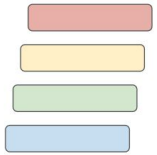
Efficiently Reproducing Distributed Workflows in Notebook-based Systems

Talha Azaz[2], **Raza Ahmad[1]**, Md Saiful Islam[3],
Douglas Thain[3], **Tanu Malik[2]**

[1]**DePaul University**, [2]University of Missouri, [3]University of Notre Dame

The 26th IEEE International Symposium on Cluster, Cloud, and Internet Computing (CCGrid), 2026



floability

Overview

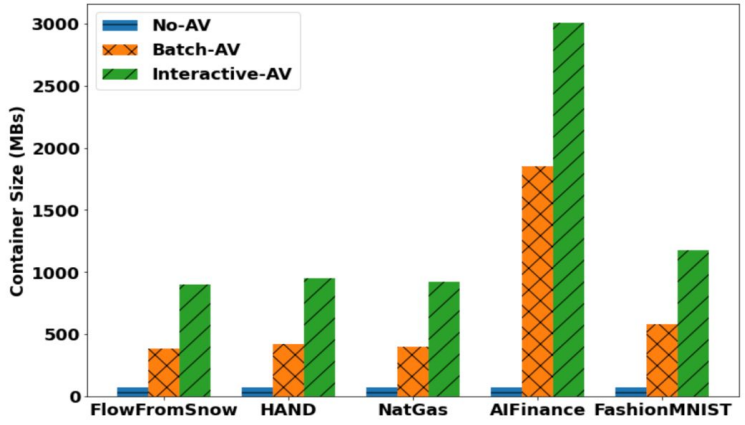
- Primary objective: Improve reproducibility of distributed workflows in notebooks
- Notebooks have become widespread in scientific computing
  - Interactive and iterative development, cell-level execution, easy dissemination
  - Distributed workflows for large-scale data analytics
- Repeating and reproducing distributed workflows is challenging
  - Most techniques like manual savepoints work well for single-node workflows
- We present NBReplay – a tool for efficient, reproducible execution of distributed workflows
  - Checkpoint and restore intermediate workflow state at each cell
  - Avoid redundant execution through selective task caching and replay
  - Shows average performance gains between 2.08x–18.4x for notebook re-execution

Structure of a Notebook

[9]:
```
plt.subplots(figsize =(12, 8))
x = np.arange(len(df2.index))
plt.bar(x, df2['Size 2'], width=bar_width, hatch='-')
plt.bar(x+(1*bar_width), df2['Size 1'], width=bar_width, hatch='x')
plt.bar(x+(2*bar_width), df2['Size 3'], width=bar_width, hatch='/')
```

[9]:
```
plt.ylabel('Container Size (MBs)', fontsize=18, fontweight='bold')
plt.xticks([r + bar_width for r in range(len(x))], df2.index)
plt.legend(labels=labels)
plt.savefig('export_size_containers_all.png', dpi=500)
df2['Size 2'].head(3)
```

[9]:
```
Use Case
FlowFromSnow    72.7
HAND            72.7
NatGas          72.7
Name: Size 2, dtype: float64
```

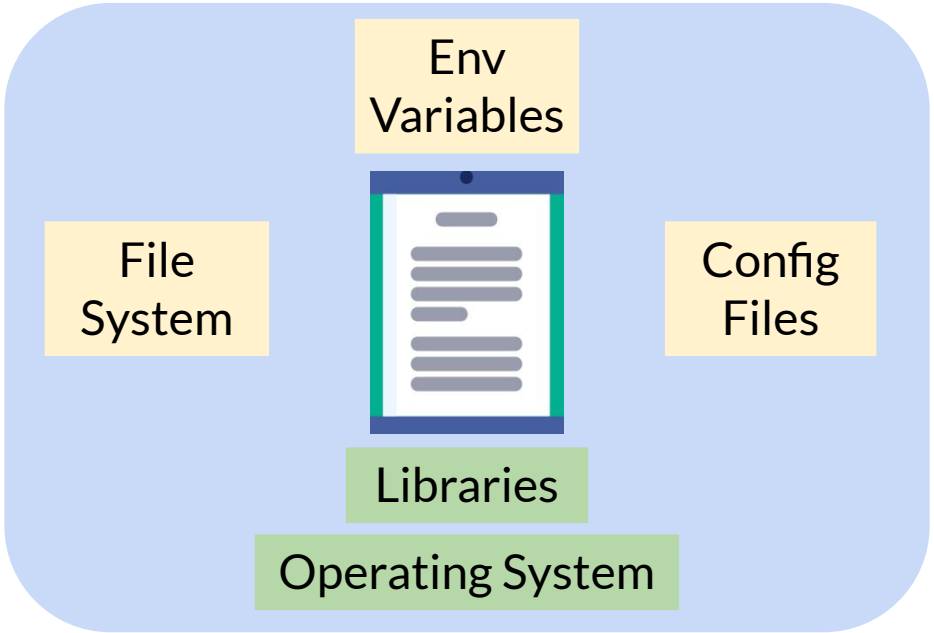


Code Cells

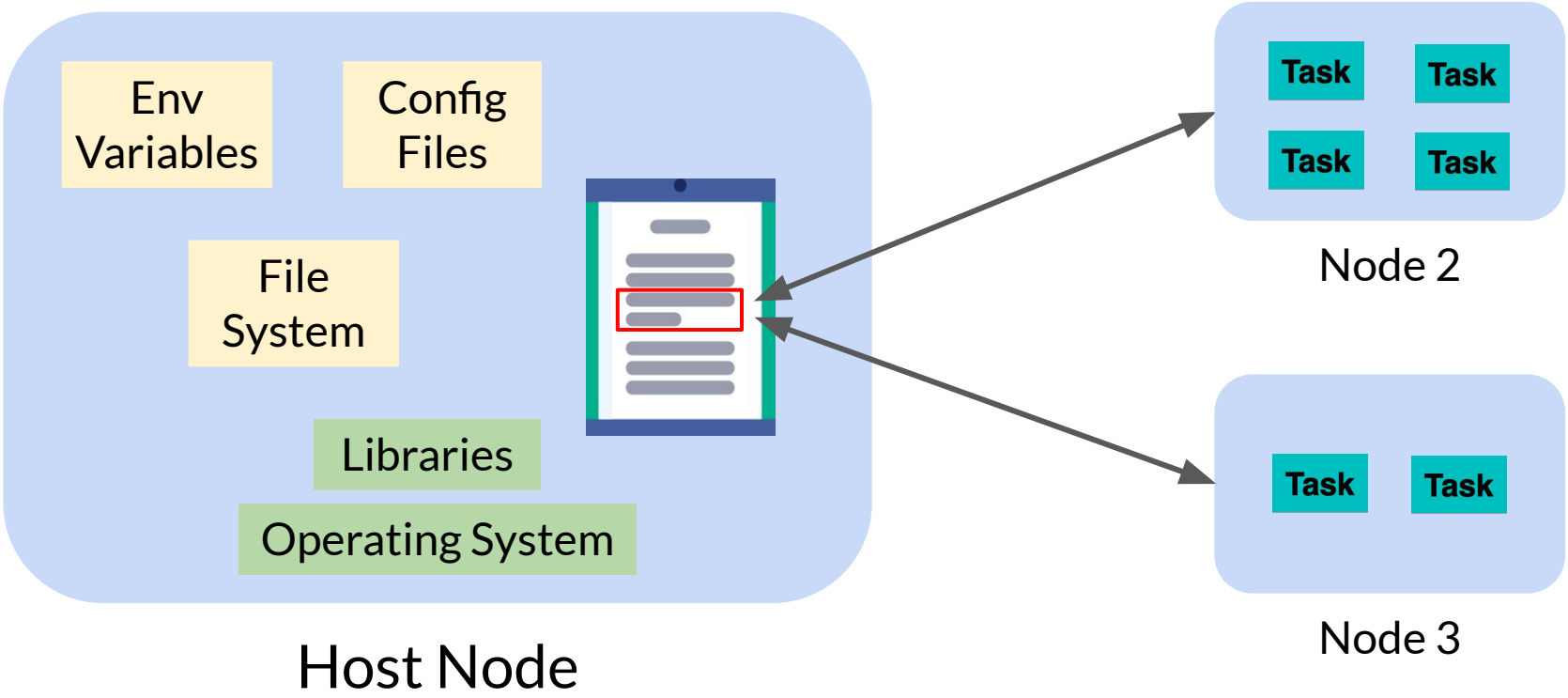
Data & Results

Visualizations

Execution in a Notebook



Distributed Execution in a Notebook



Notebooks for Exploration

Exploratory Analysis

Update the formula

→ risk_score > 10.0

→ risk_score > 5.0

[c1]
```
from dask.distributed import Client
import dask.dataframe as ddf
# Connect to an existing Dask cluster
client = Client("tcp://scheduler:8786")
# Lazy load dataset from shared filesystem
df = ddf.read_parquet("run_24-06.parquet")
# Lazy preprocessing
df = df[df["temperature"].notnull()]
df = df.assign(anomaly = df["temperature"]-
     df["temperature_baseline"])
raw_df = df
```

Data Loading & Pre-processing

[c2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
              (["region","day"])
              .agg({"anomaly":["mean","std"],
               "precipitation":"sum"})
               .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
              lambda part: part.assign(
              risk_score = part.apply(
              risk_score, axis=1)))
```

Computational Logic

[c3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

Execution, Result & Analysis

Scenario 1: Share and Repeat

[c1]
```
from dask.distributed import Client
import dask.dataframe as ddf
# Connect to an existing Dask cluster
client = Client("tcp://scheduler:8786")
# Lazy load dataset from shared filesystem
df = ddf.read_parquet("run_24-06.parquet")
# Lazy preprocessing
df = df[df["temperature"].notnull()]
df = df.assign(anomaly = df["temperature"]-
     df["temperature_baseline"])
raw_df = df
```

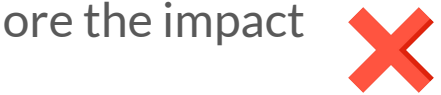
[c2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
               (["region","day"])
               .agg({"anomaly":["mean","std"],
                "precipitation":"sum"})
                .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
               lambda part: part.assign(
               risk_score = part.apply(
               risk_score, axis=1)))
```

[c3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

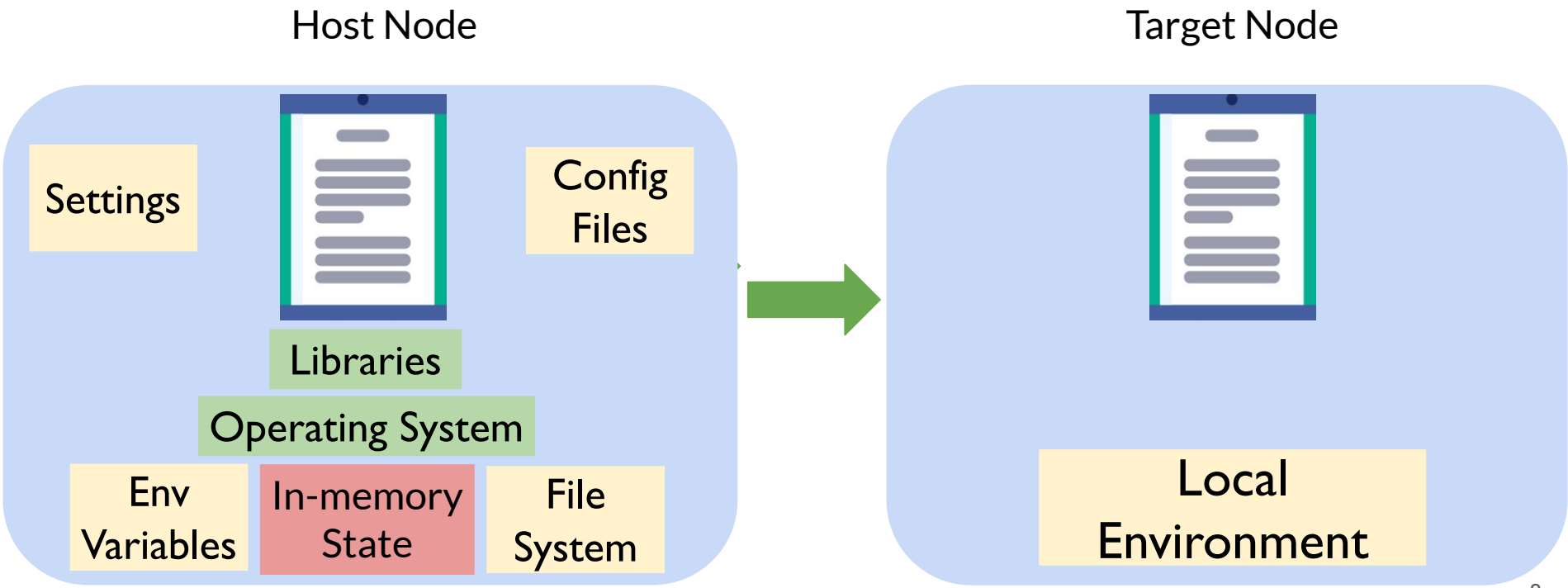
Share the notebook with a collaborator to repeat on another machine

[c3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

Only run cell 3 to explore the impact of **risk_score**



Problem: Sharing the Notebook ONLY



Scenario 2: Repeat Part of a Cell

Update **risk_score** function definition in cell 2

[c_2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
               (["region","day"])
               .agg({"anomaly":["mean","std"],
                "precipitation":"sum"})
                .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
               lambda part: part.assign(
               risk_score = part.apply(
               risk_score, axis=1)))
```

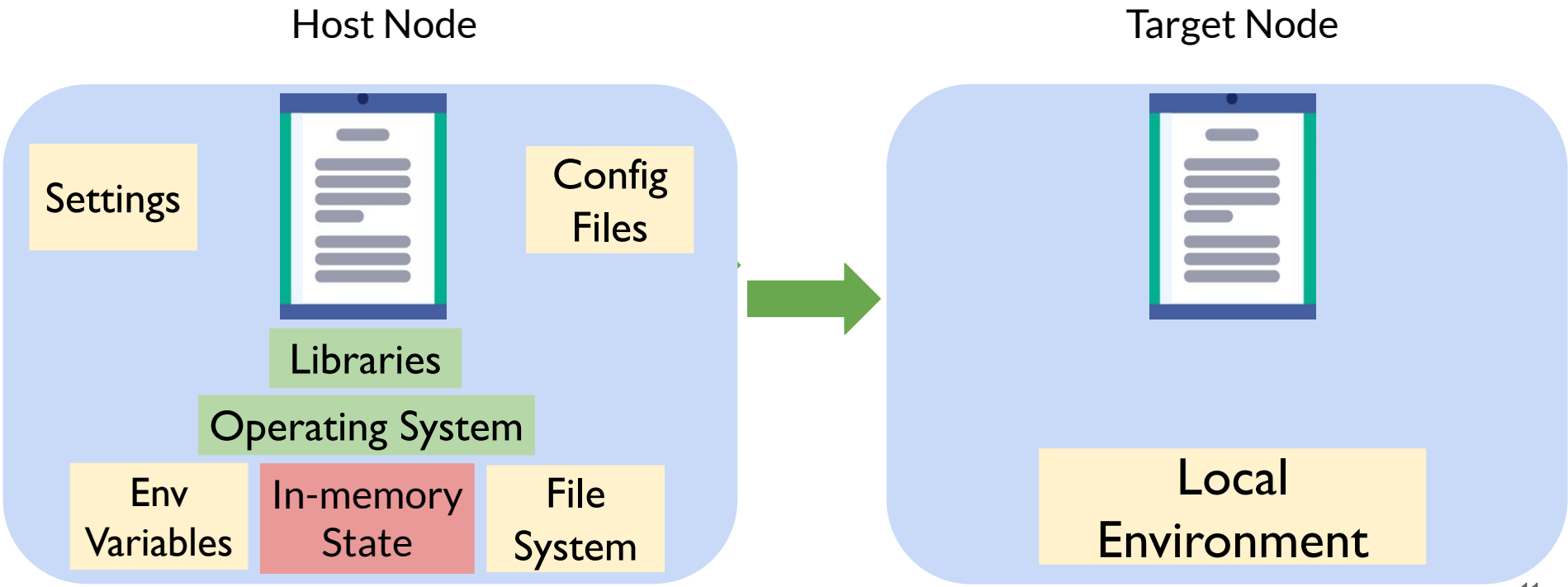
[c_3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

Only run part of distributed workflow in cell 2 

NBReplay

- NBReplay is an end-to-end system to enable efficient and reproducible execution of distributed workflows:
  - Performs cell-level checkpointing to restore intermediate workflow state
  - Performs caching of distributed tasks to avoid redundant execution
- Creates two kernels:
  - **Audit** – for executing the notebook
  - **Repeat** – for repeating the notebook, on the same or different node

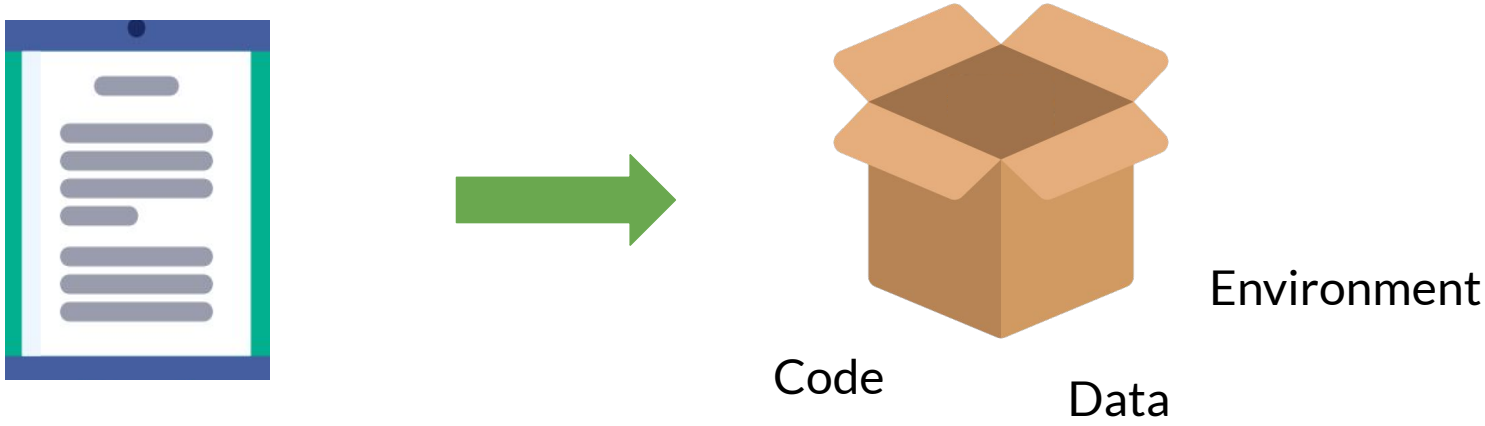
Scenario 1: Share and Repeat



Solution Scenario 1
Cell-level Checkpointing

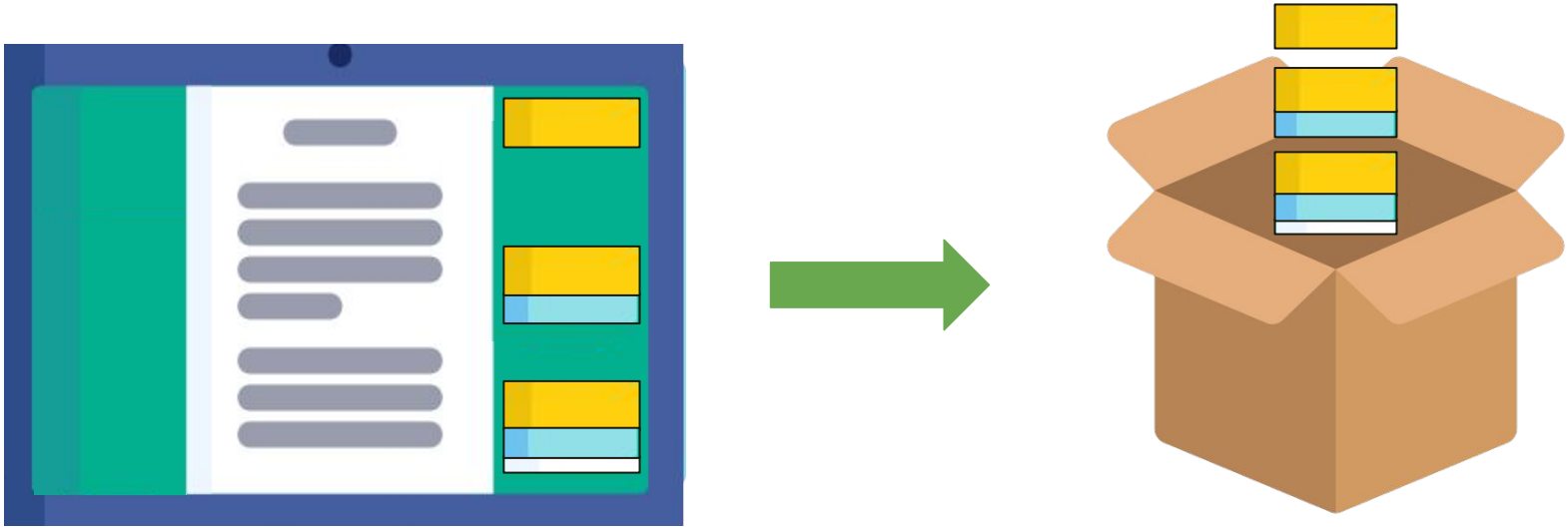
Notebook Containerization

- Audit kernel uses application virtualization to convert the notebook execution into self-contained package.



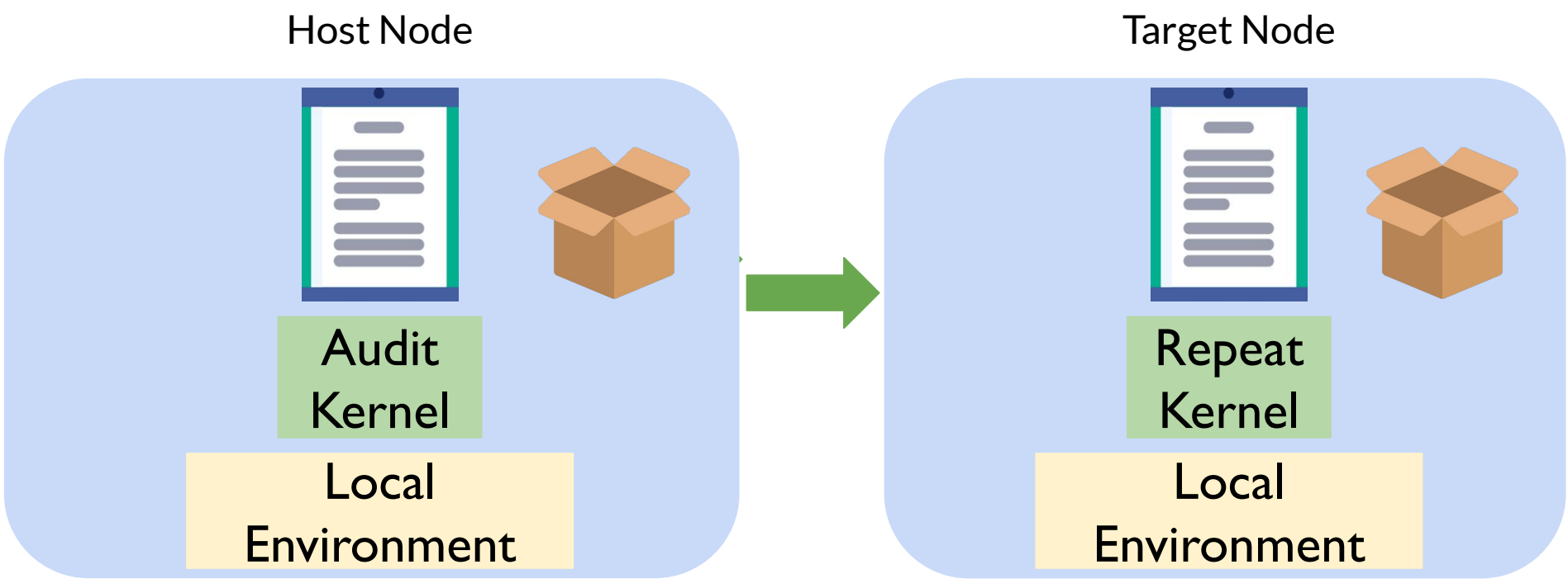
Cell-level Checkpointing

- Program state at each cell is stored in incremental checkpoints



- Checkpoints are added to the container for sharing

Sharing Checkpoints with the Notebook



Restoring Checkpoints during Repeat

- To explore the impact of **risk_score**, restore all checkpoints up to cell 2

- Re-execute cell 3 ONLY

[c_1]
```
from dask.distributed import Client
import dask.dataframe as ddf
# Connect to an existing Dask cluster
client = Client("tcp://scheduler:8786")
# Lazy load dataset from shared filesystem
df = ddf.read_parquet("run_24-06.parquet")
# Lazy preprocessing
df = df[df["temperature"].notnull()]
df = df.assign(anomaly = df["temperature"]-
     df["temperature_baseline"])
raw_df = df
```

checkpoint cell 1

[c_2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
              (["region","day"])
              .agg({"anomaly":["mean","std"],
               "precipitation":"sum"})
               .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
              lambda part: part.assign(
              risk_score = part.apply(
              risk_score, axis=1)))
```

checkpoint cell 2

[c_3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

Scenario 2: Repeat Part of a Cell

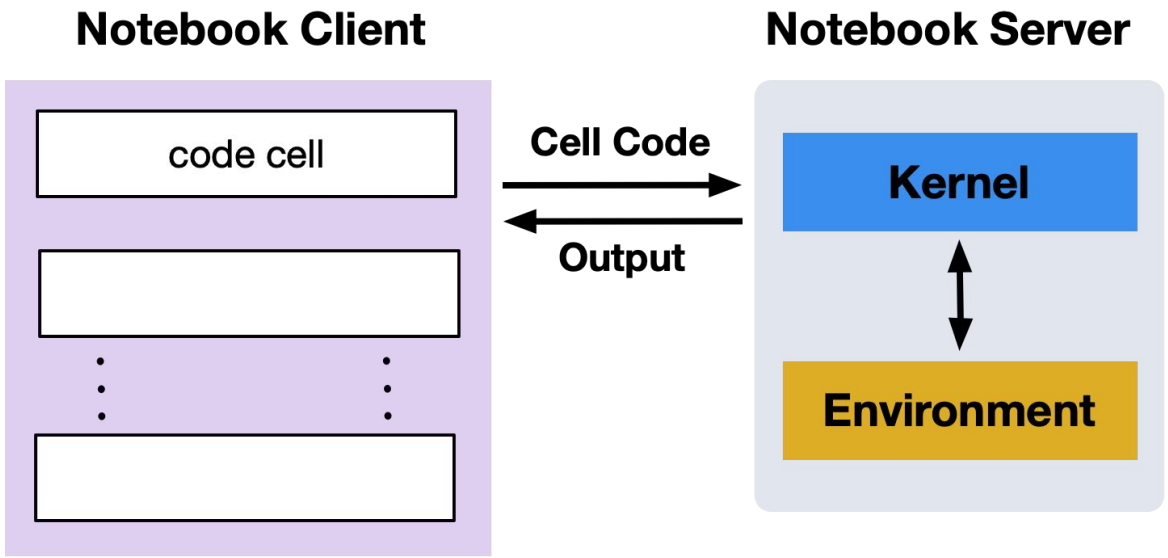
Update **risk_score** function definition in cell 2

[c_2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
               (["region","day"])
               .agg({"anomaly":["mean","std"],
                "precipitation":"sum"})
                .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
               lambda part: part.assign(
               risk_score = part.apply(
               risk_score, axis=1)))
```

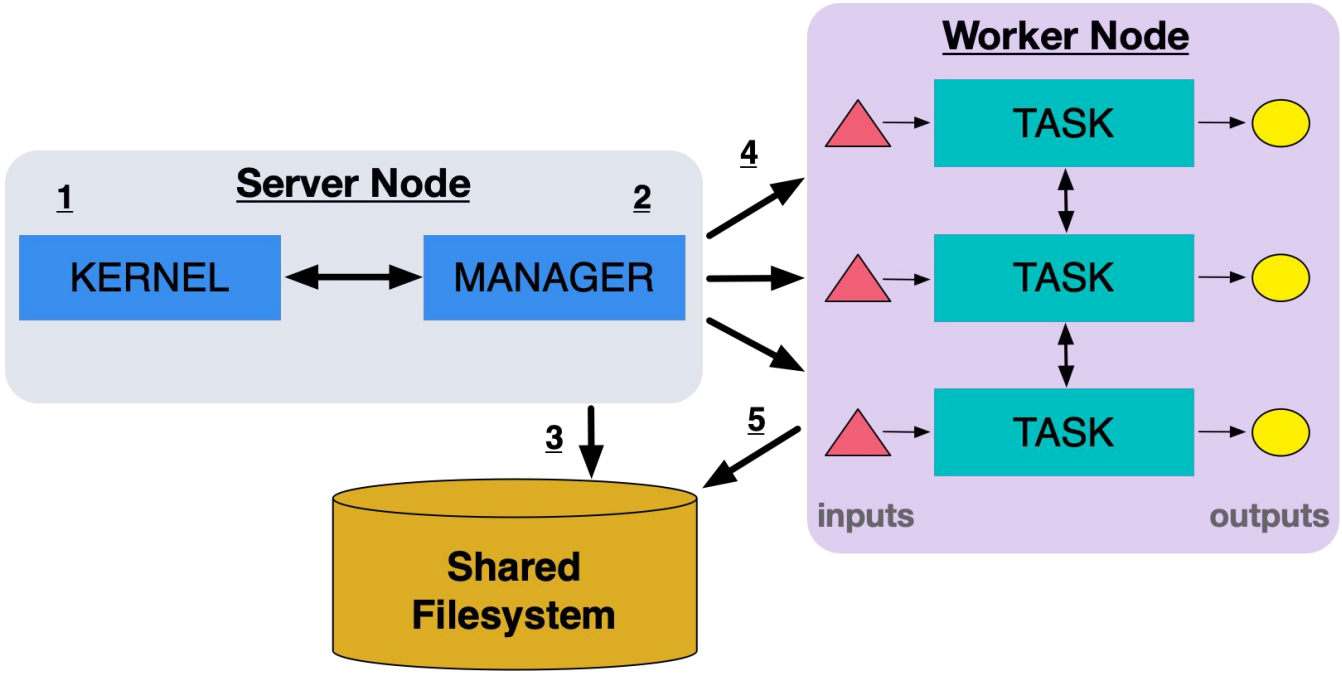
[c_3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

Only run part of distributed workflow in cell 2 

Execution in Notebook Environment



Distributed Execution in Notebooks

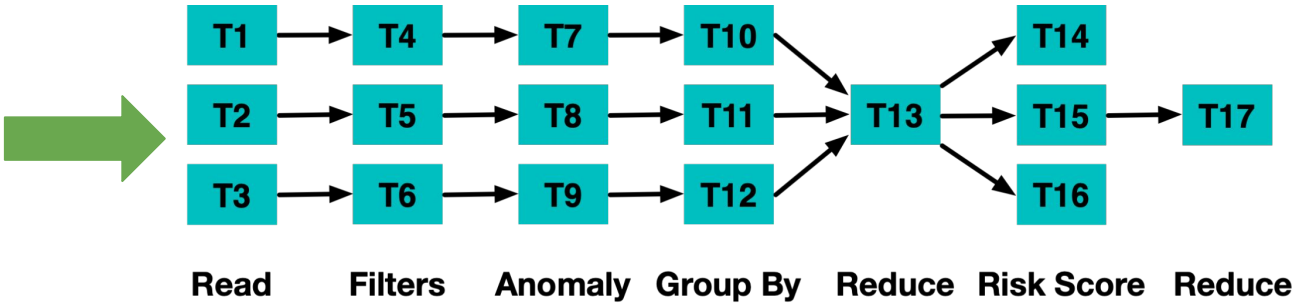


Distributed Task Generation

[c_1]
```
from dask.distributed import Client
import dask.dataframe as ddf
# Connect to an existing Dask cluster
client = Client("tcp://scheduler:8786")
# Lazy load dataset from shared filesystem
df = ddf.read_parquet("run_24-06.parquet")
# Lazy preprocessing
df = df[df["temperature"].notnull()]
df = df.assign(anomaly = df["temperature"]-
     df["temperature_baseline"])
raw_df = df
```

[c_2]
```
# Compute summary stats across worker nodes
daily_stats = (raw_df.groupby
              (["region","day"])
              .agg({"anomaly":["mean","std"],
               "precipitation":"sum"})
               .persist())
# Compute risk score
def risk_score(row):
    return 1.0*row[("anomaly", "mean")] + 0.1
    *row[("precipitation", "sum")]
stats_ref = daily_stats
daily_stats = daily_stats.map_partitions(
              lambda part: part.assign(
              risk_score = part.apply(
              risk_score, axis=1)))
```

[c_3]
```
# Run dist. computation, get subset of result
high_risk = daily_stats[daily_stats
            ["risk_score"] > 10.0]
# large-scale distributed execution
result = high_risk.compute()
# Save summary for analysis and visualization
result.to_parquet("high_risk_24-06.parquet")
```

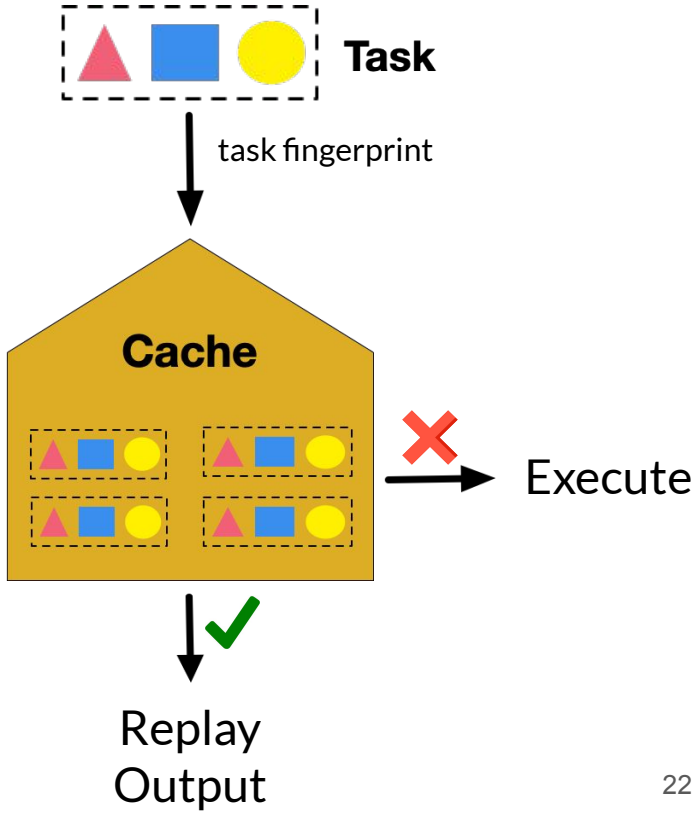


DAG of Parallel Tasks

Solution Scenario 2
Partial Re-execution with Caching

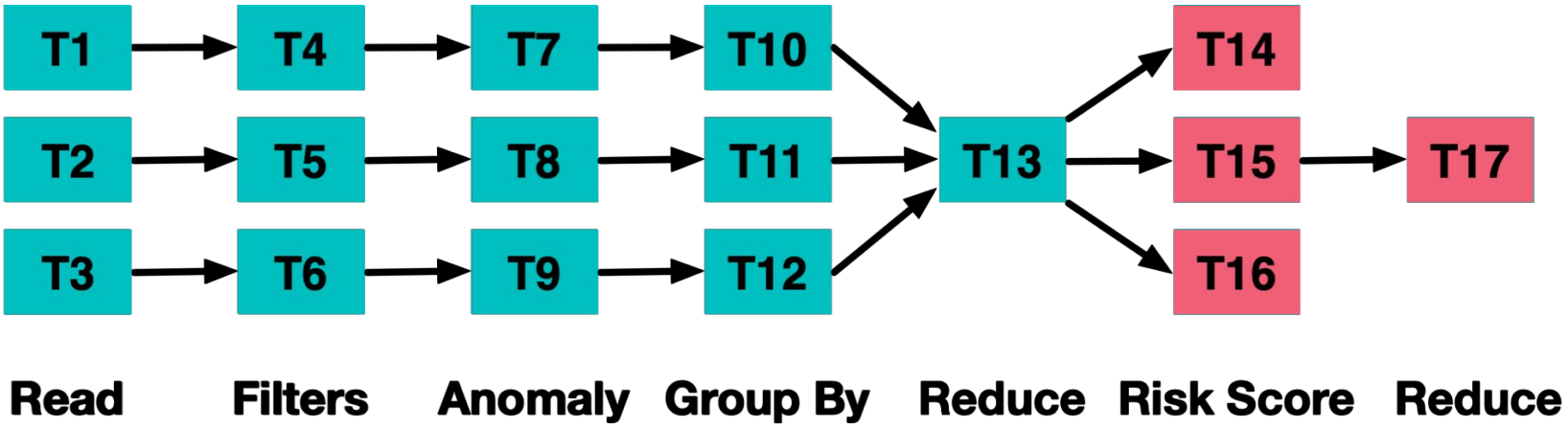
Task Fingerprinting and Caching

- Maintain a cache of tasks including their code, inputs, and outputs
- Incoming tasks are compared with the cache using a task fingerprint.
- If the task is found in cache, it is NOT executed and its cached output is replayed.

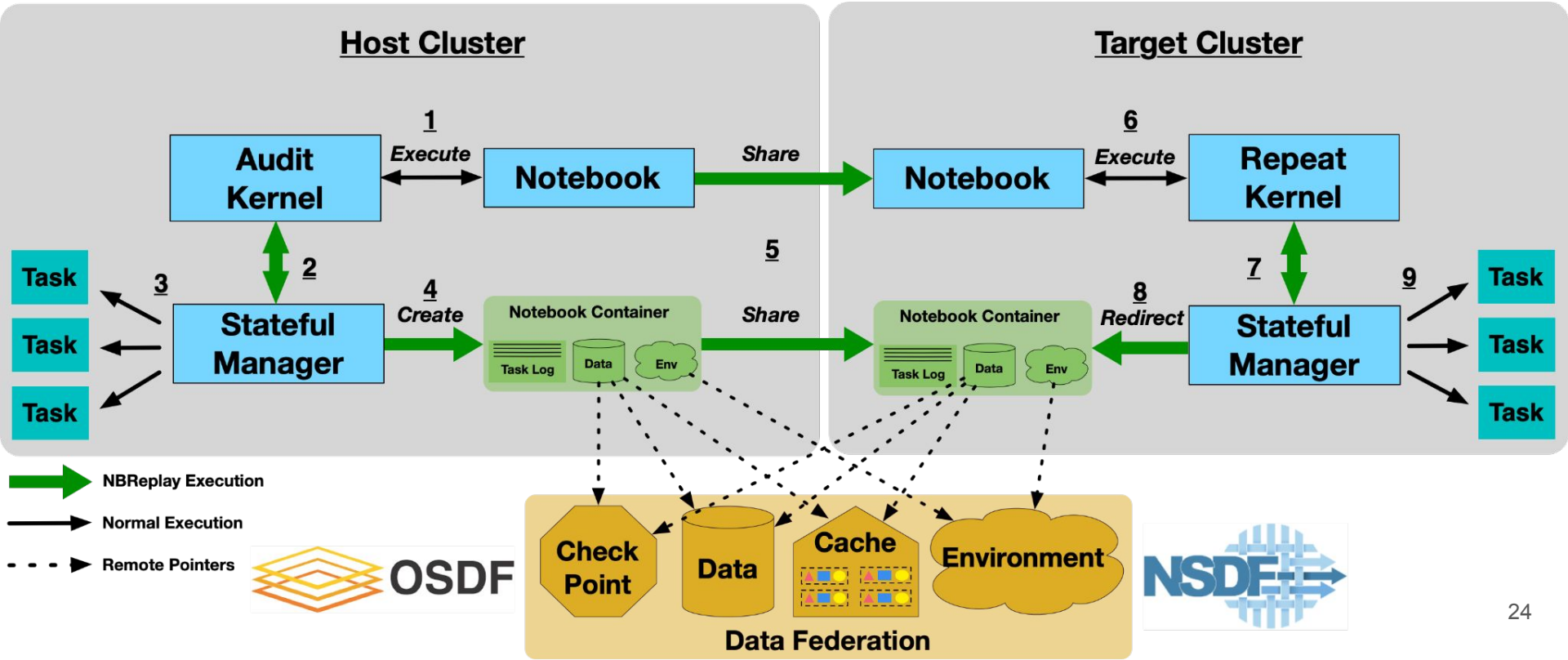


Partial Re-execution

- When **risk_score** function is updated, only the updated tasks are re-executed, shown in red.



NBReplay Architecture and Workflow



Dataset

Dataset of 5 notebooks from different domains with distributed workflows

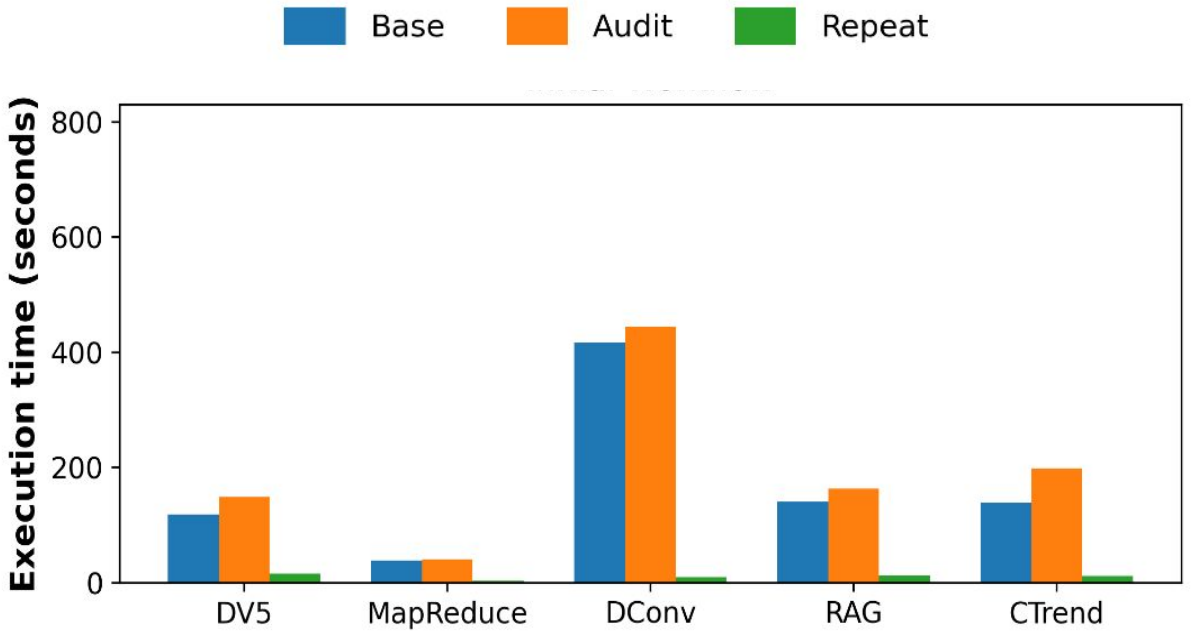
Notebook	Distributed Tasks	Domain
DV5	4	High Energy Physics
MapReduce	13	NLP
DConv	1024	Image Processing
RAG	4	ML
CTrend	120	Climate Analysis

Experiments

- **Audit – Create Checkpoints and Cache Tasks**
- **Repeat – Restore Checkpoints and Replay Tasks**
  - Repeat with change
  - Repeat without change
- **Baseline**
  - Plain notebook execution with change
  - Plain notebook execution without change

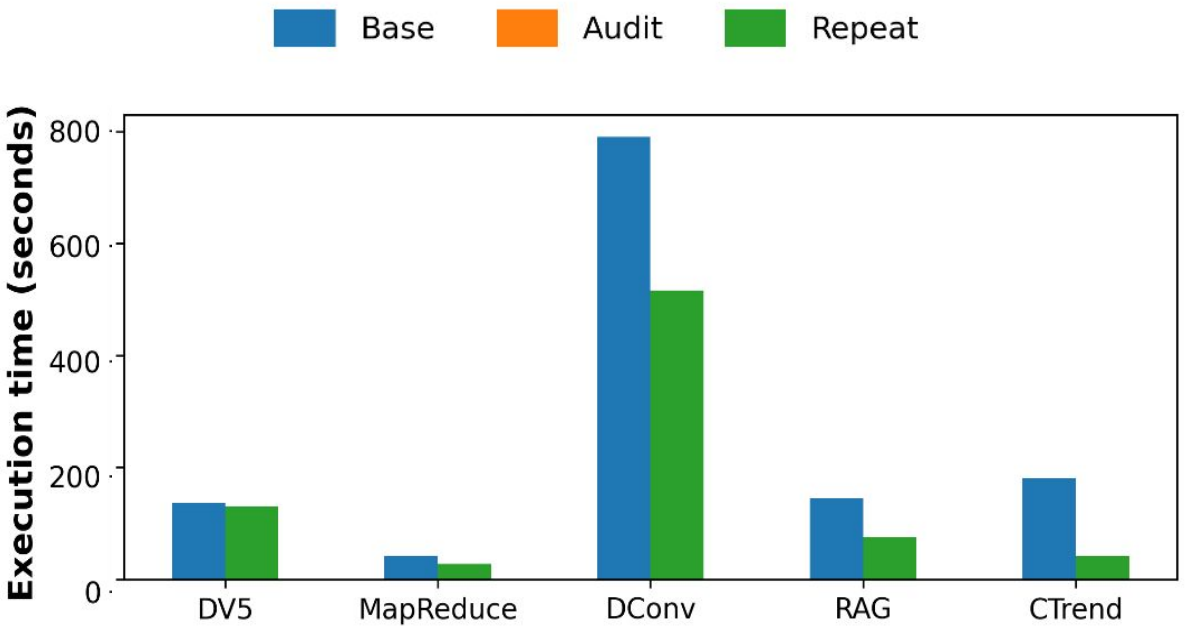
Audit vs. Repeat With Change

- An average gain of more than 2.08x on repeat with change.



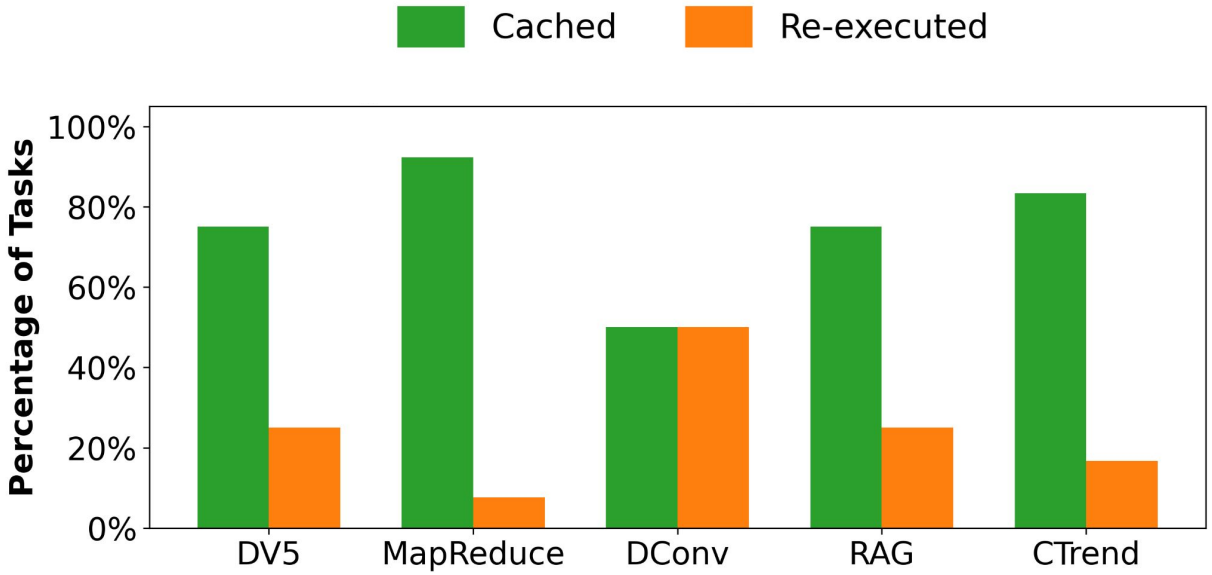
Audit vs. Repeat Without Change

- An average gain of more than 18.4x on repeat without change.



Cache Effectiveness in Partial Re-execution

- Average hit rate of 75%



Limitations & Future Work

- Overhead of auditing in compute and storage
- Assumes deterministic workflows
- Integration with data federations for data-intensive workflows
- Integretion with backpack-like mechanisms for improved collaboration
- Support a broader range of parallel and distributed computing frameworks

Conclusion

- Motivate the need for sharing and reproducing distributed workflows in notebooks successfully.
- Elaborate need for reproducibility of notebooks through different scenarios.
- Introduce NBReplay for efficient, reproducible execution for distributed workflows. It uses checkpoint and restore mechanism and avoid redundant execution through selective task caching and replay
- Experiments on notebooks using NBReplay which show good performance gains and cache effectiveness.

floability

floability.github.io

Sciunit

FLINC

github.com/radiant-systems-lab

Prof. Douglas Thain
Principal Investigator
University of Notre Dame

Prof. Tanu Malik
co-PI
University of Missouri

Prof. Kevin Lannon
co-PI
University of Notre Dame

Prof. Shaowen Wang
co-PI
University of Illinois

Md Saiful Islam
Ph.D. Student
University of Notre Dame

Talha Azaz
MS. Student
DePaul University

Ben Tovar
Research Software Engineer
University of Notre Dame

A S M Shahadat Hossain
Ph.D. Student
University of Missouri

Raza Ahmad
Ph.D. Student
DePaul University

Dr. Furqan Baig
Research Programmer
University of Illinois Urbana Champaign

This work is supported in part by the U.S. National Science Foundation under grants 251657 and 2411436