Summary of Revision
"ElasticNotebook: Enabling Live Migration for Computational Notebooks"

We thank the reviewers (R3, R4, R5) for their constructive feedback, which we all address as follows.

Meta Reviewer

MR-1: Include missing related work (R3D1 and R3D2).

We have added comparisons to materialization/reuse (R3D1) and program slicing/lineage tracing (R3D2).

MR-2: Improve the presentation throughout the paper (R3D3, R3D5, R3D6, R5D1)

We have clarified how we track variables (R3D3), added uncleaned notebooks to our experiments (R3D4), clarified our min-cut formulation (R3D5), discussed our fallback mechanism (R3D6), described a potential application to parameterized notebooks (R3D7), reworded sentences for more clarity (R4D1, R4D2), added one more baseline ("checkpoint") to file size experiment (R4D3), and discussed approaches to handling unobservable operations and side effects (R5D1).

MR-3: Clarify your approach to elasticity both in the core sections and by adding appropriate experiments (R5D2)

We have clarified our approach to elasticity, and added experiments for upscaling/downscaling (R5D2).

MR-4: Add experiments related to correctness and add a correctness metric (R5D3).

We have clarified our correctness metric, and added additional experiments comparing against ablated methods of our system without our mechanisms guaranteeing said correctness (R5D3).

Reviewer 3

***R3-D1:** Comparison with papers on the use of materialization/reuse in related data science contexts.*

Thank you for the related work. Indeed, materialization/reuse is a widely used technique in large-scale data management. We have added discussions of the additional related work (Sec 8), while also highlighting the significance of our new optimization (Sec 5.2). That is, when determining what variables to replicate, we must consider ***linked variables***, using special constraints, to ensure correct replication of notebook states. Otherwise, replicated states have wrong references, as we empirically demonstrate in updated Fig 9, Sec 7.1, where an existing technique (Helix by Xin et al., 2018) produces incorrect results for 8.3% of notebooks.

***R3-D2**: Comparing against other approaches that use static/dynamic analysis to track variables/dependencies.*

Thank you for pointing them out. We now discuss them in background (Sec 2) and related work (Sec 8). In short, our ID Graph, on top of existing syntactic code analysis, is required for correct results (Sec 4.2).

- **Identifying Accesses**: While syntactic analysis can recognize direct variable accesses, our approach allows the identification of ***indirect*** accesses (via other variables). *(Please see R3-D3 for more details)*
- **Identifying Modifications**: While syntactic or value-based analyses can only ensure each variable has the same value, our approach allows the understanding of ***inter-variable structures***, ensuring the equivalence at the level of the entire state (beyond individual variables).

Our new experiment shows that, without ID Graphs, 18.3% of notebooks are *incorrectly* replicated (Sec 7.2).

***R3-D3:** Clarification on identifying and keeping track of variables and their dependencies.*

- We have clarified the distinction between objects (including complex entities such as functions and classes) and variables (i.e., named references to same/different objects) in Sec 4.1.

- Our mechanism (clarified in Sec 4.2) is designed to detect deep accesses/modifications to the objects (recursively) reachable from variables (e.g., `mylist[0].attr`) by combining syntactic code analyses and our proposed ID Graph. *(See more details in the following item)*
- Our mechanism allows identifications of **indirect accesses** (i.e., an object referenced by **`var1`** changes via **`update(var2)`**) and **direct accesses** (i.e., an object referenced by **`var1`** changes by **`update(var1)`**). This is possible because any object (e.g., class instance) can be represented using a graph, which we traverse until we reach leaf nodes (e.g., primitive types, None) for its construction; then, we compare `IdGraph(var)` created before/after each cell execution. Our mechanism may say both `var1` and `var2` have been accessed/updated, even if only one underlying object has been changed, which is intended.

R3-D4: *Evaluation on more realistic notebooks.*

- Thank you for the suggestion. We have performed additional experiments with newly added 15 in-progress and/or uncleaned notebooks (under a new category "Homework"), demonstrating our system can reduce upscaling and restoration times by 96.8% and 98.5%, respectively (Sec 7.2, 7.3).
- We have additionally reported the memory runtime overhead after each cell execution for three of "Homework" notebooks (Sec. 7.5). Specifically, we note that while suboptimal coding habits (e.g., nested lists instead of dataframes, shallow copy for "backup") can lead to higher overhead, the overheads are still very manageable. Due to space constraints, we present these results in our technical report (citation number 72, which has a URL for this review).

R3-D5: *Clarify min-cut formulation.*

We have included the following clarifications on our min-cut formulation in Sec 5:

- Unlike incrementally maintained AHG (Application History Graph), the min-cut graph is constructed (for a replication plan) each time a replication is requested; therefore, it is never 'updated' per se in response to variable value updates. We have clarified this in the beginning of Sec 5.
- We have reworded the formulation to use **active variables snapshots** instead of variables as nodes in the min-cut graph to clarify that we aim to recompute the specific value of variables at checkpoint time.
- While inter-variable dependencies are not explicitly tracked, our AHG expresses how each active variable snapshot can be reconstructed from a series of cell executions (i.e., req(x), clarified in Sec 4.3).
- The event of each ***cell execution*** appears as a node in the min-cut graph, which is is different from a *displayed cell* in the notebook; a single *displayed cell* may result in one or more ***executions***.

R3-D6: *Clarify robust replication (i.e., skip variables that cannot be deserialized)*

We have clarified in Sec 6.1 and 7.2 that, even if there exist unexpectedly "un-desrializable" variables, our system can still recover and restore the original session by determining the minimal cells that must be re-run to reconstruct those variables. This fallback mechanism may take longer than originally expected; however, it ensures correctness and is very likely still faster than rerunning the entire notebook from scratch.

R3-D7: *For greater impact, I would encourage the authors to think about how their techniques could be implemented, for example, inside Papermill.*

We agree on the usefulness of applying our techniques to parameterized notebooks (or scripts in general). We have added a brief discussion on this topic in our related work section (Sec 8).

Reviewer 4

R4-D1: *statement "must offer significant computational benefits" on page 2 paragraph 2*

Thank you for pointing out. Our original intention was to express that our system must merit itself to be used over existing methods given additional effort for installation and incurred overhead; we have reworded

the statement to be more factual and straightforward.

R4-D2: *Rewording Scalability for Complex Notebooks (Section 7.8)*

Thank you for reminding. We have reworded Sec 7.8 to explicitly refer to the additional memory usage incurred from storing the AHG (Application History Graph) during session runtime.

R4-D3: *Including "checkpoint" strategy in section 7.6*

We agree on the usefulness of such a comparison. we have included the "checkpoint" strategy in Sec 7.6.

Reviewer 5

R5-D1-1/2: *Cell execution monitoring, External side effects during cell executions*

- **Visibility/Unobservable State:** Our approach can be used for a wide range of practical use cases since it monitors object states according to the "pickle" protocol, the de facto standard followed by almost all libraries (e.g., numpy, pytorch), meaning we capture sufficient information (Sec 6.2). Nevertheless, there could be libraries with incorrect serialization (e.g., ignoring *hidden* data); to address this, our mechanism can easily be extended to allow manual annotations to force recomputation.
- **Side Effects:** While our current prototype, focusing on read-oriented data science, does not explicitly prevent side effects (e.g., DB writes), our system can be extended to prevent them as follows:
  - **Annotation:** We can allow manual annotations for certain cells if those cells may incur side effects. Then, our system can avoid recomputing those cells during restoration (if such a plan is feasible) or throw an error (if avoiding recomputation is impossible for a given workload).
  - **System-level:** We can create a sandbox by altering a file system access (e.g., chroot) or by blocking outgoing network (e.g., ufw). Upon successful restoration inside the sandbox, its environment can get associated with regular file/network accesses. Modern clouds with dockers make this easier.

We have clarified these in Sec 6.2.

R5-D1-3: *Detecting situations when re-running is unsafe and checkpointing is impossible*

Our min-cut based solution is capable of detecting situations where no feasible migration solution exists (e.g., an unserialzable/problematic variable requires rerunning an unsave execution to recompute). We have added a description in Sec. 6.2 to detail how it accomplishes this. When this happens, our system can prompt the user to delete said problematic variable so the (majority of) the session can still be replicated as is.

R5-D2: *Claims on elasticity*

We apologize for the confusion caused. We intend to claim that our system enables elastic (or easier) scaling at the notebook service level by facilitating quicker migration to a different machine (in the cloud) than horizontally scaling an existing cluster. We have clarified our claims in introduction (Sec. 1) and background (Sec. 2.1), and updated our experiments to incorporate upscaling/downscaling cases (Sec 7.3).

R5-D3: *Correctness metric and experiments*

We have added a sub-section (Sec 5.1) describing our correctness metric, as follows:

1. **Value-Equivalence:** Each variable has the same value before and after a replication.
2. **Isomorphism:** In addition to (1), we preserve all shared references between objects (e.g., aliases).

Isomorphism is based on the notion of *graph isomorphism*, guaranteeing the structural equivalence of all the variables within the entire session before and after replication. We have also updated our experiments to reflect these refined correctness metrics (Sec 7.2).