Integrating Campaign Storage into HPC Workflows

Mohamed Najem

1 Introduction

High-Performance Computing (HPC) systems are essential for addressing the complex computational challenges encountered in various fields, such as scientific research [1] [2] and industrial applications [3]. With the rapid expansion of data-driven methodologies, HPC infrastructures must efficiently manage large-scale data processing. This requires the utilization of different storage tiering strategies to optimize data accessibility and performance across diverse storage solutions. Additionally, effective job dependency management is crucial to ensure that computational tasks are executed in the correct order, thus enhancing the efficiency and reliability of workflows. These strategies are pivotal for optimizing performance and scalability in modern HPC environments.

In this context, the effective utilization of storage resources within HPC environments—particularly through the implementation of storage tiering—has become a critical area of research and development [4] [5]. Storage tiering refers to the strategic use of multiple storage systems that differ in performance, capacity, and durability to optimize data management for diverse scientific workflows. By categorizing data based on its access patterns and performance requirements, users can allocate resources efficiently; high-demand tasks can utilize faster storage options while less critical or infrequently accessed datasets can be stored in cost-effective solutions [6]. To further enhance storage efficiency, solutions such as Campaign Storage encourage users to provide detailed specifications about their storage needs, including expiry dates of the storage. This approach helps reduce bottlenecks, particularly on faster storage tiers like scratch storage, by discouraging the accumulation of cold or long-term data in high-performance areas. As the volume and complexity of data generated by these workflows continue to increase, different data management solutions, such as Campaign Storage, become essential to support performance demands and ensure optimal performance during intensive computations. Efficiently managing data can enhance computational throughput aswell as optimize the I/O Performance [7]. The integration of these heterogeneous storage systems requires various strategies for data allocation and migration, allowing users to select the most appropriate storage tier based on their specific workload requirements.

Moreover, effective management of job dependencies—where the execution of specific tasks relies on the completion of others—remains a significant challenge in HPC environments [8]. As workflows increasingly integrate various storage solutions, addressing job dependencies becomes crucial for optimizing overall system performance. By ensuring that tasks are executed in the correct sequence and that data is readily available when needed, users can maximize resource utilization and reduce potential bottlenecks. This coordination is particularly vital when using storage tiering, as different storage types may introduce variability in data access times, which can affect the timing of subsequent tasks in a workflow.

This project aims to explore the utilization of Campaign Storage, a system designed to optimize data storage and management in HPC environments. By establishing a structured approach for requesting and allocating these resources, users will be better equipped to meet the diverse demands of their research projects. However, several challenges need to be addressed, including the seamless integration of these resource requests within existing workflows and the effective management of job dependencies, which can complicate the execution of tasks. Additionally, ensuring an efficient method for data migration is critical, as large datasets need to be managed and accessed in ways that support high-performance computing workflows while minimizing delays and resource bottlenecks.

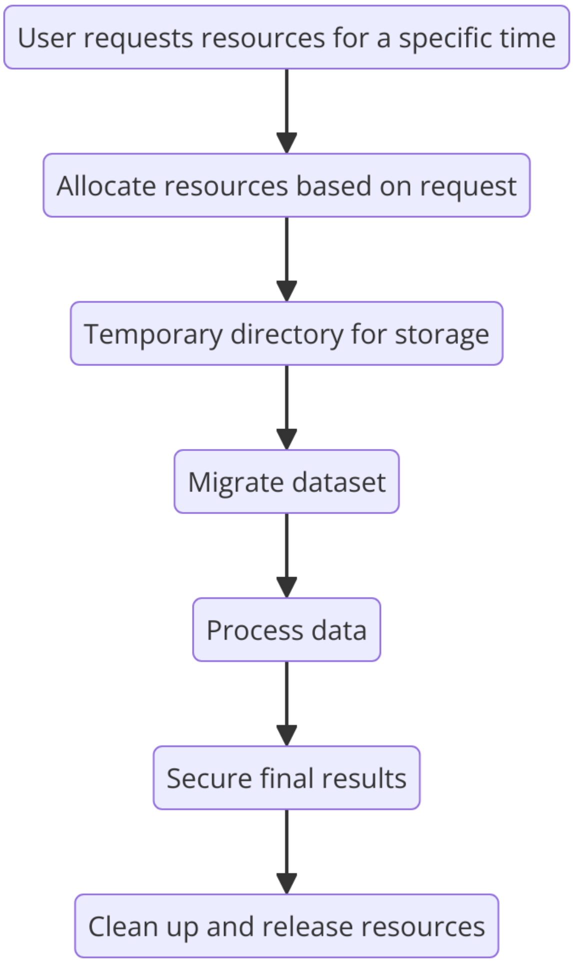

Figure 1: General Workflow.

1.1 Problem Breakdown

Objective: How can Campaign Storage systems be effectively integrated into HPC workflows in a way that is user-friendly, manages dependencies efficiently, and minimizes resource waste?

The key contributions of this work are:

- Proposing and detailing multiple design options for integrating Campaign Storage into HPC workflows.
- Evaluating the proposed designs and identifying each designs advantages and disadvantages.
- Implementing and testing the design that best meets the requirements established in Section 3.4.

Specific Use Case: To provide a consistent and practical test scenario for evaluating the design approaches, this thesis uses a machine learning workload that involves generating and processing random images. The process begins with the generation of a configurable number of random images and labels, reserving 30% for testing. During the training phase, two machine learning models are trained on 70% of the images, with each training epoch loading one random image and its metadata, including a delay between steps. In the inference phase, the remaining test images are processed in a similar manner, with a delay during loading and a slight modification to the image for explainability, followed by appending the results to a CSV file.

The structure of this thesis is as follows: Section 2 presents related work, identifying gaps that motivate this research. Section 3 analyzes the integration of Campaign Storage into HPC workflows, examining key components and identifying requirements for effective integration. Building on these requirements, Section 4 discusses different design options to address the identified challenges. Section 5 implements the selected design, detailing the approach taken and the rationale behind it. Section 6 tests the designs using the test case we just described. Section 7 evaluates the implementation, assessing its performance and suitability based on the requirements established in Section 3. Finally, Section 8 concludes the thesis and suggests directions for future work.

2 Related work

The evolution of storage technologies in High-Performance Computing (HPC) has garnered significant attention in recent years, particularly in the context of tiered storage solutions. As shown in [4], the authors outline several critical challenges faced by HPC storage systems. They emphasize the necessity for scalable storage solutions that not only enhance capacity but also minimize latency while optimizing I/O performance. The importance of multi-tiered storage systems is highlighted, as they balance fast but expensive storage with slower, more cost-effective alternatives. Furthermore, the authors discuss the automation of data management across these tiers to improve efficiency and reduce the need for manual intervention. However, the presented work lacks specific implementation strategies that can effectively integrate tiered storage solutions into existing HPC workflows, particularly with respect to specialized storage systems like Campaign Storage.

Another research [7] introduce a tiered data management system designed to optimize data movement and processing. Their structured approach for automating data placement based on access patterns effectively addresses I/O performance bottlenecks in HPC environments. By implementing a hierarchical storage model, they demonstrate how this system can reduce costs while improving performance. However, the integration of such a tiered management system into diverse existing workflows poses challenges that require further investigation, particularly regarding how specialized systems like Campaign Storage can enhance performance while managing job dependencies.

Similarly, [9] introduce Data Jockey, which automates data management across multi-tiered storage systems. The tool monitors access patterns to facilitate dynamic data movement between storage tiers, thereby optimizing system performance. While the automation provided by Data Jockey significantly enhances efficiency in large-scale HPC environments, it does not fully address the complexities of integrating such a system into existing job scheduling frameworks.

Together, these studies highlight the growing importance of tiered storage solutions in HPC while also revealing gaps related to the integration of specialized systems like Campaign Storage into existing workflows. This project aims to address these gaps by exploring how Campaign Storage can optimize data management and enhance the efficiency of HPC workflows, ultimately providing a more comprehensive solution for users working with large-scale data.

2.1 hpc-workspace Project

The [10] project provides an open-source framework for managing temporary storage directories on HPC systems. It allows users to allocate workspaces with defined expiration periods, ensuring efficient use of fast storage by preventing uncontrolled data accumulation. Key functionalities include creating, listing, and releasing workspaces through simple commands.

For this thesis, we will leverage these functionalities to integrate Campaign Storage into our HPC workflows. Specifically, the ability to allocate and release workspaces efficiently is central to the design and evaluation of the proposed solutions, enabling the management of storage throughout the training and inference processes in our use case.

3 Initial Analysis

In this section, we explore the functionalities of the HPC-Workspace discussed in the previous section. We then identify, analyze, and discuss the key components of our work. Next, we provide an example workflow, followed by an analysis of the requirements essential for this project.

3.1 HPC-Workspace

3.1.1 HPC-Workspace as Campaign Storage

The `hpc-workspace` repository provides several tools for allocating, releasing, and managing storage on HPC clusters. These tools are central to implementing Campaign Storage in an HPC environment.

The key commands provided by the repository include:

- **ws_allocate**: Allocates workspace storage for a specified duration. For example:

```
ws_allocate myworkspace 10
```

  This command allocates a workspace named `myworkspace` for 10 days.

- **ws_list**: Lists all available workspaces, allowing users to monitor and manage storage. This command helps users track their allocated workspaces and their status:

```
ws_list
```

The output shows a list of currently allocated workspaces, including their names and expiration times.

- **ws_release**: Releases previously allocated workspace storage. For example:

```
ws_release myworkspace
```

This command releases the workspace `myworkspace` after the task is complete, making the storage available for other users.

- **ws_extend**: Extends the lifetime of a workspace if the user needs more time than originally allocated. This is useful if the task takes longer than expected:

```
ws_extend myworkspace 5
```

This command extends the `myworkspace` for an additional 5 days.

- **ws_restore**: Restores data from a workspace that was deleted but is still in a recoverable state. For example:

```
ws_restore myworkspace
```

This command can restore a workspace if it was accidentally deleted and is still within the retention window.

- **ws_share**: Enables users to share access to their workspaces with others. This is particularly useful when collaborating on a project:

```
ws_share myworkspace user123
```

This command shares the workspace `myworkspace` with another user, `user123`.

These are some of the core commands provided by the HPC-Workspace tool, but there are other functionalities available for more advanced management of workspaces.

3.1.2 Configuration File

The `ws_config.conf` file is essential for configuring the workspace management system. It defines settings such as workspace expiration duration, filesystem paths, and administrative policies. Below is a simplified example highlighting key parameters, with less critical options omitted for clarity:

Listing 1: Example of ws_config.conf

```
# Workspace Configuration File

clustername: myCluster           # mandatory, a name to identify the system
smtphost: localhost              # mandatory, a host accepting smtp connections to send emails
mail_from: noreply@mydomain.com  # Sender address for email alerts
default: lustre                  # mandatory, default workspace for allocation
duration: 10                     # mandatory, max workspace duration in days
dbuid: 9999                      # mandatory, database user ID
dbgid: 9999                      # mandatory, database group ID
admins: [admin_user]             # List of admin users
adminmail: [admin@mydomain.com]  # Admin email addresses for alerts

workspaces:                      # now the list of the workspaces
  lustre:                        # name of workspace
    keeptime: 1                  # Retention time after expiration (mandatory)
    spaces: [/lustre1/ws,...]    # mandatory, list of directories
    database: /lustre-db         # mandatory, this is where DB files will end
    deleted: .removed            # mandatory, will be appended to spaces and database
    maxextensions: 5             # maximum extensions allowed
    ...
  nfs:                           # second worksapce
    keeptime: 1
    ...
```

For less critical or specialized options, such as custom ACLs, space selection rules, or additional workspace-specific settings, administrators can expand the file configuration as needed. The file can also be validated using the command `ws_validate_config`.

Furthermore, the config file can be used to store default values and can then be used alongside commands like `ws_allocate` to manage workspace allocations efficiently.

3.2 Key Components

- **Resource Manager:** The Resource Manager is essential for allocating and overseeing computing resources based on user requests and the specific requirements of workloads. It ensures that the necessary resources—such as CPUs, memory, and storage—are available for each task, optimizing resource utilization and minimizing idle time. This component plays a vital role in enhancing the efficiency and effectiveness of computational processes.
- **Workflow Engine:** The Workflow Engine manages the execution of tasks and their dependencies. It ensures that tasks are executed in the correct order, tracking the status of each task while coordinating data flow between them. By systematically identifying and resolving dependencies, the Workflow Engine enhances the reproducibility and reliability of computational tasks, making it an integral part of the workflow system.
- **Data Migration Tools:** Efficient data migration is crucial for ensuring that all required datasets are accessible during execution. Data migration tools facilitate the movement of data between campaign storage and computational resources, ensuring that the necessary data files are available when needed. By implementing effective data migration solutions, we can streamline data transfers and optimize the overall performance of the workflow system.

3.2.1 Resource Manager: Slurm

Slurm (Simple Linux Utility for Resource Management) serves as the primary resource manager in our integrated workflow system. It is responsible for allocating computing resources, managing job queues, and monitoring resource usage. Slurm supports various scheduling policies, enabling efficient allocation of resources based on user-defined priorities and job requirements. Key functionalities of Slurm include:

- **Job Scheduling:** Slurm determines the order of job execution based on resource availability and priority, optimizing resource utilization.
- **Resource Allocation:** It dynamically allocates CPUs, memory, and other resources as requested by users, ensuring that computational tasks are efficiently executed.
- **Monitoring and Reporting:** Slurm provides detailed reporting on resource usage and job status, allowing administrators and users to track performance and optimize workflows.

By using Slurm, we can effectively manage and allocate resources, enhancing the overall efficiency of the workflow system.

3.2.2 Workflow Engine: Snakemake

Snakemake [11] acts as the workflow engine in the integrated system and manages execution of computational tasks. Its primary role is to define workflows in a reproducible and efficient manner by managing dependencies between tasks. Key features of Snakemake include:

- **Dependency Management:** Snakemake automatically identifies and resolves dependencies by constructing a directed acyclic graph (DAG) from the defined rules, ensuring that tasks are executed in the correct order.
- **Resource Specification:** Users can specify the resources required for each task, allowing Snakemake to optimize resource allocation within the Slurm environment.
- **Environment Management:** Snakemake integrates with Conda and Docker to manage software environments, avoiding compatibility issues and ensuring that the required packages are available for each task.

By using Snakemake, we can enhance the reproducibility and reliability of workflows, allowing researchers to execute complex computational tasks with confidence.

3.2.3 Data Migration Tools: Rclone and Rsync

Rclone and Rsync are critical components for efficient data migration within the integrated workflow system. These tools facilitate the transfer of data between campaign storage and computational resources, ensuring that necessary datasets are accessible for execution. Key aspects of Rclone and Rsync include:

- **Rsync:** Rsync enables efficient synchronization by transferring only the differences between source and destination files. This minimizes data transfer overhead and enhances efficiency when updating large datasets.

- **Rclone:** Rclone offers flexibility in managing data across various cloud storage environments, making it suitable for workflows that involve multiple storage solutions. It supports numerous cloud providers and provides robust capabilities for data synchronization and backup.

3.2.4 Discussing Chosen Components

In selecting the components for our integrated workflow system, we prioritized solutions that offer robustness, flexibility, and efficiency in managing computational tasks and data migrations. Below, we discuss the reasons behind choosing Slurm, Snakemake, and Rclone/Rsync, aswell as showing an example of how they can be used.

- Slurm was selected for its effective management of job scheduling and resource allocation in HPC environments. Its scalability and support for diverse job types make it very suitable for executing computational tasks, thereby ensuring optimal resource utilization. Additionally, Slurm's architecture facilitates the integration of custom plugins, allowing users to extend its functionality to meet specific workflow requirements. Moreover, Slurm is the most widely used resource manager; in 2019, approximately 60% of the TOP500 supercomputers worldwide employed Slurm [12].
- Snakemake serves as our workflow engine due to its powerful dependency management capabilities, intuitive syntax, and high flexibility in handling diverse computational tasks. One of Snakemake's core features is its ability to efficiently handle complex, multi-step workflows with minimal manual intervention. It automatically tracks dependencies between tasks, ensuring that each step is executed only when its conditions are fulfilled, which avoids redundant computations. This not only guarantees reproducibility but also enhances performance by allowing parallel execution and resource optimization. Snakemake's integration with tools like Conda and containerization (Docker, Singularity) makes it highly portable and adaptable across different computational environments, enabling seamless execution on clusters, cloud systems, or local machines. Furthermore, the ease of integrating additional tools such as Rclone or Rsync into Snakemake workflows allows users to automate both computational and data transfer tasks within a single system, simplifying complex workflows for users. As demonstrated in [13], Snakemake has been successfully implemented in large-scale academic projects, making it a widely trusted and scalable solution in scientific computing.
- Rclone and Rsync were selected for their flexibility in managing data transfers between local and remote storage systems. These tools minimize data transfer overhead and streamline the movement of large datasets, which is critical for ensuring data accessibility during processing [14] [15]. Furthermore, a survey shows, these tools are the preferred choice among users at the University of Göttingen, where this work is taking place.

3.3 Example Workflow

The following example workflow expands on the process shown in Figure 1 and illustrates the proposed integration of Campaign Storage within HPC tasks.

- **Resource Request:** A user will start by requesting a resource, for example a block device, for a specific duration. This request will be made using the developed job script syntax, ensuring that the necessary campaign storage is allocated.
- **Campaign Storage:** The campaign storage will be mimicked with a workspace, specifically by creating a temporary directory on the SCRATCH filesystem. This approach provides a dedicated space for data processing tasks while maintaining efficiency in resource usage.
- **Data Migration:** A dataset will be migrated into the designated storage area using the selected data migration tools. This ensures that the data is readily accessible for computational tasks and aligns with the workflow's requirements.
- **Machine Learning Execution:** Machine learning tasks will be conducted on the migrated dataset, leveraging the computational resources effectively. This step highlights the practical application of the workflow system in executing complex data-driven tasks.
- **Model Preservation:** The optimally trained model will be securely stored after the machine learning tasks are completed. This ensures that the results of the computational processes are preserved for future use and analysis.
- **Clean Up and Release Resources:** After securing the final results, temporary data and workspaces are removed, and resources are released back to the pool.

3.4 Requirement Analysis

The design and implementation of integrating Campaign Storage into HPC workflows must meet the following key performance indicators (KPIs):

- **Usability:** The process of requesting and managing Campaign Storage should be intuitive and user-friendly, encouraging adoption and minimizing the learning curve even for non-expert users. Dependency management should be handled seamlessly, with minimal manual intervention required for defining job dependencies and data availability between interdependent jobs.
- **Performance (Time-to-Solution):** The integration should optimize the time required to complete the entire workflow, including resource allocation, data staging, and workload execution. Efficient allocation and release of storage resources should minimize idle time and resource contention, contributing to improved time-to-solution.
- **Resource Management and Allocation Efficiency:** Storage resources in HPC environments are limited and shared among multiple users and workflows. Efficient allocation ensures that Campaign Storage is available when needed while avoiding idle or wasted resources.
- **Data Access and Management:** Campaign Storage must support seamless reading, writing, and modification of data during workflow execution. The integration should ensure compatibility with existing workflow tools to prevent disruptions in data handling.
- **Portability:** The solution should be adaptable to different HPC environments with minimal configuration changes, ensuring flexibility and wider applicability.
- **User Isolation and Reproducibility:** The integration should maintain user isolation to prevent resource conflicts and ensure reproducibility of performance across different workflow executions.
- **Cost Efficiency:** Effective use of storage and computational resources is essential to reduce operational costs and optimize resource utilization across shared environments.

4 Design Options

In this section, we present various design options for integrating Campaign Storage into HPC workflows.

4.1 Manual Use of the Campaign Storage API

This design involves manually managing all aspects of the workflow. The user directly handles storage requests, data staging, and workload execution.

Process:

1. **Storage Request:**
   - The user logs into the HPC system via SSH.
   - They manually execute an API call (e.g., using curl) to request Campaign Storage. The request includes parameters like storage size, duration, and potentially metadata about the job.
   - The user monitors the allocation status by polling the API at regular intervals or checking for a notification (if supported).
2. **Data Staging:**
   - Once the storage is ready, the user manually transfers the required data from long-term storage to the allocated Campaign Storage.
   - The user ensures that all necessary input files are correctly staged for the workload.
3. **Workload Execution:**
   - After data staging is complete, the user submits a batch job to the resource manager (e.g., SLURM) to execute the workload. The batch job script references the staged data and writes outputs back to Campaign Storage or another target location.

Scheduling: Scheduling in this approach is entirely manual. Dependencies exist between the API call, data staging, and workload execution. The user must ensure that each step completes successfully before moving to the next.

- The user cannot stage data until the Campaign Storage is available.
- The workload cannot start until the required data has been staged.
- These dependencies are managed by the user, who must manually check the readiness of each step.

4.2 Using SLURM to Make API Calls

SLURM is used to submit and execute jobs, but it does not directly manage the Campaign Storage resources. Instead, API calls for storage provisioning are handled externally, either within the same job script or as a separate job.

Option 1: Single Job Script for Sequential Execution

Process:

1. **Storage Request:**
    - The user writes a batch script that performs the API call (e.g., using curl) to request Campaign Storage.
    - This script also includes commands for data staging and workload execution in sequence.
    - The script is submitted to SLURM, which schedules it like any other job.
2. **Data Staging and Workload Execution:**
    - The storage request, data staging, and workload execution are handled sequentially within the same script.
    - The dependencies are implicitly managed by the script itself, with each step waiting for the previous one to complete.

Option 2: Separate Jobs with Explicit Dependencies

Process:

1. **Storage Request:**
    - The API call is submitted as a separate SLURM job.
    - The storage allocation status can be checked through SLURM.
2. **Data Staging and Workload Execution:**
    - A second job handles data staging and workload execution.
    - This job specifies a dependency on the first job using SLURM's `--dependency` option, ensuring it only starts once the storage request job has completed successfully.

Scheduling:

- **Option 1:** SLURM schedules the script as a single job, and all steps—storage request, data staging, and workload execution—are handled sequentially within the script. Dependencies are implicitly managed by the script logic itself. SLURM only schedules the job as a whole, without explicit knowledge of the individual steps.
- **Option 2:** SLURM schedules the jobs separately, with dependencies explicitly managed through its `--dependency` option. For example, a job for the storage request is submitted first, and the subsequent job (for data staging and workload execution) is set to run only after the completion of the storage request job.
- **Storage and Workload Separation:** SLURM schedules a staging job first and then a separate workload execution job, with storage availability acting as the dependency.

4.3 SLURM Plugin Integration

This design integrates Campaign Storage directly into SLURM using a custom plugin. SLURM manages Campaign Storage as a resource, similar to CPUs or memory. A plugin would need to be developed using the SLURM Plugin API (SPANK), which provides a framework for extending SLURM's functionality by allowing custom actions during job lifecycle events (e.g., submission, start, or completion). Developing such a plugin requires integrating the Campaign Storage API with SLURM's resource management system.

Process:

1. **Storage Request:**
    - The user specifies Campaign Storage requirements in their SLURM job script.
2. **Data Staging and Workload Execution:**
    - When the job is submitted, the SLURM plugin automatically handles the API call to request Campaign Storage.
    - The plugin monitors the status of the storage request and ensures that the job does not start until the requested storage is available.
    - The same job handles data staging and workload execution once the storage is ready.

SLURM Workflow:

- **SLURMCTLD**, the central management daemon, is responsible for accepting job requests and allocating resources. It interacts with **SLURMD**, the compute node daemon, to ensure jobs are distributed to the appropriate nodes for execution.
- Initially, a job to stage the data will be submitted. This job will ensure that the required data is placed into the allocated Campaign Storage.
- Once the data is staged, the workload job is executed, using the `--reservation` flag to ensure that the storage remains allocated and that the workload runs in the appropriate environment.

Scheduling: The SLURM plugin integrates scheduling for Campaign Storage directly into SLURM's scheduling system. Dependencies between storage readiness and workload execution are managed by the plugin. Users do not need to wait for storage availability, as SLURM ensures the necessary resources are allocated automatically. Additionally, the plugin facilitates the separation of job stages—first staging, then execution—through resource reservation.

4.4 Using Snakemake

Snakemake manages the entire workflow, including Campaign Storage requests, data staging, and workload execution. It automates dependency tracking and ensures steps are executed in the correct order.

Process

1. **Storage Request:**
   - A rule in the Snakefile handles the API call for Campaign Storage using tools like curl or Python's libraries. This rule defines the storage allocation as its output, ensuring following steps wait for the storage to be ready.
2. **Data Staging and Workload Execution:**
   - Subsequent rules depend on the storage rule's output. These rules stage data to Campaign Storage and execute the workload.
   - For complex workflows, Python logic can be used in the run directive to dynamically manage steps.
   - **Workload Execution as a Separate SLURM Job:** The workload execution step is treated as a separate SLURM job, independent of the storage request and data staging steps. Snakemake ensures that these jobs are submitted in the correct order, but the workload itself is executed as a distinct SLURM job once the storage is ready and data has been staged.

Scheduling: Snakemake inherently handles scheduling through its dependency management system. Each rule is executed only when its conditions (e.g., Campaign Storage readiness) are satisfied. This ensures that the entire workflow, from storage requests to workload execution, is automatically and efficiently managed. Since Snakemake automatically manages job sequencing and storage allocation, the user does not need to manually wait for storage or manage dependencies. No '–dependency' flag in SLURM is necessary, as Snakemake ensures the proper order of execution without requiring explicit job dependencies to be set in SLURM. Notifications can be configured to inform the user when the storage is allocated or when the job is complete.

4.5 Preliminary Analysis

This section analyzes the proposed designs, focusing on their feasibility, complexity, and applicability to the project. Based on this analysis, we select the following three designs for implementation: Manual Use of the Campaign Storage API, SLURM with Explicit Job Dependencies (Option 2), and Using Snakemake. The selection logic for each design is discussed below, along with reasons for excluding other designs.

4.5.1 Selected Designs

1. Manual Use of the Campaign Storage API

- This design is the simplest and most direct approach, providing a baseline implementation to understand the fundamental aspects of integrating Campaign Storage without additional dependencies. By manually managing storage requests, job execution, and storage release, this design offers insights into the mechanics of the system and user experience without automation. Although less practical for complex workflows, it is ideal for testing core system functions and serves as a foundation for comparing more advanced approaches.

2. Option 2: SLURM with Explicit Job Dependencies

- This design leverages SLURM's `--dependency` flag to explicitly manage dependencies between the storage request and subsequent jobs. Compared to Option 1, this approach offers better resource control and job sequencing, ensuring that storage is allocated before subsequent tasks are executed. This explicit dependency management reduces errors caused by premature job execution and enhances overall workflow efficiency. Additionally, it provides an intermediate solution between full manual management and complete automation.

3. Using Snakemake

- Snakemake excels in automating complex workflows with multiple dependencies. Its rule-based system aligns with the project's goal of improving automation and user-friendliness in HPC workflows. Snakemake also simplifies dependency tracking and execution, reducing the potential for user error in managing Campaign Storage and workflow dependencies.

4.5.2 Excluded Designs

1. SLURM Batch Script without Explicit Dependencies (Option 1)

- This design was excluded as it does not fully leverage SLURM's dependency management features. Without explicit dependency management, there is a higher risk of workflow errors caused by storage unavailability. Additionally, this design is not practical for complex workflows and offers less control over resource scheduling compared to Option 2.

2. SLURM Plugin Integration

- Developing a SLURM plugin is a highly complex and resource-intensive task, requiring in-depth knowledge of SLURM's Plugin API. This design is not feasible within the project's scope and exceeds the level of complexity necessary to achieve the project's goals.

5 Implementation

This section focuses on implementing three selected designs for integrating HPC-Workspace as Campaign Storage in HPC workflows: (1) Manual Use of the Campaign Storage API, (2) Separate Jobs with Explicit Dependencies, and (3) Snakemake Workflow. It details how the `hpc-workspace` repository is used to manage workspace storage and provides examples of workflow implementation.

5.1 Design 1: Manual Use of Campaign Storage API

This design offers a straightforward approach where the user manually interacts with the Campaign Storage system via command-line tools to allocate and release storage. The process requires the user to SSH into the cluster, allocate storage, run tasks, and finally release the storage.

5.1.1 Steps for Manual Use of Campaign Storage API

1. **SSH into the Cluster:** To begin, the user logs into the HPC cluster using SSH:

```
ssh user@cluster-address
```

2. **Allocate Campaign Storage:** The user allocates a workspace for a specific duration (e.g., 10 days) using the `ws_allocate` command:

```
ws_allocate myworkspace 10
```

3. **Run the Task:** The user then submits a batch job to execute the required computations.

```
sbatch job_1.sh
sbatch job_2.sh
sbatch job_3.sh
```

4. **Release Campaign Storage:** After completing the task, the user releases the allocated storage to free up resources:

```
ws_release myworkspace
```

5.2 Design 2: Separate Jobs with Explicit Dependencies

This design is slightly more automated than the first, as it utilizes **SLURM** job dependencies to ensure a structured workflow. Two separate jobs are used—one for storage allocation and another for data staging and task execution. The second job is dependent on the successful completion of the first job using the --dependency=afterok:<job_id> flag.

5.2.1 Sample SLURM Job Scripts

1. **Storage Allocation Job (`storage_request.sh`):**

Listing 2: Storage Allocation Job (storage_request.sh)

```
#!/bin/bash
#SBATCH --job-name=storage_request
#SBATCH --output=storage_request.out
#SBATCH --time=00:05:00
#SBATCH --partition=standard

# Allocate workspace storage for 10 days
ws_allocate myworkspace 10
```

This script performs the following key operations:

- `#!/bin/bash`: Specifies that the script should be executed in the bash shell.
- `#SBATCH --job-name=storage_request`: Names the SLURM job for easier tracking.
- `#SBATCH --output=storage_request.out`: Specifies the file to store job output.
- `#SBATCH --time=00:05:00`: Sets the wall time for the job.
- `ws_allocate myworkspace 10`: Allocates workspace storage for 10 days.

2. **Data Staging and Workload Execution Job (`workload.sh`)**

Listing 3: Data Staging and Workload Execution Job

```
#!/bin/bash
#SBATCH --job-name=workload_execution
...
#SBATCH --dependency=afterok:<job_id>

# Stage data and run task
cp input_data/* $SCRATCH
python3 run_task.py

# Release storage
ws_release myworkspace
```

This job script executes the task only after the successful completion of the storage request job using the `--dependency=afterok:` parameter. The key operations include:

- `#SBATCH --job-name=workload_execution`: Names the SLURM job.
- #SBATCH --dependency=afterok:<job_id>: Sets a dependency ensuring this job starts only after the previous job completes successfully.
- `cp input_data/* $SCRATCH`: Copies input data for processing.
- `python3 run_task.py`: Runs the required task.
- `ws_release myworkspace`: Releases the workspace after the task completes.

Additional SLURM Parameters : SLURM provides numerous options for job submission beyond those demonstrated. Some examples include:

- `-N`: Specifies the number of nodes for the job.
- `--mem`: Sets the memory per node required for the job.
- `--mail-type`: Configures email notifications for job start, end, or failure events.

5.2.2 Extended Job Submission Example

The following listing demonstrates another example that chains multiple jobs together, including storage allocation, task execution, and cleanup into a single file:

Listing 4: Extended Modular SLURM Job Example

```
#!/bin/bash

# Job A - Allocate Storage
job_A=$(sbatch --job-name=allocate_storage \
               ...
               --wrap="ws_allocate myworkspace 10")

# Extract the job ID for Job A
job_A_ID=$(echo $job_A | awk '{print $4}')

# Job A failure handler (runs if Job A fails)
sbatch --job-name=allocate_storage_failed \
       ...
       --dependency=afternotok:$job_A_ID \
       --wrap="echo 'Job A failed.'"

# Job B - Perform Task (depends on successful storage allocation)
job_B=$(sbatch --job-name=perform_task \
               ...
               --dependency=afterok:$job_A_ID \
               --wrap="python3 task.py")

# Extract the job ID for Job B
job_B_ID=$(echo $job_B | awk '{print $4}')

# Job B failure handler (runs if Job B fails)
sbatch --job-name=perform_task_failed \
       ...
       --dependency=afternotok:$job_B_ID \
       --wrap="echo 'Job B failed.'"

# Job C - Release Storage (depends on task completion)
sbatch --job-name=release_storage \
       ...
       --dependency=afterok:$job_B_ID \
       --wrap="ws_release myworkspace"

# Job C failure handler (runs if Job C fails)
sbatch --job-name=release_storage_failed \
       ...
       --dependency=afternotok:$job_C_ID \
       --wrap="echo 'Job C failed.'"
```

This workflow demonstrates three sequential jobs:

- `job_A`: Allocates workspace storage using the `ws_allocate` command.
- `job_B`: Runs a task (in this case, a Python script) after successful storage allocation.
- `job_C`: Releases the workspace storage after the task completes.
- `job_A_failed`: Handles the case when `job_A` fails. This job runs only if `job_A` fails.
- `job_B_failed`: Handles the case when `job_B` fails. This job runs only if `job_B` fails.
- `job_C_failed`: Handles the case when `job_C` fails. This job runs only if `job_C` fails.

The key elements of this workflow are as follows:

- **Job ID Extraction:** After each job submission, the SLURM scheduler returns a message in the format:

```
Submitted batch job <job_id>
```

To extract the numerical job ID, we use the command:

```
echo $job_A | awk '{print $4}'
```

Explanation:

- `echo $job_A` displays the output string of the SLURM submission command.
- The `|` (pipe) operator passes this output as input to the next command.
- `awk '{print $4}'` extracts the fourth field (i.e., the job ID) from the output string, assuming whitespace is the delimiter.

This job ID is required for setting dependencies between jobs.

- **Setting Dependencies:** The `--dependency=afterok:$job_ID` parameter ensures that a job only starts after the successful completion of a preceding job. This prevents resource conflicts and allows for proper sequencing of tasks. In addition, failure handling is implemented using the `--dependency=afternotok:$job_ID` parameter, which ensures that a failure in a previous job triggers a corresponding failure handler.
- **Failure Handlers:** To handle job failures and take appropriate action, we use the `afternotok` dependency flag. This causes a job to execute only if the previous job fails. Specifically:
  - `job_A_failed`: This job will only run if `job_A` fails. It handles failure scenarios for the storage allocation step.
  - `job_B_failed`: This job will only run if `job_B` fails. It handles failure scenarios for the task execution step.
  - `job_C_failed`: This job will only run if `job_C` fails. It handles failure scenarios for the storage release step.
- **Efficient Resource Management:** By separating storage allocation, task execution, and cleanup into distinct jobs, this workflow allows for better control and monitoring of each phase. Failure handling jobs ensure that resources can be freed or alternate actions taken in case of failures.
- **The `--wrap` Option:** The `--wrap` option in SLURM allows you to run a single command or script within the job without needing to create a separate script file. It effectively "wraps" the command in a SLURM batch job, which is particularly useful for short commands or for keeping job scripts concise. In this example:
  - `--wrap="ws_allocate myworkspace 10"` runs the `ws_allocate` command to allocate workspace storage.
  - `--wrap="python3 task.py"` runs a Python script as part of the task execution.
  - `--wrap="ws_release myworkspace"` runs the command to release the storage after the task is complete.

  This approach simplifies job submission since you don't need to manage separate script files for each task, and it helps reduce overhead in workflows where commands are simple or self-contained.
- **Problem:** While this example may work for simple workflows, it becomes error-prone and impractical for more complex workflows, especially when users need to modify the job sequence or dependency structure. Failure handling ensures that jobs can be re-run or remedied without manual intervention.

5.3 Design 3: Snakemake Workflow

The third design automates the entire process using **Snakemake**, which defines rules for allocating storage, executing tasks, and releasing storage in a systematic and repeatable manner. Snakemake handles the dependency management automatically, allowing users to focus on task definitions without worrying about the sequence of operations.

5.3.1 Sample Snakemake Rule

Below is an example Snakemake rule that handles the entire process of allocating storage, running a task, and releasing storage:

Listing 5: Snakemake Rule Example

```
rule all:
    input:
        "storage_released.txt"
```

```
rule allocate_storage:
    output:
        "storage_allocated.txt"
    shell:
        "ws_allocate myworkspace 10 && echo 'done' > {output}"

rule job_a:
    input:
        "storage_allocated.txt"
    output:
        "job_a_complete.txt"
    shell:
        "echo 'Running job A' > {output}"

rule job_b:
    input:
        "storage_allocated.txt"
    output:
        "job_b_complete.txt"
    shell:
        "echo 'Running job B' > {output}"

rule release_storage:
    input:
        "job_a_complete.txt", "job_b_complete.txt"
    output:
        "storage_released.txt"
    shell:
        "ws_release myworkspace && echo 'done' > {output}"
```

- **rule all:** This is the entry point for Snakemake. It defines the final target output for the entire workflow (`storage_released.txt`). Snakemake resolves the necessary steps to achieve this target by considering the dependencies between the various rules. The workflow will only be complete once the storage is released.
- **allocate_storage rule:** This rule is responsible for allocating workspace storage using the `ws_allocate` command. Once the storage is successfully allocated, it generates the file `storage_allocated.txt`, indicating the completion of the storage allocation step. The rule is simple and does not depend on any prior tasks, but it serves as the foundation for subsequent tasks that require the allocated storage.
- **job_a rule:** This rule represents the first task (job A) that depends on the successful allocation of storage (`storage_allocated.txt`). Once storage is available, this job executes and generates `job_a_complete.txt`. In practice, this could be any task, such as preprocessing, model setup, or data preparation. The output file signals the successful completion of the task.
- **job_b rule:** Similarly to `job_a`, the `job_b` rule depends on the completion of storage allocation (`storage_allocated.txt`). It generates `job_b_complete.txt` upon successful completion. Both `job_a` and `job_b` can run in parallel after storage allocation.
- **release_storage rule:** This rule depends on the successful completion of both `job_a` and `job_b`. It ensures that storage is only released after all the critical tasks have been completed. The release process is handled by the `ws_release` command, and the file `storage_released.txt` is generated, indicating the successful release of the allocated storage.
- **Automatic Dependency Management:** Snakemake handles the execution order by automatically resolving the dependencies between rules. For instance:
  - If storage is not yet allocated, Snakemake ensures that the `allocate_storage` rule runs before any other jobs.
  - `job_a` and `job_b` are executed in parallel once storage is allocated.
  - The `release_storage` rule waits until both `job_a` and `job_b` are completed before releasing the storage.
- **Shell Block Usage:** Each rule uses a shell block to specify the sequence of commands that should be executed. For example, in the `allocate_storage` rule, storage is allocated and, if successful, a confirmation message is written to the output file. This is achieved using the && operator, which ensures that the `echo` command only runs if the storage allocation completes successfully. In the `release_storage` rule, the storage is released after

confirming the completion of prior tasks. Using shell blocks allows for easy definition of multi-step tasks within a single Snakemake rule.

- **Reproducibility and Scalability:** Snakemake facilitates the creation of reproducible workflows that can be easily scaled to fit complex HPC jobs. This structure (separating storage allocation, task execution, and cleanup) ensures that the workflow remains flexible and efficient even as the number of tasks grows.

6 Testing

This chapter evaluates the three implemented designs through a specific use case: a machine learning (ML) workflow involving image data. As outlined in Chapter 1.2, this workflow aims to test Campaign Storage integration while simulating typical ML tasks, such as data generation, model training, and inference. For simplicity and testing purposes, random images are generated instead of using a real-world dataset.

6.1 Use Case Overview

The workflow comprises the following stages:

1. **Preprocessing:** Generate random images and labels, store them in a structured directory, and prepare the data for training.
2. **Training:** Train machine learning models using the generated data.
3. **Inference:** Evaluate models on test data, simulate explainability, and store results.

Each stage is executed via a Python driver script:

Listing 6: Workflow Execution Command

```
python3 workflow.py --task <task_name>
```

Tasks include `--preprocess`, `--train`, and `--inference`, detailed below.

6.2 Task Explanations

6.2.1 Preprocessing

The preprocessing task generates random images and labels, which are then saved in a directory called `generated_images`. Labels are written to a CSV file for later use. This step ensures data organization for efficient training and testing.

Listing 7: Directory Creation for Image Storage

```
storage_path/
    generated_images/
        img_0.png
        img_1.png
        ...
    labels.csv
```

Core Operation: The following code snippet demonstrates how images and labels are generated and stored:

Listing 8: Generating and Storing Images and Labels

```
images = []
labels = []
for _ in range(n):
    # Generate a random image
    img = np.random.randint(0, 256, size=(image_size[0], image_size[1], 3), dtype=np.uint8)
    images.append(img)

    # Create a random label (0 or 1)
    label = random.randint(0, 1)
    labels.append(label)
```

- The code generates n images of the specified size with pixel values between 0 and 255.
- Each image is paired with a random label (0 or 1), simulating a binary classification task.
- The generated images are saved in the `generated_images` directory, and labels are appended to a CSV file for future use.

6.2.2 Training

In this task, machine learning models are trained using the preprocessed data. To simulate computational time, delays are introduced with sleep functions.

Listing 9: Simulated Training Process

```
for epoch in range(epochs):
    for step in range(steps_per_epoch):
        img_idx = random.randint(0, len(train_images) - 1)
        img = train_images[img_idx]
        label = train_labels[img_idx]

        # Simulate training time
        time.sleep(0.1)
```

- The code iterates over a predefined number of epochs and steps per epoch.
- In each step, a random training sample (image and label) is selected for processing.
- The `time.sleep(0.1)` simulates computation time, mimicking real model training.

6.2.3 Inference

The inference task evaluates the trained models on test data. Each image is slightly modified to simulate explainability, and the results are saved in a CSV file.

Listing 10: Inference Workflow

```
for idx, (img, label) in enumerate(zip(test_images, test_labels)):
    # Simulate inference with a small transformation
    modified_img = img + 1

    # Save the modified image
    img_path = os.path.join(modified_images_path, f"modified_{idx}.png")
    save_image(modified_img, img_path)

    # Append the result to a CSV file
    results.append([img_path, label])
```

- Images are incrementally modified (by adding 1 to pixel values) to simulate an explainability step.
- The modified images are saved in the `modified_images` directory.
- Results, including image paths and labels, are stored in a CSV file for further analysis.

6.3 Design 1: Manual Use of the Campaign Storage API

This design assumes direct user interaction with the Campaign Storage API to allocate and release storage before and after each task. The user manually executes commands for storage management and workflow steps, providing complete control but requiring active intervention.

Workflow Example:

- **Allocate storage:**

  Listing 11: Allocating workspace for 10 days

  ```
  ws_allocate myworkspace 10
  ```

  As explained in Chapter 5 this command creates a workspace named `myworkspace` with a duration of 10 days.

- **Run preprocessing:**

  Listing 12: Running the preprocessing task

  ```
  python3 workflow.py --task preprocess
  ```

This step executes the preprocessing task using a Python driver script. In this case, random image data is generated and stored in the allocated workspace.

- **Run training:**

Listing 13: Running the training task

```
python3 workflow.py --task train
```

Here, machine learning models are trained using the previously generated data.

- **Run inference:**

Listing 14: Running the inference task

```
python3 workflow.py --task inference
```

This step performs inference on test data using the trained models and saves the results.

- **Release storage:**

Listing 15: Releasing the allocated storage

```
ws_release myworkspace
```

After all tasks are complete, this command releases the storage.

The commands must be executed in the specified order. While this approach is manageable for simple workflows, it becomes impractical for those with multiple dependencies or frequent changes in task order, as manually managing each step increases the risk of errors and inefficiencies, particularly in more complex scenarios.

6.4 Design 2: Separate Jobs with Explicit Dependencies

This design uses SLURM job scheduling to automate task execution by chaining tasks through job dependencies. Each task is defined in its own job script, improving clarity and maintainability.

6.4.1 Job Scripts

Listing 16: allocate_storage.sh - Allocating Storage

```
#!/bin/bash

#SBATCH --job-name=storage_request
#SBATCH --output=storage_request.out
#SBATCH --time=00:05:00
#SBATCH --partition=standard

# Allocate storage for 10 days
ws_allocate myworkspace 10
```

This script creates a job to allocate storage for 10 days using the 'ws_allocate' command.

Listing 17: preprocess.sh - Preprocessing Task

```
#!/bin/bash
#SBATCH --job-name=preprocess_task
#SBATCH --partition=normal
#SBATCH --cpus-per-task=4
#SBATCH --time=02:00:00
#SBATCH --output=preprocess_task.out
#SBATCH --error=preprocess_task.err

# Preprocessing Task
python3 workflow.py --task preprocess
```

This job script runs the preprocessing step by executing the 'workflow.py' script with the '–task preprocess' argument.

Listing 18: train.sh - Training Task

```
#!/bin/bash
#SBATCH --job-name=train_task
#SBATCH --partition=gpu
#SBATCH --gpus=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=32G
#SBATCH --time=04:00:00
#SBATCH --output=train_task.out
#SBATCH --error=train_task.err

# Load any required modules
module load python

# Activate the virtual environment if necessary
# source /path/to/your/env

# Training Task
python3 workflow.py --task train
```

The training task is executed using the '–task train' argument.

Listing 19: inference.sh - Inference Task

```
#!/bin/bash
#SBATCH --job-name=inference_task
#SBATCH --partition=cpu
#SBATCH --cpus-per-task=4
#SBATCH --mem=16G
#SBATCH --time=02:00:00
#SBATCH --output=inference_task.out
#SBATCH --error=inference_task.err

# Load any required modules
module load python

# Activate the virtual environment if necessary
# source /path/to/your/env

# Inference Task
python3 workflow.py --task inference

# Release the workspace storage after completion
ws_release myworkspace
```

This job script performs inference using the '–task inference' argument and releases the allocated storage using 'ws_release myworkspace'.

6.4.2 Job Submission Script

Listing 20: submit_jobs.sh - Submitting Jobs Sequentially

```
#!/bin/bash

# Submit storage allocation job
job_1=$(sbatch --parsable allocate_storage.sh)

# Submit preprocessing job dependent on Job 1
job_2=$(sbatch --parsable --dependency=afterok:$job_1 preprocess.sh)

# Submit preprocessing failure handler job for Job 1
sbatch --dependency=afternotok:$job_1 preprocess_failed.sh

# Submit training job dependent on Job 2
```

```
job_3=$(sbatch --parsable --dependency=afterok:$job_2 train.sh)

# Submit training failure handler job for Job 2
sbatch --dependency=afternotok:$job_2 train_failed.sh

# Submit inference job dependent on Job 3
sbatch --dependency=afterok:$job_3 inference.sh

# Submit inference failure handler job for Job 3
sbatch --dependency=afternotok:$job_3 inference_failed.sh
```

- The script first submits ‘allocate_storage.sh‘ to allocate the necessary workspace and captures the job ID in ‘job_1‘.
- The ‘–dependency=afterok:$job_1‘ flag ensures that ‘preprocess.sh‘ starts only if ‘allocate_storage.sh‘ completes successfully.
- This dependency chain continues for the ‘train.sh‘ job, ensuring that preprocessing completes before training begins.
- The ‘inference.sh‘ job is also dependent on the successful completion of the training job, ensuring that inference only starts after training is done.
- Using ‘–parsable‘ enables capturing job IDs directly for further dependencies.
- No failure handler jobs are included in this script, as the assumption is that each step should proceed to the next unless explicitly handled elsewhere.

6.5 Design 3: Snakemake Workflow

This design automates the workflow using Snakemake, including task execution and storage management.

6.5.1 Sample Workflow for Snakemake

Listing 21: Snakefile

```
rule all:
    input:
        "inference_complete.txt"

rule allocate_storage:
    output:
        "storage_allocated.txt"
    shell:
        "ws_allocate myworkspace 10 && echo 'done' > {output}"

rule preprocess:
    input:
        "storage_allocated.txt"
    output:
        "preprocessing_complete.txt"
    shell:
        "python3 workflow.py --task preprocess && echo 'done' > {output}"

rule train:
    input:
        "preprocessing_complete.txt"
    output:
        "training_complete.txt"
    shell:
        "python3 workflow.py --task train && echo 'done' > {output}"

rule inference:
    input:
        "training_complete.txt"
    output:
```

```
        "inference_complete.txt"
    shell:
        "python3 workflow.py --task inference && echo 'done' > {output}"

rule release_storage:
    input:
        "inference_complete.txt"
    output:
        "storage_released.txt"
    shell:
        "ws_release myworkspace && echo 'done' > {output}"
```

- **allocate_storage**: Allocates storage using the `ws_allocate` command and creates a flag file to indicate completion.
- **preprocess**: Preprocessing depends on the successful allocation of storage. It processes data and creates a flag file to indicate the task's completion.
- **train**: Training begins after preprocessing is complete, ensuring proper sequencing.
- **inference**: Inference is the final task in the data processing pipeline, where predictions are made. The flag 'inference_complete.txt' indicates completion.
- **release_storage**: After inference is complete, the storage is released using the `ws_release` command, and a flag file ('storage_released.txt') is created to mark the end of the workflow.

7 Evaluation

This section evaluates the three proposed designs for integrating Campaign Storage into HPC workflows, based on the key requirements discussed in Section 3.4. The evaluation is primarily qualitative, focusing on usability, performance, resource efficiency, and other qualitative KPIs.

7.1 Manual Use of the Campaign Storage API

The first design relies on direct interaction with the Campaign Storage API using CLI tools. Users manually allocate storage before each task and release it after completion, executing each job individually. While this approach provides full transparency and control over storage lifecycle and workflow execution, it introduces significant operational complexity.

From a usability perspective, this design is impractical and error-prone. Users must track job dependencies, ensure correct timing between storage allocation and job execution, and manually manage data movement. This increases the risk of human error and discourages adoption among less experienced users. Performance is also negatively impacted, as manual coordination leads to idle time between tasks, delaying the overall workflow. The lack of automation makes this approach particularly inefficient for larger workflows with many interdependent steps.

Resource allocation under this design is fragile: users might allocate storage for longer than necessary or forget to release it, resulting in wasted capacity. While the method is highly portable—since it doesn't rely on any specific workflow engine—it lacks structure, making reproducibility and user isolation difficult to guarantee. In collaborative environments, unstructured manual execution can lead to inconsistent results and unintended side effects, such as conflicting storage usage or path mismanagement.

7.2 Job Submission with SLURM Dependencies

The second design improves on the manual approach by using SLURM's job scheduler and dependency management features. Tasks are defined in separate job scripts or submitted inline using the `--dependency` flag to ensure that each job runs only after the successful completion of the previous one. Storage allocation and release are embedded within the job sequence, reducing the need for manual coordination.

This design significantly enhances usability and performance compared to the manual method. Users no longer need to track job completion or manually trigger the next step. By chaining jobs through SLURM dependencies, the workflow becomes more structured and less error-prone. However, it still requires explicit management of job IDs and dependency chains, which can be unintuitive for complex workflows or for users unfamiliar with SLURM scripting.

In terms of resource efficiency, this approach ties storage usage more closely to execution, reducing the likelihood of long, unnecessary allocations. Nevertheless, if a job fails and no failure-handling logic is included, storage may remain allocated indefinitely. Portability is good in any SLURM-based environment, but adapting the setup to non-SLURM clusters may require significant changes. Reproducibility improves thanks to the scripting structure, but the workflow remains somewhat fragmented, especially if job logic is split across multiple files.

7.3 Snakemake Workflow Integration

The third design leverages Snakemake to define a rule-based, automated workflow. Snakemake manages task dependencies internally and supports integration of external commands, including Campaign Storage commands like `ws_allocate` and `ws_release`. Each rule specifies its inputs and outputs, allowing Snakemake to determine execution order automatically.

This design provides the highest usability of the three. Users define rules once, and Snakemake coordinates their execution, managing dependencies, file paths, and partial re-execution if needed. The rule-based structure also makes the workflow easier to read, modify, and debug. Performance is optimized as jobs execute back-to-back without idle periods, and Snakemake can also parallelize independent steps where possible.

Resource efficiency is further improved through automation: storage allocation and release can be embedded directly in the relevant rules, ensuring minimal overhead and reducing the risk of forgotten releases. Portability is excellent, as Snakemake workflows can be moved between environments with only minor path or config changes. Reproducibility is also a key strength, as the workflow definition fully captures the task sequence, expected outputs, and dependencies.

Overall, the Snakemake design introduces structure, automation, and robustness that are especially valuable for complex or collaborative workflows. It balances flexibility and control, making it the most effective integration strategy among the three explored designs.

8 Conclusion and Future Work

This report explored and evaluated different strategies for integrating Campaign Storage into HPC workflows. Starting with a manual approach and progressively introducing automation, three designs were implemented and evaluated.

The evaluation revealed that while the manual approach offers full control, it is error-prone and inefficient for larger workflows. The SLURM-based design improved automation but still required explicit handling of dependencies and lacked flexibility. The Snakemake-based solution emerged as the most effective approach, offering a clean rule-based structure, automatic dependency handling, and high reproducibility. Its alignment with modern workflow management practices and minimal user interaction make it particularly well-suited for Campaign Storage integration in HPC environments.

One logical next step is to provide users with an interface that no longer relies on direct SSH access to interact with the system. Instead, workflows could be submitted and managed through a REST API. This would lower the entry barrier for users with less technical experience, improve accessibility across different platforms, and enable better integration with external tools or services. It would also support more flexible usage patterns, such as submitting jobs remotely or from web-based portals.

Overall, this report has shown how Campaign Storage can be effectively integrated into HPC workflows by comparing different architectural approaches. Through implementation and evaluation, it became evident that combining structured workflow systems like Snakemake with Campaign Storage can significantly reduce user overhead while improving reproducibility and efficiency.

References

[1] Jack Dongarra and David Keyes. The co-evolution of computational physics and high-performance computing. *Nature Reviews Physics*, 2024.

[2] Domenico Elia, Salvatore Fiore, and Giovanni Aloisio. Towards hpc and big data analytics convergence: Design and experimental evaluation of a hpda framework for e-science at scale. *IEEE Access*, 2021.

[3] DA Bader, BME Moret, and L Vawter. Industrial applications of high-performance computing for phylogeny reconstruction. *Commercial Applications for High-Performance Computing*, 2001.

[4] GK Lockwood, D Hazen, Q Koziol, RS Canon, K Antypas, J Balewski, N Balthaser, and W Bhimji. Storage 2020: A vision for the future of hpc storage, 2017.

[5] L Thomas, S Gougeaud, S Rubini, and P Deniel. Predicting file lifetimes for data placement in multi-tiered storage systems for hpc. In *Proceedings of the Conference on High-Performance Storage and Computing*. ACM, 2021.

[6] M. Hoseinzadeh. A survey on tiering and caching in high-performance storage systems. *arXiv preprint arXiv:1904.11560*, 2019.

[7] Peng Cheng, Yutong Lu, Yunfei Du, and Zhiguang Chen. Accelerating scientific workflows with tiered data management system. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 75–82, 2018. doi: 10.1109/HPCC/SmartCity/DSS.2018.00042.

[8] F. Zakaria, T. R. W. Scogland, T. Gamblin, and C. Maltzahn. Mapping out the hpc dependency chaos. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022.

[9] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan. Data jockey: Automatic data management for hpc multi-tiered storage systems. In *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2019.

[10] Holger Berger. hpc-workspace. GitHub repository, 2023. URL https://github.com/holgerBerger/hpc-workspace.

[11] Snakemake documentation (version 8.20.6), 2024.

[12] USC University of Southern California, 2019.

[13] Taylor Reiter, Phillip T Brooks, Luiz Irber, Shannon E K Joslin, Charles M Reid, Camille Scott, C Titus Brown, and N Tessa Pierce-Ward. Streamlining data-intensive biology with workflow systems. *GigaScience*, 10(1):giaa140, 2021. doi: 10.1093/gigascience/giaa140. URL https://doi.org/10.1093/gigascience/giaa140.

[14] Y.C. Yu. Container migration for mobile services, 2022.

[15] N. Padhy. An automation api to optimize the rate of transmission using rclone from local system to cloud storage environment. *Materials Today: Proceedings*, 2021.