GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

DevOps-Integrated HPC Pipeline with Apptainer, GitLab CI/CD, and SLURM

Mohamed Basuony

MatrNr: 12647187

Supervisor: Chirag Mandal

Georg-August-Universität Göttingen
Institute of Computer Science

September 28, 2025

Abstract

High-performance computing (High-Performance Computing (HPC)) workflows have traditionally relied on manual job management and custom environments, which often leads to errors and impedes reproducibility. This report presents a DevOps-integrated HPC pipeline that treats scientific workflows like production code by incorporating containerization, continuous integration/continuous delivery (Continuous Integration/Continuous Delivery (CI/CD)), and automated job scheduling. We designed a matrix multiplication workload pipeline using Apptainer containers for a portable, reproducible environment and GitLab CI/CD for automated testing and deployment to an HPC cluster managed by the SLURM scheduler. Three matrix multiplication workloads (single-core, multi-threaded, and Message Passing Interface (MPI)-distributed) were implemented to validate the pipeline across scaling levels. Our results demonstrate that automated unit tests (via PyTest) reliably verify correctness on every code commit, and containerized jobs achieve consistent performance from one node up to multiple nodes. The pipeline highlights both the benefits and limitations of applying DevOps practices in HPC: while continuous integration improved stability and reproducibility, certain deployment steps could not be fully automated due to HPC security constraints. We discuss how emerging HPC-DevOps approaches and custom CI runners can overcome these barriers. We also examine the growing impact of Artificial Intelligence (AI) workloads on HPC pipelines, noting that the DevOps methods used here can help manage the complexity and scale of AI training jobs. Overall, our DevOps-enabled HPC workflow markedly improves automation, reproducibility, and collaboration in scientific computing, indicating a promising direction for future HPC systems engineering.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ✓ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
CI/CD	Continuous Integration/Continuous Delivery
MPI	Message Passing Interface
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
SLURM	Simple Linux Utility for Resource Management
GPU	Graphics Processing Unit
CPU	Central Processing Unit
SSH	Secure Shell
YAML	YAML Ain't Markup Language
API	Application Programming Interface

1 Introduction

High-performance computing environments are critical for scientific and engineering workloads, yet many workflows remain manual, error-prone, and difficult to reproduce[1, 2]. Computations are typically executed via custom scripts on shared clusters with ad-hoc configuration of software and parameters. This lack of standardization impedes reproducibility and complicates replication or reuse of software by others. In the wider software industry, DevOps practices—combining software development and IT operations—have improved the reliability of deployments through automation, continuous testing, and version control[3]. The adoption of DevOps principles in HPC is therefore expected to yield similar benefits of automation, consistency, and collaboration for scientific workflows[4, 5]. For example, automatic building and testing of code changes can detect issues early and ensure that computational experiments are conducted in known, versioned environments, thereby enhancing scientific rigor[6, 2].

Integration of DevOps tools into HPC remains non-trivial. HPC systems differ substantially from the cloud environments where DevOps originated: clusters rely on batch schedulers rather than persistent services, and security policies frequently prohibit typical cloud tools such as Docker daemons. Consequently, key DevOps components such as CI/CD pipelines have achieved only limited adoption in scientific computing to date[6, 2]. Furthermore, DevOps expertise may be lacking in research settings, where emphasis is often placed on proving scientific concepts rather than packaging and deploying software for reuse[2]. Despite these constraints, increasing recognition has emerged that improvements in automation and reproducibility are vital for HPC. The objective of this work is to treat an HPC workflow "like production code"—that is, to make it testable, portable, and version-controlled[7]. An HPC pipeline integrating DevOps techniques (containerization, continuous integration, and automated deployment) is demonstrated to streamline the workflow and reduce human error.

Concretely, an HPC pipeline was designed for a set of matrix multiplication workloads, using Apptainer containers to encapsulate the software environment and ensure cross-platform reproducibility. Apptainer (formerly Singularity) is a container platform tailored for HPC that allows containerized applications to run without privileged access, thereby providing a secure, portable environment on shared clusters[1, 8]. Containers have emerged as an effective mechanism to improve portability and reproducibility in scientific computing[1, 8]. Git repository version control and GitLab CI/CD are leveraged to automatically build and test code on each commit. A GitLab continuous integration pipeline executes unit tests inside a container to verify correctness prior to cluster deployment. Finally, the pipeline employs the SLURM workload manager to schedule and run containerized jobs on HPC nodes. In combination, these elements bring the benefits of DevOps—automation, consistent environments, and continuous testing—to the HPC domain.

The remainder of this report is organized as follows. In *Methods and Pipeline*, the architecture of the DevOps-integrated pipeline is described, including the repository structure, containerization process, CI/CD configuration, and job scheduling with SLURM. *Experimental Workloads* details the three matrix multiplication programs used to exercise the pipeline and the execution procedure. *Results* presents key outcomes, including workload performance and CI/CD behavior. In *Discussion*, the findings are interpreted, highlighting effective elements (e.g., automated testing, scalable containers) and encountered challenges

(e.g., deployment constraints), followed by related work. A dedicated section on the *Impact of AI Workloads on HPC* examines how the rise of machine learning tasks influences HPC pipeline design and how DevOps practices can manage AI resource demands. *Threats to Validity* addresses limitations of scope and assumptions. *Conclusion* summarizes key takeaways and outlines future directions for DevOps in HPC.

2 Methods and Pipeline

Pipeline Design: The DevOps-integrated pipeline was designed to advance code from development through automated testing to execution on an HPC cluster. The workflow consists of the following stages: code is pushed to a Git repository (GitLab); continuous integration triggers automated tests in a container; upon success, a container image is built for the application; and the application is then executed on the HPC cluster via SLURM job scheduling[7]. This can be summarized as Dev → GitLab CI → Container Build → SLURM Job[7]. The rationale is to enforce checks at each step: any code change is immediately validated by tests, the environment remains consistent through containerization, and job submission is automated rather than manual. By structuring the pipeline in this way, the HPC workflow is treated analogously to a production software deployment pipeline, aiming for repeatability and minimal manual intervention[7].

Repository Structure: The repository was organized to support this pipeline, with concerns separated into dedicated directories for clarity. Figure 1 illustrates the repository structure, which includes a `container/` directory containing the Apptainer definition file, a `matrix/` directory with Python source code for workloads, a `tests/` directory with unit test scripts, and `job_scripts/` with SLURM submission scripts for each workload. This modular layout aligns with DevOps principles by clearly delineating infrastructure (container and job scripts), application code, and tests.

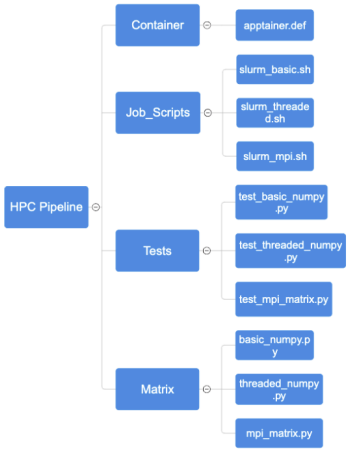

Figure 1: Repository structure of the project.

Containerization with Apptainer: Apptainer was used to containerize the HPC workloads for reproducibility and ease of deployment. Apptainer is specifically designed for HPC and operates without elevated privileges, in contrast to Docker, which is generally not allowed on multi-user clusters[2, 1]. The container environment is defined in an Apptainer

recipe file (`apptainer.def`), specifying the OS base image and all dependencies. In the present case, the definition installs Python 3, NumPy, mpi4py (for MPI support), OpenMPI, and PyTest, and copies the project's code and tests into the container image[7, 7]. This ensures that the identical software stack is available wherever the container runs—on a local machine or on any node of the cluster—achieving portability and consistency for the workloads[1].

Container images were built on the HPC cluster's login node (Apptainer converts the definition into a binary image file). The build command used was `apptainer build matrix-demo.sif container/apptainer.def`, producing a portable image file (`matrix-demo.sif`)[7]. This step was initially performed manually due to restrictions on where Apptainer could run (cluster policy required image builds to occur on designated nodes). The resulting container image encapsulates the environment needed for all workloads and tests. Workloads are executed inside the container using `apptainer exec matrix-demo.sif python3 matrix/`*`program`*`.py`. By using a single container image for both local testing and cluster execution, the "works on my machine" discrepancy is mitigated; code that passes tests in the container on GitLab should behave equivalently on the HPC cluster[1, 8]. Containerization therefore addresses a key reproducibility challenge in HPC by standardizing the runtime environment.

Continuous Integration (CI) with GitLab: All code changes trigger an automated CI pipeline on GitLab. GitLab CI was configured with a YAML pipeline definition specifying three stages: *test*, *build*, and *deploy*[9]. In the *test* stage, a Docker-based executor runs the PyTest suite inside a lightweight container (distinct from Apptainer; a standard Python Docker image is used for speed). Tests on GitLab's CI runner ensure that code is sanity-checked prior to any interaction with the HPC cluster. Each commit or merge request initiates these tests. This approach follows best practices in scientific software development, where continuous tests enhance reproducibility[5, 3]. The unit tests cover each workload script: verifying correct shapes and values for the single-core case, checking that multi-threaded results are correct and utilize the expected number of threads, and confirming that MPI-distributed computation yields consistent results across processes[7, 7]. The CI pipeline thus functions as an automated gatekeeper, permitting only code that passes all tests to proceed toward deployment.

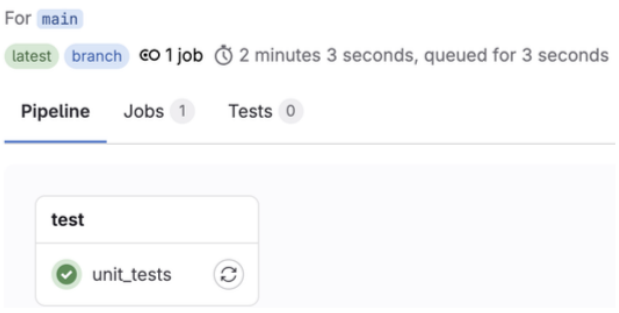

Figure 2: GitLab CI/CD test stage passing.

Once tests pass, the *build* stage can construct the container image. Owing to cluster restrictions (discussed later), the container build was not fully automated in CI. Instead, CI triggers a placeholder or manual step for image construction. Ideally, this stage would employ Apptainer on the cluster or a CI runner with Apptainer support. In practice, the

build was performed on the cluster manually because the GitLab runner lacked direct access to the cluster's Apptainer environment. The *deploy* stage was similarly set up as a manual trigger: it was intended to transfer the built image and job scripts to the cluster and execute SLURM jobs. However, direct deployment from GitLab to the HPC system encountered obstacles (SSH key and environment issues; see Discussion), so final deployment was executed via a manual process outside the CI pipeline[9]. Despite this, the CI pipeline provided significant benefits in earlier phases: immediate test feedback and assurance of a working container recipe.

SLURM Job Scripts and Scheduling: Three SLURM batch scripts (single-core, multi-thread, and MPI) were created to automate job submission on the HPC cluster[10, 7]. These scripts encapsulate resource requirements and execution commands for each run type. For example, `slurm_basic.sh` requests 1 CPU and runs the single-core Python program inside the Apptainer container. `slurm_threaded.sh` requests multiple CPUs on one node (e.g., 8 cores) and runs the threaded program with `OMP_NUM_THREADS=8` inside the container. `slurm_mpi.sh` requests multiple nodes (e.g., 2 nodes with 4 tasks total) and uses `srun` to launch the MPI program inside the container across allocated nodes. Each script loads any required environment modules (e.g., the Apptainer module) and then executes `srun apptainer exec matrix-demo.sif python3 matrix/`*`program`*`.py`[7]. Using `srun` ensures that MPI ranks are launched correctly across nodes and that tasks are bound to the allocated CPUs. The SLURM scheduler handles queueing, resource allocation, and enforcement of requested resources for each job.

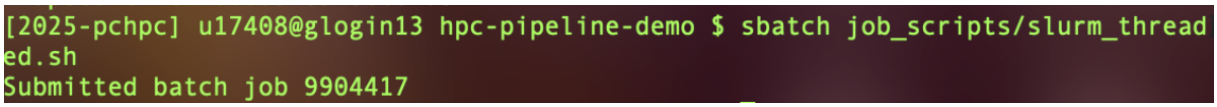

Figure 3: SLURM submission returning a job ID.

To run the workloads on HPC, batch scripts are submitted using `sbatch`. This was integrated into the workflow such that deployment would issue `sbatch job_scripts/slurm_basic.sh`, etc., thereby automating what was previously a manual step. Because automated deploy from GitLab was problematic, submissions were executed manually as the final workflow step. The overall process remained straightforward: pushing code to GitLab triggers tests; if tests pass and a container is built, `sbatch` commands execute the jobs on HPC. SLURM logging captures job output for later inspection.

Continuous Delivery and Deployment Considerations: The final stage of a full DevOps pipeline is automated deployment. The intended design was for the GitLab CI pipeline to deploy the container and run the SLURM jobs after building. An SSH-based approach from the GitLab runner to the HPC cluster was attempted so that the CI job could transfer files and invoke `sbatch` remotely. In practice, several issues arose: the HPC cluster required an SSH key format incompatible with GitLab's default, and GitLab's handling of multiline secure variables complicated private key injection[9]. Additionally, the non-interactive login environment did not load required modules (e.g., Apptainer), causing remote `sbatch` attempts to fail due to missing `apptainer`. These hurdles prevented an automated deploy stage. As a result, a workaround was implemented: following CI testing and image build, deployment steps were executed manually—transferring code or container to the cluster, building on the cluster if necessary, and submitting jobs. While

not fully automated, this compromise reflects a common situation in current HPC DevOps integration, where institutional security policies or infrastructure limitations constrain automation[7, 10]. Despite partial automation, the pipeline streamlined substantial portions of the HPC workflow. All code changes passed through an automated test suite, and the container ensured that any cluster run occurred in a vetted environment. The remaining manual tasks—file transfer and job triggering—could be improved with administrative support.

3 Experimental Workloads

To evaluate the pipeline, three matrix multiplication workloads were developed in Python, representing increasing levels of parallelism and complexity. Each script multiplies two large matrices and verifies correctness, differing in resource utilization (single core, multi-threading, or distributed MPI):

`basic_numpy.py`: A single-core workload that multiplies two 1000×1000 matrices using NumPy's `dot` function. Only one CPU core is used, serving as a quick check with minimal computational load[7]. This program is suitable for validating that the container environment and pipeline function end-to-end.

`threaded_numpy.py`: A multi-threaded workload that multiplies two 4000×4000 matrices using NumPy with OpenBLAS multi-threading. `OMP_NUM_THREADS=8` is set to utilize 8 threads on a single node[7]. This workload is CPU-intensive and tests multi-core execution and parallel speedup on one machine, while also validating threading behavior within the container and cluster CPU affinity.

`mpi_matrix.py`: A distributed-memory workload that multiplies two 4000×4000 matrices using MPI (via `mpi4py`). The matrix is partitioned by rows, each MPI process computes a subset of the result, and the partial results are gathered[7]. The program was executed on multiple nodes (2 nodes with 2 MPI ranks each) to validate support for multi-node, multi-process jobs under SLURM and to test MPI operation within the container.

These workloads were selected to verify pipeline correctness and scalability across different levels of parallelism[7]. The single-core job checks base functionality; the threaded job examines multi-core performance on one node; and the MPI job evaluates multi-node distributed execution. All three yield the same logical result ($C = A \times B$) for known inputs, enabling cross-validation. Execution time was logged, and assertions confirmed expected results within floating-point tolerances.

PyTest unit tests were written to ensure correctness. For `basic_numpy.py`, tests assert the expected output shape (1000×1000) and verify sample entries against a reference multiplication. The threaded and MPI versions are checked for consistency: identical inputs to all three programs yield identical outputs. The MPI test launches `mpi_matrix.py` under an MPI executor (using `mpi4py.run`) on a single machine with a few processes, confirming that the assembled result matches a single-process computation. These tests were integrated into CI (executed in a single-node context for simplicity), providing confidence that the MPI job would behave correctly across nodes in the cluster.

Execution on HPC proceeded via the SLURM scripts. Jobs for all three workloads were submitted to observe behavior and performance on actual cluster hardware. The basic and

threaded jobs were allocated to a single compute node (1 core and 8 cores, respectively); the MPI job was allocated to 2 nodes with 2 cores each. Submissions were managed concurrently by SLURM. The container image `matrix-demo.sif` resided on a shared filesystem, enabling identical runtime environments for all jobs. Batch scripts captured console output (including timing) via SLURM logging.

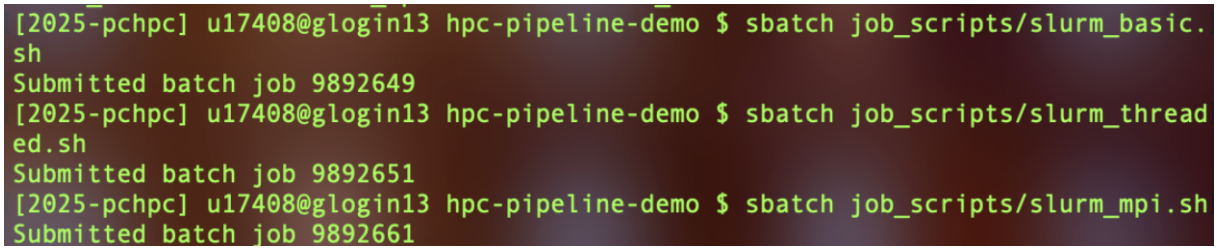

Figure 4: Batch submission of the three jobs.

4 Results

All stages of the DevOps-integrated pipeline were executed, and the outcomes validate the approach. The continuous integration tests on GitLab ensured that each workload produced correct results in the containerized environment. When code commits introduced errors (e.g., bugs in matrix multiplication logic or incorrect environment variables), the CI pipeline detected them through failing unit tests. Over the development cycle, most commits were tested automatically, which increased confidence in code stability. These observations are consistent with reports that CI and containers enhance reproducibility and reliability of scientific software[5, 10].

On the HPC cluster, all three workloads ran successfully using the Apptainer container and SLURM job scripts. Table 1 summarizes the execution times for each workload (each run was performed once on the allocated resources):

`basic_numpy.py`: ∼0.05 seconds (1 core)
`threaded_numpy.py`: ∼1.26 seconds (8 threads)
`mpi_matrix.py`: ∼0.29 seconds (4 MPI processes across 2 nodes)

These timings were obtained from program output logs. The single-core job executes in tens of milliseconds for a 1000×1000 multiply, as expected. The multi-threaded job (4000×4000 on 8 threads) takes ∼1.3 seconds, reflecting the $16\times$ increase in elements and thread coordination overhead. The MPI-distributed job (4000×4000 split among 4 processes) completes in ∼0.29 seconds, faster than the threaded run. This likely results from dividing the workload across nodes and achieving low communication overhead for this problem size, indicating effective scaling on 2 nodes. The outcome further suggests that MPI within Apptainer imposes no noticeable performance penalty. Prior studies similarly report minimal container overhead for HPC workloads, preserving near-native performance[1, 8].

Table 1: Execution times on the cluster using the shared Apptainer image.

Workload	Resources	Execution Time (s)
`basic_numpy.py`	1 core	0.05
`threaded_numpy.py`	8 threads	1.26
`mpi_matrix.py`	4 MPI processes (2 nodes)	0.29

Figure 5 shows an excerpt from the `basic_numpy.py` log, indicating container invocation (`apptainer exec ... python3 matrix/basic_numpy.py`) and an elapsed time of approximately 0.05 seconds; analogous logs for the threaded and MPI runs report ∼1.26 s and ∼0.29 s, respectively.

```
[2025-pchpc] u17408@glogin13 hpc-pipeline-demo $ apptainer exec matrix-demo.sif
python3 matrix/basic_numpy.py
Running single-threaded NumPy multiplication for 1000×1000 matrices...
Elapsed time: 0.053283 seconds
```

Figure 5: Output snippet from the basic run.

From a DevOps perspective, automated testing and containerization were effective in detecting issues early and ensuring consistency. A deliberate discrepancy (e.g., scaling the output matrix) was introduced during development to validate test sensitivity; the CI pipeline correctly failed that commit[5, 3]. Furthermore, the same Apptainer image functioned across systems, confirming environment portability without re-customizing packages for the cluster. The container image, built on a compatible OS base, executed cleanly on compute nodes.

The deploy stage of GitLab CI did not execute automated cluster runs, due to authentication and environment constraints. Manual deployment was therefore employed[9]. This limitation underscores that certain DevOps automation steps are not yet turnkey in traditional HPC environments[7, 10]. Nonetheless, once jobs were submitted, the assurances provided by CI (code correctness) and containerization (environment consistency) held. Overall, the pipeline maintained software correctness and environment consistency and enabled scaling from one core to multiple nodes without code modification.

5 Discussion

Implementation of a DevOps-integrated HPC pipeline provided several insights into the benefits and challenges of applying DevOps practices in HPC. The primary objective—improved automation and reproducibility—was achieved. Automatic testing of every code update in a controlled environment reduced the likelihood of runtime errors on the HPC cluster caused by untested changes, a critical advantage in scientific computing where undetected bugs or environment inconsistencies can lead to invalid results or wasted computation[5]. Continuous integration thus served as a safety net for the HPC workflow, reflecting the concept of "scientific CI," in which CI and containers continually verify scientific results[5, 3].

Use of Apptainer containers ensured consistency and portability. The application environment was built once and executed across systems, eliminating a common class of HPC failures arising from library/version mismatches. This observation aligns with broader trends in HPC, where container adoption is increasing to manage complex software stacks across heterogeneous systems[1, 8]. Apptainer also enabled the use of modern user-space stacks (e.g., recent Python/NumPy) on systems with older modules, providing agility without administrative intervention[1].

Practical challenges remain that help explain why DevOps is not yet ubiquitous in HPC. The foremost issue involves automated deployment to the cluster. Many HPC centers are closed environments with stringent security; external GitLab runners must SSH into clusters, encountering key-format incompatibilities and non-interactive shells that do not load required modules[9]. Such constraints hinder direct CI integration at numerous sites. A viable mitigation is to provision a GitLab runner on the cluster with appropriate permissions, enabling native job submission and fully automated deployment.

Another challenge stems from batch scheduling. Unlike enterprise deployments, cluster jobs may queue for minutes or hours, making synchronous CI/CD fragile due to potential timeouts. More robust designs may require asynchronous notifications or decoupled stages (trigger, then separate result verification) rather than end-to-end synchronous pipelines.

Cultural and skills-related factors also play a role. Many scientific teams lack deep familiarity with DevOps tooling. Survey evidence indicates that limited DevOps expertise constrains container and CI adoption among researchers[2]. Addressing this gap will require training and simplified tooling. While the present pipeline targeted a relatively modest application (matrix multiplication), the core principles generalize to more complex scientific codes, including those with GPUs and substantial I/O demands[11, 12].

The empirical comparison between threading (8 threads) and MPI (4 processes over 2 nodes) revealed different performance characteristics (1.26 s vs. 0.29 s) for the 4000×4000 case. Such differences are expected and highlight an additional benefit of the pipeline: multiple execution modalities can be evaluated with minimal friction, since they are driven by the same repository and container. This facilitates fair comparisons and repeatable performance studies.

Key lessons, consistent with the presentation's takeaways, can be summarized as:

- CI/CD workflows improve stability and reproducibility in HPC.
- Testing and containerization reduce human error and setup time.
- Manual fallbacks remain essential in restricted environments.
- Full automation depends on system-level support.
- Infrastructure constraints are as consequential as tooling choices.

6 Impact of AI Workloads on HPC

The growing prevalence of artificial intelligence (AI) and machine learning workloads is reshaping HPC infrastructure and workflows. Modern AI training frequently requires

HPC-class resources: multiple GPUs, large memory, high-speed interconnects, and parallel I/O. Organizations are scaling clusters specifically for AI, leading to a convergence of HPC and AI infrastructure[11, 12]. The demonstrated pipeline and practices translate directly to AI contexts: containerized ML frameworks and CI-driven checks can validate small samples in CI, while full training runs are executed on the cluster, preserving reproducibility and provenance. Apptainer supports GPUs via `-nv`[13], and SLURM job scripts can request GPUs analogously to CPUs[10].

AI workloads introduce iterative, data-intensive patterns that complicate CI. End-to-end retraining on every commit is infeasible for large models; instead, CI can validate reduced datasets or short training epochs, with scheduled or event-driven full runs. Data versioning and artifact tracking (MLOps) complement DevOps in this setting, reinforcing traceability and repeatability. HPC centers are adapting accordingly by deploying more GPUs and high-bandwidth storage, while schedulers evolve to manage heterogeneous resources and interactive workflows (e.g., Jupyter on HPC). Distributed training frameworks (e.g., Horovod, PyTorch Distributed) can be launched via `srun` or `sbatch` within containers[13, 10].

A shift toward throughput and workflow automation is also observable: AI research often entails many experiments (e.g., hyperparameter sweeps). CI/CD orchestration can queue and monitor such experiments efficiently, improving cluster utilization and experimental rigor.

7 Threats to Validity

Several limitations and potential threats to validity should be noted:

Generality of Workloads: Experiments employed relatively small matrix multiplication programs by HPC standards. The largest problem (4000×4000) and the modest node count (2) are not representative of large-scale workloads. Container overhead and CI costs were not stress-tested under extended runtimes or massive I/O.

Cluster-Specific Behavior: The pipeline reflects one cluster's configuration (scheduler policies, module environment). Issues encountered (e.g., SSH key handling, non-interactive module loading) may vary elsewhere. Replication on other systems may require environment-specific adjustments.

Measurement of "Success": While qualitative improvements in workflow were observed, formal metrics (e.g., developer time saved, defects caught pre-deployment) were not collected. Stronger quantitative evidence would further substantiate the benefits.

Incomplete Automation: End-to-end automation was not achieved due to deploy-stage constraints. Some error classes might only manifest at execution time on the cluster, outside CI's purview. Automated post-submission verification and asynchronous result collection remain open engineering tasks.

Reproducibility of the Pipeline Itself: Although the approach is described conceptually, full configuration artifacts (CI YAML, Apptainer recipe, etc.) are not reproduced here. Small differences in runner setup or permissions can affect replicability.

Performance Overheads Not Profiled: Container startup overhead, image size effects, and network staging costs were not systematically profiled. Results should be interpreted in the context of a relatively small image on a fast shared filesystem.

Limited Scope of Testing: Fault tolerance, concurrent multi-developer scenarios, and container versioning strategies were not evaluated. Production usage would require robust image tagging and isolation to avoid conflicts.

8 Conclusion

A DevOps-integrated HPC pipeline combining Apptainer, GitLab CI, and SLURM was designed and implemented. Using a matrix multiplication case study, HPC workflows were treated as production-quality pipelines, demonstrating feasibility and benefits. Notable contributions include: (1) **Improved reproducibility and reliability:** continuous integration testing and containerized environments ensured that computations were performed in controlled, consistent settings across systems, increasing trust in results and reducing undetected errors; (2) **Workflow automation:** tasks traditionally performed manually in HPC (installation, submission, verification) were partially or fully automated, accelerating research cycles and reducing human error; (3) **Integration challenges identified:** gaps between DevOps tooling and HPC policies were documented, informing both practitioners and facility administrators about required support (e.g., on-cluster CI runners, secure automation pathways).

In practical terms, a set of HPC jobs—including multi-node MPI runs—was executed through a largely hands-off process once configured, representing a marked improvement over manual practices of module management and ad-hoc testing. These findings align with literature on research software engineering and reproducible computing, and are consistent with evidence that containers preserve near-native performance for compute-bound workloads while CI increases the repeatability of scientific results[1, 8, 5].

Future work includes integrating GPU-accelerated and AI workloads (with staged CI strategies), enabling fully automated deployment via on-cluster runners or HPC-aware CI frameworks, and scaling the approach for collaborative teams using container registries and infrastructure-as-code. Overall, the presented pipeline illustrates a viable path to modernizing scientific workflows: as HPC workloads diversify and grow—particularly with AI—integrated DevOps practices will be increasingly essential for managing complexity and ensuring trustworthy computational results.

References

[1] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLoS ONE* 12.5 (2017), e0177459.

[2] HPC Containers Working Group. *The HPC Container Community Survey 2024*. 2024. URL: https://supercontainers.github.io/hpc-containers-survey/2024/two-thousand-twenty-four/.

[3] Christof Ebert et al. "DevOps". In: *IEEE Software* 33.3 (2016), pp. 94–100.

[4] Ioannis K Moutsatsos et al. "Jenkins-CI as a scientific data and image-processing platform". In: *SLAS Discovery* 22.3 (2017), pp. 238–249.

[5] Matthew S Krafczyk et al. "Scientific tests and continuous integration strategies to enhance reproducibility in scientific software". In: *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems (P-RECS'19)*. 2019, pp. 23–28.

[6] Paul Nuyujukian. "Leveraging DevOps for Scientific Computing". In: *arXiv preprint arXiv:2310.08247* (2023).

[7] Valerie Hayot-Sasson et al. "Addressing reproducibility challenges in HPC with continuous integration". In: *arXiv preprint arXiv:2508.21289* (2025).

[8] Jack S Hale et al. "Containers for portable, productive, and performant scientific computing". In: *Computing in Science & Engineering* 19.6 (2017), pp. 40–50.

[9] GitLab Inc. *GitLab CI/CD Pipelines Documentation*. 2024. URL: https://docs.gitlab.com/ee/ci/.

[10] SchedMD LLC. *SLURM Workload Manager Documentation*. 2024. URL: https://slurm.schedmd.com/documentation.html.

[11] DDN (DataDirect Networks). *Why HPC Is Your Path to AI*. Whitepaper. 2023. URL: https://www.ddn.com/resources/whitepapers/why-hpc-is-your-path-to-ai/.

[12] Intel Corporation. *Scale AI Workloads within an HPC Environment*. 2023. URL: https://www.intel.com/content/www/us/en/high-performance-computing/hpc-artificial-intelligence.html.

[13] Apptainer Community. *Apptainer User Guide*. 2024. URL: https://apptainer.org/docs/.

A Code samples

https://gitlab.gwdg.de/basuony/hpc-pipeline