, 62]	Saves memory dump of computing session (high network cost and low portability)
Execution environment migration [1, 113]	Migrates installed modules; useful in conjunction with (but orthogonal to) session state replication
Ours (ElasticNotebook)	**Optimally combines copy/recompute for reliability, efficiency, and platform independence**

Our Goal. We propose ElasticNotebook, a notebook system that offers live state migration via checkpointing/restoration using a reliable, efficient, and platform-independent state replication mechanism. ***Reliability:*** It enables correct/successful replication for (almost) all notebooks. ***Efficiency:*** It is significantly more efficient than others. ***Platform-independence:*** It does not rely on platform-/architecture-specific features. That is, ElasticNotebook enables *live notebook replication* for potentially all notebook workloads by introducing a novel data management layer. For example, if a user specifies a new machine to run a currently active notebook, the system transparently replicates the notebook, including all of its variables, as if the notebook has been running on the new machine. If we can provide this capability with little to no modifications to existing systems (e.g., Jupyter), we can offer benefits to a large number of data scientists and educators who use notebooks. To achieve this, we must overcome the following technical challenges.

Challenge. Creating a reliable, efficient, and platform-independent replication mechanism is challenging. First, the mechanism must offer high coverage. That is, for almost all notebooks people create, we should be able to successfully replicate them across machines. Second, the mechanism should be significantly faster than straightforward approaches—rerunning all the cells exactly as they were run in the past, or copying, if possible, all the variables with serialization/deserialization. Third, the mechanism should integrate with existing notebook systems with clean separation for sustainable development and easier adoption.

Our Approach. Our core idea is that by observing the evolution of session states via lightweight monitoring, we can address the three important challenges—reliability, efficiency, and platform-independence—by combining program language techniques (i.e., on-the-fly code analyses) and novel algorithmic solutions (i.e., graph-based mathematical optimization). Specifically, to represent session state changes, we introduce the *application history*, a special form of bipartite graph expressing the dependencies among variables and cell executions. Using this graph, we take the following approach.

First, we achieve *reliability* and *platform independence* by choosing a computational plan (or *replication plan*) that can safely reconstruct platform-dependent variables (e.g., Python generators, incompletely defined custom classes) based on the other platform-independent variables. That is, in the presence of variables that cannot be serialized for platform-independent replication, ElasticNotebook uses the application history to recompute them dynamically on a target machine. In this process, ElasticNotebook optimizes for the collective cost of recomputing all such variables while still maintaining their correctness (§4).

Second, for *efficiency*, ElasticNotebook optimizes its replication plan to determine (1) the variables that will be copied, and (2) the variables that will be recomputed based on the copied variables, to minimize the end-to-end migration (or restoration) time in consideration of serialization costs, recomputation costs, data transfer costs, etc. For example, even if a variable can be reliably transferred across machines, the variable may still be dynamically constructed if doing so results in a lower total cost. To make this decision in a principled way, we devise a new graph-based optimization problem, which reduces to a well-established min-cut problem (§5).

Implementation: While our contributions can apply to many dynamically analyzable languages (e.g., Python/R, LLVM-based ones), we implement our prototype (in C and Python) for the Python user interface, which is widely used for data science, machine learning, statistical analysis, etc. Specifically, ElasticNotebook provides a data management layer to Jupyter as a hidden *cell magic* [101] to transparently monitor cell executions and offer efficient replication.

Difference from Existing Work. Compared to existing work, we pursue a significantly different direction. For example, there are tools that make data serialization more convenient [38, 102]; however, they fail if a session contains non-serializable variables, and are inefficient because they do not consider opportunities for dynamic recomputation. Alternatively, system-level checkpointing [3, 17, 42, 59] is platform-dependent, limited to checkpointing memory (e.g., not GPU), less efficient than ours since dynamic recomputation is impossible. Building on top of result reuse [41, 114] and lineage tracing [52, 82, 92], we introduce deeper (reference-aware) analyses (§4.2) and novel optimization techniques to incorporate unique constraints such as inter-variable dependencies (§5) and also empirically confirm their effectiveness (§7.2). Completely orthogonal work includes library migration [1, 113] and scalable data science [78, 80, 115]. Table 1 summarizes differences.

Contributions. Our contributions are as follows:
- **Motivation.** We discuss alternative approaches and explain the advantage of our approach. (§2)
- **Architecture.** We describe our system architecture for achieving efficient and robust session replication. (§3)
- **Data Model.** We introduce a novel data model (*Application History Graph*) for expression session history, which enables efficient and accurate state replication. (§4)
- **Optimization Problem and Solution.** We formally define the optimization problem of minimizing state replication cost through balancing variable copying and recomputation. We propose an efficient and effective solution. (§5)
- **Evaluation.** We show ElasticNotebook reduces upscale, downscale, and restore times by 85%-98%, 84%-99%, and 94%-99%, respectively. Overheads are negligible (<2.5% runtime). (§7)
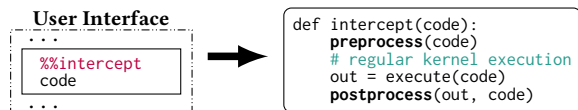
```
def intercept(code):
    preprocess(code)
    # regular kernel execution
    out = execute(code)
    postprocess(out, code)
```

Figure 2: For every cell run, we can inject custom pre-/post-processing logic. "%%intercept" is hidden to users.

2 MOTIVATION

This section describes use cases (§2.1) and requirements (§2.2) for session replication, and our intuition for higher efficiency (§2.3).

2.1 Why is Live Migration Useful?

A seamless state replication for computational notebooks can allow easier infrastructure scaling and frequent session suspension, without interrupting user workflow, as described below.

Fast Replication for Elastic Computing. The ability to move a state across machines is useful for scaling resources [20, 62], allowing us to migrate a live session to the machines with the right equipment/resources (e.g., GPU [21], specific architectures [117]). For interruption-free scaling, we can copy data $\mathcal{D}$ from a source machine to a target machine in a way that the original session state can be restored from $\mathcal{D}$. In this process, we want to minimize the end-to-end time for creating $\mathcal{D}$, transferring $\mathcal{D}$ to a target machine, reconstructing the state from $\mathcal{D}$ on the target machine. This is the first use case we empirically study (§7.3).

Fast Restart for On-demand Computing. Leveraging pay-as-you-go pricing model offered by many cloud vendors [5, 47], suspending sessions (and VMs) when not in use is an effective way for reducing charges (e.g., up to 6× [113]). With the ability to create data $\mathcal{D}$ sufficient for reconstructing the current session state, we can persist $\mathcal{D}$ prior to either manual or automated suspension [19, 46, 57], to quickly resume, when needed, the session in the same state. This achieves on-demand, granular computing with fast session restart times without impacting user experience due to frequent session suspensions [56, 96]. In this process, we want to restore the session as quickly as possible by minimizing the time it takes for downloading $\mathcal{D}$ and reconstructing a state from it. This is the second use case we empirically study (§7.4).

2.2 How to Enable Data Management Layer?

We discuss the pros and cons of several different approaches to enabling a data management layer.

OS-level Checkpointing. To save the current session state, we can checkpoint the entire memory space associated with the underlying Python/R kernels. To make the process more efficient, existing tools like CRIU patch the Linux kernel to trace dirty pages. However, as described in §1, this approach is platform-independent, incurs higher space cost, and is limited to storing the state of primary memory (not GPU or other devices). We empirically compare our approach to CRIU to understand reliability and efficiency (§7).

Object wrappers. Watchpoint object wrappers [40, 43] are commonly used for debugging purposes [82] and program slicing [52, 92]: they maintain deep copies for objects in the session state, which
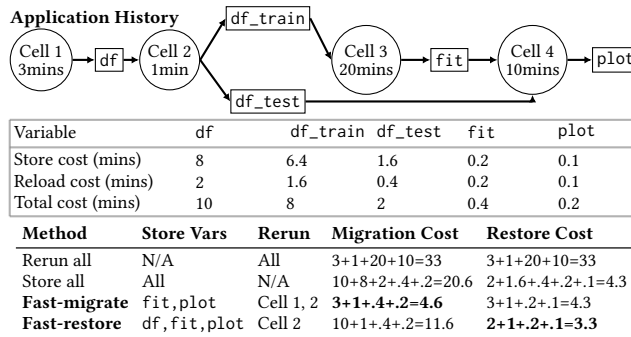

Figure 3: Example app history (top) and different replication plan costs (bottom). Combining recompute/copy allows faster migration (Fast-migrate). Alternatively, the optimal plan changes if the restoration is prioritized (Fast-restore).

are compared to check for changes after each frame execution; however, they are unsuitable for use during data science workflows due to the unacceptable ~20× runtime overhead in our preliminary tests.

Monitoring Cell Executions (Ours). In order to trace cell executions and their effects on variables, we can add a lightweight wrapper (i.e., our data management layer) that functions before and after each cell execution to monitor the cell code, runtime, and variable changes. This idea is depicted conceptually in Fig 2. Specifically, our implementation uses *cell magics*, a Jupyter-native mechanism that allows arbitrary modification to cell statements when the cell is executed. With this, we add pre-/post-processing steps to capture cell code and resulting session state modifications.

2.3 Fast Replication with Application History

This section describes our core idea for devising an efficient replication strategy by leveraging the ability to monitor cell executions.

Application History. An *application history graph* (AHG) is a bipartite graph for expressing session states changes with respect to cell runs. There are two node types: variables and transformations. A transformation node connects input variables to output variables (see example in Fig 3). AHG aims to achieve two properties required for correct state reconstruction (§4): **Completeness:** No false negatives. All input/output variables for each transformation must be captured. **Minimal:** Minimal false positives. The number of variables that are incorrectly identified as accessed/modified, while variables are not actually accessed/modified, must be minimized.

Core Optimization Idea. AHG allows for efficient state replication with a combination of (1) recompute and (2) copy.

Motivating Example. Suppose a data analyst fitting a regression model (Fig 3). The notebook contains 4 cell runs: data load (Cell 1), train-test split (Cell 2), fitting (Cell 3), and evaluation (Cell 4). After fitting, the analyst decides to move the session to a new machine for GPU. Simply rerunning the entire notebook incurs ***33 minutes***. Alternatively, serializing/copying variables takes ***20.6 minutes***.

However, there is a more efficient approach. By copying only `fit` and `plot` and recomputing others on a new machine (Fast-migrate), ***we can complete end-to-end migration in 4.6 minutes.*** Or, if we prioritize restoration time (to reduce user-perceived restart

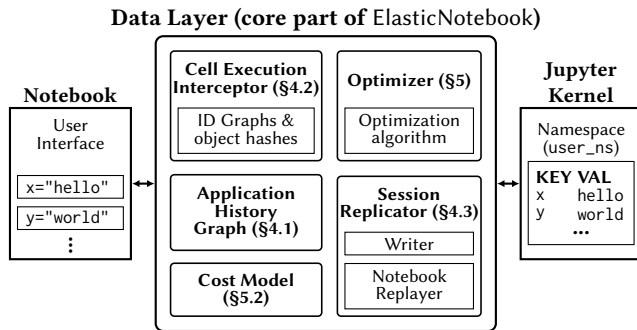**Data Layer (core part of** ElasticNotebook**)**

Figure 4: ElasticNotebook architecture. Its data layer acts as a gateway between the user interface and the kernel: cell executions are intercepted to observe session state changes.

time for on-demand computing), our optimized plan (Fast-restore) takes 3.3 minutes. This example illustrates significant optimization opportunities in session replication. Our goal is to have the ability to find the best replication plan for arbitrarily complex AHGs.

3 SYSTEM OVERVIEW

This section presents ElasticNotebook at a high level by describing its components (§3.1) and operations (§3.2).

3.1 ElasticNotebook Components

ElasticNotebook introduces a unique data layer that acts as a gateway between the user and the kernel (See Fig 4): it monitors every cell execution, observing code and resulting session state changes.

Cell Execution Interceptor. The Cell Execution Interceptor intercepts cell execution requests and adds pre-/post-processing scripts before rerouting it into the underlying kernel for regular execution. The added scripts perform (1) cell code analyses and the AHG updates, and (2) cell runtime recordings.

Application History Graph (AHG). The AHG is incrementally built by the Cell Execution Interceptor to record how variables have been accessed/modified by each cell execution (§4). The AHG is used by the Optimizer to compute replication plans (§5).

Cost Model. The cost model stores profiled metrics (i.e., cell runtimes, variable sizes, network bandwidth), serving as the hyperparameters for the Optimizer (§5.2).

Optimizer. The Optimizer uses the AHG and the Cost Model to determine the most efficient replication plan consisting of (1) variables to store and (2) cells to re-run. We discuss ElasticNotebook's cost model and optimization in detail in §5.

Session Replicator. The Session Replicator replicates a notebook session according to the Optimizer's plan. Specifically, the Writer creates and writes a checkpoint file to storage (e.g., SSD, cloud storage), while the Notebook Replayer reads the file and restores the session, both following the replication plan. We discuss ElasticNotebook's session replication in detail in §3.2.

3.2 ElasticNotebook Workflow

This section describes ElasticNotebook's operations. ElasticNotebook monitors every cell execution during a session lifecycle, then performs on-request replication of the session in two steps: *checkpointing* (writing to the checkpoint file) and *restoration*.

Monitoring Cell Executions. Upon each cell execution by the user, ElasticNotebook performs the following steps:

1. Accessed variables of the cell execution are identified via AST analysis (described in §4.2).
2. The cell code is executed by the Jupyter kernel.
3. Variable changes (i.e., creation/deletion/modification) are identified within the global namespace (§4.2).
4. The AHG is updated using (1) the cell code and (2) modified variables by the cell execution.
5. The Cost Model is updated to record cell runtime.

Initiating Replication. When replication is requested, ElasticNotebook creates and writes a *checkpoint file* to storage, which can be restored later to exactly and efficiently reconstruct the current session. ElasticNotebook first completes the Cost Model by profiling variable sizes and network bandwidth to storage; then, the Optimizer utilizes the AHG and Cost model to compute a replication plan, according to which the Writer creates the checkpoint file: it consists of (1) a subset of stored variables from the session state, (2) cells to rerun, (3) the AHG, and (4) the Cost Model.

Restoring a Session. When requested, ElasticNotebook restores the notebook session from the checkpoint file according to the replication plan. The Notebook Replayer reconstructs variables in the order they appeared in the original session by combining (1) cell reruns and (2) data deserialization followed by variable redeclaration (into the kernel). Finally, ElasticNotebook loads the AHG and Cost Model for future replications.

Accuracy Guarantee: ElasticNotebook's state reconstructing is effectively the same as re-running all the cells from scratch exactly in the order they were run in the past. That is, ElasticNotebook shortens the end-to-end reconstruction time by loading saved variables (into the kernel namespace) if doing so achieves time savings. §4.3 presents formal correctness analysis. §6.1 discusses how we address external resources, side effects, and deserialization failures.

4 APPLICATION HISTORY GRAPH

This section formally defines the Application History Graph (§4.1), and describes how we achieve exact state replication (§4.3).

4.1 AHG Formal Definition

The AHG is a directed acyclic graph expressing how a session state has changed with respect to cell executions. Fig 5 is an example.

Definition 1. A **variable** is a named entity (e.g., df) referencing an **object** (which can be uniquely identified by its object ID).

A variable can be primitive (e.g., int, string) or complex (e.g., list, dataframe). Multiple variables may point to the same object. The set of all variables (i.e., $\mathcal{X}$) defined in the global namespace forms a session state. Cell executions may modify the values of variables (or referenced objects) without changes to their names, which we recognize in AHG using *variable snapshot*, as follows.

Table 2: Notations and their meaning

Symbols	Definition
$\mathcal{X}$	Set of Variables
$\mathcal{V}$	Set of Variable Snapshots (VSs)
$\mathcal{V}_a$	Set of Active Variable Snapshots
$C (= c_{t_1}, c_{t_2}, \ldots)$	Set of Cell Executions (CEs)
$\mathcal{E}_w$	Set of write dependencies
$\mathcal{E}_r$	Set of read dependencies
$\mathcal{G} := \{\mathcal{V} \cup C, \mathcal{E}_w \cup \mathcal{E}_r\}$	Application History Graph (AHG)
$req : \mathcal{X} \to 2^C$	Reconstruction mapping function
$w_{store} : \mathcal{X} \to \mathbb{R}^+$	Variable storage cost
$w_{rerun} : C \to \mathbb{R}^+$	Cell Rerun cost
$w_M : 2^{\mathcal{X}} \to \mathbb{R}^+$	Migration cost function
$w_R : 2^{\mathcal{X}} \to \mathbb{R}^+$	Recomputation cost function
$\mathcal{L} \subseteq \mathcal{X} \times \mathcal{X}$	Pairs of linked variables
$\mathcal{H} := \{\mathcal{V}_H, \mathcal{E}_H\}$	Flow graph
$c : \mathcal{E}_H \to \mathbb{R}^+$	Flow graph edge capacity function

Definition 2. A **variable snapshot** (VS) is a name-timestamp pair, (x, t), representing the variable x created/modified at t. We denote the set of VSes as $\mathcal{V}$.

Definition 3. A **cell execution** (CE) c_t represents a cell execution that finishes at timestamp t.

All cell executions are *linear*; that is, for each session, there is at most one cell running at a time, and their executions are totally ordered. We denote the list of CEs by C. Each CE also stores executed cell code, which can be used for re-runs (§3.2).

Definition 4. A **write dependency** $(c_t \to (x, t))$ indicates CE c_t may have modified/created at time t the object(s) reachable from the variable x. We denote the set of write dependencies as $\mathcal{E}_w$.

In Fig 5, c_{t_3} modifies x with "x += 1"; hence, $(c_{t_3} \to (x, c_{t_3}))$.

Definition 5. A **read dependency** $((x, s) \to c_t)$ indicates CE c_t may have accessed object(s) reachable from x last created/modified at time s. We denote the set of read dependencies by $\mathcal{E}_r$.

In Fig 5, "gen=(i for i in l1)" in C_{t_4} accesses elements in the list l1 after its creation in c_{t_3}; hence there is $((x \to c_{t_3}), c_{t_4})$. Note that write/read dependencies are allowed to contain false positives; nevertheless, our replication ensures correctness (§4.3).

Definition 6. The **AHG** $G := \{\mathcal{V} \cup C, \mathcal{E}_w \cup \mathcal{E}_r\}$ is a bipartite graph, where $\mathcal{V}$ is VSes, C is CEs; $\mathcal{E}_w$ and $\mathcal{E}_r$ are write/read dependencies, respectively. It models the lineage of the notebook session.

In sum, AHG formalizes variable accesses/modifications with respect to cell executions. at the variable level (not object level), theoretically bounding the size of AHG to scale linearly with the number of defined variables, not the number of underlying objects (which can be very large for lists, dataframes, and so on). We empirically verify AHG's low memory overhead in §7.5.

4.2 Dynamic AHG Construction

We describe how ElasticNotebook constructs the AHG accurately.

Constructing the AHG. The AHG is incrementally built with accessed/created/modified variables by each cell execution:

- A new CE c_t is created; t is an execution completion time.
- Read dependencies are created from VSes $(x_1, t_{x_1}), ..., (x_k, t_{x_k})$ to c_t, where $x_1, ..., x_k$ are variables *possibly* accessed by c_t.
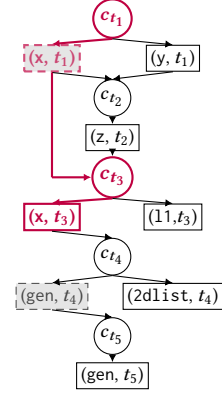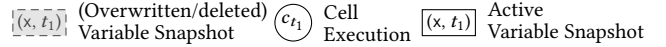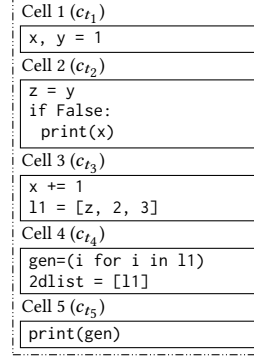
Notebook

Figure 5: An example notebook and its corresponding Application History Graph. The AHG tells ElasticNotebook how to recompute variables; for example, rerunning c_{t_1} and c_{t_3} is necessary for recomputing x (red).

- VSes $(y_1, t), ..., (y_k, t)$ are created, where $y_1, ..., y_k$ are variables *possibly* modified and created by c_t. Write dependencies are added from c_t to each of the newly created VSes.

Fig 5 (right) shows an example AHG. Identifying access/modified variables is crucial for its construction, which we describe below.

ID Graph. The ID Graph aims to to detect changes at the reference level (in addition to values). For instance, conventional equality checks (e.g., based on serialization) will return True for "[a] == [b]" if a and b have the same value (e.g., a = [1] and b = [1]), whereas we ensure it returns True only if a and b refer to the same *object*, i.e., id(a)==id(b), where id is the object's unique ID. This is because for correct state replication, shared references (e.g. aliases) and inter-variable relationships must be captured precisely.

Identifying Accessed Variables. ElasticNotebook identifies both *directly accessed* variables (via AST [31] parsing) and *indirectly accessed* variables (with ID Graphs), as follows.

Direct Accesses: Cell code is analyzed with AST, stepping also into user-defined functions (potentially nested) to check for accesses to variables not explicitly passed in as parameters (e.g., global x).

Indirect Accesses: The object(s) reachable from a variable X may be accessed indirectly via another variable Y if X and Y reference common object(s) (e.g., when aliases exist, Fig 6a), which cannot be identified via parsing only. To recognize indirect accesses, we check the existence of overlaps between the ID Graphs of X and Y.

Our approach is conservative; that is, it may over-identify variables by including, for example, ones reachable from control flow branches that were not taken during cell executions. However, these false positives do not affect accuracy of state replication (§4.3).

Identifying Modified Variables. Variable modifications are identified using a combination of (1) object hashes and (2) ID Graphs.

Value Changes: ElasticNotebook identifies value modifications by comparing hashes (by xxHash [116]) before and after each cell execution while using deep copy as a fallback. If the deep copy fails (e.g., unserializable or uncomparable variables), we consider them
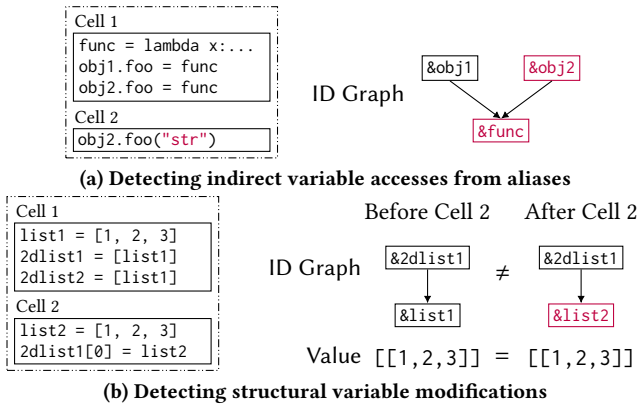
(a) Detecting indirect variable accesses from aliases

(b) Detecting structural variable modifications

Figure 6: Two uses of the ID Graph during AHG construction.

Figure 7: Two variables sharing references (in Fig 5). They must be migrated/recomputed together for the correct replication, serving as constraints to our opt problem (see §5.3).

to be modified-on-access using results from AST and ID Graph (§6.1). This may result in false positives; however, as previously mentioned, these false positives do not affect the accuracy.

Structural Changes: The ID Graph enables detecting structural changes (Fig 6b). After each cell execution, the current variables' ID Graphs are compared to the ones created before to identify reference swaps. In Fig 6b, while the value of 2dlist1 remains unchanged after execution after executing Cell 2, the memory address of its nested list has been changed, no longer referencing list1.

4.3 State Reconstruction with AHG

This section describes how we reconstruct variable(s). We focus on reconstructing the latest version of each variable, as defined in *active variable snapshot* (VS) in an AHG.

Definition 7. VS (x, t_i) is **active** if x is in the system (i.e., not deleted), and there is no VS (x, t_j) such that $t_i < t_j$.

An active VS, (x, t_i), represents the current version of x. For example, even if we checkpoint after c_{t_5} (in Fig 5), "(x, t_3)" is active since x was last modified by c_{t_3}. We denote the set of active VSes as $\mathcal{V}_a$.

Reconstruction Algorithm. Our goal is to identify the most efficient computation strategy for reconstructing one or more active variables. Note that we do not reconstruct non-active variables since they are not part of the current session state. In achieving this goal, the AHG allows us to avoid unnecessary cell executions (e.g., because their outcomes have been overwritten) and to learn proper execution orders. Moreover, this process can be extended to reconstruct a set of variables more efficiently than computing them one by one. while still ensuring correctness.

Specifically, to recompute VS (x, t), we traverse back to its ancestors in the AHG (e.g., using the breadth-first search), collecting all CEs into a list $req(x, t)$, until we find a *ground variable* for every path, where the ground variable is a variable whose value is available in the system, i.e., either another active VS or copied variable. By rerunning all the CEs in $req(x, t)$ in the order of their completion times, we can obtain the target VS (x, t). To extend this algorithm to multiple VSes, say $(x1, t_{x1})$, $(x2, t_{x2})$, and $(x3, t_{x3})$, we obtain req for each VS and union them into a merged set (that is, identical CEs collapse into one). By rerunning all the CEs in the merged set, we obtain all target VSes. Fig 5 shows an example. To recompute (x, t_3),

we rerun c_{t_3} which requires the previous version (x, t_1) as input, which in turn requires c_{t_1} to be rerun. Notably, it is not necessary to rerun c_{t_2} as its output z is available in the namespace. Finally, §6.1 discusses how this approach can recover even if some ground variables are unexpectedly unobtainable.

Why Only Use Active VSes? Theoretically, it is possible to use non-active variables as ground variables. That is, by preserving deleted/overwritten variables (e.g., in a cache), we may be able to speed up the recomputation of active variables [41, 114]. However, we don't consider this approach as many data science workloads are memory-hungry with large training data and model sizes. Still, there might be cases where we can speed up recomputation by storing small overwritten variables, which we leave as future work.

Correctness of Reconstruction. As stated in §2.3, the AHG is allowed to have false positives, meaning it may indicate a cell accessed/modified variables that were not actually accessed/modified. While the false positives have a performance impact, they do not affect the correctness of identification.

THEOREM 4.1. *Given the approximate AHG $\mathcal{G}$ of* ElasticNotebook *with false positives, and the true AHG $\mathcal{G}^*$, there is $req^*(x, t^*) \subseteq req(x, t)$ for any variable $x \in X$, where (x, t) and (x, t^*), req and req^* are the active VSes of x and reconstruction mapping functions defined on $\mathcal{G}$ and $\mathcal{G}^*$ respectively.*

That is, for any arbitrary variable x, while $req(x, t)$ may contain cell executions unnecessary for recomputing x, it will never miss any necessary cell executions (i.e., those in $req(x, t^*)$). The proof is presented in our technical report [71].

5 CORRECT & EFFICIENT REPLICATION

This section covers how ElasticNotebook computes an efficient and correct plan for state replication with the AHG and profiled metrics. We describe correctness requirements in §5.1, the cost model in §5.2, the optimization problem in §5.3, and our solution in §5.4.

5.1 Correctness Requirements

ElasticNotebook aims to *correctly* replicate session states. which we define the notion of in this section:

Definition 8. A replication of state X is **value-equivalent** if $\forall x \in X, x = new(x)$, where $new(x)$ is the value of x post-replication.

A value-equivalent replication preserves the value of each individual variable and is guaranteed by the correct identification of $req(x, t)$ for each variable x (§4.3). However, it is additionally important that shared references are preserved, as defined below.
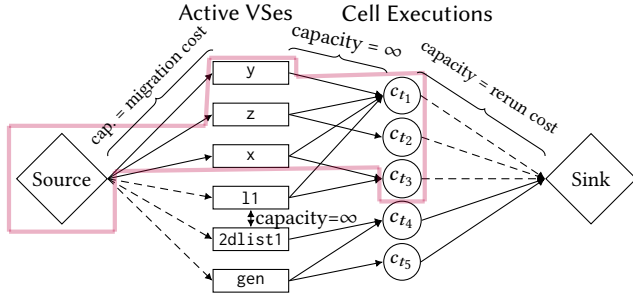
Figure 8: Running min-cut on the flow graph constructed from the AHG in Fig 5. The partition (red) defined by the minimum cut (dashed edges) determines the replication plan.

Definition 9. A value-equivalent replication of a session state $\mathcal{X}$ is additionally **isomorphic** if $\forall a, b,\ id(a) = id(b) \rightarrow id\_new(a) = id\_new(b)$, where a, b are arbitrary references (e.g., x[0], y.foo), and $id(a), id\_new(a)$ are the unique IDs (i.e., memory addresses) of the objects pointed to by a before and after replication.

ElasticNotebook defines replication as 'correct' only if it is isomorphic, requiring all shared references to be preserved: two references pointing to the same object pre-replication will still do so post-replication. That is, inter-object relations are identical (analogous to graph isomorphism). We describe how ElasticNotebook ensures isomorphic replication via its *linked variable constraint* in §5.3.

5.2 Cost Model

Our model captures the costs associated with (1) serializing variables, (2) writing byte data into storage (e.g., local SSD, cloud storage) and (3) rerunning cell executions. These costs are computed using the AHG and profiled system metrics.

Variable Migration Cost. *Migrating* a variable (from one session to another) includes serializing it to the checkpoint file, then loading it into a new session. Given a subset of variables to migrate $\mathcal{S} \subseteq \mathcal{X}$, the migration cost w_M can be expressed as follows:

$$w_M(\mathcal{S}) = \sum_{x \in \mathcal{S}} \alpha \times w_{store}(x) + w_{load}(x) \tag{1}$$

Where $w_{store}(x)$ and $w_{load}(x)$ are the time costs for serializing the value of x at checkpointing time into a file and unpacking into the new session, respectively. These times are estimated using the size of x and storage latency/bandwidth from ElasticNotebook's Profiler (§3.1). The time costs for unserializable variables are set to infinity. α is a coefficient for adjusting the time cost of storage; for example, if ElasticNotebook is to be invoked upon auto-suspension, α can be set to a low value to discount the user-perceived time of storing variables prior to completely suspending a session (as the user is likely away).

Variable Recomputation Cost. The Interceptor records cell runtimes during a session lifecycle (§3.1). Combined with the reconstruction mapping $req()$ for the AHG (§4.3), the cost w_R for recomputing a subset of variables $\mathcal{S} \subseteq \mathcal{X}$ can be defined as follows:

$$w_R(\mathcal{S}) = \sum_{c \in req(\mathcal{S})} w_{rerun}(c), \text{ where } req(\mathcal{S}) = \bigcup_{x \in \mathcal{S}} req(x, t) \tag{2}$$

where (x, t) is the active VS of x and $w_{rerun}(c) : C \rightarrow \mathbb{R}^+$ is the estimated time to rerun the CE c in the new session.

Replication Plan Cost. Using migration and recomputation costs (i.e., Eqs. (1) and (2)), the total cost w—with variables to migrate $\mathcal{S}$ and variables to recompute $\mathcal{X} - \mathcal{S}$—is expressed as:

$$w(\mathcal{S}) = w_M(\mathcal{S}) + w_R(\mathcal{X} - \mathcal{S}) \tag{3}$$

5.3 Optimization Problem for State Replication

The goal is to find the variables to migrate $\mathcal{S} \subseteq \mathcal{X}$ that minimizes the cost Eq. (3). To ensure isomorphic replication in consideration of variable inter-dependencies, additional constraints are added.

Constraint for Linked Variables. Two variables containing references to the same object (which we refer to as *linked variables*, e.g., l1 and 2dlist1 in Fig 7) must be either both migrated or recomputed, as migrating one and recomputing the other may result in their contained shared reference/alias being broken, as illustrated in Fig 7. Let the set of linked variable pairs be denoted as $\mathcal{L}$, then the constraint can be formally expressed as follows:

$$(x_1 \in \mathcal{S} \wedge x_2 \in \mathcal{S}) \vee (x_1 \notin \mathcal{S} \wedge x_2 \notin \mathcal{S}) \ \forall (x_1, x_2) \in \mathcal{L} \tag{4}$$

Problem definition. Using the cost model in Eq. (3) and the constraint in Eq. (4), we formally define the state replication problem:

Problem 1. Optimal State Replication

Input:
    1. AHG $\mathcal{G} = \{\mathcal{V} \cup C, \mathcal{E}\}$
    2. Migration cost function $w_M : 2^{\mathcal{X}} \rightarrow \mathbb{R}^+$
    3. Recompute cost function $w_R : 2^{\mathcal{X}} \rightarrow \mathbb{R}^+$
    4. Linked variables $\mathcal{L} \subseteq \mathcal{X} \times \mathcal{X}$

Output:
    A replication plan of subset of variables $\mathcal{S} \subseteq \mathcal{X}$ for which we migrate (and another subset $\mathcal{X} - \mathcal{S}$ which we recompute)

Objective:
    Minimize replication cost $w_M(\mathcal{S}) + w_R(\mathcal{X} - \mathcal{S})$

Constraint:
    Linked variables are either both migrated or recomputed: $(x_1, x_2 \in \mathcal{S}) \vee (x_1, x_2 \notin \mathcal{S}) \ \forall (x_1, x_2) \in \mathcal{L}$

The next section (§5.4) presents our solution to Prob 1.

5.4 Solving State Replication Opt. Problem

We solve Prob 1 by reducing it to a min-cut problem, with a *src-sink* flow graph constructed from the AHG such that each *src-sink* cut (a subset of edges, which, when removed from the flow graph, disconnects source s and sink t) corresponds to a replication plan $\mathcal{S}$, while the cost of the cut is equal to the replication cost $w_M(\mathcal{S}) + w_R(\mathcal{X} - \mathcal{S})$. Therefore, finding the minimum cost *src-sink* cut is equivalent to finding the optimal replication plan.

Flow Graph Construction. A flow graph $H := \{\mathcal{V}_H, \mathcal{E}_H\}$ and its edge capacity $\phi : \mathcal{E}_H \rightarrow \mathbb{R}^+$ are defined as follows:

- $\mathcal{V}_H = \mathcal{V}_a \cup C \cup \{src, sink\}$: $\mathcal{V}_a$ is active VSes, C is cell executions, and src and $sink$ are *dummy* source and sink nodes.
- $\forall x \in \mathcal{V}_a, (src, (x, t)) \in \mathcal{E}_H$ and $\phi(src, (x, t)) = w_M(x)$: We add an edge from the source to each active VS with a capacity equal to the migration cost of the variable.
- $\forall c \in C, (c, sink) \in \mathcal{E}_H$ and $\phi(c, sink) = w_{rerun}(c)$: We add an edge with capacity from each CE to the sink with a capacity equal to the rerun cost of the CE.

- $\forall c \in C, c \in req(x,t) \rightarrow ((x,t),c) \in \mathcal{E}_H$ and $\phi((x,t),c) = \infty$ and $(x,t) \in \mathcal{V}_a$: We add an edge with infinite capacity from an active VS (x,t) to a CE c if (x,t) must be recomputed.
- $\forall (x_1, x_2) \in \mathcal{L}, \ ((x_1,t_1) \leftrightarrow (x_2,t_2)) \in \mathcal{E}_H$ and $\phi((x_1,t_1) \leftrightarrow (x_2,t_2)) = \infty$: We add a *bi-directional* edge with an infinite capacity between each pair of active VSes corresponding to linked variables x_1 and x_2, e.g., l1 and 2dlist1.

The flow graph $\mathcal{H}$ for the AHG in Fig 5 is depicted in Fig 8.

Solution. We can now solve Prob 1 by running a *src—sink* min-cut algorithm (e.g., Ford-Fulkerson [29]) on H. The set of edges that form the *src—sink* min-cut (dashed edges), when removed, disconnects *src* from *sink*; therefore, it defines a partition (in red) of the nodes into nodes reachable from *src*, $\mathcal{V}_{H_{src}}$ and nodes unreachable from *src*, $\mathcal{V}_{H_{sink}}$. The partition defines the replication plan:

- $\mathcal{S} = \{x \mid (x,t) \in \mathcal{V}_{H_{sink}} \cap \mathcal{V}_a\}$ are the active variable snapshots (and thus variables) that we want to migrate; in the example, these variables are l1, 2dlist1, and gen.
- $\mathcal{V}_{H_{src}} \cap C$ are the CEs which we will rerun post-migration to recompute $X - \mathcal{S}$. In the example, these CEs are t_1, t_2, and t_3; when rerun, they recompute y, z, and x.[2]

By construction of $\mathcal{H}$, the sum of migration and recomputation costs of this configuration $w_M(\{x \mid (x,t) \in \mathcal{V}_{H_{sink}}) + w_R(C_a - (\mathcal{V}_{H_{src}} \cap C))$ is precisely the cost of the found *src—sink* min-cut.

6 IMPLEMENTATION AND DISCUSSION

This section describes ElasticNotebook's implementation details (§6.1) and design considerations (§6.2).

6.1 Implementation

Integrating with Jupyter. For seamless integration, ElasticNotebook's data layer is implemented using a magic extension [101], which is loaded into the kernel upon session initialization. The cell magic is automatically added to each cell (§2.2) to transparently intercept user cell executions, perform code analyses, create ID Graphs and object hashes, and so on.

Serialization Protocol. The Pickle protocol (e.g., __reduce__) is employed for (1) object serialization and (2) definition of reachable objects, i.e., an object y is reachable from a variable x if pickle(x) includes y. As Pickle is the de-facto standard (in Python) observed by almost all data science libraries (e.g., NumPy, PyTorch [28]), ElasticNotebook can be used for almost all use cases.

Handling Undeserializable variables. Certain variables can be serialized but contain errors in its *deserialization* instructions (which we refer to as *undeserializable* variables), and are typically caused by oversights in incompletely implemented libraries [14, 67]. While undetectable via serializability checks prior to checkpointing, ElasticNotebook handles them via fallback recomputation: if ElasticNotebook encounters an error while deserializing a stored variable during session restoration, it will trace the AHG to determine and rerun (only) necessary cell executions to recompute said variable, which is still faster than recomputing the session from scratch.

6.2 Design Considerations

Definition of Session State. In ElasticNotebook, the session state is formally defined as the contents of the user namespace dictionary (user_ns), which contains key-value pairs of variable names to their values (i.e., reachable objects). The session state does not include local/module/hidden variables, which we do not aim to capture.

Unobservable State / External Functions. Although the Pickle protocol is followed by almost all libraries, there could be lesser-known ones with incorrect serialization (e.g., ignoring data in C stack). To address this, ElasticNotebook can be easily extended to allow users to annotate cells/variables to inform our system that they must be recomputed for proper reconstruction. Mathematically, this is effectively setting their recomputation costs to infinity in Eq. (2).

Cell Executions with Side Effects. Certain cell executions may cause external changes outside a notebook session (e.g., filesystem) and may not be desirable to rerun (e.g., uploading items to a repository). Our prototype currently does not identify these side effects as our focus is read-oriented data science and analytics workloads. Nevertheless, our system can be extended at least in two ways to prevent them. *(1: Annotation)* We can allow users to add manual annotations to the cells that may cause side effects; then, our system will never re-run them during replications[3] *(2: Sandbook)* We can block external changes by replicating a notebook into a sandbox with altered file system access (e.g., chroot [73]) and blocked outgoing network (e.g., ufw [26]). The sandbox can then be associated with regular file/network accesses upon successful restoration.

Non-deterministic Operations. The replication has the same effect as rerunning the cells in the exact same order as they occurred in the past; thus, under the existence of nondeterministic operations (e.g., randint()), the reconstructed variables may have different values than the original ones. Users can avoid this by using annotations to inform ElasticNotebook to always copy them.

Library Version Compatibility. Accurate replication is ensured when external resources (e.g., installed modules, database tables) remain the same before and after the replication. While there are existing tools (i.e., pip freeze [83]) for reproducing computational environments on existing data science platforms (i.e., Jupyter Notebook, Colab) [1, 113], this work does not incorporate such tools.

7 EXPERIMENTAL EVALUATION

In this section, we empirically study the effectiveness of Elastic-Notebook's session replication. We make the following claims:

1. **Robust Replication:** Unlike existing mechanisms, ElasticNotebook is capable of replicating almost all notebooks. (§7.2)
2. **Faster Migration:** ElasticNotebook reduces session migration time to upscaled/downscaled machines by 85%–98%/84%-99% compared to rerunning all cells and is up to 2.07×/2.00× faster than the next best alternative, respectively. (§7.3)

---

[2]Rerunning t_3 also recomputes l1; however, it will be overwritten with the stored l1 in the checkpoint file following the procedure in §3.2. This is to preserve the link between l1 and 2dlist1.

[3]Replication may be unfeasible due to annotations, e.g., an unserializable variable requiring an cell execution annotated 'never-rerun' to recompute. ElasticNotebook can detect these cases as they have infinite min-cut cost (§5.4), upon which the user can be warned to delete the problematic variable to proceed with replicating the remaining (majority of) variables in the state.

Table 3: Summary of datasets for evaluation.

Dataset	Notebooks	Runtime (s)	Input data (MB)	Cell count
Kaggle [55]	35	178-31831	107−12,560	15−103
JWST [58]	5	25−323	2−109	21−44
Tutorial [105]	5	10−96	1−139	10−48
HW [13, 45, 64]	15	9−1203	16−439	11−160

3. **Faster Resumption:** ElasticNotebook reduces session restoration time by 94%–99% compared to rerunning all cells and is up to 3.92× faster than the next best alternative. (§7.4)

4. **Low Runtime Overhead:** ElasticNotebook incurs negligible overhead—amortized runtime and memory overhead of <2.5% and <10%, respectively. (§7.5)

5. **Low Storage Overhead:** ElasticNotebook's checkpoint sizes are up to 66% smaller compared to existing tools. (§7.6)

6. **Adaptability to System Environments:** ElasticNotebook achieves consistent savings across various environments with different network speeds and available compute resources. (§7.7)

7. **Scalability for Complex Notebooks**: ElasticNotebook's runtime and memory overheads remain negligible (<150ms, <4MB) even for complex notebooks with 2000 cells. (§7.8)

7.1 Experiment Setup

Datasets. We select a total of 60 notebooks from 4 datasets:

- Kaggle [55]: We select 35 popular notebooks on the topic of EDA (exploratory data analysis) + machine learning from Kaggle created by Grandmaster/Master-level users.
- JWST [58]: We select 5 notebooks on the topic of data pipelining from the example notebooks provided on investigating data from the James Webb Space Telescope (JWST).
- Tutorial [105]: We select 5 notebooks from the Cornell Virtual Workshop Tutorial. These notebooks are lightweight and introduce tools (i.e., clustering, graph analysis) to the user.
- Homework [13, 45, 64]: 15 in-progress notebooks are chosen from data science exercises. They contain out-of-order cell executions, runtime errors, and mistakes.

Table 3 reports our selected notebooks' dataset sizes and runtimes.

Methods. We evaluate ElasticNotebook against existing tools capable of performing session replication:

- RerunAll [100]: Save (only) cell code and outputs as an `ipynb` file. All cells are rerun to restore the session state.
- CRIU [17]: Performs a system-level memory dump of the process hosting the notebook session. The session state is restored by loading the memory dump and reviving the process.
- %Store [102]: A checkpointing tool that serializes variables one by one into storage. We use a modified version using Dill [39] instead of Pickle [37] for robustness.[4]
- DumpSession [38]: Unlike %Store, DumpSession packs the entire session state into one single file.

Ablation Study. We additionally compare against the following ablated implementations of ElasticNotebook:
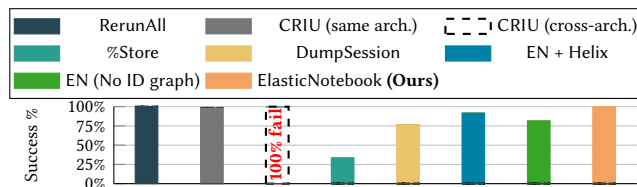

Figure 9: Ratio of correct replications. ElasticNotebook achieves 100% correctness, on par with full rerun (RerunAll).

Table 4: Existing work fails for these cases. Ours works.

Notebook(s)	Type	Description and purpose
NFL [87]	hashlib [30]	Dropdown list in plot
All 5 JWST notebooks [58]	mmap [35]	Helps avoid reading large file into memory
Arxiv [68] Plant [93]	generator [32]	Speedup iterable comprehension via lazy element generation

- EN + Helix [114]: We replace our min-cut solution with Helix, which does not consider linked variables (§5.3).
- EN (No ID graph): This method omits ID Graphs, relying only on AST analysis and object hashes for detecting variable accesses and modifications, respectively.

We consider these methods regarding replication correctness (§7.2) to gauge the impact of ignoring (1) the linked constraint and (2) implicit accesses and structural modifications, respectively.

Environment. We use an Azure Standard D32as v5 VM instance with 32 vCPUs and 128 GB RAM. For the migration experiment (§7.3), we migrate sessions from D32as to D64as/D16as with 64/16 vCPUs and 256/64 GB RAM for upscaling/downscaling, respectively. Input data and checkpoints are read/stored from/to an Azure storage with block blobs configuration (NFS). Its network bandwidth is 274 MB/s with a read latency of 175 μs.

Time measurement. We measure (1) *migration time* as the time from starting the checkpointing process to having the state restored (i.e., all variables declared into the namespace) in the destination session and (2) *restoration time* as the time to restore the state from a checkpoint file. We clear our cache between (1) checkpointing and restoring a notebook and (2) between subsequent runs.

7.2 Robust Session Replication

This section compares the robustness of ElasticNotebook's session replication to existing methods. We count the number of isomorphic (thus, *correct*) replications (§5.1) achieved with each method on the 60 notebooks and report the results in Fig 9.

ElasticNotebook correctly replicates all sessions, on par with full rerun from checkpoint file (which almost always works). Notably, it replicates 19, 25, and 2 notebooks containing unserializable variables, variable aliases, and undeserializable variables (§6.1), respectively. **DumpSession** and **%Store** fail on 19/60 notebooks containing unserializable variables (examples in Table 4); ElasticNotebook successfully replicates them as it can bypass the serialization of these variables through recomputation. **%Store** additionally fails on 21/60 notebooks (total 40/60) without unserializable variables but contain variable aliases (i.e., Timeseries [88] notebook, Cell 15,
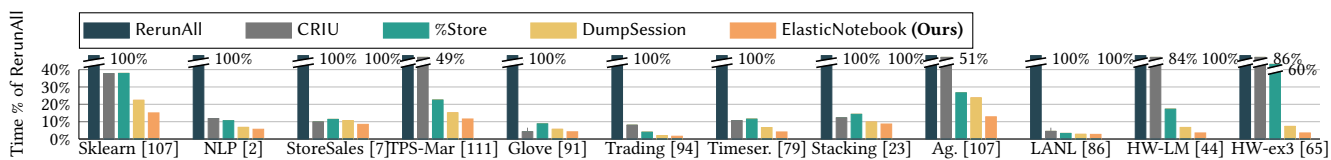
Figure 10: ElasticNotebook's session upscaling time (D32as v5 VM→D64as v5 VM) vs. existing tools. Times normalized w.r.t. RerunAll. ElasticNotebook speeds up migration by 85%-98% and is up to 2.07× faster than the next best alternative.
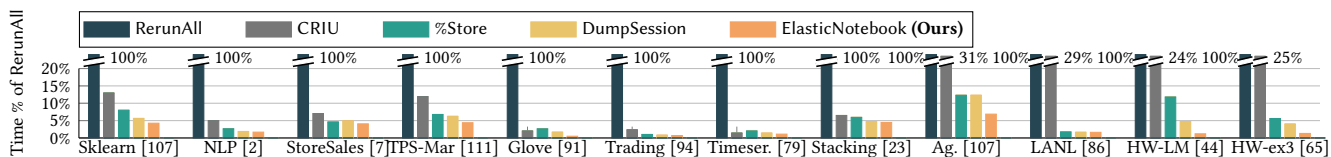

Figure 11: ElasticNotebook's session restoration time vs. existing tools. Times normalized w.r.t. RerunAll. ElasticNotebook speeds up session restore by 94%-99%, and is up to 3.92× faster compared to the next best alternative.

Table 5: Runtime and memory overhead of ElasticNotebook's workflow monitoring on selected notebooks.

	Sklearn	NLP	StoreSales	TPS-Mar	Glove	Trading	Timeser.	Stacking	Ag.	LANL	HW-LM	HW-ex3
Notebook runtime (s)	58.48	1016.77	283.06	178.42	696.64	687.54	204.10	788.54	269.40	1437.87	22.54	27.29
Total cell monitoring time (s)	1.26	4.30	0.81	1.34	6.43	0.46	0.60	2.13	3.08	0.19	0.50	0.09
Runtime overhead (%)	**2.14**	**0.42**	**0.28**	**0.78**	**0.92**	**0.07**	**0.29**	**0.27**	**1.14**	**0.01**	**2.21**	**0.32**
User Namespace memory usage (MB)	1021.45	325.82	6732.17	1558.52	347.16	1363.32	130.27	20211.51	5026.48	7641.19	31.28	19.06
ElasticNotebook memory usage (MB)	19.16	4.73	0.14	1.69	33.25	4.09	0.28	0.33	0.06	0.14	0.99	0.47
Memory overhead (%)	**1.88**	**1.45**	**0.002**	**0.11**	**9.58**	**0.30**	**0.21**	**0.002**	**0.001**	**0.001**	**3.16**	**2.45**

linked components of a Matplotlib [103] plot—f, fig, ax); it serializes variables into individual files, which breaks object references and isomorphism. ElasticNotebook's linked variables constraint (§5.3) ensures that it does not do so. **ElasticNotebook + Helix** fails to correctly replicate 5/60 notebooks containing variable aliases due to its lacking of the linked variable constraint. **EN (No ID graph)** fails to correctly replicate 11/60 sessions due to it missing indirect accesses and structural modifications causing incorrect construction of the AHG, which in turn leads it to recompute some variables value-incorrectly. **CRIU** fails on one notebook [89] which contains an invisible file; however, unlike ElasticNotebook's failures, this failure is currently a fundamental limitation in CRIU [16].

Robust Migration across System Architectures. We additionally performed session replication from our D32as VM (x64 architecture) to a D32pds V5 VM instance (arm64 architecture). The CRIU images cannot be replicated across machines with different architectures. In contrast, ElasticNotebook does not have such a limitation.

7.3 Faster Session Migration

This section compares the efficiency of ElasticNotebook's session migration to existing methods. We choose 10 notebooks with no unserializable variables (otherwise, existing methods fail) to compare the end-to-end session migration time achieved by different methods. We report upscaling and downscaling results in Fig 10 and our technical report [71], respectively.

The design goal of ElasticNotebook is to reduce session replication time through balancing variable storage and recomputation, which is successfully reflected as follows. ElasticNotebook is able

to reduce session migration time to the upscaled/downscaled VMs by 85%–98%/84%-99% compared RerunAll. Compared to DumpSession, %Store, and CRIU, which store all variables in the checkpoint file, ElasticNotebook upscales/downscales up to 2.07×/2.00× faster than the best of the three. DumpSession, while being the next best alternative for upscaling/downscaling on 8/9 notebooks, falls short in robustness as demonstrated in §7.2. %Store's individual reading and writing of each variable results in high overhead from multiple calls to the NFS for each migration. CRIU is the slowest non-rerun method for upscaling/downscaling on 6/7 notebooks, due to the size of its memory dump (higher I/O during migration) being up to 10× larger compared to checkpoint files from native tools (§7.6).

7.4 Faster Session Restoration

In this section, we compare the efficiency of ElasticNotebook's session restoration to existing methods. We generate checkpoint files using each method, then compare the time taken to restore the session from the checkpoint files on the 10 notebooks from §7.3. For ElasticNotebook, we set the coefficient α to 0.05 (§5.2) to emphasize session restoration time heavily.

We report the results in Fig 10. ElasticNotebook's restoration time is 94%–99% faster compared to full rerun, while being 3.92× faster than the next best alternative. ElasticNotebook is capable of fast restorations as it adapts to the new optimization objective unlike the baselines: for example, on the Sklearn [107] notebook, instead of rerunning cell 3 (df = pd.read_csv(...)) to re-read the dataframe df into the session as in the migration-centric plan, the restoration-centric plan opts to store df instead: despite the sum of serialization and deserialization times of df being greater
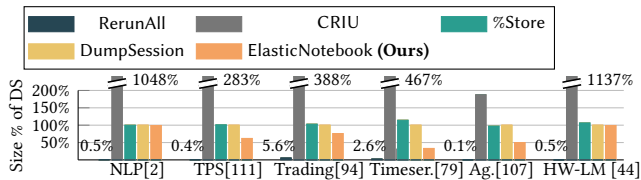
Figure 12: ElasticNotebook's checkpoint file size vs. existing tools. Times normalized w.r.t. output from DumpSession. ElasticNotebook's checkpoint file size is up to 67% smaller compared to those from existing tools (excluding RerunAll).

than the re-reading time with pd.read_csv (6.19s + 1.17s > 5.5s), the deserialization time by itself is less than the re-reading time (1.17s < 5.5s); hence, storing df is the optimal choice.

7.5 Low Runtime Overhead

This section investigates the overhead of ElasticNotebook's notebook workflow monitoring. We measure ElasticNotebook's total time spent in pre/post-processing steps before/after each cell execution for updating the AHG and cell runtimes (*Total cell monitoring time*), and total storage space taken to store the AHG, ID Graphs, and hashes at checkpoint time (ElasticNotebook *memory usage*).

We report the results in Table 5. ElasticNotebook's cell monitoring incurs a maximum and median runtime overhead of (only) 2.21% and 0.6%; thus, ElasticNotebook can be seamlessly integrated into existing workflow. ElasticNotebook is similarly memory-efficient as its stored items (AHG, ID Graphs, and hashes) are all metadata largely independent of the size of items in the session: the median memory overhead is 0.25%, with the worst case being 9.58%.

Fine-grained Analysis. To study the per-cell time and memory overheads during experimental notebook usage, we examined three notebooks from Homework category to confirm the maximum time and memory overheads were 92ms and 4.9MB, respectively. We report details in our technical report [71].

7.6 Lower Storage Overhead

This section measures the storage cost of ElasticNotebook's checkpoint files: we compare the migration-centric checkpoint file sizes from ElasticNotebook and those from other baseline methods.

We report select results in Fig 12. ElasticNotebook's AHG allows it to choose between storing and recomputing each variable, reflected in ElasticNotebook's checkpoint files being up to 67% smaller compared to DumpSession's. For example, on the Agriculture [88] notebook, ElasticNotebook recomputes the train-test splits of the input dataframes X and Y (Cell 5, x_train, x_test,... = train_test_split(X, Y)) instead of storing them in the checkpoint file: this saves considerable storage space (2.5GB) in addition to speeding up migration. Conversely, CRIU's checkpoint file sizes can be 10× larger than ElasticNotebook's as it additionally dumps memory occupied by the Python process itself and imported modules, no matter necessary or not, into the checkpoint file. Output sizes from RerunAll (i.e., notebook metadata size consisting of cell code and outputs) are provided for comparison. While metadata are significantly smaller than checkpoint files, the storage benefit is offset by significantly slower session recovery times (§7.4).
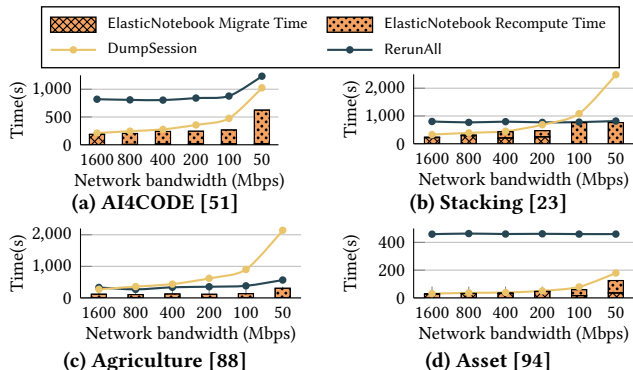

Figure 13: ElasticNotebook adapts to different environments for its replication plan. The lower the network bandwidth, the more variables are recomputed.
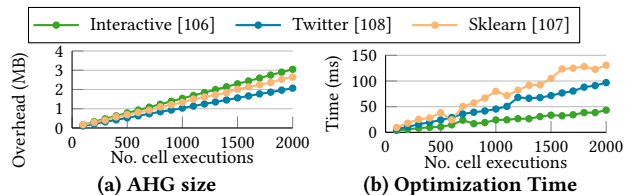

Figure 14: Scalability of ElasticNotebook with cell execution count. The size of AHG increases linearly. Replication plan optimization time increases sub-linearly.

7.7 Performance Gains Across Environments

This section demonstrates ElasticNotebook's operation in environments with varying specifications. We parameter sweep the NFS network bandwidth via rate limiting [10] and compare the migration time of ElasticNotebook, DumpSession, and RerunAll.

We report the results in Fig 13. ElasticNotebook's balancing of variables storage and recomputation ensures that it is always at least as fast as the faster of DumpSession and RerunAll. Notably, ElasticNotebook can adapt to the relative availability between network bandwidth and compute power: as the bandwidth decreases, the replication plan is changed accordingly to migrate more variables through recomputation rather than storage. For example, on the Stacking [23] notebook, at regular bandwidth (>400Mbps), ElasticNotebook's replication plan includes migrating most of the session state, opting only to recompute certain train/test splits (i.e., Cell 37, Y_train, Y_validation). At <400 Mbps, ElasticNotebook modifies its plan to recompute instead of store a computationally expensive processed dataframe (Cell 39, latest_record). At <100 Mbps, ElasticNotebook modifies its plan again to only store the imported class and function definitions (i.e., XGBRegressor, mean_squared_error in Cell 1) while recomputing the rest of the notebook.

7.8 Scaling to Complex Workloads

In this section, we test the scalability of ElasticNotebook's session replication on complex notebook sessions with a large number of cell executions and re-executions. Specifically, we choose 3 tutorial notebooks, on which we randomly re-execute cells and measure

the (1) size of ElasticNotebook's AHG and (2) optimization time for computing the replication plan at up to 2000 cell re-executions[5].

We report the results in Fig 14. The memory consumption of ElasticNotebook's AHG exhibits linear scaling vs. the number of cell executions reaching only <4MB at 2000 cell re-executions, which is negligible compared to the memory consumption of the notebook session (>1GB) itself. ElasticNotebook's optimization time for computing the replication plan similarly exhibits linear scaling, reaching a negligible <150ms at 2000 cell re-executions: ElasticNotebook's chosen algorithm for solving min-cut, Ford-Fulkerson [29], has time complexity $O(Ef)$, where E is the number of edges in the AHG and f is the cost of the optimal replication plan: The former scales linearly while the latter is largely constant.

8 RELATED WORK

Intermediate Result Reuse in Data Science. The storage of intermediate results has been explored in various contexts in Data Science due to the incremental and feed-forward nature of tasks, which allows outputs from prior operations to be useful for speeding up future operations [41, 53, 63, 72, 109, 114, 115, 120]. Examples include caching to speed up model training replay for ML model diagnosis [41, 109], caching to speedup materialized view refresh workloaods [72], caching to speed up anticipated future dataframe operations in notebook workflows [115], and storage of cell outputs to facilitate graphical exploration of the notebook's execution history for convenient cell re-runs [53, 63]. There are related works [114, 120] which algorithmically explore the most efficient way to (re)compute a state given currently stored items; compared to our work, while Helix [114] similarly features balancing loading and recomputation, its model lacks the linked variable constraint which may result in silently incorrect replication if directly applied to the computational notebook problem setting.

Data-level Session Replication. Session replication on Jupyter-based platforms can be performed with serialization libraries [33, 34, 37, 39, 77]. There exists a variety of checkpoint tools built on these serialization libraries: IPython's %Store [102] is a Pickle-based [37] interface for saving variables to a key-value store; however, it breaks object references as linked variables are serialized into separate files. The Dill-based [39] DumpSession [38] correctly resolves object references, yet it still fails if the session contains unserializable objects. Jupyter's native checkpointing mechanism [100] only saves cell metadata and often fails to exactly restore a session due to the common presence of hidden states. Compared to existing data-level tools, session replication with ElasticNotebook is both more efficient and robust: the Application History Graph enables balancing state storage and recomputation, which achieves considerable speedup while avoiding failure on unserializable objects.

System-Level Session Replication. Session replication can similarly be performed using system-level checkpoint/restart (C/R) tools, on which there is much existing work [6, 6, 8, 11, 22, 50, 70, 76, 95]. Applicable tools include DMTCP [3] and CRIU [17]; recently, CRUM [42] and CRAC [59] have explored extending C/R to CUDA applications. Elsa [62] integrates CRIU with JupyterHub to enable

C/R of JupyterHub servers. Compared to ElasticNotebook, system-level tools are less efficient and robust due to their large memory dump sizes and limited cross-platform portability, respectively.

Lineage Tracing. Lineage tracing has seen extensive use in state management to enable recomputation of data for more efficient storage of state or fault tolerance [15, 49, 69, 81, 104, 109, 118] Recently, the usage of data lineage in computational notebooks has enabled notebook replay [75], recommending notebook interactions [74], reproducing notebook containers [1], and program slicing [49, 52, 63, 82, 92]. This work adopts lineage tracing techniques to capturing inter-variable dependencies (the Application History Graph) for optimization; to the best of our knowledge, existing works on Python programs focus on capturing value changes (via equality comparisons); however, our work additionally captures *strucal changes* via the ID graph, which is crucial for preserving variable aliases and avoiding silent errors during state replication.

Replicating Execution Environment. An identical execution environment may be necessary for session replication on a different machine. There is some recent work exploring environment replication for Jupyter Notebook via containerizing input files and modules [1, 113]. While useful in conjunction with ElasticNotebook, we consider these works to be largely orthogonal.

Notebook Parameterization and Scripts. There exists works on executing notebooks in parameterized form for systematic experimentation [12, 97, 98]. While ElasticNotebook is designed for use within interactive notebook interfaces, it is similarly applicable for the migration of parameterized notebook execution results.

9 CONCLUSION

In this work, we have proposed ElasticNotebook, a new computational notebook system that newly offers elastic scaling and checkpointing/restoration. To achieve this, ElasticNotebook introduces a transparent data management layer between the user interface and the underlying kernel, enabling robust, efficient, and platform-independent state replication for notebook sessions. Its core contributions include (1) low-overhead, on-the-fly application history construction and (2) a new optimization for combining copying and re-computation of variables that comprise session states. We have demonstrated that ElasticNotebook can reduce upscaling, downscaling, and restoration times by 85%-98%, 84%-99%, and 94%-99%, respectively, on real-world data science notebooks with negligible runtime and memory overheads of <2.5% and <10%, respectively.

In the future, we plan to achieve higher efficiency and usability by tracing state changes at a finer level. Specifically, we will introduce *micro-cells* to capture code blocks inside a cell that repeatedly runs (e.g., `for`-loop for machine learning training). Then, the system will automatically store intermediate models (along with other metadata) that will enable live migration and checkpointing/restoration for long-running cell executions.

---

[5]This is twice the length of the longest observed notebook on Kaggle [48].

REFERENCES

[1] Raza Ahmad, Naga Nithin Manne, and Tanu Malik. 2022. Reproducible Notebook Containers using Application Virtualization. In *18th IEEE International Conference on e-Science, e-Science 2022, Salt Lake City, UT, USA, October 11-14, 2022.* IEEE, Salt Lake City, UT, USA, 1–10. https://doi.org/10.1109/eScience55777.2022.00015

[2] AndresHG. 2021. NLP, GloVe, BERT, TF-IDF, LSTM... Explained. www.kaggle.com/code/andreshg/nlp-glove-bert-tf-idf-lstm-explained/notebook.

[3] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE, IEEE Computer Society, Rome, Italy, 1–12.

[4] Microsoft Azure. 2023. Azure ML Studio. learn.microsoft.com/en-us/azure/machine-learning/how-to-run-jupyter-notebooks.

[5] Microsoft Azure. 2023. Microsoft Azure pay-as-you-go. azure.microsoft.com/en-us/pricing/purchase-options/pay-as-you-go/.

[6] Anju Bala and Inderveer Chana. 2012. Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues (IJCSI)* 9, 1 (2012), 288.

[7] Ekrem Bayar. 2022. Store Sales TS Forecasting - A Comprehensive Guide. www.kaggle.com/code/ekrembayar/store-sales-ts-forecasting-a-comprehensive-guide/notebook.

[8] Mohammad Riyaz Belgaum, Safeeullah Soomro, Zainab Alansari, and Muhammad Alam. 2019. Cloud Service ranking using Checkpoint based Load balancing in real time scheduling of Cloud Computing. *CoRR* abs/1905.03093 (2019), 667–676. arXiv:1905.03093 arxiv.org/abs/1905.03093

[9] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.* 13, null (feb 2012), 281–305.

[10] Simon Séhier Bert Hubert, Jacco Geul. 2020. WonderShaper. github.com/magnific0/wondershaper.

[11] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Weizhong Qiang, and Gang Hu. 2010. SHelp: Automatic Self-Healing for Multiple Application Instances in a Virtual Machine Environment. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing (CLUSTER '10)*. IEEE Computer Society, USA, 97–106. https://doi.org/10.1109/CLUSTER.2010.18

[12] Supawit Chockchowwat, Zhaoheng Li, and Yongjoo Park. 2023. Transactional Python for Durable Machine Learning: Vision, Challenges, and Feasibility. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning* (Seattle, WA, USA) *(DEEM '23)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3595360.3595855

[13] Chhaya Choudhary. 2023. Machine Learning and Deep learning Notebooks. github.com/chhayac/Machine-Learning-Notebooks.

[14] Bokeh Contributors. 2023. Bokeh - Interaction. docs.bokeh.org/en/latest/docs/user_guide/interaction.html.

[15] Iván Cores, Gabriel Rodríguez, Mará J Martín, Patricia González, and Roberto R Osorio. 2013. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Computing* 31 (2013), 163–185.

[16] CRIU. 2023. CRIU - Invisible file. criu.org/Invisible_files.

[17] CRIU. 2023. Linux CRIU. criu.org/Main_Page.

[18] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2024–2027.

[19] JupyterHub Idle Culler. 2023. JupyterHub Idle Culler. github.com/jupyterhub/jupyterhub-idle-culler.

[20] Renato Luiz de Freitas Cunha, Lucas Correia Villa Real, Renan Souza, Bruno Silva, and Marco A. S. Netto. 2021. Context-aware Execution Migration Tool for Data Science Jupyter Notebooks on Hybrid Clouds. In *17th IEEE International Conference on eScience, eScience 2021, Innsbruck, Austria, September 20-23, 2021*. IEEE, Innsbruck, Austria, 30–39. https://doi.org/10.1109/eScience51609.2021.00013

[21] Nvidia Developer. 2023. Nvidia - CUDA. developer.nvidia.com/cuda-toolkit.

[22] Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello. 2013. Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 64, 12 pages. https://doi.org/10.1145/2503210.2503217

[23] DimitreOliveira. 2019. Model stacking, feature engineering and EDA. www.kaggle.com/code/dimitreoliveira/model-stacking-feature-engineering-and-eda/notebook.

[24] Docker. 2013-2023. Docker documentation - Swarm mode overview. docs.docker.com/engine/swarm/.

[25] Cody Dunne, Nathalie Henry Riche, Bongshin Lee, Ronald Metoyer, and George Robertson. 2012. GraphTrail: Analyzing Large Multivariate, Heterogeneous Networks While Supporting Exploration History. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI*

'12). Association for Computing Machinery, New York, NY, USA, 1663–1672. https://doi.org/10.1145/2207676.2208293

[26] dwd-daniel. 2022. UncomplicatedFirewall. wiki.ubuntu.com/UncomplicatedFirewall.

[27] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. IDEBench: A Benchmark for Interactive Data Exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1555–1569. https://doi.org/10.1145/3318464.3380574

[28] Lightning AI et al. 2018. PyTorch ModelCheckpoint. pytorch-lightning.readthedocs.io/en/stable/api/pytorch_lightning.callbacks.ModelCheckpoint.html.

[29] LRDR FORD-FULKERSON. 1962. Flows in Networks.

[30] Python Software Foundation. 2023. Python Hashlib. docs.python.org/3/library/hashlib.html.

[31] Python Software Foundation. 2023. Python - AST. docs.python.org/3/library/ast.html.

[32] Python Software Foundation. 2023. Python - Generators. wiki.python.org/moin/Generators.

[33] Python Software Foundation. 2023. Python JSON. docs.python.org/3/library/json.html.

[34] Python Software Foundation. 2023. Python Marshal. docs.python.org/3/library/marshal.html.

[35] Python Software Foundation. 2023. Python Mmap. docs.python.org/3/library/mmap.html.

[36] Python Software Foundation. 2023. Python Object Reduction. docs.python.org/3/library/pickle.html#object.__reduce__.

[37] Python Software Foundation. 2023. Python Pickle Documentation. docs.python.org/3/library/pickle.html.

[38] The Uncertainty Quantification Foundation. 2023. Dill dump session. dill.readthedocs.io/en/latest/dill.html.

[39] The Uncertainty Quantification Foundation. 2023. Dill - PyPi. pypi.org/project/dill/.

[40] Tian Gao. 2020. Python Watchpoints. pypi.org/project/watchpoints/.

[41] Rolando Garcia, Eric Liu, Vikram Sreekanti, Bobby Yan, Anusha Dandamudi, Joseph E. Gonzalez, Joseph M. Hellerstein, and Koushik Sen. 2020. Hindsight Logging for Model Training. *Proc. VLDB Endow.* 14, 4 (dec 2020), 682–693. https://doi.org/10.14778/3436905.3436925

[42] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman. 2018. CRUM: Checkpoint-Restart Support for CUDA's Unified Memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 302–313. https://doi.org/10.1109/CLUSTER.2018.00047

[43] GDB. 2022. GDB Watchpoints. sourceware.org/gdb/download/onlinedocs/gdb/Set-Watchpoints.html.

[44] Aurélien Geron. 2023. Chapter 4 – Training Models. github.com/ageron/handson-ml3/blob/main/04_training_linear_models.ipynb.

[45] Aurélien Geron. 2023. Machine Learning Notebooks, 3rd edition. github.com/ageron/handson-ml3.

[46] Google. 2023. Google Colab. colab.research.google.com/.

[47] Google. 2023. Google Colab pay-as-you-go. colab.research.google.com/signup.

[48] Google and X. 2022. Google AI4Code – Understand Code in Python Notebooks. www.kaggle.com/competitions/AI4Code.

[49] Philip J. Guo and Margo Seltzer. 2012. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance* (Boston, MA) *(TaPP'12)*. USENIX Association, USA, 7.

[50] HAProxy. 2023. HAProxy. www.haproxy.org/.

[51] Sanskar Hasija. 2022. AI4Code Detailed EDA. www.kaggle.com/code/odins0n/ai4code-detailed-eda.

[52] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[53] Inc. Hex Technologies. 2023. Hex 2.0: Reactivity, Graphs, and a little bit of Magic. hex.tech/blog/hex-two-point-oh/.

[54] IBM. 2022. IBM Watson Studio Service. www.ibm.com/docs/en/knowledge-accelerators/1.0.0?topic=catalog-jupyter-notebook.

[55] Kaggle Inc. 2023. Kaggle. www.kaggle.com/.

[56] Kaggle Inc. 2023. Kaggle Forums - Product Feedback. www.kaggle.com/discussions/product-feedback.

[57] Kaggle Inc. 2023. Kaggle Notebook Specifications. www.kaggle.com/docs/notebooks#technical-specifications.

[58] Space Telescope Science Institute. 2023. JWST Data Analysis Example. jwst-docs.stsci.edu/jwst-post-pipeline-data-analysis/data-analysis-example-jupyter-notebooks.

[59] Twinkle Jain and Gene Cooperman. 2020. CRAC: Checkpoint-Restart Architecture for CUDA with Streams and UVM. arXiv:2008.10596 [cs.DC]

[60] Jeremiah W. Johnson. 2020. Benefits and Pitfalls of Jupyter Notebooks in the Classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education* (Virtual Event, USA) *(SIGITE '20)*. Association for Computing Machinery, New York, NY, USA, 32–37. https://doi.org/10.1145/3368308.3415397

[61] Project Jupyter. 2023. Jupyter Notebook. jupyter.org/.

[62] Mario Juric, Steven Stetzler, and Colin T. Slater. 2021. Checkpoint, Restore, and Live Migration for Science Platforms. arXiv:2101.05782 [astro-ph.IM]

[63] David Koop and Jay Patel. 2017. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance* (Seattle, WA) *(TaPP'17)*. USENIX Association, USA, 17.

[64] Martin Krasser. 2023. Machine learning notebooks. github.com/krasserm/machine-learning-notebook.

[65] Martin Krasser. 2023. Multi-class Classification. github.com/krasserm/machine-learning-notebooks/blob/master/ml-ex3.ipynb.

[66] Kubernetes. 2023. Kubernetes. kubernetes.io/.

[67] SFU Database System Lab. 2022. Dataprep - Low-Code Data Preparation. dataprep.ai/.

[68] Colin Lagator. 2020. Arxiv Data Processing. www.kaggle.com/code/colinlagator/arxiv-data-processing.

[69] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2670979.2670985

[70] Yawei Li and Zhiling Lan. 2010. FREM: A fast restart mechanism for general checkpoint/restart. *IEEE Trans. Comput.* 60, 5 (2010), 639–652.

[71] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks (Technical Report). arxiv.org/abs/2309.11083.

[72] Z. Li, X. Pi, and Y. Park. 2023. S/C: Speeding up Data Materialization with Bounded Memory. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1981–1994. https://doi.org/10.1109/ICDE55515.2023.00393

[73] Arch Linux. 2023. chroot. wiki.archlinux.org/title/chroot.

[74] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-Grained Lineage for Safer Notebook Interactions. *Proc. VLDB Endow.* 14, 6 (feb 2021), 1093–1101. https://doi.org/10.14778/3447689.3447712

[75] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: Multiversion Replay with Ordered Checkpoints. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1297–1310. https://doi.org/10.14778/3514061.3514075

[76] Anjali D Meshram, AS Sambare, and SD Zade. 2013. Fault tolerance model for reliable cloud computing. *International Journal on Recent and Innovation Trends in Computing and Communication* 1, 7 (2013), 600–603.

[77] Inc. MongoDB. 2023. BSON. pymongo.readthedocs.io/en/stable/api/bson/index.html.

[78] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[79] Rob Mulla. 2020. Time Series forecasting with Prophet. www.kaggle.com/code/robikscube/time-series-forecasting-with-prophet.

[80] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[81] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-Grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1426–1439. https://doi.org/10.1145/3448016.3452788

[82] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. NoWorkflow: A Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1841–1844. https://doi.org/10.14778/3137765.3137789

[83] The pip developers. 2023. Pip Freeze. pip.pypa.io/en/stable/cli/pip_freeze/.

[84] Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. 2022. Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1279–1287.

[85] PBC Posit Software, PBC formerly RStudio. 2023. Posit RStudio. posit.co/.

[86] Gabriel Preda. 2019. LANL Earthquake EDA and Prediction. www.kaggle.com/code/gpreda/lanl-earthquake-eda-and-prediction.

[87] Kalilur Rahman. 2022. NFL Data Bowl 2023 - Offensive Plays EDA. www.kaggle.com/code/kalilurrahman/nfl-data-bowl-2023-offensive-plays-eda/notebook.

[88] DS Rahul. 2020. Agricultural Drought Prediction. www.kaggle.com/code/dsrhul/agricultural-drought-prediction.

[89] Mani Raj. 2022. Amex Dataset. www.kaggle.com/code/manirajheerakar/amex-dataset.

[90] Amazon Web Services. 2023. AWS JupyterHub. docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-jupyterhub.html.

[91] Shahules. 2022. Basic EDA,Cleaning and GloVe. www.kaggle.com/code/shahules/basic-eda-cleaning-and-glove/notebook.

[92] Shreya Shankar, Stephen Macke, Sarah Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, compact, and rapid program slicing for notebooks. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4038–4047.

[93] shreyas thorat30. 2023. Plant disease classification SDP. www.kaggle.com/code/shreyasthorat30/plant-disease-classification-sdp.

[94] Andrey Shtrauss. 2022. Building an Asset Trading Strategy. www.kaggle.com/code/shtrausslearning/building-an-asset-trading-strategy/notebook.

[95] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. 2009. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 37–48.

[96] StackOverflow. 2019. Colab Session Timeout. stackoverflow.com/questions/57113226/how-can-i-prevent-google-colab-from-disconnecting.

[97] Jupyter Development Team. 2023. nbconvert - Jupyter Notebook Conversion. github.com/jupyter/nbconvert.

[98] Nteract Team. 2023. Welcome to papermill. papermill.readthedocs.io/en/latest/.

[99] The IPython Development Team. 2023. IPython Interactive Computing. ipython.org/.

[100] The IPython Development Team. 2023. Jupyter checkpoint. jupyter-server.readthedocs.io/en/latest/developers/contents.html.

[101] The IPython Development Team. 2023. Jupyter Magics Class. ipython.readthedocs.io/en/stable/config/custommagics.html.

[102] The IPython Development Team. 2023. Jupyter store magic. ipython.readthedocs.io/en/stable/config/extensions/storemagic.html.

[103] The Matplotlib Development Team. 2023. Matplotlib. matplotlib.org/.

[104] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (2018), 847–872.

[105] Cornell University. 2021. Cornell Virtual Workshop Tutorial Notebooks. github.com/CornellCAC/CVW_PyDataSci2.

[106] Cornell University. 2021. Investigating Tweet Timelines Using Interactive Bokeh Scatterplots. github.com/CornellCAC/CVW_PyDataSci2/blob/master/code/interactive_visualization_with_bokeh.ipynb.

[107] Cornell University. 2021. SKLearn Tweet Classification. github.com/CornellCAC/CVW_PyDataSci2/blob/master/code/sklearn_tweet_classification.ipynb.

[108] Cornell University. 2021. Twitter Networks. github.com/CornellCAC/CVW_PyDataSci2/blob/master/code/twitter_networks.ipynb.

[109] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1285–1300. https://doi.org/10.1145/3183713.3196934

[110] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[111] Devlikamov Vlad. 2022. [TPS-Mar] Fast workflow using scikit-learn-intelex. www.kaggle.com/code/lordozvlad/tps-mar-fast-workflow-using-scikit-learn-intelex/notebook.

[112] Eric-Jan Wagenmakers and Simon Farrell. 2004. AIC model selection using Akaike weights. *Psychonomic bulletin & review* 11, 1 (2004), 192–196.

[113] 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* (Boston, MA, USA) *(PEARC '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3491418.3530296

[114] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (dec 2018), 446–460. https://doi.org/10.14778/3297753.3297763

[115] Doris Xin, Devin Petersohn, Dixin Tang, Yifan Wu, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time. *IEEE Data Eng. Bull.* 44, 1 (2021), 66–78. sites.computer.org/debull/A21mar/p66.pdf

[116] xxHash. 2023. xxHash - Extremely fast non-cryptographic hash algorithm. github.com/Cyan4973/xxHash.

[117] Yandex. 2023. CatBoost - open-source gradient boosting library. catboost.ai/.

[118] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.

[119] Emanuel Zgraggen, Robert Zeleznik, and Steven M. Drucker. 2014. Panoramic-Data: Data Analysis through Pen & Touch. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2112–2121. https://doi.org/10.1109/TVCG.2014.2346293

[120] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 1–32.