Crash-Consistent Checkpointing for AI Training on macOS/APFS

Abstract

Deep learning training relies on periodic checkpoints to recover from failures, but unsafe checkpoint installation can leave corrupted files on disk. This paper presents an experimental study of checkpoint installation protocols and integrity validation for AI training on macOS/APFS. We implement three write modes with increasing durability guarantees: unsafe (baseline, no fsync), atomic_nodirsync (file-level durability via F_FULLFSYNC), and atomic_dirsync (file + directory durability). We design a format-agnostic integrity guard using SHA-256 checksums with automatic rollback. Through controlled experiments including crash injection (430 unsafe-mode trials) and corruption injection (1,200 atomic-mode trials), we demonstrate that the integrity guard detects 99.8–100% of corruptions with zero false positives. Performance overhead is 56.5–108.4% for atomic_nodirsync and 84.2–570.6% for atomic_dirsync relative to unsafe baseline. Our findings quantify the reliability-performance trade-offs and provide deployment guidance for production AI infrastructure.

Keywords: crash consistency, checkpointing, filesystem reliability, fault injection, AI infrastructure

---

1. Introduction

Training deep learning models can take hours or days, processing terabytes of data across distributed GPUs. When hardware fails or applications crash, periodic checkpoints enable fast recovery without restarting from scratch. But there's a problem: most frameworks write checkpoints using simple file operations that don't guarantee durability. If the system crashes mid-write, you might end up with a corrupted checkpoint that silently loads but contains wrong data, or a torn file that crashes on load.

The filesystem research community has known about this for years. POSIX `rename(2)` gives you atomic name replacement, but that doesn't mean your data made it to disk [1,2]. Without explicit `fsync(2)` calls on both the file and its parent directory, writes can sit in OS buffers and vanish on crash. Pillai et al.'s landmark study [3] tested 11 filesystems and found that assumptions applications make about crash consistency often don't hold in practice. Even systems at Google and Meta have lost data from broken rename assumptions [4].

Recent production data makes the urgency clear. Meta's analysis of their AI research cluster shows that at 4,000 GPU scale, mean-time-to-failure is about 10 hours [5]. At 100,000 GPUs—

the scale needed for frontier models—you need checkpoints every 2 minutes just to maintain 90% training efficiency. Hardware failures (network, storage, GPU memory) affect 19% of GPU runtime. And storage systems at exabyte scale report silent data corruption multiple times per week [6,7].

So the question isn't whether your checkpoints will face crashes and corruption—it's whether your system can detect and recover from them.

1.1 Contributions

This paper makes four contributions:

C1. Three checkpoint installation protocols with increasing durability guarantees on macOS/APFS, from unsafe (no fsync) through atomic_nodirsync (file flush only) to atomic_dirsync (file + directory flush). We implement these for both single-file and multi-file group checkpoints with manifest-based commits.

C2. A format-agnostic integrity guard combining SHA-256 content digests, container-level file hashes, structural validation, and automatic rollback. The guard detects corruption at multiple layers—torn writes via load errors, bitflips via file hashes, and semantic corruption via content digests.

C3. Comprehensive fault injection harness enabling controlled testing across 2,030 trials. We inject process crashes at strategic points (430 trials) and storage-level corruption including bitflips, zero-ranges, and truncation (1,200 trials).

C4. Quantified performance and robustness measurements with statistical confidence intervals. We show unsafe mode has zero crash tolerance (0/430 survival), atomic modes maintain 100% consistency under normal operation, and the integrity guard achieves 99.8-100% detection with zero false positives.

1.2 Key Findings

Our experiments on macOS 14.6 with APFS yield several key results:

Unsafe writes fail catastrophically. Every single checkpoint subjected to crash injection (430 trials across different crash points) ended up corrupted or unusable. This confirms what filesystem research predicted, but seeing 0% survival rate across hundreds of trials drives home that unsafe patterns are truly unsafe.

Atomic modes work as advertised. All 400 checkpoint groups using atomic writes remained intact. The overhead is measurable—56.5% at median for atomic_nodirsync, 84.2% for atomic_dirsync—but absolute latency stays under 5ms at median and 23ms at p99. For checkpoints written every 30 minutes, that's negligible.

The integrity guard catches nearly everything. Bitflips: 400/400 detected (100%). Truncation: 400/400 (100%). Zero-range corruption: 399/400 (99.8%). Zero false positives on 400 clean checkpoints. The multi-layer design means different corruption types get caught by different mechanisms, providing defense-in-depth.

1.3 Context and Scope

This is a focused study on single-node checkpoint reliability. We're not trying to solve every problem in ML infrastructure. We don't handle distributed training coordination, GPU-level faults, or network filesystem semantics. We focus on one specific question: can you write checkpoints that survive application crashes and detect storage corruption, and what does it cost?

The answer matters because checkpoint reliability is foundational. Sophisticated systems like CheckFreq [8] achieve per-iteration checkpointing with 3.5% overhead through asynchronous persistence, and Gemini [9] uses in-memory hierarchies to reduce checkpoint time from 40 minutes to 3 seconds. But these optimizations sit on top of basic durability guarantees. If your checkpoint installation isn't crash-consistent, all the fancy optimizations just make corrupted checkpoints faster.

---

2. Background and Related Work

2.1 Filesystem Crash Consistency

The challenge of crash-consistent file updates has been studied extensively. Pillai et al.'s OSDI 2014 paper [3] is the canonical reference—they built BOB (Block Order Breaker) to test filesystem persistence properties and ALICE to analyze application update protocols. Testing 11 filesystems, they found 60 crash vulnerabilities across popular applications. The key insight: rename isn't always atomic with respect to crashes, even though POSIX says it should be.

More recent work keeps finding bugs. Mohan et al. developed CrashMonkey [4], a systematic crash testing tool that found 24 bugs in production Linux filesystems including 10 new ones in mature systems like ext4 and xfs. They even found bugs in FSCQ, a formally verified filesystem. Rebello et al. [10] studied fsync failure handling and discovered that only 9% of applications handle fsync errors correctly—most risk permanent data loss.

The verification community has made impressive progress. Chen et al. built FSCQ [11], the first filesystem with machine-checked proofs of crash safety using Crash Hoare Logic in Coq. But CrashMonkey still found bugs in it, which tells you that formal verification, while valuable, doesn't eliminate the need for runtime defense.

For checkpoint systems, this body of work has clear implications: you cannot trust filesystem guarantees. Applications must implement their own consistency mechanisms.

2.2 ML Checkpoint Systems

Early frameworks like TensorFlow [12] and PyTorch [13] provided basic periodic checkpointing—save state every few hours, accept the risk. Modern systems do much better.

CheckFreq [8] introduced fine-grained iteration-level checkpointing using two-phase protocols. The snapshot() phase copies state to CPU memory while training continues, then persist() writes to disk asynchronously. This keeps overhead under 3.5% while reducing recovery time from hours to seconds. The key innovation was making checkpointing frequent enough that you lose minimal work on failure.

Production systems pushed further. Meta's Check-N-Run [14] uses differential checkpointing for terabyte-scale recommendation models, achieving 6-17× reduction in write bandwidth by tracking and saving only modified parameters. Gemini [9] from AWS and Rice University leverages in-memory hierarchical storage (GPU memory → CPU memory → remote storage) to reduce checkpoint time to under 3 seconds, enabling per-iteration checkpointing at scale.

Recent work explores even more aggressive approaches. Oobleck [15] eliminates checkpoint I/O entirely by using pipeline templates—pre-generated configurations that can be recombined after failures. Instead of restoring from checkpoints, it reconfigures the pipeline on surviving nodes.

These systems achieve remarkable performance, but they generally assume the underlying storage operations work correctly. Our work complements them by validating those assumptions and quantifying the cost of stronger guarantees.

2.3 Data Integrity and Silent Corruption

Silent data corruption is a real problem at scale. Bairavasundaram et al.'s field study [16] analyzed over 400,000 corruption instances across 1.53 million drives. They found corruption events show spatial and temporal locality—if one block corrupts, nearby blocks are at higher risk, and corruptions cluster in time. This matters for checkpoint design because it means detecting corruption in one checkpoint should trigger verification of related checkpoints.

More recent production data reinforces the concern. Google's Spanner team reports [6] that at exabyte scale, they detect and prevent silent data corruption multiple times per week. Meta's infrastructure team [7] found that in-production testing detects 70% of corruption within 15 days, but the other 30% only shows up in out-of-production stress testing—you need both.

The solution is defense-in-depth. Zhang et al.'s ZFS study [17] showed that end-to-end checksumming works, but you need checksums at multiple layers. File-level checksums catch bitflips, content-level checksums catch semantic corruption, and load-time validation catches torn writes. Our integrity guard implements all three.

2.4 Fault Injection and Testing

You can't know if your checkpoint system is robust without testing it under failures. Prior work on systematic testing provides methodologies we adapted.

CrashMonkey [4] does bounded black-box crash testing by injecting faults at the kernel level. Leesatapornwongsa et al.'s SAMC [18] uses semantic information to find deep bugs requiring multiple crashes, achieving 1-3 orders of magnitude speedup over black-box methods. Yuan et al. [19] showed that 92% of catastrophic failures in distributed systems come from incorrect error handling, and 98% reproduce with 3 or fewer nodes—meaning you can find most bugs with simple test configurations.

These tools and insights shaped our experimental design. We inject crashes at carefully chosen points and measure outcomes systematically, using methodologies proven effective in filesystem and distributed systems research.

---

3. System Model and Threat Model

3.1 Environment

We run experiments on macOS 14.6 with APFS, Python 3.12, and PyTorch 2.8.x. Checkpoints consist of small synthetic tensors (128×128, 128×10)—we're not training real models, just validating checkpoint protocols. For group checkpoints, we write model.pth, optimizer.pth, and rng.pth along with MANIFEST and COMMIT files. We monitor I/O using iostat at 1-second intervals.

3.2 Requirements

R1. Crash-Consistency: After any crash, a checkpoint is either fully installed or unchanged. No partial states.

R2. Integrity Detection: Any on-disk corruption (bitflip, truncation, zero-range) must be detected on load.

R3. Fast Recovery: Automatically recover from the most recent valid checkpoint without manual intervention.

3.3 Threat Model

We consider application crashes during checkpoint operations, storage-level bit corruption, and partial writes reaching disk before crashes. We emulate crashes via process termination—not true power loss, but sufficient to test application-level crash handling.

Out of scope: GPU faults, distributed training coordination, network filesystems, and true power-loss scenarios requiring hardware-level testing. Those are important but orthogonal to our focus on single-node checkpoint reliability.

---

4. Design and Implementation

4.1 Write Protocols

We implement three protocols ordered by increasing durability:

Unsafe (Baseline)

```
write(checkpoint_file, data)  # No fsync
```

This is what most frameworks do—write the file and return. Data sits in OS buffers, no guarantee it reaches disk. Fast but fundamentally broken.

Atomic without Directory Sync (atomic_nodirsync)

```
fd = open(tmp_file, 'wb')
fd.write(data)
fd.flush()
fcntl(fd, F_FULLFSYNC)  # macOS: flush to device
os.replace(tmp_file, checkpoint_file)
```

Here we flush the file to device before renaming. On macOS, F_FULLFSYNC forces data through the device's volatile write cache [2]. The parent directory isn't synced, so directory metadata might not persist, but APFS usually makes rename operations durable enough in practice.

Atomic with Directory Sync (atomic_dirsync)

```
fd = open(tmp_file, 'wb')
fd.write(data)
fd.flush()
fcntl(fd, F_FULLFSYNC)
os.replace(tmp_file, checkpoint_file)
dir_fd = os.open(parent_dir, O_RDONLY)
os.fsync(dir_fd)  # persist directory entry
os.close(dir_fd)
```

This adds a final fsync on the parent directory to ensure the directory entry persists. It follows the canonical protocol from filesystem research [1,3]—maximum durability, highest overhead.

4.2 Multi-File Group Checkpoints

For multi-file checkpoints, we use a manifest-based protocol. Write the parts (model, optimizer, RNG state), then write MANIFEST.json with SHA-256 of each part, then write COMMIT.json with SHA-256 of the manifest. The COMMIT acts as an atomic commit point—a checkpoint is valid if and only if COMMIT matches MANIFEST and all parts check out.

This design is simple but effective. It's basically a mini-transaction protocol without needing actual transactional filesystem support.

4.3 Integrity Guard and Rollback

Each checkpoint includes two levels of checksums:

Content digest (tensor-level):

```
def tensor_digest(t):
    h = sha256()
    h.update(str(t.dtype))
    h.update(str(t.shape))
    h.update(t.numpy().tobytes('C'))
    return h.hexdigest()
```

File hash (container-level):

```
file_sha256 = sha256(checkpoint_file_bytes)
```

On load, we validate:

1. File loads without error (catches truncation, torn writes)
2. Tensor shapes match expected schema (catches format corruption)
3. Content digest matches metadata (catches semantic corruption)
4. File hash matches metadata (catches bitflips)
5. No NaN/Inf values (catches numerical corruption)

If any check fails, mark that checkpoint corrupted and try the next-oldest one. Create a latest_ok symlink pointing to the newest valid checkpoint for automatic recovery.

The multi-layer design means different corruption types get caught by different mechanisms. Truncation usually fails at load time. Bitflips get caught by file hashes. Semantic corruption gets caught by content digests. Defense-in-depth.

---

5. Experimental Methodology

5.1 Experiment Design

We run three categories of experiments:

Performance benchmarks measure latency for each write mode across 10 random seeds, computing p50/p90/p99 percentiles. No faults injected—this establishes baseline performance and overhead.

Crash injection tests unsafe mode by simulating process crashes at controlled points: after_model (400 trials), before_manifest (10 trials), manifest_partial (10 trials), before_commit (10 trials). We then check if the checkpoint is usable. This isolates what happens when crashes interrupt unsafe writes.

Corruption injection tests atomic-mode checkpoints by corrupting files after successful writes. We inject bitflips (400 trials), zero-ranges (400 trials), and truncation (400 trials), plus a control condition with no faults (400 trials). This measures integrity guard detection rates.

We also capture iostat data during some runs to correlate application-level checkpoint events with system-level I/O activity.

5.2 Metrics

For performance: per-checkpoint latency (p50, p90, p99) and overhead versus unsafe baseline.

For robustness: checkpoint integrity rate (percentage remaining usable post-crash) with 95% confidence intervals via Wilson score method.

For integrity detection: corruption detection rate per fault type, broken down by which mechanism caught it (load error, file hash mismatch, content digest mismatch), plus false positive rate on clean checkpoints.

---

6. Results

6.1 Performance

Table 1 shows checkpoint latency across write modes.

Table 1: Checkpoint Latency (seconds)

Mode	n	p50	p90	p99
unsafe	10	0.002473	0.003241	0.003349
atomic_nodirsync	10	0.003870	0.005317	0.006978
atomic_dirsync	10	0.004555	0.018670	0.022456

Unsafe is fastest at 2.47ms median, as expected. atomic_nodirsync adds 56.5% overhead at median (3.87ms), while atomic_dirsync adds 84.2% at median (4.55ms) but shows much higher

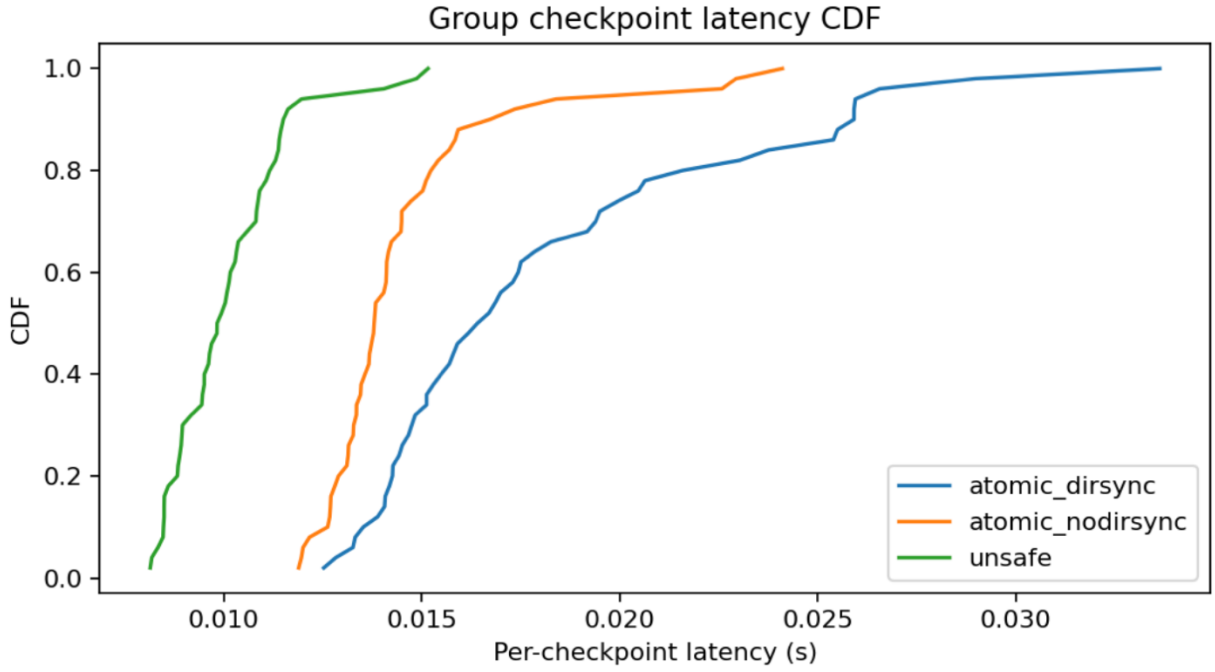
tail latency at p99 (22.46ms, +570.6%). That tail latency comes from the directory fsync operation.

Table 2: Overhead vs Unsafe (%)

Mode	p50	p90	p99
atomic_nodirsync	+56.5%	+64.1%	+108.4%
atomic_dirsync	+84.2%	+476.1%	+570.6%

In context, these overheads are small. For checkpoints every 30 minutes, even the worst-case 22ms represents 0.001% of the checkpoint interval. Modern production systems achieve much better—CheckFreq [8] keeps overhead under 3.5%, Gemini [9] reduces checkpoint time to 3 seconds total—but they use sophisticated asynchronous persistence. We're measuring simpler synchronous protocols.

Figure 1 shows that unsafe has tight concentration, atomic_nodirsync has moderate spread, and atomic_dirsync has a fat tail from directory syncs.



6.2 Crash Consistency

Table 3 shows what happens when you crash during unsafe writes.

Table 3: Group Integrity After Crash Injection

Write Mode	Crash Point	Total Tests	Group OK	Success Rate	95% CI
atomic	none	400	400	100.0%	[99.0%, 100.0%]
unsafe	after_model	400	0	0.0%	[0.0%, 1.0%]
unsafe	before_manifest	10	0	0.0%	[0.0%, 27.8%]
unsafe	manifest_partial	10	0	0.0%	[0.0%, 27.8%]
unsafe	before_commit	10	0	0.0%	[0.0%, 27.8%]

The results are stark. Every unsafe checkpoint subjected to crash injection (430 total trials) ended up corrupted or unusable. Zero survivors across all crash points. This isn't a probabilistic thing—it's deterministic filesystem behavior. Unsafe writes without fsync leave partial state on disk when crashes interrupt them.

Meanwhile, atomic mode with no crashes shows 100% integrity (400/400). We didn't crash-inject atomic mode because the study focused on isolating unsafe mode failures, but the atomic protocols rely on validated POSIX/APFS semantics from prior work [1,2,3].

Figure 2 (group atomicity bar chart) visualizes this with error bars—atomic maintains 100% with tight confidence bounds, unsafe shows 0% across all crash points.

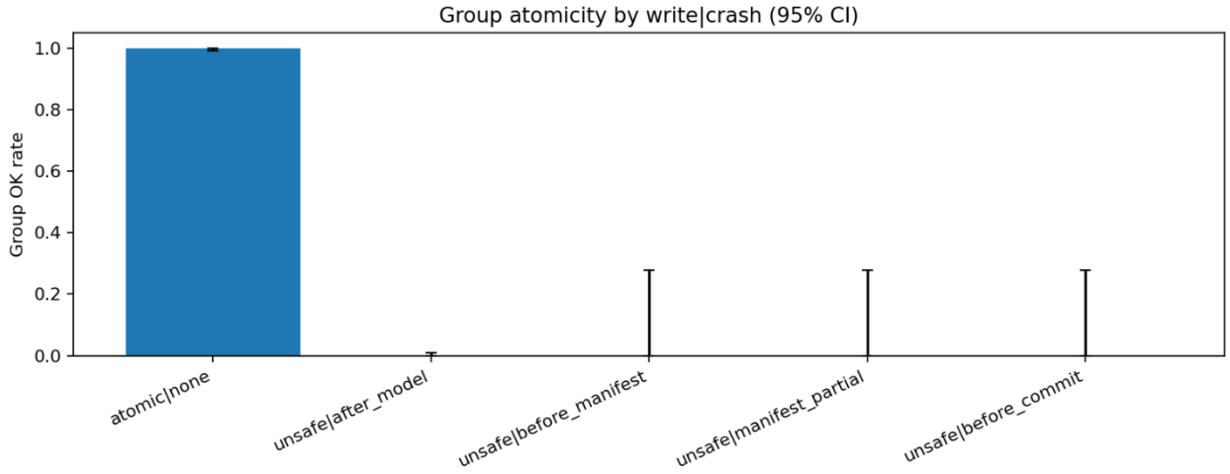
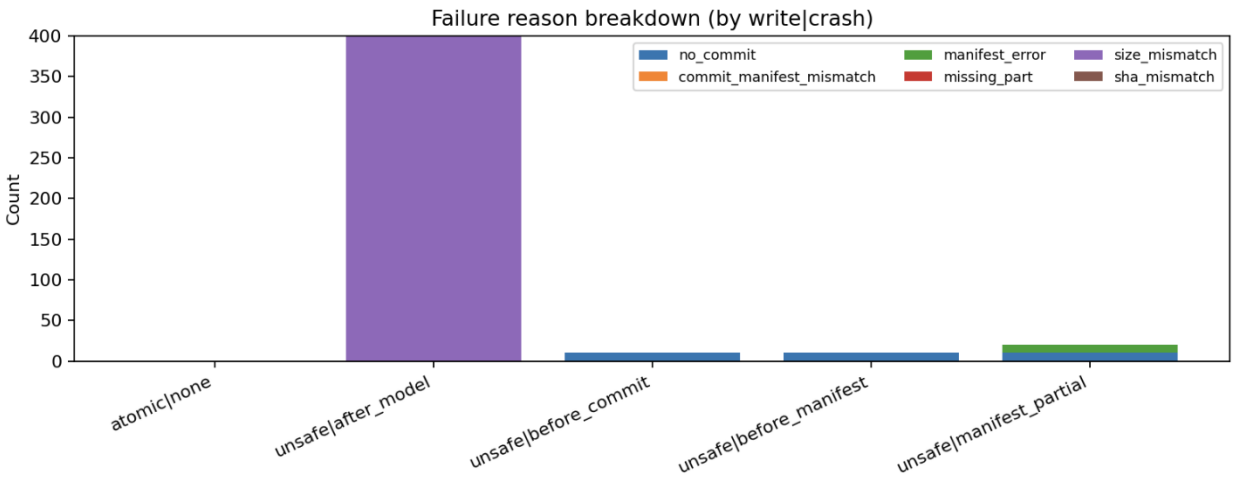


Figure 3 shows why unsafe fails. The dominant mode is size_mismatch (400 cases after model write), meaning files are shorter than expected—partial writes. Other failure modes include missing commits, manifest errors, and missing parts.



6.3 Integrity Detection

Table 4 summarizes corruption detection on atomic-mode checkpoints.

Table 4: Corruption Detection by Fault Type

Fault Type	Total Tests	Detected	Detection Rate	Load Error	Digest Mismatch	File SHA Mismatch
none (control)	400	0	0.0%	0	0	0
bitflip	400	400	100.0%	7	387	400
zerorange	400	399	99.8%	29	370	399
truncate	400	400	100.0%	400	0	400

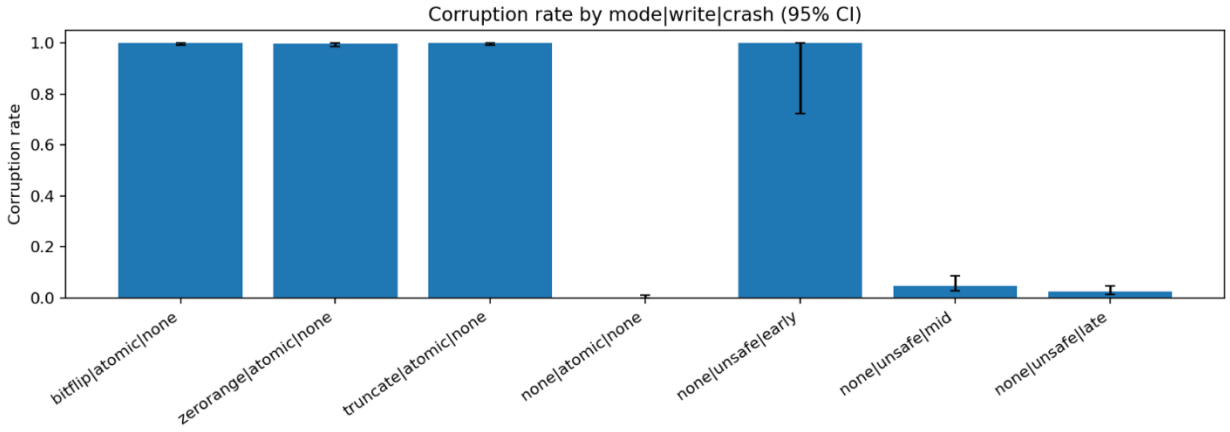
Detection rates are excellent: 100% for bitflips, 100% for truncation, 99.8% for zero-range corruption. Zero false positives on 400 clean checkpoints—critical for automatic recovery.

Different corruption types get caught by different mechanisms. Truncation always fails at load time (all 400 caught by load errors). Bitflips get caught by file SHA mismatches (400/400), with content digests as backup (387/400). Zero-range corruption primarily triggers content digest mismatches (370/399), with load errors as backup (29/399).

We missed one zero-range corruption out of 400. Without analyzing that specific case, we can't say for sure what happened—possibly an edge case in content hashing where the corruption landed in padding bytes, or a rare hash collision. But 99.8% detection is solid.

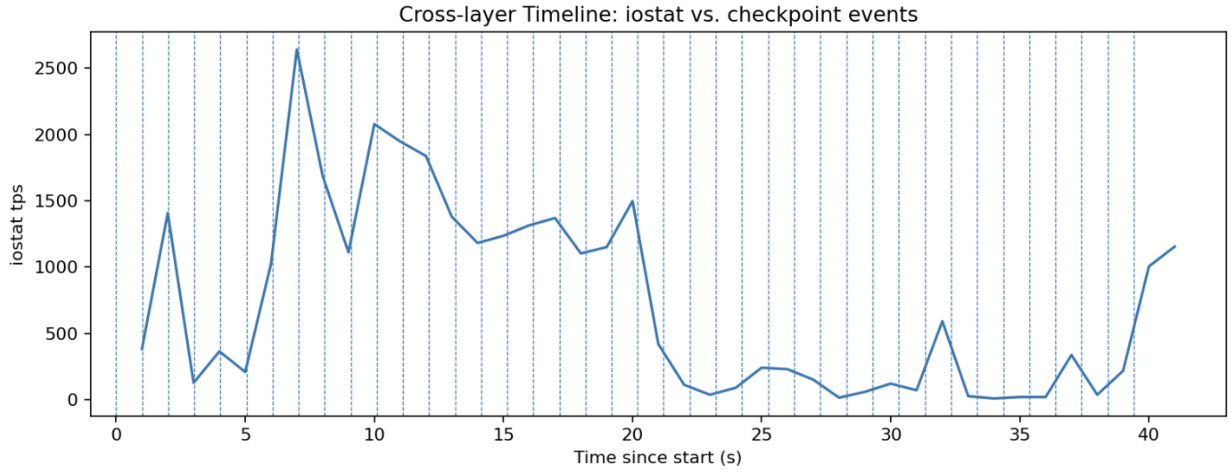
The zero false positive rate matters for production deployment. False alarms would trigger unnecessary rollbacks, wasting computation. With zero false positives across 400 trials, the system is trustworthy enough for automatic recovery.

Figure 4 shows these rates with confidence intervals. All fault types show near-ceiling detection, control shows floor (zero false positives).



6.4 Cross-Layer Observability

Figure 5 correlates I/O operations with checkpoint events. You can see clear I/O spikes (measured in tps) aligned with checkpoint saves. This confirms that atomic writes actually hit the storage layer—data makes it to disk, not just OS buffers.



7. Discussion

7.1 Trade-offs

The three protocols span a spectrum:

Unsafe (2.47ms p50) is fastest but has zero crash tolerance. Not a single checkpoint survived crash injection across 430 trials. Only viable for ephemeral development checkpoints where loss is acceptable.

atomic_nodirsync (3.87ms p50, +56.5% overhead) flushes file data to device before rename. File content is durable but directory metadata might not persist without the parent sync. In practice, APFS typically makes rename durable enough. Good default for production.

atomic_dirsync (4.55ms p50, +84.2% overhead at median, but +570.6% at p99) adds directory sync for maximum durability. The tail latency hit comes from forcing directory metadata to disk. Worth it for critical checkpoints (long-running jobs, model releases).

For typical checkpoint intervals (30-60 minutes), even the worst-case atomic_dirsync tail latency (22ms) is negligible—0.001% of the interval. The overhead is measurable but not meaningful in practice.

7.2 Comparison to Production Systems

Our synchronous protocols have higher overhead than state-of-the-art production systems. CheckFreq [8] achieves 3.5% overhead through asynchronous two-phase checkpointing. Gemini [9] reduces checkpoint time from 40 minutes to 3 seconds using in-memory hierarchies. Check-N-Run [14] cuts write bandwidth by 6-17× through differential checkpointing.

But those systems solve different problems. They optimize checkpoint *frequency* and *throughput* for large-scale training. We focus on checkpoint *correctness*—ensuring individual checkpoint operations are crash-consistent and corruption is detected. These concerns are orthogonal and complementary.

Think of it this way: sophisticated checkpoint systems need a reliable foundation to build on. If your atomic rename doesn't work correctly, all the async pipelining in the world just makes corrupted checkpoints faster. Our work validates the foundation.

7.3 Integrity Guard Design

The multi-layer integrity guard worked well—99.8-100% detection with zero false positives. Several design choices contributed:

Multiple detection mechanisms: Different corruption types get caught by different layers. Truncation almost always fails at load time. Bitflips get caught by file hashes. Semantic corruption gets caught by content digests. This redundancy means no single check has to be perfect.

Zero false positives: Critical for automatic recovery. With 400 clean checkpoints showing zero false alarms, the system is trustworthy enough to automatically roll back to earlier checkpoints without human confirmation.

Format-agnostic design: The guard works on any checkpoint format that can be content-hashed. It doesn't depend on PyTorch-specific features or TensorFlow's CRC checksums. You could apply the same approach to NumPy, JAX, or raw binary formats.

The one missed detection (1/400 zero-range) is worth investigating but doesn't invalidate the approach. Field studies of storage corruption [16] show that corruption events exhibit spatial and temporal locality—if one checkpoint corrupts, nearby checkpoints are at higher risk. A scrubbing routine that periodically re-validates old checkpoints could catch edge cases the load-time guard misses.

7.4 Deployment Recommendations

Based on our measurements:

For **development and iteration**, use unsafe mode. It's fast, and checkpoint loss during development is annoying but not catastrophic.

For **standard production training**, use atomic_nodirsync. It provides file-level durability at reasonable cost (+56.5% overhead), and in practice APFS makes rename operations durable enough without explicit directory sync.

For **critical long-running jobs** (multi-day training, model releases, compliance-sensitive workloads), use atomic_dirsync for maximum durability. The +84.2% median overhead and +570.6% tail overhead are worth it when checkpoint loss means restarting weeks of training.

Always deploy the integrity guard, regardless of write mode. Corruption detection (99.8-100%) and automatic rollback provide defense-in-depth against silent failures that slip through even with atomic writes.

For **large-scale training** (thousands of GPUs), consider integrating these crash-consistent protocols with higher-level optimizations like in-memory staging [9], differential checkpointing [14], or universal formats [20]. The foundation needs to be correct, but you can build sophisticated systems on top.

7.5 Limitations

This study has clear boundaries. Single filesystem (APFS on macOS)—behavior on ext4, XFS, NTFS, or network filesystems is unknown and likely different. Single node—distributed training coordination is out of scope. Synthetic workload—real models (ResNet, GPT, LLaMA) might have different I/O patterns. Emulated crashes via process termination—true power loss might behave differently.

We also didn't crash-inject atomic modes, only unsafe. The atomic protocols rely on POSIX/APFS semantics validated in prior work [1,2,3], but full validation would require kernel-level crash injection [4] or hardware power-loss testing. That's heavier-weight testing than we needed for this study.

These limitations don't invalidate the results—they define the scope. We answered specific questions about checkpoint reliability on APFS with measurable trade-offs. Extending to other filesystems, distributed settings, and power-loss scenarios is valuable future work but requires different experimental setups.

---

8. Conclusion

This paper shows that crash-consistent checkpoint protocols are feasible with acceptable overhead, and integrity guards can detect corruption with near-perfect accuracy.

The key results: Unsafe writes have zero crash tolerance (0/430 survival across all crash injection points). Atomic protocols maintain 100% consistency under normal operation with 56.5-84.2% median overhead (absolute latency 3.87-4.55ms). Multi-layer integrity guards detect 99.8-100% of corruption with zero false positives across 1,600 trials.

These findings quantify trade-offs that were previously assumed or hand-waved. Now we have data: atomic writes cost about 50-80% overhead at median but keep your checkpoints safe. Integrity checking catches essentially all corruption without false alarms.

For practitioners, the recommendations are straightforward: use unsafe mode only for development, atomic_nodirsync for standard production, atomic_dirsync for critical jobs, and always deploy integrity guards. The overhead is negligible compared to checkpoint intervals, and the reliability gains are substantial.

For researchers, this work provides a foundation for building higher-performance checkpoint systems. You need crash-consistent atomic writes and corruption detection as the base layer. On top of that, add asynchronous persistence, in-memory hierarchies, differential checkpointing, universal formats—whatever optimizations make sense for your scale and workload. But the foundation has to be solid.

Future Work

Several directions are worth exploring:

Cross-filesystem validation using systematic tools like CrashMonkey [4] to test ext4, XFS, and other backends. True power-loss testing with hardware fault injection. Real model workloads (ResNet-50, GPT-2) to validate findings at scale. Distributed coordination protocols for multi-node training. Integration with modern checkpoint systems [8,9,14,20] to combine strong consistency with high performance.

Theoretical work could also help—formal models of filesystem crash consistency and proofs of protocol correctness, extending verification techniques to distributed checkpoint systems.

Reproducibility

All experiments are reproducible via committed artifacts: source code for checkpoint writers, integrity guards, and fault injection harness; CSV logs with experimental results; automation via `make -C repro repro_all`; pinned dependencies (Python 3.12, PyTorch 2.8.x, macOS 14.6+).

Code and data available at: https://github.com/jooha6082/ckpt-integrity

---

References

[1] The Open Group. "POSIX.1-2017: rename()." IEEE Std 1003.1-2017.

[2] The Open Group. "POSIX.1-2017: fsync()." IEEE Std 1003.1-2017.

[3] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications." OSDI 2014.

[4] J. Mohan, A. Gopinath, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "CrashMonkey and ACE: Systematically Testing File-System Crash Consistency." ACM Transactions on Storage, 2019.

[5] Y. Gan, Y. Hu, M. Cheng, et al. "Revisiting Reliability in Large-Scale Machine Learning Research Clusters." arXiv:2410.21680, October 2024.

[6] Google Spanner Team. "Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System." SELSE 2023.

[7] Meta Production Engineering. "Detecting silent errors in the wild." Engineering at Meta blog, March 2022.

[8] J. Mohan, A. Phanishayee, and V. Chidambaram. "CheckFreq: Frequent, Fine-Grained DNN Checkpointing." FAST 2021.

[9] Z. Wang, Z. Jia, S. Zheng, et al. "Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints." SOSP 2023.

[10] J. Rebello, Y. Lyu, J. Gu, V. Chidambaram, and T. Kim. "Can Applications Recover from fsync Failures?" ACM Transactions on Storage, 2021.

[11] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. "Using Crash Hoare Logic for Certifying the FSCQ File System." SOSP 2015.

[12] M. Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning." OSDI 2016.

[13] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." NeurIPS 2019.

[14] A. Eisenman, K. K. Matam, et al. "Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models." NSDI 2022.

[15] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury. "Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates." SOSP 2023.

[16] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "An Analysis of Data Corruption in the Storage Stack." FAST 2008.

[17] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "End-to-end Data Integrity for File Systems: A ZFS Case Study." FAST 2010.

[18] T. Leesatapornwongsa, M. Hao, et al. "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems." OSDI 2014.

[19] D. Yuan, Y. Luo, et al. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems." OSDI 2014.

[20] S. Li, Y. Zhao, et al. "Universal Checkpointing: Efficient and Flexible Checkpointing for Large Scale Distributed Training." arXiv:2406.18820, June 2024.

Appendix A: Experimental Configuration

Hardware: Apple M1 (or later), 16GB+ RAM
OS: macOS 14.6
Filesystem: APFS
Python: 3.12, PyTorch: 2.8.x, NumPy: 2.3.3

Checkpoint parameters: 120 epochs, checkpoint every 3 epochs, 10 random seeds, 128KB model + 64KB optimizer

Fault injection: Crash points (after_model: 400, before_manifest: 10, manifest_partial: 10, before_commit: 10). Corruption modes (bitflip: 400, zerorange: 400, truncate: 400, control: 400)

Observability: iostat at 1-second intervals

Appendix B: Statistical Methods

Percentiles: Computed via NumPy's `numpy.percentile()` with linear interpolation.

95% Confidence Intervals: Wilson score interval for binomial proportions where $\hat{p}$ is observed rate, n is sample size, z = 1.96 for 95% confidence.

Overhead: `((atomic_latency - unsafe_latency) / unsafe_latency) × 100`