ElasticNotebook: Enabling Live Migration for Computational Notebooks

Zhaoheng Li*, Pranav Gor*, Rahul Prabhu*, Hui Yu*, Yuzhou Mao†, Yongjoo Park*
University of Illinois at Urbana-Champaign* University of Michigan†
{zl20,gor2,rprabhu5,huiy3,yongjoo}@illinois.edu,yuzhom@umich.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, ElasticNotebook, 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%.

PVLDB Reference Format:

1 INTRODUCTION
Computational notebooks1 (e.g., Jupyter [62, 100], Rstudio [85]) are widely used in data science and machine learning for interactive tutorials [61], data exploration [18, 25, 120], visualization [27], model

Figure 1: Our transparent data layer (in the middle) enables robust, efficient, and platform-independent live migration.

1In 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.
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 a 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 reconstitute 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 ($\S$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 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 [39, 103]; 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, 60] 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, 115] and lineage tracing [53, 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, 114] and scalable data science [78, 80, 116]. 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 upscaling, downscaling, and restore times by 85%-98%, 84%-99%, and 94%-99%, respectively. Overheads are negligible (<2.5% runtime). (§7)

## 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 stateless 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, 63], allowing us to migrate a live session to the machines with the right equipment/resources (e.g., GPU [21], specific architectures [118]). For interruption-free scaling, we can copy data $D$ from a source...
We discuss the pros and cons of several different approaches to when the cell is executed. With this, we add pre-/post-processing 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.

**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× [114]). With the ability to create data D sufficient for reconstructing the current session state, we can persist D prior to either manual or automated suspension [19, 46, 58], 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 [57, 96]. In this process, we want to restore the session as quickly as possible by minimizing the time it takes for downloading 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 [53, 92]; they maintain deep copies for objects in the session state, which 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Store Vars</th>
<th>Rerun cells</th>
<th>Migration Cost</th>
<th>Restore Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store all</td>
<td>N/A</td>
<td>All</td>
<td>3+1×2+10=16</td>
<td>3+1×2+10=16</td>
</tr>
<tr>
<td>Store all</td>
<td>All</td>
<td>N/A</td>
<td>10×6+2×4+2×20=66</td>
<td>2×6+2×4+2×2=20</td>
</tr>
<tr>
<td>Fast-migrate</td>
<td>df,plot</td>
<td>df_train</td>
<td>3+1×4+2×4.2=16</td>
<td>3+1×2+1×4.2=16</td>
</tr>
<tr>
<td>Fast-restore</td>
<td>df,plot</td>
<td>df,plot</td>
<td>10×1+4×2×11.6=31</td>
<td>2×1+2×1=3.2</td>
</tr>
</tbody>
</table>

**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 types of nodes: variables and transformations. A transformation node connects input variables to output variables (see an example in Fig 3). AHG aims to achieve two properties:

- **Completeness:** No false negatives. All input/output variable 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.

These properties are required for correct state reconstruction (§4).

**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 model 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 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 re-declaration (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., $X$) 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.

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

**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 code, which can be used for re-runs ($\S$3.2).

**Definition 4.** A write dependency $(c_t \rightarrow (x, t))$ 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 $E_w$.

In Fig 5, $c_{t_2}$ modifies $x$ with "$x \leftarrow 1$"; hence, $(c_{t_2} \rightarrow (x, c_{t_2}))$.

**Definition 5.** A read dependency $((x, s) \rightarrow c_t)$ 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 $E_r$.

In Fig 5, "gen=(i for i in l1)" in $c_{t_3}$ accesses elements in the list $l1$ after its creation in $c_{t_1}$; hence there is $((x \rightarrow c_{t_1}), c_{t_3})$. Note that write/read dependencies are allowed to contain false positives; nevertheless, our replication ensures correctness ($\S$4.3).

**Definition 6.** The AHG $G := \{V \cup C, E_w \cup E_r\}$ is a bipartite graph, where $V$ is VSes, $C$ is CEs; $E_w$ and $E_r$ 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 $\S$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 $(x_1, t_1), ..., (x_k, t_k)$ to $c_t$, where $x_1, ..., x_k$ are variables possibly accessed by $c_t$.
- 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 detect changes at the reference level (in addition to values). For instance, conventional equality checks (e.g., based on serialization) will return True for "$a = [1, 2, 3] \neq \text{[1]}$" 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., $\text{id}(a) = \text{id}(b)$, where $\text{id}$ 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 [30] 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 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 ($\S$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 [117]`) before and after each cell execution while using deep copy as a fallback. If the deep copy fails (e.g., unserializable or comparable variables), we consider them to be modified-on-access using results from AST and ID Graph ($\S$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 Cell 2

we rerun 𝑐 was last modified by An active VS, (variables are unexpectedly unobtainable.

of the objects pointed to by 𝑛𝑒𝑤

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) is active if 𝑥 is in the system (i.e., not deleted), and there is no VS (𝑥, 𝑡𝑗) such that 𝑡𝑗 < 𝑡.

An active VS, (𝑥, 𝑡), represents the current version of 𝑥. For example, even if we checkpoint after 𝑐2, (in Fig 5), “(𝑥, 𝑡)” is active since 𝑥 was last modified by 𝑐2. We denote the set of active VSes as 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, 𝑡1), (𝑥2, 𝑡2), and (𝑥3, 𝑡3), 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), we rerun 𝑐2, which requires the previous version (𝑥, 𝑡1) as input, which in turn requires 𝑐1 to be rerun. Notably, it is not necessary to rerun 𝑐3 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.

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

after execution after executing Cell 2, the memory address of its nested list has been changed, no longer referencing list1.

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).

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, 115]. However, we don’t consider this approach as many data science workloads are memory-hungry with large training data and model sizes.

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 G of ElasticNotebook with false positives, and the true AHG G*, there is req∗(𝑥, 𝑡’) ⊆ req(𝑥, 𝑡) for any variable 𝑥 ∈ X, where (𝑥, 𝑡) and (𝑥, 𝑡’), req and req* are the active VSs of 𝑥 and reconstruction mapping functions defined on G and 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 [72].

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 ∀𝑥 ∈ 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.

Definition 9. A value-equivalent replication of a session state 𝑋 is additionally isomorphic if ∀a, b, id(a) = id(b) → id:new(a) = id:new(b), where a, b are arbitrary references (e.g., x[8][1], 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: Enabling Live Migration for Computational Notebooks

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 \( S \subseteq X \), the migration cost \( w_M(S) \) can be expressed as follows:

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

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. \( \alpha \) is a coefficient for adjusting the time cost of storage; for example, if ElasticNotebook is to be invoked upon auto-suspension, \( \alpha \) 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 run-times during a session lifecycle (§3.1). Combined with the reconstruction mapping \( \text{req()} \) for the AHG (§4.3), the cost \( w_R \) for recomputing a subset of variables \( S \subseteq X \) can be defined as follows:

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

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 \( S \) and variables to recompute \( X - S \)—is expressed as:

\[
w(S) = w_M(S) + w_R(X - S) \tag{3}
\]

5.3 Optimization Problem for State Replication

The goal is to find the variables to migrate \( S \subseteq X \) 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., 11 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:

\[
(x_1 \in S \land x_2 \in S) \lor \neg(x_1 \in S \land x_2 \in S) \lor (x_1, x_2) \in \mathcal{L} \tag{4}
\]

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

**Output:** A replication plan of subset of variables \( S \subseteq X \) for which we migrate (and another subset \( X - S \) which we recompute).

**Objective:** Minimize replication cost \( w_M(S) + w_R(X - S) \)

**Constraint:** Linked variables are either both migrated or recomputed: \((x_1, x_2) \in \mathcal{L} \lor \neg(x_1 \in S \land x_2 \in S) \lor (x_1, x_2) \notin \mathcal{L} \)

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

5.4 Solving State Replication Optim. 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 \( S \), while the cost of the cut is equal to the replication cost \( w_M(S) + w_R(X - S) \). Therefore, finding the minimum cost src-sink cut is equivalent to finding the optimal replication plan.

**Flow Graph Construction.** A flow graph \( H = (V_H, E_H) \) and its edge capacity \( \phi : E_H \rightarrow \mathbb{R}^+ \) are defined as follows:

- \( V_H = V_a \cup C \cup \{\text{src, sink}\} \): \( V_a \) is active VSes, \( C \) is cell executions, and src and sink are dummy source and sink nodes.
- \( \forall x \in V_a, (\text{src}, (x, t)) \in E_H \) and \( \phi(\text{src}, (x, t)) = w_M(x) \): We add an edge from the source to each active VS with a capacity equal to the migration cost of the variable.
- \( \forall c \in C, (c, \text{sink}) \in E_H \) and \( \phi(c, \text{sink}) = w_{rerun}(c) \): We add an edge with capacity from each CE to the sink with a capacity equal to the rerun cost of the CE.
- \( \forall C \subseteq V_a, (x, t) \in \mathbb{E}_{\text{req}} \) and \( \phi((x, t), c) = \infty \) and \( (x, t) \in V_H \): We add an edge with infinite capacity from an active VS \((x, t)\) to a CE \( c \) if \((x, t)\) must be recomputed.
- \( \forall (x_1, x_2) \in \mathcal{L}, ((x_1, t_1) \leftrightarrow (x_2, t_2)) \in E_H \) and \( \phi((x_1, t_1) \leftrightarrow (x_2, t_2)) = \infty \): We add a bidirectional edge with an infinite

[Diagram: 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.]
capacity between each pair of active VSes corresponding to linked variables $x_1$ and $x_2$, e.g., $l_1$ 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 solving algorithm (i.e., 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 replication plan can be obtained from the partition:

- $S = \{x \mid (x, t) \in \mathcal{V}_{H_{sink}} \cap \mathcal{V}_a\}$ are the active variable snapshots (and thus variables) that we want to migrate; in the example, these variables are $l_1$, $2dlist1$, and gen.
- $\mathcal{V}_{H_{src}} \cap C$ are the CEs which we will rerun post-migration to recompute $X - S$. In the example, these CEs are $t_1$, $t_2$, and $t_3$; when rerun, they recompute $y$, $z$, and $x$.\footnote{Rerunning $t_3$ also recomputes $l_1$; however, it will be overwritten with the stored $l_1$ in the checkpoint file following the procedure in $\S3.2$. This is to preserve the link between $l_1$ and $2dlist1$.}

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 ($\S6.1$) and design considerations ($\S6.2$).

6.1 Implementation

**Integrating with Jupyter.** For seamless integration, ElasticNotebook’s data layer is implemented using a magic extension [102], which is loaded into the kernel upon session initialization. The cell magic is automatically added to each cell ($\S2.2$) 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 underserializable variables), and are typically caused by oversights in incompletely implemented libraries [14, 68]. 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 recompiling 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 defined in a 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 has the same effect as 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\footnote{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 ($\S5.4$), upon which the user can be warned to delete the problematic variable to proceed with replicating the remaining (majority of) variables in the state.} (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 non-deterministic 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, 114], this work does not incorporate such tools.

7 EXPERIMENTAL EVALUATION

In this section, we empirically study the effectiveness of ElasticNotebook’s session replication. We make the following claims:

1. **Robust Replication:** Unlike existing mechanisms, ElasticNotebook is capable of replicating almost all notebooks. ($\S7.2$)
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. ($\S7.3$)
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. ($\S7.4$)
4. **Low Runtime Overhead:** ElasticNotebook incurs negligible overhead—amortized runtime and memory overhead of <2.5% and <10%, respectively. ($\S7.5$)
ElasticNotebook: Enabling Live Migration for Computational Notebooks

Table 3: Summary of datasets for evaluation.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Notebooks</th>
<th>Runtime (s)</th>
<th>Input data (MB)</th>
<th>Cell count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial [106]</td>
<td>5</td>
<td>10–96</td>
<td>1–139</td>
<td>10–48</td>
</tr>
<tr>
<td>HW [13, 45, 65]</td>
<td>15</td>
<td>9–1203</td>
<td>16–439</td>
<td>11–160</td>
</tr>
</tbody>
</table>

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 [56]: We select 35 popular notebooks on the topic of EDA (exploratory data analysis) + machine learning from Kaggle created by Grandmaster/Master-level users.
- JWST [59]: 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 [106]: 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, 65]: 15 in-progress notebooks are chosen from data science exercises. They contain out-of-order cell executions, runtime errors, and mistakes (e.g., $df_{\text{backup}}=df^4$).

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

**Methods.** We evaluate ElasticNotebook against existing tools capable of performing session replication:
- RerunAll [101]: 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 [103]: A checkpointing tool that serializes variables one by one into storage. We use a modified version using Dill [38] instead of Pickle [37] for robustness.\(^5\)
- DumpSession [39]: Unlike %Store, DumpSession packs the entire session state into one single file.

**Ablation Study.** We additionally compare against the following ablated implementations of ElasticNotebook:
- ElasticNotebook + Helix [115]: 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.

\(^5\)This creates a shallow copy of $df$, which does not serve the purpose of backup.

\(^6\)The original implementation of %store uses Python Pickle [37], and fails on too many notebooks to give meaningful results.

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.

<table>
<thead>
<tr>
<th>Notebook(s)</th>
<th>Type</th>
<th>Description and purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFL [87]</td>
<td>hashlib [32]</td>
<td>Drop down list in plot</td>
</tr>
<tr>
<td>All 3 JWST books</td>
<td>mmap [35]</td>
<td>Helps avoiding reading large file into memory</td>
</tr>
<tr>
<td>Arxiv [69]</td>
<td>generator [31]</td>
<td>Speedup iterative comprehension</td>
</tr>
<tr>
<td>Plant [93]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 upsizing/downsizing, 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 $\mu$s.

**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.

**Reproducibility.** Our implementation of ElasticNotebook, experiment notebooks, and scripts can be found in our Github repository.\(^6\)

### 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 unpickleable variables (§6.1), respectively. **DumpSession** and %Store fail on 19/60 notebooks containing unserializable variables, many of which are used to enhance data science workflow efficiency (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, https://anonymous.4open.science/r/ElasticNotebook-63E7
linked components of a Matplotlib [104] plot—f, f1g, 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 to correctly replicate 11/60 sessions containing variable aliases due to its lacking of the linked variables constraint.

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 upsampling and downsampling results in Fig 10 and our technical report [72], 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 RunAll. Compared to DumpSession, %Store, and CRIU, which store all variables in the checkpoint file, ElasticNotebook upscale/downscale up to 2.07×/2.00× faster than the best of the three. DumpSession, while being the next best alternative for upsampling/downsampling 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 upsampling/downsampling 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 $\alpha$ 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. Compared to the baselines, ElasticNotebook is 3.92× faster than the next best alternative. These fast restoration can be attributed to ElasticNotebook capable of adapting to the new optimization objective, unlike the baselines: for example, on the Sklearn [108] notebook, instead of re-running cell 3 ($df = pd.read_csv(\ldots)$) to re-read the dataframe $df$ into the session as in the migration-centric plan, the restoration-centric plan opts to store $df$ instead. The reasoning is that despite the sum of serialization and deserialization times of $df$ being greater 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 taken to store the AHG, ID Graphs, and hashes at checkpoint time (ElasticNotebook memory usage).

We report the results in Table 5. ElasticNotebook’s 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 [72].

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 file sizes 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 10X 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).

7.7 Performance Gains Across Environments

This section demonstrates ElasticNotebook’s operation in environments with varying specifications. We perform a parameter sweep on the NFS network bandwidth via rate limiting [10] and compare the migration time of ElasticNotebook, DumpSession (migrating all variables), 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_r_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.\(^7\)

\(^7\)This is twice the length of the longest observed notebook on Kaggle [49].
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, 54, 64, 110, 115, 116, 121]. Examples include caching to speed up model training for ML model diagnosis [41, 110], caching to speed up anticipated future dataframe operations in notebook workflows [116], and storage of cell outputs to facilitate graphical exploration of the notebook’s execution history for convenient cell re-runs [54, 64]. There are related works [115, 121] which algorithmically explore the most efficient way to (re)compute a state given currently stored items; compared to our work, while Helix [115] 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, 38, 77]. There exists a variety of checkpoint tools built on these serialization libraries: IPython’s %Store [103] 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 [38] DumpSession [39] correctly resolves object references, yet it still fails if the session contains unserializable objects. Tensorflow [48] and Pytorch [28] offer periodical checkpointing during ML model training limited to objects within the same library. Jupyter’s native checkpointing mechanism [101] 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, 12, 22, 51, 71, 76, 95]. Applicable tools include DMTCP [3] and CRIU [17]; recently, CRUM [42] and CRAC [60] have explored extending C/R to CUDA applications. Elsa [63] 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, 50, 70, 81, 105, 110, 119] Recently, the usage of data lineage in computational notebooks has enabled multi-version notebook replay [75], recommending notebook interactions [74], and creating reproducible notebook containers [1], and program slicing, i.e., finding the minimal set of code to run to compute certain variable(s) [50, 53, 64, 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 modifications (via equality comparisons); however, our techniques additionally identifies and captures structural 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, 114]. 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 (e.g., in the form of a script via [98] or papermill [99]). 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 upsampling, downsampling, 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.

ACKNOWLEDGMENTS

The authors are grateful to Chandra Chekuri and Kent Quanrud for assistance with the derivation of the reduction to min-cut employed in ElasticNotebook. This work is supported in part by the National Center for Supercomputing Applications and Microsoft Azure.
REFERENCES


