SLoFI: Low-Privilege Fault Injection for Reliability Testing in Production Supercomputers

Yongqian Sun†, Shaoyu Hu†, Lei Tao†, Xijie Pan†, Yuan Yuan‡, Wenwei Gu†*
Shenglin Zhang†, Yuqi Li‡§, Jian Zhang§
†*Nankai University* ‡*National University of Defense Technology* §*National Supercomputer Center in Tianjin*

***Abstract*—Production supercomputers often expose software-reliability risks as silent application slowdowns rather than explicit crashes, while active testing is constrained by user privileges and co-running jobs. We present *SLoFI*, a low-privilege fault-injection framework for Slurm-managed production supercomputers. *SLoFI* emulates application-visible degradation symptoms from user space, confines perturbations to the current job allocation, and aligns injection events, node telemetry, and workload iterations on a common wall-clock timeline. On the Tianhe production supercomputer, *SLoFI* executed over 1,800 workload iterations across nine representative workloads, providing broad workload coverage to validate that the injected perturbations are realistic, observable, and effective under production constraints without unintended node failures. In May 2026, *SLoFI* was deployed and validated by the Tianhe production team, covering all supported perturbation types, 568 nodes, and 30 injections for each perturbation type. It has also been integrated into a closed-loop reliability workflow for anomaly detection, root-cause localization, isolation, cleanup, and recovery validation.**

***Index Terms*—Supercomputer reliability, low-privilege fault injection, performance degradation, production deployment**

I. Introduction

Production supercomputers now host reliability-critical computational software, including simulation, data analytics, throughput-oriented batch workloads, and AI training. They aggregate massive numbers of compute nodes, high-speed interconnects, storage systems, schedulers, and monitoring facilities, while concurrently serving many users and applications. Reliability problems in such systems directly affect scientific productivity and service continuity. Prior studies have shown that high-performance computing systems experience diverse failures and performance anomalies at scale [1], [2], and recent production studies further highlight the importance of telemetry-driven health analysis and anomaly diagnosis [3]–[5].

In production operation, however, reliability loss does not always appear as explicit node crashes, kernel panics, or job failures. A more subtle and common form is *silent performance degradation*: CPU contention, memory pressure, operating-system noise, process stalls, or communication slowdown may significantly prolong jobs while leaving only weak evidence in logs or scheduler summaries [6]–[8]. This problem is particularly severe for tightly synchronized MPI and distributed-training workloads. A local slowdown on one victim node can propagate through barriers, collective communication, or straggler effects, turning a small disturbance into job-wide inefficiency [9]–[11]. For operators and reliability engineers, such degradation is difficult to reproduce, label, and diagnose from passive observation alone.

Controlled injection turns rare degradation incidents into repeatable software-reliability tests. By introducing perturbations with controlled type, timing, duration, and scope, fault injection can test whether monitoring, anomaly detection, diagnosis, isolation, and recovery mechanisms respond as expected [12]. It can also generate labeled evidence for evaluating AIOps components, which is important because production telemetry is usually weakly labeled or unlabeled [4], [5]. Existing fault-injection and chaos-engineering tools provide useful foundations, but many assume simulators, compiler instrumentation, administrator-controlled system mechanisms, or cloud orchestration planes [13]–[19]. As summarized in Table I, where "User (partial)" denotes user-level support for only a subset of functions, these assumptions do not match the operational interface available to low-privilege users on bare-metal production supercomputers.

TABLE I
Comparison with representative fault-injection approaches.

Tool	Domain & Scope	Privilege
HDFIT [13]	Hardware simulation	N/A
LLFI [14]	Compiler IR	User
ChaosBlade [17]	Host / application	User (partial)
Chaos Mesh [18]	Kubernetes pod / node	Admin
LitmusChaos [19]	Kubernetes pod / node	Admin
HPCArrow [16]	Petascale HPC network	Admin
FINJ [15]	HPC experiment orchestration	User (partial)
***SLoFI* (Ours)**	**Supercomputer job allocation**	**User**

The need for low privilege applies not only to application users, but also to operators and reliability engineers who want to run controlled perturbation tests without root-level authority. In practice, avoiding privileged operations reduces approval overhead, limits the blast radius of experiments, and helps prevent irreversible system changes. This mismatch leaves two production reliability-engineering challenges:

(1) Safe low-privilege perturbation is difficult. Production users cannot modify kernels, reboot nodes, install persistent daemons, or use privileged traffic-control mechanisms. Even operator-approved tests should remain job-scoped when possible. Thus, injection must run in user space, stay within the target Slurm allocation, support controllable intensity and

*Wenwei Gu is the corresponding author.

duration, and be cleaned up after execution [20].

(2) Verifiable degradation ground truth is difficult. The objective is not merely to create load, but to produce labels for application-visible slowdown. Yet silent degradation often lacks typical failure signals, while schedulers, monitoring systems, and applications record observations at different layers and granularities [20]–[22]. Across MPI, HPDA, HTC, and AI workloads, progress signals also differ, making unified degradation evidence hard to obtain.

This paper presents *SLoFI* (**S**upercomputer **Lo**w-Privilege **F**ault **I**njection), a production-safe reliability testing framework for shared Slurm-managed supercomputers. To address the first challenge, *SLoFI* uses a symptom-oriented perturbation model and treats the Slurm allocation as the safety boundary, emulating resource pressure, stalled progress, and communication interference through temporary user-space processes. To address the second challenge, *SLoFI* uses job-scoped orchestration and absolute-clock alignment to correlate injection events, node telemetry, and workload iterations, turning low-privilege perturbations into labeled cross-layer reliability evidence.

The main contributions are as follows:

- **A production-safe low-privilege fault-injection framework for supercomputers.** *SLoFI* enables controlled degradation experiments in shared production supercomputers without root privileges, kernel modification, persistent daemons, or intrusive system changes. We plan to release the fault-injection code[1] and scripts for selected workloads through a public code repository.
- **A job-centric workflow for verifiable cross-layer fault evidence.** *SLoFI* couples Slurm-scoped orchestration, user-space perturbation, telemetry collection, and absolute-clock alignment, enabling each injected event to be checked against both node-level symptoms and workload-visible slowdown across representative production workloads.
- **Industrial validation and deployment on Tianhe.** We validate *SLoFI* on the Tianhe production supercomputer using nine representative workloads spanning scientific simulation, HPDA, HTC, and CPU-based AI training, with over 1,800 workload iterations. The results show that the injected perturbations are realistic, observable, and effective under production constraints. In May 2026, *SLoFI* was deployed and validated by the Tianhe production team, covering all supported perturbation types, 568 nodes, and 30 injections for each perturbation type. It has also been integrated into a closed-loop reliability workflow covering fault injection, anomaly detection, root-cause localization, isolation, cleanup, and recovery validation.

II. *SLoFI* Design

A. Design Overview

Figure 1 shows the workflow of *SLoFI*. The design addresses the two challenges in Section I: safe low-privilege perturbation in a shared production system, and verifiable cross-layer degradation evidence for each injected event. A user submits a standard Slurm job [20] that contains the target workload, cyclic execution wrapper, telemetry configuration, and perturbation policy. Once the allocation is granted, *SLoFI* starts temporary controllers within the allocation. The workload wrapper records iteration boundaries, the controller triggers perturbations, and the telemetry pipeline queries production monitoring systems, such as Prometheus-style metric collection or HPC monitoring services [21], [22]. After the job completes, *SLoFI* correlates these streams into a job-centric evidence record.

The key design choice is to use the *Slurm job allocation* as the boundary for safety, orchestration, and measurement. Perturbation processes are launched only on allocated nodes and are terminated during cleanup. This matches the interface that ordinary users legitimately possess in production supercomputers: before allocation, the nodes are inaccessible; after allocation, users can run job-scoped programs but still cannot modify system-wide control paths. Designing around this boundary makes *SLoFI* acceptable to operators and repeatable for reliability engineers.

B. Perturbation Model

SLoFI targets silent performance degradation rather than catastrophic hardware failures. Its perturbation model is symptom-oriented: instead of reproducing every physical fault source, *SLoFI* creates observable degradation symptoms with controlled timing, intensity, and scope. All perturbations are executed as user-space programs within the current Slurm allocation and are removed during cleanup, making the injected events safe and auditable in a shared production environment.

Table II summarizes the supported perturbations. For each perturbation, we report the layer where the symptom is introduced, the main control parameters, and the resulting observable symptom. The concrete parameter ranges are configured according to operator-approved safety limits at the deployment site. The planned public artifact will include the corresponding fault-injection wrappers and selected workload scripts, while site-specific configurations such as node lists, telemetry endpoints, and approved parameter ranges remain local to the production environment.

TABLE II
User-space perturbations supported by *SLoFI*.

Perturbation	Layer	Main parameters	Symptom
CPU contention	Node	CPU load ratio, duration	compute slowdown
Memory pressure	Node	memory ratio, duration	reduced free memory
Process noise	Process	duration, freeze length	progress stalls / jitter
Network delay	MPI	fault window, delay bound	higher latency
Network loss	MPI	fault window, loss rate	loss-like disruption
Network noise	MPI	packet size, interval	traffic contention
Disk I/O / space fill	Node	fill ratio or write size, duration	I/O or space pressure

CPU and memory perturbations emulate local resource contention on selected allocated nodes. Process noise emulates transient progress stalls and scheduling jitter by perturbing target processes within the job. The three communication

[1]https://github.com/ustinian-yun/SLoFI

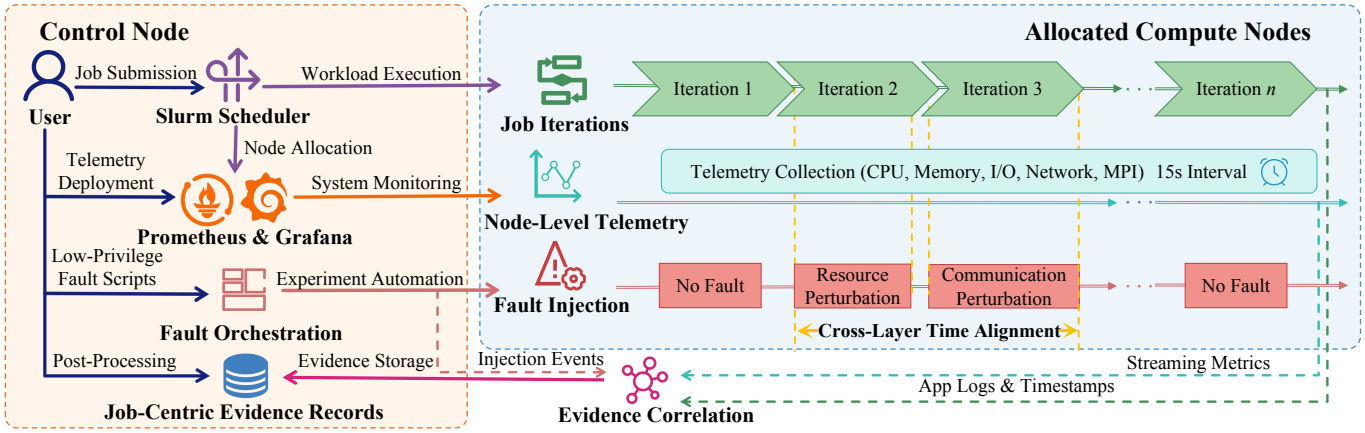

Fig. 1. Job-centric execution and cross-layer alignment workflow of *SLoFI*. The framework operates through a standard Slurm allocation and correlates injection events, telemetry samples, and workload iteration records after the job completes.

perturbations emulate delay-, loss-, and noise-like symptoms at the MPI/application layer, because ordinary users cannot directly manipulate production interconnect control paths. Disk perturbation is restricted to safe local paths to avoid affecting the shared parallel file system.

The main PDR evaluation in Section III focuses on the first six perturbation types because they consistently affect resource usage, process progress, or communication behavior across the selected workloads. Disk perturbation is excluded from the main PDR study because the evaluated workloads are dominated by CPU, memory, process-progress, and communication behavior; it is instead covered in the Tianhe deployment.

C. Job-Scoped Orchestration

SLoFI treats one scheduled Slurm job as the unit of experiment. Each configuration specifies the workload, iteration count, allocated nodes, victim node or process group, perturbation type, injection window, duration, and intensity. The orchestration script records the actual start and end time of each injection, target scope, perturbation parameters, and cleanup status. For local resource or process perturbations, *SLoFI* launches perturbation processes only on selected allocated nodes. For multi-node communication perturbations, *SLoFI* launches the MPI-level interference utility only across nodes granted to the current allocation.

To obtain comparable observations, workloads are executed cyclically within the same allocation. One iteration denotes one complete repeated execution of the target workload, rather than an application-internal epoch, timestep, or mini-batch. The workload wrapper continuously appends iteration records to a job-local log, including the iteration identifier, start time, end time, exit status, and wall-clock duration. For an injected run, the controller triggers the perturbation only after the start of the target iteration has been recorded, so the injected window can be associated with a concrete workload iteration rather than only with the entire Slurm job. Neighboring fault-free iterations provide a within-allocation baseline for estimating performance degradation, reducing the influence of coarse-grained variation across jobs and production time periods.

This job-scoped orchestration distinguishes *SLoFI* from ad-hoc stress scripts: each perturbation is controlled, logged with actual timestamps, cleaned up, and converted into repeatable reliability evidence.

D. Cross-Layer Time Alignment

Reliability testing requires connecting injected events to both node-level symptoms and application-visible behavior. *SLoFI* records three timestamped streams during each job: injection events, node telemetry, and workload iteration boundaries. The injection log contains the perturbation type, target scope, actual start time, actual end time, and parameters. The telemetry stream is queried from the production monitoring system over the allocated nodes, and the workload log records the start and end of every repeated iteration.

These streams are not naturally aligned in production because node-level monitoring, workload logs, and telemetry queries are generated at different granularities. *SLoFI* aligns them on a common wall-clock timeline: instead of chaining relative `sleep()` calls, the telemetry scraper recomputes each collection boundary from absolute time. In our Tianhe deployment, the sampling interval is 15 seconds, matching the production monitoring granularity. For each injected event, *SLoFI* associates the injection log with victim-node telemetry samples and the corresponding workload iteration, producing an evidence record with the injected target, injection window, telemetry symptom, affected iteration, and cleanup status.

This mechanism supports operator review: engineers can check whether the expected telemetry symptom appears within the injection window, whether the corresponding iteration slows down, and whether the symptom disappears after cleanup.

E. Extensibility and Workflow Integration

Extending *SLoFI* to a new workload requires a Slurm job wrapper, iteration-boundary logging, and a workload-specific

success or timeout rule. The core framework is application-agnostic: scientific simulations, HPDA pipelines, HTC tasks, and AI training jobs can be supported as long as repeated iterations can be logged consistently. Adding a new perturbation requires a user-space executable and a configuration entry specifying its target scope, duration, and intensity.

This design separates reusable framework logic from site-specific configuration. The reusable part includes job-scoped orchestration, injection-event logging, cleanup, telemetry-window extraction, and evidence alignment. The site-specific part includes node naming rules, telemetry endpoints, workload paths, and operator-approved parameter ranges. Therefore, the planned public artifact focuses on the fault-injection framework and selected workload scripts, while production-specific configurations remain local to the deployment site.

III. Evaluation

This section evaluates whether *SLoFI* can generate effective and verifiable reliability-testing evidence under production constraints. We focus on two research questions:

RQ1: Can *SLoFI* safely create application-visible degradation evidence across production workloads?

RQ2: Can *SLoFI* align injection, telemetry, and workload records into operator-reviewable evidence?

A. Experimental Setup

We evaluate *SLoFI* on the Tianhe production Slurm-managed supercomputer. Table III summarizes the workload suite, including each workload's application class, run scope, and injected fault classes. The suite covers four classes: scientific simulation, HPDA, HTC, and CPU-based AI training.

TABLE III
Main experimental matrix used in the PDR study.

Workload	Class	Run Scope	Injected Fault Classes
GROMACS	Sci.	Multi-node	Resource, Process, Comm.
LAMMPS	Sci.	Multi-node	Resource, Process, Comm.
QE	Sci.	Multi-node	Resource, Process, Comm.
NAMD	Sci.	Multi-node	Resource, Process, Comm.
AI_DDP	AI	Multi-node	Resource, Process, Comm.
AI_ResNet18	AI	Single-node	Resource, Process
HPDA_DASK	HPDA	Single-node	Resource, Process
HPDA_Dist	HPDA	Multi-node	Resource, Process, Comm.
HTC_GRO	HTC	Multi-node	Resource, Process, Comm.

"Resource" faults include CPU and memory perturbations; "Process" denotes process noise; "Comm." denotes inter-node communication perturbations.

Each workload is executed cyclically within the same allocation. One iteration denotes one complete repeated execution of the workload, and its wall-clock duration is the basic observation unit. Disk perturbation is implemented but excluded from the main PDR study: for safety, it is confined to local storage pressure on the target node and does not perturb the shared parallel file system. Since the evaluated workloads are dominated by CPU, memory, and communication behavior, disk perturbation is instead covered in the Tianhe deployment. Across the workload suite, the campaign contains at least 200 workload iterations per workload and more than 1,800 workload iterations in total. Each included workload–perturbation pair is injected 10 times.

We use the Performance Degradation Ratio (PDR) to quantify slowdown:

$$\mathrm{PDR} = \frac{T_{fault} - T_{normal}}{T_{normal}} \times 100\%, \tag{1}$$

where T_{fault} is the injected iteration duration and T_{normal} is the average duration of neighboring fault-free iterations in the same allocation. Using neighboring iterations avoids comparison with unrelated jobs and makes the metric suitable for production environments with background variation.

B. Injection Effectiveness across Workloads (RQ1)

Figure 2 summarizes the PDR results across workload–perturbation pairs. We use this heatmap as coverage-oriented validation. The result shows that *SLoFI* can expose diverse and interpretable degradation paths across scientific simulation, HPDA, HTC, and CPU-based AI training workloads. Communication-intensive MPI and distributed-training workloads amplify communication and CPU-side perturbations; for example, the strongest observed mean slowdown exceeds 140% for a GROMACS–network-loss setting. Loosely coupled HTC-style jobs show more localized effects. These observations demonstrate that *SLoFI* can generate realistic and workload-visible degradation evidence across representative production workload classes.

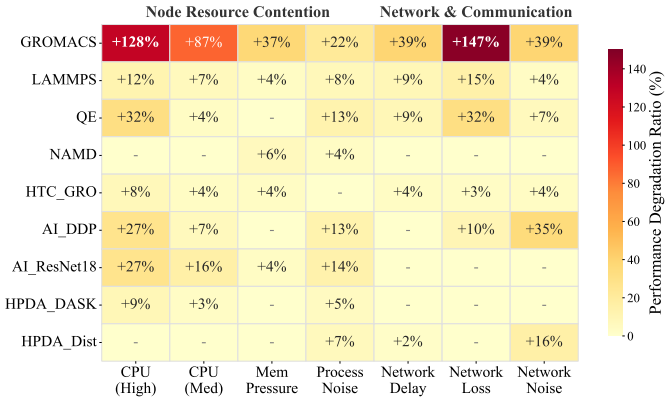

Fig. 2. Coverage-oriented PDR view across workload–perturbation pairs. Each reported cell gives the average PDR of one evaluated pair; cells marked with "–" indicate non-applicable or negligible-impact settings.

The safety boundary also holds in this campaign. We did not observe unintended node failures, scheduler-triggered node outages, or reported cross-job interference caused by the injections. Perturbation processes were confined to the target allocation and cleaned up after execution. The larger Tianhe deployment in Section IV further validates this safety boundary in the production reliability-testing pipeline.

C. Cross-Layer Alignment Validation (RQ2)

RQ2 is grounded in the same injection campaign as the PDR study. Across the reported workload–perturbation pairs,

the injected runs produce evidence records that link injection logs, victim-node telemetry windows, and workload iteration logs.

Figure 3 compares *SLoFI*'s absolute-clock alignment with a naive chained-sleep strategy on a representative injected run. Although both settings use the same injection and workload timestamps, sleep-based scraping accumulates query latency and drifts from the intended boundaries. By recomputing collection boundaries from wall-clock time, *SLoFI* keeps telemetry samples closer to the injection and iteration windows, making the generated labels usable for operator review and downstream anomaly-detection and root-cause-localization evaluation.

Fig. 3. Representative alignment comparison under collection latency. Absolute-clock scraping avoids additional drift introduced by chained sleep-based scraping.

IV. Deployment Experience and Lessons Learned

A. Deployment on Tianhe

Figure 4 shows the production deployment workflow of *SLoFI*. A reliability engineer submits a Slurm job with the target workload and injection configuration. After allocation, *SLoFI* selects victim nodes within the job scope, triggers job-scoped perturbations, and collects injection events, workload logs, and real-time telemetry. The aligned evidence is then used by downstream anomaly detection and root-cause localization modules, followed by cleanup and recovery validation.

SLoFI has been deployed on the Tianhe production supercomputer and validated by Tianhe engineers. The deployment covered all perturbation types in Table II, including resource, process, communication, and disk-related perturbations. Each perturbation type was exercised 30 times, covering 568 production nodes in total. To avoid mutual interference, each test targeted only one Slurm allocation. Across the deployment tests, engineers did not observe unintended node failures or reported cross-job interference, and all perturbation processes were removed after execution.

Tianhe engineers further confirmed the realism of injected symptoms through the resource-monitoring platform and the states of corresponding nodes and network devices. For the evaluated closed-loop cases, the anomaly detection module

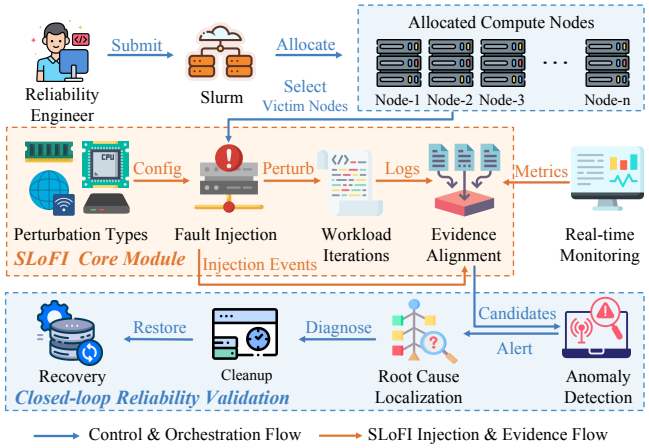

Fig. 4. Deployment workflow of *SLoFI* in the Tianhe reliability-testing pipeline.

reported injected abnormal symptoms with an average detection latency of 32.43 seconds [23], the root-cause localization module ranked the injected nodes within the Top-2 candidates in 86.67% of cases, and the system typically returned to normal within 40–60 seconds after cleanup. These results indicate that job-scoped low-privilege injection can provide practical evidence for reliability testing in shared Slurm-managed production supercomputers.

B. Case Study

Figure 5 shows a representative case from the deployed workflow. The target job is an 8-node AI_DDP workload, and Node 4 is selected as the victim node. *SLoFI* injects a CPU-high perturbation asynchronously during Iteration 70. The normal iterations take 638 seconds on average, while the injected iteration lasts 808 seconds and becomes a straggler.

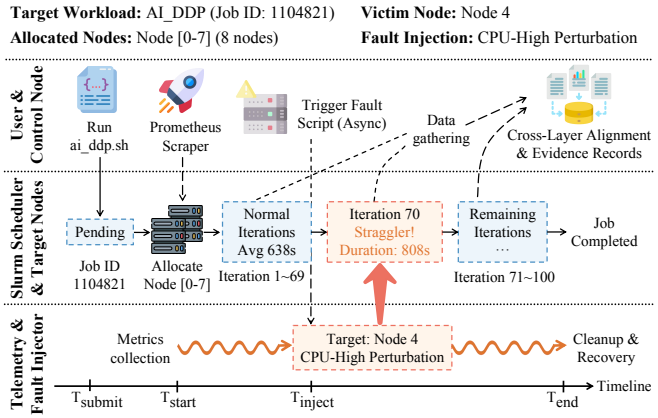

Fig. 5. Representative case study of a *SLoFI* experiment.

This case illustrates the operational value of *SLoFI*. The perturbation is confined to the allocated job, requires no privileged daemon, and is removed after the injection window. At the same time, the injected event is visible at three layers:

the injection log records the target node and fault type, the telemetry stream shows the CPU-side symptom during the injection window, and the workload log captures the delayed iteration. This cross-layer evidence allows engineers to verify whether the detector reports the abnormal symptom and whether the diagnosis module localizes the injected node.

C. Lessons Learned

Use the scheduler allocation as the safety contract. In a shared supercomputer, the Slurm allocation is not only a resource boundary but also an operational contract between users and operators. Binding injection, logging, and cleanup to this boundary makes low-privilege testing reproducible at other Slurm-managed sites without changing system control paths.

Validate symptoms, not hardware mechanisms. For production reliability workflows, operators care whether monitoring and diagnosis modules observe the right degradation symptoms at the right time and scope. Reproducing such symptoms in user space is often more deployable than emulating every physical fault source with privileged mechanisms.

Make every injection auditable. A useful production injection is not just a stress event. It should leave an auditable trail linking the injected target, telemetry symptom, workload slowdown, cleanup status, and recovery result. This evidence trail is what makes fault injection acceptable for operator review and reusable for downstream AIOps evaluation.

V. Conclusion

This paper presented *SLoFI*, a low-privilege fault-injection framework for reliability testing in production supercomputers. By combining user-space symptom emulation, Slurm job-scoped orchestration, and absolute-clock alignment, *SLoFI* generates controlled and verifiable degradation evidence without privileged system modification. Evaluation and deployment on Tianhe show that *SLoFI* can safely induce observable performance degradation across representative workloads and integrate into a closed-loop reliability workflow covering anomaly detection, root-cause localization, cleanup, and recovery validation.

References

[1] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belakaria, P. Bose, F. Cappello, P. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[3] B. Aksar, E. Sencan, B. Schwaller, O. Aaziz, V. J. Leung, J. M. Brandt, B. Kulis, M. Egele, and A. K. Coskun, "Prodigy: Towards unsupervised anomaly detection in production HPC systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 26:1–26:14.

[4] B. Aksar, E. Sencan, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, B. Kulis, M. Egele, and A. K. Coskun, "Runtime performance anomaly diagnosis in production HPC systems using active learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 693–706, 2024.

[5] E. Sencan, Y.-C. Lee, C. Casey, B. Schwaller, V. J. Leung, J. Brandt, B. Kulis, M. Egele, and A. K. Coskun, "Refine: A robust approach to unsupervised anomaly detection for production HPC systems," in *ISC High Performance 2025 Research Paper Proceedings*, 2025, pp. 1–12.

[6] P. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. Association for Computing Machinery, 2021, pp. 9–16.

[7] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[8] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 3241. Springer, 2004, pp. 97–104.

[10] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "PyTorch distributed: Experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, 2020.

[11] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[12] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[13] P. Omland, A. Netti, Y. Peng, A. Baldovin, M. Paulitsch, G. Espinosa, J. Parra, G. Hinz, and A. Knoll, "HPC hardware design reliability benchmarking with HDFIT," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 995–1006, 2023.

[14] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic: System Effects (SELSE)*, 2013.

[15] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sirbu, A. Bartolini, and A. Borghesi, "FINJ: A fault injection tool for hpc systems," in *Euro-Par 2018: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, vol. 11339. Springer, 2019, pp. 800–812.

[16] V. Formicola, S. Jha, D. Chen, F. Deng, A. Bonnie, M. Mason, J. Brandt, A. Gentile, L. Kaplan, J. Repik *et al.*, "Understanding fault scenarios and impacts through fault injection experiments in Cielo," *arXiv preprint arXiv:1907.01019*, 2019.

[17] Alibaba Group, "ChaosBlade: An easy to use and powerful chaos engineering toolkit," https://github.com/chaosblade-io/chaosblade, 2025, accessed: 2025-02.

[18] Cloud Native Computing Foundation (CNCF), "Chaos Mesh: A powerful chaos engineering platform for Kubernetes," https://chaos-mesh.org/, 2025, accessed: 2025-02.

[19] LitmusChaos, "LitmusChaos: Cloud-Native chaos engineering," https://litmuschaos.io/, 2025, accessed: 2025-02.

[20] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 2862. Springer, 2003, pp. 44–60.

[21] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 154–165.

[22] Cloud Native Computing Foundation (CNCF), "Prometheus: From metrics to insight. power your metrics and alerting with a leading open-source monitoring solution," https://prometheus.io/, 2025, accessed: 2025-02.

[23] S. Xia, Y. Sun, X. Pan, Y. Yuan, S. Zhang, S. Hu, L. Tao, Y. Li, and J. Feng, "Effective node-level anomaly detection in HPC systems via coarse-grained clustering and fine-grained model sharing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1180–1194.