

Silent Data Corruption in AI

Authors:

Nishant George, NVIDIA
Harish Dattatraya Dixit, Emel Goksu, Meta Platforms Inc.
Bharath Parthasarathy, Amber Huffman, Google
Sudhanva Gurumurthi, Vilas Sridharan, AMD
Thiago Macieira, Arani Sinha, Intel
Lisa Minwell, Dean Liberty, ARM
Rob Chappell, Microsoft

Dec 16, 2025

Version 1.1

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Version History

Date	Version #	Author (s)	Description
July 25, 2025	0.3	Above Listed Authors	Draft adhering to OCP template
August 8, 2025	1.0 RC	Nishant George	Ready for approvals before release
August 26, 2025	1.0	Nishant George	Released
October 31, 2025	1.1	Nishant George, Sudhanva Gurumurthi, Vilas Sridharan	Added section 5.3 and more references
December 16, 2025	1.1	Bharath Parthasarathy	Addressed OCP reviewer comments

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Executive Summary and Problem Statement

This Open Compute Project (OCP) white paper addresses the critical impact of Silent Data Corruption (SDC) on large-scale AI/ML systems. SDCs or hardware faults that go undetected subtly undermine the accuracy and trustworthiness of AI models during training and inference. This paper enumerates various hardware fault types and explores the significant challenge of correlating these low-level errors with their actual effect on AI performance. It outlines best-practices for SDC mitigation, including innovative detection and recovery strategies designed for high-performance computing and machine learning environments.

A key challenge is the misalignment between hardware fault metrics and AI correctness metrics. Hardware metrics like FIT and AVF focus on low-level failures such as bit flips, while AI metrics, like accuracy and loss, measure aggregate model performance. This gap makes it difficult to quantify the real-world impact of hardware faults on AI outcomes. The inherent fault-masking behavior of AI systems can also obscure SDCs, delaying or preventing their detection. At the massive scale of today's AI deployments, the probability of an SDC event becomes substantial, creating a "Needle in a Haystack" challenge where rare errors are difficult to detect. A crucial open research question is how to bridge this gap to develop actionable methodologies that account for the non-determinism and scale of modern AI workloads.

Call to Action

Addressing the SDC challenge requires a collaborative approach that extends beyond traditional hardware solutions. This paper serves as a direct call to action to the AI/ML developer community to integrate resilience into their code and training pipelines from the outset. We encourage developers to:

- Implement novel detection mechanisms: Proactively use workload-specific methods like algorithmic gradient clipping to mitigate NaN propagation, or divergence detection to monitor neuron outputs.
- Adopt scalable mitigation strategies: Employ infrastructure-level strategies such as reductive triage or hyper checkpointing to accelerate the identification and quarantine of faulty nodes. For synchronous training, consider tri-variate computational architectures with shadow nodes to verify results and contain errors.
- Participate in open research: Contribute to the development of standardized resilience benchmarks, similar to MLPerf, to quantify resilience and evaluate SDC impact in a repeatable, scalable, and representative manner.

By encouraging open collaboration and sharing innovative mitigation methods, we can accelerate progress and ensure the integrity and trustworthiness of AI models at scale.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Table of Contents

1.0 Introduction....6
2.0 Compliance with Open Compute Project Tenets....6
2.1 Openness....6
2.2 Efficiency....6
2.3 Impact....7
2.4 Scale....7
2.5 Sustainability....7
3.0 Background on Hardware Faults....7
3.1 Defining Silent Data Corruption....8
3.2 Root Causes of SDC....9
3.2.1 Permanent Faults....9
3.2.2 Transient Faults....10
3.3 Hardware Fault Metrics and AI Correctness Metrics....10
3.3.1 Hardware Reliability and Fault Metrics....11
3.3.1.1 Failures in Time (FIT)....11
3.3.1.2 Architectural Vulnerability Factor (AVF)....11
3.3.1.3 DelayAVF....12
3.3.2 AI Correctness Metrics....12
4.0 Background on AI....13
4.1 Characteristics of AI Workloads....13
4.2 Hardware Usage Patterns in AI....13
4.3 AI Software Stacks and Execution Models....14
5.0 Best Practice Mitigations....14
5.1 Novel Detection Mechanisms....15
5.1.1 SDCs in Training Workloads....16
5.1.1.1 NaN (Not-a-Number) Propagation....16
5.1.1.2 Corrupted Gradient Variance....17
5.1.2 SDCs in Inference Workloads....17
5.1.3 Impact of SDCs....18
5.1.4 Detection of SDCs in AI hardware....18
5.1.4.1 Infrastructure strategies....18
5.1.4.2 Stack Strategies....19
5.2 Error Detection....20
5.2.1 End-to-End Integrity Checks (Informational Redundancy)....20
5.2.2 Replication and Consistency Checks (Spatial Redundancy)....21
5.2.3 Integer Residue Checking....21

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

5.3 Proactive Testing and Screening..... 21
5.4 Error Recovery.....22
5.4.1 Checkpoint/Restart (Backward Error Recovery, Temporal Redundancy)..... 22
5.4.2 Algorithm-Based Fault Tolerance (ABFT) (Forward Error Recovery, Informational Redundancy).....22
5.5 Implications for AI..... 22
6.0 Open Research Questions.....24
6.1 Defining the Core Challenge.....24
6.2 Gap Between Hardware Fault Metrics and AI Correctness..... 24
6.2.1 Partial Tolerance to Errors..... 25
6.2.2 Layer-Specific Sensitivity.....25
6.2.3 Model Mapping and Accelerator-Aware Fault Effects.....25
6.2.4 Fault-Injection Studies & Standardized Benchmarks.....26
6.3 The “Needle in a Haystack” Challenge..... 27
6.3.1 Detection of Rare Events is Expensive.....27
6.3.2 Delayed or Non-Observable Impact.....27
6.3.3 Scalable Mitigation Strategies.....27
7.0 Conclusion and Future Direction.....27
8.0 References..... 28
9.0 License..... 32

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

1.0 Introduction

High-Performance Computing (HPC) has traditionally been the domain of national laboratories, scientific research institutions, and specialized industries. These fields routinely employ massive computational resources to solve complex simulations in fields like climate modeling, nuclear physics, and genomics. However, the advent of large-scale Artificial Intelligence (AI) and Machine Learning (ML) workloads has effectively brought HPC to the commercial and consumer spheres. Modern AI models, such as Convolutional Neural Networks (CNNs) or large language models (LLMs), can contain billions of parameters, requiring extensive computational power during both training and inference. The proliferation of GPUs, TPUs, and other specialized accelerators has enabled even small- to medium-scale organizations to tap into HPC-level resources through cloud services and specialized on-premise hardware.

This mainstream adoption of HPC-like infrastructure for AI introduces questions about reliability and data integrity that were once mostly confined to scientific supercomputing environments. In particular, Silent Data Corruption (SDC), in which errors occur without triggering hardware detection, poses challenges to AI workloads, where small perturbations in certain model parameters or intermediate data can have outsized impact on accuracy (during, say, inference,) and even on validity of models during training because of unintended biases [2].

2.0 Compliance with Open Compute Project Tenets

This aligns with the Server Resilience Working Group's initiatives to address SDC challenges.

2.1 Openness

We are driving for increased openness to tackle SDC in large-scale AI. Through this initiative, we aim to accelerate collective progress by encouraging open collaboration, sharing innovative mitigation methods, and disseminating best practices. Transparent communication regarding hardware fault and AI correctness metrics will be key to preserving the integrity and trustworthiness of AI models, benefiting the entire community.

2.2 Efficiency

Mitigating SDC in AI demands significant efficiency. Given the immense scale and computational intensity of modern AI, traditional fault-tolerance methods are simply unfeasible due to prohibitive overheads in performance, power, and resources. Our focus is on developing solutions that achieve robust data integrity while optimizing the balance between error protection and operational cost. This ensures that our resilience mechanisms enhance, rather than impede, the speed and cost-effectiveness of AI training and inference.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

2.3 Impact

SDC profoundly harms AI model integrity, accuracy, and trustworthiness. In training, SDC can waste weeks or even months of valuable computational resources. During inference, it can lead to incorrect results with severe consequences, especially in critical applications like autonomous driving. By effectively mitigating SDC through rapid faulty hardware detection, we can significantly reduce downtime, accelerate AI development and deployment, and ensure new technologies are accessible and deliver positive industry impact.

2.4 Scale

Designing SDC solutions for massive scale is a central theme due to the high probability of faults in deployments with "hundreds or thousands of nodes". The "Needle in a Haystack" challenge describes the difficulty of detecting rare, individual errors at this scale. The whitepaper emphasizes the need for "efficient, scalable approaches" rather than costly brute-force methods. Strategies like "reductive triage" aim to contain and isolate issues within smaller cluster subsets, allowing the rest of the system to continue operation and thereby facilitating incremental scaling and minimizing overall disruption

2.5 Sustainability

Nothing can be sustainable if the fundamental tenet of computing is in jeopardy. So, any effort towards sustainability must overcome the SDC challenges. By ensuring there is sharing of best practice and open framework to advance the state of art in SDC detection and mitigation across a wide variety of hardware, SDC specification can help improve the reliability and sustainability for future data center infrastructure. Addressing SDC is critical for both operational carbon, by preventing energy waste from recomputation, and embodied carbon, by reducing hardware waste from defective units being pulled from service.

3.0 Background on Hardware Faults

While data corruption, in general, can stem from a variety of issues including software bugs or systematic hardware design errors, the focus of this exploration is on random (both transient and permanent) and systematic hardware faults that go undetected due to insufficient error-checking or because they initially manifest in ways that do not trigger conventional alarms. This section presents a taxonomy typically adopted in the hardware reliability community and provides an overview of two typical fault models encountered in modern hardware, i.e., permanent and transient, and how they may manifest as silent data corruption in the context of AI applications.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

3.1 Defining Silent Data Corruption

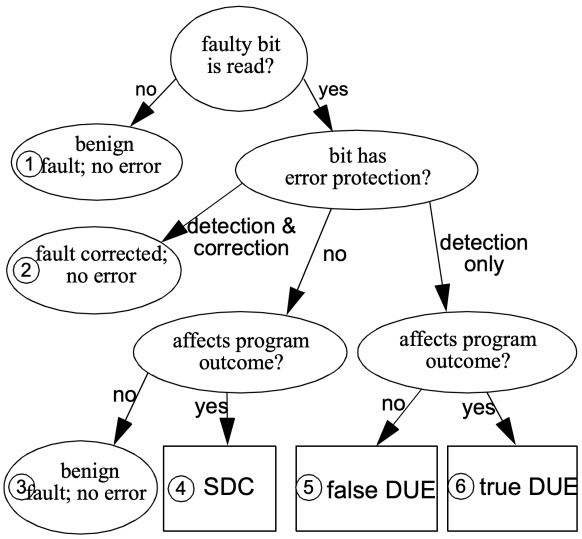


Figure 1: Classification of the possible outcomes of a faulty bit in a microprocessor. SDC = silent data corruption. DUE = detected unrecoverable error [1]

Based on terminology used in the hardware reliability community, it is helpful to define the possible outcomes of a hardware fault within a system. When a single-bit fault occurs, such as one induced by particle strikes from cosmic rays or alpha particles, there are several possible outcomes.

1. Benign fault; no error: The fault occurs, but it does not affect the program's outcome.
2. Fault corrected; no error: The fault is detected by error detection/correction mechanisms (like ECC), and the system corrects it without affecting the program's outcome.
3. Detected unrecoverable error (DUE): The fault is detected by error detection mechanisms (like parity) but cannot be corrected. This avoids generating incorrect outputs but often prevents recovery, resulting in fail-stop behavior. Errors in this category can be further subdivided into benign detected errors (false DUE) or those that would have affected the final outcome (true DUE).
4. Silent data corruption (SDC): The fault occurs and induces the system to generate erroneous outputs, but it does so without triggering hardware detection.

Essentially, SDC represents the most insidious form of error because it goes undetected by the hardware's built-in checking mechanisms and leads directly to incorrect program results. Designers often employ basic error detection like parity specifically to avoid SDC, accepting the trade-off of potential DUE events instead. Errors in hardware structures that can be handled by error detection or correction mechanisms are not treated as SDC events.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

3.2 Root Causes of SDC

3.2.1 Permanent Faults

Permanent faults are typically rooted in physical defects within the silicon and have been extensively studied in the hardware literature. These defects can be abstracted by a variety of fault models, including stuck-at faults, bridging faults, and delay faults, each capturing different manifestations of erroneous circuit behavior. Of these, the *stuck-at* fault model, where a signal is permanently held at logic '1' or '0', has been widely adopted in industry due to its simplicity and effectiveness in guiding test generation and coverage analysis.

However, as process geometries shrink and designs operate closer to timing and power limits, delay faults, particularly from *small-delay defects*, are emerging as a more relevant fault model. These are caused by permanent physical defects and process marginalities, but rather than producing consistent logical failures, they may introduce **intermittent faults**, i.e., faults that occur only under specific operating conditions such as voltage droop, temperature variation, or high switching activity. This makes them significantly harder to detect, as their activation may depend on context rather than an always-on failure.

As power density increases in each process generation, the ability to test logic for faults that vary in timing sensitivity over time of use continues to become more challenging. Small-delay defects may arise from manufacturing defects, process variation or progressive wear-out (e.g., electromigration). The net result is that a transistor or interconnect occasionally fails to switch within the expected timing window, potentially causing single-bit errors, or even multi-bit errors if the affected node or net fans out to multiple destinations. While small-delay defects are indeed permanent, they differ from stuck-at faults because they may not be consistently activated, and their impact often appears intermittent. Some defects degrade over time until their visibility increases, while others remain hidden unless triggered by events such as voltage droop or elevated temperature.

AI accelerators often implement aggressive voltage and frequency scaling to optimize power consumption and throughput. This can reduce safety margins, increasing the likelihood of small-delay defects going undetected. The confluence of wider parallel execution and higher clock frequencies results in increased current draw and noise in the power delivery network. The increased noise along with increased power density on processes with smaller fabrication detail makes the system more prone to activation of permanent timing susceptible faults. In the context of large-scale computing infrastructure, such as those used for training or inference, these errors may go undetected for extended periods, allowing them to accumulate or propagate in unpredictable ways [2].

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

3.2.2 Transient Faults

Transient or soft errors can be caused by cosmic ray-induced neutron strikes, alpha particles, or other environmental factors. These single-event upsets (SEUs) are typically non-destructive but can silently alter data values or logic states. The architectural vulnerability of various processor components to such faults has been extensively studied [3]. These studies show that arithmetic units generally exhibit low vulnerability to transient faults under typical workloads. More recent research [5] has shown that the opposite is true in the context of small-delay faults (SDF) as they apply to ALUs. ALUs tend to be most vulnerable to SDF compared to array structures.

AI workloads introduce new considerations. Although compute units such as ALUs and MACs often exhibit lower Architectural Vulnerability Factor (AVF) to transient faults in traditional CPUs, AI accelerators tend to dedicate a larger proportion of their area to these dense arithmetic units to maximize throughput for matrix-heavy operations. This architectural emphasis combined with aggressive performance optimizations, shift reliability concerns to surrounding structures like register files and intermediate buffers, which may become new points of vulnerability under fault conditions.

Additionally, many accelerators support limited-precision arithmetic (e.g., 16-bit or 8-bit representations) to improve performance and efficiency, and are known to be widely used in large-scale AI training [4]. While narrower data formats improve compute and memory efficiency, their effect on fault resilience varies depending on the specific fault model and workload characteristics. Some studies suggest that they may increase sensitivity to certain types of bit-level corruptions due to reduced numerical guard bands, while others indicate that reduced precision can limit the absolute magnitude of errors, potentially improving resilience under specific fault scenarios [12, 13]. Given the range of observations, the implications of reduced precision on fault tolerance remain an open area of investigation.

3.3 Hardware Fault Metrics and AI Correctness Metrics

A significant challenge in managing Silent Data Corruption (SDC) in AI lies in aligning hardware fault detection methodologies with the ways in which AI systems assess correctness. Hardware metrics focus on low-level failure modes, often in terms of raw bit flips or timing violations, whereas AI workloads measure success in terms of model accuracy, robustness, and convergence. These two layers of abstraction do not naturally align, making it difficult to quantify the real-world impact of hardware faults on AI outcomes. Furthermore, the inherent fault-masking behavior of AI systems, especially during training, can obscure the presence of SDCs, delaying or even preventing their detection. Understanding how hardware faults propagate through an AI stack to affect model behavior requires a holistic view that combines traditional architectural vulnerability modeling with AI workload-specific sensitivity analysis.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

This section outlines the key hardware-level reliability metrics used to characterize fault behavior, such as FIT, AVF, and DelayAVF, and contrasts them with the performance and correctness metrics commonly used in AI. It also discusses the challenges of mapping low-level hardware errors to their high-level impact on AI model behavior and introduces the need for cross-layer analysis to bridge this gap.

3.3.1 Hardware Reliability and Fault Metrics

This subsection describes a few of the metrics that HPC and hardware communities typically rely on to describe and quantify error and failure behavior.

3.3.1.1 Failures in Time (FIT)

FIT is a field reliability metric that quantifies the number of failures per billion (10^9) device-hours. It is widely used in industry to estimate the long-term failure rate of deployed silicon. For permanent failures, FIT can be derived from empirical data gathered through burn-in testing, qualification screening, and in-field return analysis. In the context of transient faults, such as those caused by cosmic rays or alpha particles, FIT is typically estimated through accelerated radiation testing (e.g., neutron beam experiments) or modeled using environmental exposure data combined with device-level sensitivity parameters (e.g., critical charge, cross-sectional area).

The use of FIT to describe failure rate is applicable to a broad range of failure mechanisms, including both wear-out mechanisms (e.g., electromigration, bias temperature instability, hot carrier injection) and transient error sources. However, it does not account for the architectural masking or application-level impact of faults. That is, FIT quantifies the rate at which raw errors are expected to occur in hardware, but it does not indicate how many of those errors will result in silent data corruption observable at the software level. Bridging this gap requires complementary metrics such as AVF.

3.3.1.2 Architectural Vulnerability Factor (AVF)

AVF is a design-time metric used to estimate the probability that a fault in a particular hardware structure (e.g., register file, cache, pipeline stage) will propagate and produce a user-visible error at the application level [1, 3]. AVF is workload-dependent. It reflects the fraction of time a hardware element contains data essential for architectural correctness. This makes it a useful guide for architects to identify which components are most critical to protect through error detection or correction mechanisms.

AVF was originally developed in the context of transient faults, such as soft errors caused by particle strikes, where the fault itself is momentary but its impact can propagate. However, traditional AVF does not account for faults that manifest through timing violations, particularly those caused by small-delay defects. These faults arise from permanent defects but produce timing-related errors and may propagate in ways that are different from the errors produced by

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

transient faults, which are often abstracted as single bit flips. This gap motivated the development of DelayAVF, which extends the AVF concept to address vulnerabilities arising from subtle, delay-based failures.

3.3.1.3 DelayAVF

DelayAVF [5] is a design-time metric that quantifies the vulnerability of microarchitectural structures to small-delay faults (SDFs), which are a class of permanent defects that manifest as subtle, timing-induced errors under marginal conditions. Unlike traditional AVF, which models the likelihood that a transient bit-flip leads to a program-visible failure, DelayAVF incorporates both circuit-level timing characteristics and dynamic masking effects (i.e., does not affect program outcome) to capture whether a delay-induced fault results in an incorrect value being latched and ultimately observed at the application level. It enables architects to identify which structures are most at risk from SDFs and to guide mitigation efforts accordingly.

The metrics described in this section (3.3.1.3) focus on bitwise correctness and the likelihood of hardware faults leading to program-visible errors, but they do not account for the intrinsic resiliency of AI models or the fault-tolerant nature of the training process, which can vary significantly depending on the architecture, data, and optimization dynamics. As a result, not all hardware-level errors may manifest as degraded AI performance. To fully understand the implications of SDCs in AI systems, we must also examine the correctness metrics used within AI, which operate at a higher level of abstraction and reflect how such faults may (or may not) affect learning outcomes and model predictions.

3.3.2 AI Correctness Metrics

AI workloads can be broadly divided into training and inference, each with different sensitivity profiles and correctness metrics. During training, faults can silently corrupt weight updates, gradients, or intermediate activations, potentially leading to model divergence or instability. These effects may not be immediately obvious from standard performance metrics but can degrade the model's long-term learning trajectory or robustness. Metrics commonly used to monitor correctness during training include cross-entropy loss, mean squared error, and Kullback-Leibler (KL) divergence, which provide a more continuous signal of model behavior than discrete classification metrics.

In contrast, inference, where a trained model is deployed for prediction, faces different reliability risks. Here, even minor silent data corruptions can result in biased or incorrect outputs [9], which may have especially serious consequences in safety-critical or high-stakes domains like autonomous driving or medical diagnostics. Evaluation during inference or validation is typically based on discrete performance metrics, such as accuracy, precision/recall, F1-score (for classification), BLEU score (for translation), or perplexity (for language modeling). Subtle faults might not significantly shift these aggregate metrics in the short term but can disproportionately

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

affect edge cases or underrepresented inputs, which may often be the most sensitive in real-world deployments.

4.0 Background on AI

4.1 Characteristics of AI Workloads

AI workloads generally fall into categories like supervised learning, reinforcement learning, and generative AI, and are deployed across a wide range of use cases from computer vision and language modeling to recommendation systems and robotics. Regardless of domain, AI models are typically composed of layers of numerical computations, often implemented using frameworks like TensorFlow or PyTorch, which abstract away much of the underlying hardware complexity.

One defining feature of AI training workloads is their iterative nature. Models are trained over many epochs, with parameters updated gradually based on incoming data. This iterative feedback loop, combined with stochastic elements like random initialization and data shuffling, makes AI training inherently non-deterministic. While this randomness enables generalization and robustness in the learning process, it also complicates efforts to isolate and attribute the effects of underlying hardware faults.

Inference workloads, in contrast, are generally more deterministic, latency-sensitive, and focused on delivering predictions at scale. While they avoid some of the long-running complexity of training, inference systems operate under tight correctness and response-time constraints, making them susceptible to SDCs due to faults in the model parameters and/or computation.

4.2 Hardware Usage Patterns in AI

Modern AI workloads rely heavily on dense numerical operations, especially matrix multiplication and fused multiply-accumulate (MAC) operations. To accelerate these operations, AI systems increasingly use specialized hardware such as GPUs, TPUs and dedicated ML accelerators that expose high degrees of parallelism and throughput. These devices often operate near the limits of power and thermal envelopes, leading to higher sensitivity to voltage droop and temperature variation.

A hallmark of AI hardware is the widespread use of limited-precision arithmetic, such as FP16 or INT8, which reduces memory footprint and improves compute efficiency. These data formats allow for faster training and inference. The importance of these limited-precision formats have led to standardization efforts, such as the OCP microscaled data formats [14]. However, their narrower representational range can influence how faults are manifested and observed.

Distributed training workloads introduce additional reliability complexity through tight synchronization primitives like all-reduce operations, which are used to share gradients across

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

devices. Faults in a single node can therefore propagate across an entire cluster if not caught sufficiently early. Furthermore, because AI models are often trained on custom kernels or fused operators, the underlying instruction and data access patterns can be opaque to traditional error checking mechanisms.

4.3 AI Software Stacks and Execution Models

The AI execution pipeline spans multiple abstraction layers–from high-level model code written in Python, to ML frameworks, to compiler toolchains, and finally down to hardware drivers and runtime execution. Each layer introduces opportunities for optimization, such as operator fusion, graph transformations and quantization, which can alter the runtime behavior of models at the higher levels in ways that complicate debugging and fault attribution.

Because of this layered execution model, faults introduced at the hardware level may not become visible until they propagate through several transformations. For example, a low-level numerical error in a fused kernel may manifest as a divergence in training loss only after many iterations, or result in subtle shifts in inference outputs that evade standard correctness checks.

As AI frameworks continue to optimize for performance, the opportunities for silent faults to go undetected increase. To mitigate this risk, it becomes important to treat interfaces between layers of the software stack as detection boundaries. If faults can be detected when they cross these boundaries, such as from kernel-level computation to framework-level tensor operations, the system can catch silent errors earlier and reduce the chances of them silently corrupting downstream model behavior. This highlights the need for SDC-aware tools and test strategies at each layer of the software stack, as well as interface-level checks that provide fault containment across abstraction boundaries.

5.0 Best Practice Mitigations

Mitigating Silent Data Corruption (SDC) requires a multi-layered strategy that includes both *detection* and *recovery* techniques. These strategies often draw from practices established in high-performance computing (HPC) and large-scale cloud systems. Conceptually, resilience mechanisms fall into three broad categories based on the type of redundancy they employ.

- Spatial redundancy: Replicating hardware or computation to cross-check results.
- Temporal redundancy: Repeating operations over time to detect transient anomalies.
- Informational redundancy: Embedding additional information (e.g., checksums, codes) into the data stream and hardware structures for fault detection and/or correction.

These mechanisms may be applied for either error detection or recovery. Furthermore, recovery itself can take two forms.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

- Backward error recovery, where the system rolls back to a previously known good state (e.g., checkpointing.)
- Forward error recovery, where the system detects and corrects the error in place and continues execution.

This section reviews common techniques through that lens and considers their applicability to AI workloads.

Silent data corruptions pose an incredibly difficult set of challenges across hyperscale infrastructure. The corruptions are inherently data dependent, and as a result have an exponentially large test space, as it is impractical to test all the possible data values for a computational path. In addition, these faults are dependent on the voltage and frequency of the device operation as well as the operating temperature. It has been observed that device life cycle is also a meaningful contributor. In a specific case study, where Meta ran the same computational checks on a device, it failed only after six months, pointing to wear out with usage for the previously computationally correct set of values [15, 18, 22].

As a result, at scale, these computational inaccuracies have to be tested for with consistent, periodic, and frequent testing within a random state space of operation throughout the life cycle of the device.

5.1 Novel Detection Mechanisms

A variety of detection mechanisms to protect applications from the harmful effects of silent data corruption have been published in the past [11, 23-26].

The first approach implemented at fleet scale is periodic screening with methods like Fleetscanner [16]. This is a method where an infrastructure consistently takes pit-stops to run a set of directed tests, with test durations in the order of minutes. Infrastructure typically has a set of maintenance operations such as firmware and kernel upgrades, and hardware repairs. These methods utilize these states to reserve and execute hardware tests at scale. These are scheduled periodically to ensure coverage for the entire fleet within a period of 45 to 60 days.

While this approach allows for dedicated testing on a set of hosts, for some types of SDCs waiting 45 to 60 days for coverage can be detrimental. As a result, alternative approaches [16] colocate with the workload and execute tests in the orders of milliseconds to seconds. This approach can scale to the entire fleet within a matter of days. The tests ripple through the infrastructure based on the scheduler executing the tests.

Both methods can offer 70% overlapping coverage [16], with the remaining coverage being unique to each test mechanism. While in-production testing allows for heterogeneous instructions to overlap test instructions across cores and threads, in out-of-production testing the entire device is executing test instructions.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

A third and novel approach is analytical [17]. This is a test-and-architecture-agnostic approach that evaluates applications across scale and their exception behavior within the kernel space. If many sets of applications happen to encounter frequent exceptions on specific fleet cores, the methodology triages core-based anomalies as silent data corruption, since applications are failing with no hardware-failure signature. This methodology does not require testing allocation or reservations and thus can operate outside of the application plane and solely within the analytical plane. It has been observed that this approach outperforms testing-based approaches across architectures, applications, and data-centers by 41% [17].

The methodologies described above execute across hyperscaler fleets, detecting SDCs across AI and non-AI infrastructure. However, AI applications such as training and inference have unique and more challenging implications for SDCs.

5.1.1 SDCs in Training Workloads

Silent Data Corruptions in training workloads have drastic implications for training efficacy [10, 20]. SDCs fundamentally result in incorrect computations, which are consumed by a training iteration. This incorrect consumption of the corrupted iteration in forward and backward passes (in any particular pass, in any accelerator, in any node) creates a divergence in the training path that the iteration, as well as the workload, would have taken instead. It has been asserted that AI training workloads are self-resilient amidst silent data corruptions. However, this assertion holds true in a fairly small subset of the possible SDC manifestations. For example, self-resilience is insufficient in most of the realistic training use cases and SDC properties. SDCs tend to be persistent throughout many iterations, and AI training workloads pursue quantization of data values representing a higher amount of information per bit, both of which exacerbate the implication of an SDC in a training iteration, and continuously increase the divergence rates of training workloads.

In the subsequent subsections, we present the two most common cases of training divergence due to SDCs.

5.1.1.1 NaN (Not-a-Number) Propagation

In this first scenario, we explore NaN propagation. During the computational passes in training iterations, a majority of the computations are conducted with fixed-point or floating-point numerics. These numeric representations are typically range-bound on the minimum and maximum value representations based on bit-widths. NaN is the representation of numbers that do not have a valid representation in the specified numeric scheme. As and when an SDC occurs during a computation, the computational error pushes a representable value into an incorrect representation, thereby generating a NaN.

Furthermore, once a NaN is generated in any part of the training iteration, subsequent computations involving this NaN also result in additional NaNs, thus propagating the NaNs first

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

to the training iteration in flight, then to the accelerator domain, then subsequently to the host domain consisting of many accelerators, and eventually to the cluster domain, which consists of thousands of nodes and many thousands of accelerators. During this process, since the source of the NaNs is due to a few specific computations on a single accelerator, which may or may not be individually traceable amidst cluster scale, the cluster has a NaN contagion, but the source of the contagion is not traceable. Thus, with SDCs in AI training, the NaN propagations eventually lead to a cluster halt and debug until the offending accelerator and offending nodes are correctly identified and quarantined.

5.1.1.2 Corrupted Gradient Variance

A second and more complex scenario is that of a corrupted gradient variance. Gradient variance refers to a gradient explosion or gradient implosion or a local minimum because of silent data corruptions. Within the floating- and fixed-point ranges, a corruption can manifest within the numeric-representation bounds. This SDC, while representable, is consumed by the training iteration and eventually by the entire cluster as the correct outcome of the training step. However, in synchronous training, a corrupting device with a persistent SDC will continue to corrupt the iteration and have a cascaded impact on the training values across all accelerators within the cluster. The synchronization results in the exchange of corrupted values as true values throughout the cluster.

As a result, the training will continue to execute but is not guaranteed to make any forward progress, as intended by the current state of the hyperparameters. In simple terms, this results in an illusion of forward progress and experimentation, until the SDCs aggregate and result in major divergences to gradients. Depending on the scale, scope, layer, and consumption of the corruption, the gradient descent algorithm is stuck at a local minimum, or the gradients are exploding or imploding beyond the expected ranges.

The observability of these SDCs is also dependent on time and computational intensity, and based on the range of the corruptions from true value, identifying the existence of a corruption can take weeks or even months. Similar to the previous case, the source of the corruption is not traceable, yet this scenario is an order of magnitude harder to rectify, since trapping on NaNs will be insufficient to detect corruption. So a true SDC can result in many hours or weeks of cluster computational-resource and training iterations rendered unproductive. In addition, the existence of these corruptions without any detection makes the root-cause and subsequent training iterations risky, until the offender is quarantined.

5.1.2 SDCs in Inference Workloads

In inference applications, SDCs result in incorrect end results for the application. Inference applications typically execute hundreds of thousands of inferences at scale in any given hour, and a corrupted device results in corruptions across thousands of inference consumers. Due to the persistent nature of defects, these incorrect inferences can have direct consequences to

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

decisions associated with inference, such as a recommendation system or LLM output. Inference corruptions lead to second-order effects where policies associated with privacy or integrity potentially can be bypassed, since the corruption knows no bounds. Corruptions in inference fundamentally reduce the efficacy of models that have been trained with large computational resources. As a result, inference use cases asserted to be benign are anything but benign at scale.

5.1.3 Impact of SDCs

The presence of SDCs in training and inference clusters results in a cluster-level debugging scenario across thousands of components in the entire cluster.

In training, while visible faults halt the cluster and require attention to the fault source, SDCs result in an illusion of forward progress while hiding that source. In case of NaN propagation, until the NaN generating node is identified, a training restart from a previous checkpoint without the elimination of the offending accelerator leads to subsequent training runs that eventually generate NaNs. Similarly, in the case of corrupted gradient variance, the illusion of training progress persists longer until the eventual aggregation of the variances repeatedly manifest in subsequent training iterations that render ineffective all of the restarts from past known good checkpoints. As a result, SDCs can result in significant computational inefficiency at cluster scale having a larger temporal impact than all of the visible faults.

In inference, the triage process leads to computationally expensive telemetry enablement at each of the sub-stages of the inference. And until the offending node and the offending devices are identified, the inference clusters cannot be allocated for inference, due to the possibility of a repeat corruption. Specifically, in inference, large deviations are easier to detect than smaller deviations, as anomaly detectors can be set up to limit the expected ranges of inference. Inference-debug triage requires hundreds of engineers across teams to identify the root cause while the production use cases are halted, directly impacting available reliable capacity for serving production.

5.1.4 Detection of SDCs in AI hardware

In this section, we'll provide an overview of mitigation strategies to deal with SDCs in AI training workloads. We classify them into infrastructure strategies and stack strategies. Infrastructure strategies are applicable in operational triage at the cluster level, while stack strategies require coordination with the workload.

5.1.4.1 Infrastructure strategies

Reductive triage: In the case of NaN propagation, a naive approach is to conduct a binary search with mini-training iterations across continuously decreasing cluster sizes. Since the corruption is due to a NaN propagation, the intent with this approach is to arrive at a small enough cluster that replicates a NaN propagation. Upon this replication, the smaller cluster, or

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

the offending node, is quarantined for deeper investigation, while a reconstituted cluster with new nodes executes training from a previously saved checkpoint. However, this approach assumes a strong ability to reproduce the SDCs. Since SDCs are data-dependent, electrical and temperature variation dependent, the reproducibility is not guaranteed. Even for corrupted gradient variance scenarios, a similar divide-and-triage approach can work; however, the gradient variances and the intentions fundamentally vary with the training input data, as does the size of the cluster, in spite of consistent hyperparameter configurations across all nodes.

Deterministic training: A known effective model can be executed for a few training iterations to validate the absence of NaNs and divergence of gradients. This approach can be used to verify computational failures that are not dependent, since this approach is fundamentally providing a guarantee for a single set of values and training inputs.

Taking Google's Gemini model training as an example, training AI models at an unprecedented scale often uncovers novel system failure modes. One of the problems that needed to be addressed during the development of Google's Gemini models was SDC. While extremely rare, the sheer scale of Gemini's training means SDC events can be anticipated to impact training every one to two weeks. Rapidly detecting and removing faulty hardware required several new techniques that exploit deterministic replay to isolate incorrect computations, combined with proactive SDC scanners on idle machines and hot standbys. A fully deterministic infrastructure proved crucial during the development of the Ultra model, allowing for quick identification of root causes, including hardware failures, and ultimately contributing to stable training for Gemini [10].

Hyper checkpointing: An alternative method is to checkpoint at increasingly high frequency to allow for accelerated triage of the offending corrupting node, and also allow for lower throughput reduction for the training. In cases of NaN propagation, this method can help to contain the NaN to an accelerator or a host, thereby significantly accelerating the triage and quarantine process.

5.1.4.2 Stack Strategies

Gradient clipping: A conclusive way to mitigate the NaN propagation scenario for SDCs is to algorithmically enforce gradient clipping as part of the training workload expectations, and limit the training within a certain value range. Any computation that exceeds the range is clipped, and an example computation generating a NaN can also be detected at the clipping step with a NaN detection that's set to a max-value of a min-value based on the operand sign. However, while this potentially works for a subset of NaNs based on representation format, it can sometimes introduce partial error.

Algorithmic fault tolerance: A more robust approach is to enable training algorithms with built-in fault tolerance for a range of data corruptions. This would not only reduce the operational burden of detection and triage, but also, due to lower overheads, allow for larger computational efficiency throughout the training workload. Previously, this approach has been shown to be successful in training on CPUs with minimal overheads [21]. It requires understanding the most

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

commonly observed defect modes occurring with engineering investments across the stack, and modified guarantees to the training workloads, at an overhead to the overall training footprint.

Tri-variate computational training architecture: A key live approach to mitigate SDCs is with shadow nodes in synchronous training architectures, where training steps are repeated across a different set of nodes at random iteration counts. This ensures that the training progress happens as intended after a verification step, and it helps in cases associated with NaN propagation as well as corrupted gradient variance. In both cases, if the shadow nodes and the live nodes produce different results, the training is immediately halted, and only the shadow and live nodes conducting specific training steps are investigated, while the rest of the training continues with new node additions. A tri-variate computational flow performs this operation with multiple shadow node pools, a randomly chosen training node pool of the same size, and a specified number of training steps with the same checkpoint origin. This approach provides a robust training-infrastructure, forward-progress methodology but requires significant algorithmic changes to the cluster trainer and larger data movement and infrastructure overheads.

Parameter vulnerability factors: A novel approach for machine-learning architectures is to identify which layers are more vulnerable to corruption and which are resilient. This enables the end user to map specific vulnerable layers to more resilient hardware while allowing the resilient layers to map to unprotected hardware. This is tricky, though, as it has to be evaluated dynamically at scale in conjunction with the evolution of the architectures. In most cases associated with hardware design, resilience is available at a cost to the area or power or performance, hence approaches such as PVF allow for targeted resilient design especially for inference use cases [9].

Divergence Detection: An in-line inference corruption detection mechanism is to consistently maintain a distribution map for each neuron and conduct divergence detection from the typical distribution observed for neuron outputs. This approach, while expensive, can be executed at chosen sampling rates for inference applications at scale. Overall, in preserving the DNA of each neuron's behavior for a specific workload family, a divergence can lead to detection of inference corruptions while the workload is under execution [19]

All the above methodologies for mitigation provide varying levels of resilience with different operating points as well as different engineering and infrastructure overheads. Based on the scale and the intensity of the training and inference workloads, employing a combination of these strategies can help mitigate the adverse effects of SDCs at scale in AI applications.

5.2 Error Detection

5.2.1 End-to-End Integrity Checks (Informational Redundancy)

Checksums or hashes can be stored and validated at multiple stages of a pipeline to detect unexpected changes in data or intermediate states. This technique, commonly used in

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

distributed data systems and cloud-scale infrastructure, provides low-cost error detection but requires follow-on mechanisms for correction or rollback. It is especially relevant in large-scale AI pipelines involving multi-node communication or storage layers.

5.2.2 Replication and Consistency Checks (Spatial Redundancy)

Some cloud systems employ redundancy, i.e., running the same service or computation on multiple replicas and comparing outputs to detect divergences. In AI, this could involve redundantly executing critical layers of a model in parallel across different nodes or using shadow models to cross-check outputs. While effective for detection, it introduces overhead and may be prohibitive at scale.

5.2.3 Integer Residue Checking

Integer residue codes exploit properties of modular arithmetic to detect errors in arithmetic operations, particularly over integers or quantized data. By maintaining parallel computations modulo small primes (e.g., mod 3, mod 7), systems can validate whether computed results satisfy residue constraints. This technique is computationally inexpensive and can detect a variety of common fault patterns, such as bit flips or stuck-at faults, especially in arithmetic-heavy pipelines.

5.3 Proactive Testing and Screening

The hardware industry best-practice to catch latent defects has been through testing. Silicon designers typically use both structural testing (e.g., via Automated Test Pattern Generation) and functional testing.

Structural testing uses specific fault models, such as the stuck-at model or a path-delay fault model. Structural testing can provide high coverage to catch defects that trigger such faults. However, structural testing can involve high costs for large designs, require specialized automated test equipment, and are typically carried out under controlled conditions that may not represent the mission-mode usage of the part where intermittent faults may manifest. Functional testing, on the other hand, involves running program code that can be run under mission-mode conditions. However, crafting high-coverage functional tests is challenging and is often a manual process involving expert engineers. Recent research has shown that it may be possible to generate high-coverage functional tests in an automated manner [28] [29]. For a more detailed discussion on structural vs. functional testing in the context of SDC at scale, the reader is referred to [30].

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

An added benefit of functional testing is that it also allows testing hardware after it has been deployed in the data center, as part of acceptance testing or periodic fleet-level testing. Large cloud operators, for instance, perform periodic fleet-wide diagnostics [15] [16]. In these screening protocols, servers or accelerators are taken offline briefly to run stress tests (for example, linear algebra routines, memory checks, etc.). Such proactive testing can detect a portion of latent errors before they wreak havoc in a long training job. The downside is that exhaustive testing of an entire fleet might take weeks and some issues only appear under specific data or timing conditions that tests may not cover.

5.4 Error Recovery

5.4.1 Checkpoint/Restart (Backward Error Recovery, Temporal Redundancy)

Originally developed for HPC simulations, periodic checkpointing allows the system to roll back to a previously correct state after detecting an error. In AI training, saving model states only at the end of an epoch may be insufficient. More frequent checkpointing, for example, per mini-batch or per iteration can limit loss when errors are eventually discovered. However, this increases storage and I/O overhead and can reduce *runtime goodput* [27], which reflects the fraction of allocated compute time spent making useful forward progress. In large-scale ML systems, excessive checkpointing can introduce significant orchestration overheads and lower system throughput if not carefully optimized.

5.4.2 Algorithm-Based Fault Tolerance (ABFT) (Forward Error Recovery, Informational Redundancy)

ABFT introduces checksums or encoded forms directly into the algorithm (e.g., matrix operations in linear algebra) to detect and sometimes correct faults without needing to revert to a prior state. These techniques can be adapted to neural network training, especially in matrix multiplication or attention layers, and offer a promising path to resilience with modest overhead if tailored carefully.

5.5 Implications for AI

Although HPC and cloud reliability solutions provide a strong foundation, adapting them to AI workloads presents unique challenges. The most significant of these is *overhead*. Frequent checkpointing, a common fault-tolerance strategy in HPC, becomes increasingly expensive at scale. For example, in exascale systems, the time required to perform checkpoints can approach the Mean Time To Failure (MTTF) of the system itself. In such cases, the system risks spending most of its runtime checkpointing and restarting, making little forward computational progress [7]. A similar situation could arise in large-scale AI training, where the overhead is not only due to storage demands but also to synchronization delays across distributed nodes.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Likewise, full replication, while widely used by cloud providers to ensure high availability, can be cost-prohibitive in AI contexts, where models are extremely large and training consumes substantial compute and memory resources.

Another challenge specific to AI workloads is the inherent non-determinism in model execution. Factors such as random weight initialization, data shuffling, and asynchronous updates introduce variability that can mask or confound the detection of fault-induced deviations. This same non-determinism also complicates resilience *analysis* i.e., the impact of a hardware fault (whether measured by model accuracy, loss, or another metric) can vary significantly depending on the specific values of random seeds, batch orderings, or hyperparameters. As a result, reproducing faults and drawing consistent conclusions about fault severity or coverage becomes difficult. While re-architecting algorithms or frameworks to enforce determinism may improve fault detectability, it may incur significant performance tradeoffs and limit scalability. How to effectively account for this stochastic behavior and high-dimensional parameter space remains an open research question in developing actionable and generalizable insights about AI system resilience.

Moreover, AI workflows are fundamentally iterative: model parameters including weights, biases, and other internal states are updated continuously over many training steps. This introduces complexity in reasoning about fault impact. For instance, an error injected into a frequently updated parameter early in training might be overwritten quickly and have no observable effect, while a similar error introduced into a critical layer late in training could cause divergence or degrade accuracy in subtle ways. This variability makes it difficult to generalize fault-tolerance strategies. As a result, traditional one-size-fits-all techniques from HPC or transactional systems may either overprotect in low-risk regions or fail to catch impactful corruptions, depending on the AI workload, model architecture, and training dynamics.

In practice, a layered approach may offer the best tradeoff: combining lightweight error detection (e.g., checksums or parity checks on critical data paths), selective checkpointing, and targeted redundancy for the most vulnerable components or stages of training. Such a strategy may help mitigate error risks without imposing undue resource demands.

Finally, emerging research on algorithm-based fault tolerance (ABFT,) especially when integrated directly into core AI kernels like matrix multiplications, offers promising pathways to forward error recovery that align with AI's compute patterns. These methods can provide error resilience at low overhead by detecting and correcting errors in-flight, reducing the need for rollback and enabling fault-aware training strategies.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

6.0 Open Research Questions

6.1 Defining the Core Challenge

How can the AI community ensure that subtle hardware errors manifesting as SDC do not undermine the integrity, accuracy, and trustworthiness of models deployed at scale, given the unique workload characteristics of AI? Additionally, how can discovered SDC events, which may remain benign to some AI models, be leveraged by data center operators to identify and isolate faulty components, thereby preserving overall cluster health?

Several key obstacles need to be overcome in tackling this challenge.

- Complexity of AI models: Large neural networks may partially mask faults, making discovery and localization difficult.
- Stochastic training: Randomness in weight initialization and batch sampling can either hide or exacerbate certain errors.
- Operational scale: Commercial deployments often involve hundreds or thousands of nodes, elevating both the probability of SDC and the difficulty of detection.
- Attribution Challenges: Distinguishing hardware-induced corruptions (e.g., NaNs, loss spikes, model divergence) from other sources, such as poor training data, software bugs, or inherently unstable model configurations, can be extremely difficult.

6.2 Gap Between Hardware Fault Metrics and AI Correctness

A crucial open question is whether hardware faults have a direct, consistent, and measurable impact on AI model behavior. Hardware-level metrics such as AVF and FIT are designed to estimate the probability and rate of bit-level or timing-related faults, while AI correctness metrics, such as accuracy, BLEU score, or loss, summarize model performance at an aggregate level. These two layers of abstraction operate independently, and how faults in low-level hardware manifest as degradations in high-level AI correctness remains poorly understood. Bridging this gap requires both deeper empirical studies and the development of new or adapted metrics that can capture how Silent Data Corruption (SDC) affects the fidelity of AI models during both training and inference.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

6.2.1 Partial Tolerance to Errors

Neural networks are not explicitly designed to be fault-tolerant, but they often exhibit resilience to small perturbations due to their statistical and redundant nature. For example, many architectures may tolerate small variations in weights or activations without meaningful loss of performance, especially in overparameterized models. However, this resilience is incidental, not by design or guaranteed. If corruption occurs in parts of the model not expected to carry noise (e.g., weight matrices, backward-pass updates), it can result in unbounded degradation.

Several research efforts have proposed metrics such as “loss error” (i.e., corrupted_loss – correct_loss) to quantify how a perturbation impacts model outputs in the forward pass. While useful for gauging the impact of faults during inference or evaluation, these measures often overlook how corruption propagates during training, where errors in gradients or parameter updates may accumulate over many iterations. A comprehensive assessment must capture both forward and backward-pass sensitivities over time, particularly in large models trained across multiple nodes.

6.2.2 Layer-Specific Sensitivity

The impact of faults is not uniform across all model layers or architectures. Some components, such as normalization layers or attention heads, may be more tolerant of perturbations due to averaging effects or redundancy, while others, such as output (logit) layers in language models, are highly sensitive, with even minor corruptions potentially causing large output deviations [6].

Additionally, model architecture plays a role in fault sensitivity. For example, dense models (e.g., LLaMA) involve computation across all parameters at each step, making them potentially more vulnerable to uniformly distributed faults. In contrast, Mixture-of-Experts (MoE) models (e.g., DeepSeek, Mistral) activate only a subset of parameters per input, potentially localizing the impact of faults; but also raising questions about how expert routing decisions themselves tolerate or amplify faults. Understanding the fault sensitivity of these architectures is an emerging area for empirical and theoretical research.

6.2.3 Model Mapping and Accelerator-Aware Fault Effects

The way a deep neural network maps onto hardware strongly influences how faults arise and propagate. While some effects of SDCs could be generalized, a model’s fault sensitivity likely depends on the underlying accelerator architecture. Systolic arrays, for example, process data in tightly structured patterns that can amplify faults in shared elements. GPUs, in contrast, use wide SIMD execution, where faults may be masked by redundancy or averaging across threads.

Compiler-level choices also affect how faults are expressed. Operator fusion, layer scheduling, and memory layout optimizations can change which parts of a model are exposed to hardware

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

vulnerabilities. Sensitive operations like softmax or batch normalization may be assigned to different hardware units depending on platform capabilities, creating variation in fault behavior even for the same model.

Some models can handle small errors because of built-in redundancy, but this is not always reliable. The design of the hardware, such as how memory is used or how data flows through the system, can affect how faults show up and how much they matter. A key open question is whether we can find a general way to describe how faults affect models, no matter what type of hardware they run on, or if we always need to tailor our understanding to each accelerator's design.

6.2.4 Fault-Injection Studies & Standardized Benchmarks

Fault injection remains one of the most effective tools for characterizing how faults propagate and affect AI workloads [2]. Targeted experiments, e.g., flipping bits in specific layers or injecting delays during weight updates, can reveal which parameters, layers, or computation phases are most vulnerable. To support reproducibility and cross-comparison, the community would benefit from standardized resilience benchmarks and datasets, similar in spirit to how MLPerf standardized performance measurement.

While MLPerf [8] has advanced the benchmarking of AI system performance, it does not explicitly measure model robustness under fault conditions. As such, there is a need for complementary frameworks that quantify resilience, evaluate SDC impact, and guide fault-tolerance design tradeoffs. These benchmarks would need to be repeatable, scalable, and representative of real-world deployment settings.

A key enabler of meaningful benchmarking is determinism, i.e., ensuring that the same model, data, and training setup produce repeatable outcomes. Deterministic runs allow researchers to isolate and attribute observed deviations to fault injection rather than randomness. This typically involves fixing random seeds, controlling thread scheduling, and eliminating non-deterministic operations. However, this approach diverges from real-world AI workflows, which are often non-deterministic due to performance optimizations, distributed training dynamics, or hardware variability.

This mismatch presents a grand challenge for AI resilience research: how can we develop benchmarking and analysis methodologies that yield actionable insights despite non-determinism? While constrained, deterministic environments are useful for controlled evaluation, future work must also account for stochasticity in practice, potentially by quantifying the robustness of findings across random seeds, hyperparameters, and system configurations. Tackling this challenge will require a new generation of tools and methodologies, and presents a rich opportunity for research.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

6.3 The “Needle in a Haystack” Challenge

Despite being relatively rare on a per-node basis, SDC events can become significant when AI applications run at massive scale (tens or even hundreds of thousands of nodes). When an individual node experiences a transient or timing-induced fault, that single error might go unnoticed if it does not immediately compromise the output. However, given the large number of nodes operating continuously, the probability of encountering at least one such corruption in any large-scale run becomes substantial. Detecting or discovering these one-off errors, akin to finding a needle in a haystack, poses the following challenges.

6.3.1 Detection of Rare Events is Expensive

Because these events are relatively rare, the challenge is to design techniques that balance cost and benefit, i.e., offering high coverage with minimal overhead. Brute-force approaches like replication or frequent checkpointing across thousands of nodes can be prohibitively expensive in storage, time, and other resources.

6.3.2 Delayed or Non-Observable Impact

It is challenging to precisely define the point at which an SDC-attributable deviation of model behavior should be treated as anomalous. For example, during model training, subtle corruptions may not cause immediate failures; instead, they can gradually degrade model fidelity or introduce bias. This delay complicates discovery since by the time an issue becomes apparent, rolling back may require retraining over many epochs or verifying huge volumes of data.

6.3.3 Scalable Mitigation Strategies

Many known fault-tolerance techniques (e.g., triple modular redundancy or continuous replication) become prohibitively expensive at extreme scales. Efficient, scalable approaches such as lightweight anomaly detection at the node level, selective checkpointing, or adaptive error correction require further research and rigorous validation.

7.0 Conclusion and Future Direction

This exploration of Silent Data Corruption (SDC) underscores the growing urgency of addressing reliability issues typically found in traditional HPC contexts, particularly as AI workloads proliferate at unprecedented scale in mainstream computing. The convergence of large-scale AI workloads and HPC-like hardware infrastructure amplifies the impact of hardware-level faults, such as small-delay defects and transient errors, on model accuracy, data integrity, and trustworthiness.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

To address the SDC challenge, the Open Compute Project as well as many of its member companies have collaborated on or sponsored research to advance the state of the art. This research includes work on proactive testing and screening [28] [35] [36]; advances in modeling, measuring, and quantifying the rates of SDC-causing faults in silicon [5] [30] [31] [32] [33] [37] [38]; and runtime error detection [34].

But more research is required. For instance, we have seen that bridging hardware reliability metrics and AI performance metrics remains an unresolved challenge, highlighting the need for novel approaches that capture the nuanced ways in which SDC events can emerge, propagate, and remain undetected in both training and inference scenarios.

By integrating best practices from HPC and cloud computing with AI-specific considerations, researchers and practitioners may collaboratively develop more robust detection, diagnostic, and mitigation methods. This, in turn, will enable AI systems to operate reliably at unprecedented scale while preserving the performance gains that have made AI such a transformative force in modern computing.

8.0 References

[1] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee and R. Rangan, "Computing architectural vulnerability factors for address-based structures," 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 2005, pp. 532-543, doi: 10.1109/ISCA.2005.18.

[2] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023. “Understanding and Mitigating Hardware Failures in Deep Learning Training Systems,” In Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 70, 1–16. https://doi.org/10.1145/3579371.3589105

[3] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, San Diego, CA, USA, 2003, pp. 29-40, doi: 10.1109/MICRO.2003.1253181.

[4] Liu, Aixin, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao et al. "Deepseek-v3 technical report." *arXiv preprint arXiv:2412.19437* (2024).

[5] P. W. Deutsch, V. Q. Ulitzsch, S. Gurumurthi, V. Sridharan, J. S. Emer and M. Yan, "DelayAVF: Calculating Architectural Vulnerability Factors for Delay Faults," 2024 57th

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

IEEE/ACM International Symposium on Microarchitecture (MICRO), Austin, TX, USA, 2024, pp. 231-245, doi: 10.1109/MICRO61859.2024.00026.

[6] Yu, Mengxia, De Wang, Qi Shan, and Alvin Wan. "The super weight in large language models." *arXiv preprint arXiv:2411.07191* (2024).

[7] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. “Toward Exascale Resilience: 2014 Update.” Supercomput. Front. Innov.: Int. J. 1, 1 (April 2014), 5–28. https://doi.org/10.14529/jsfi140101

[8] https://mlcommons.org/benchmarks/

[9] Jiao, Xun, Fred Lin, Harish D. Dixit, Joel Coburn, Abhinav Pandey, Han Wang, Venkat Ramesh et al. "PVF (Parameter Vulnerability Factor): A Scalable Metric for Understanding AI Vulnerability Against SDCs in Model Parameters." *arXiv preprint arXiv:2405.01741* (2024).

[10] Gemini Team, Google, “Gemini: A family of highly capable multimodal models”, https://arxiv.org/pdf/2312.11805

[11] Shubhendu Mukherjee, “Architecture Design for Soft Errors”, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Behnam Ghavami, Mohammad Shahidzadeh, Lesley Shannon, Steve Wilton: "Gemini: A family of highly capable multimodal models", IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), 2024.

[13] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. "Ares: a framework for quantifying the resilience of deep neural networks", Design Automation Conference (DAC), 2018.

[14] Rouhani et al., “Microscaling Data Formats for Deep Learning”, https://arxiv.org/pdf/2310.10537

[15] Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B. and Sankar, S., 2021. Silent data corruptions at scale. arXiv preprint arXiv:2102.11245.

[16] Dixit, H.D., Boyle, L., Vunnam, G., Pendharkar, S., Beadon, M. and Sankar, S., 2022. Detecting silent data corruptions in the wild. arXiv preprint arXiv:2203.08989.

[17] Rhea Dutta, Harish Dattatraya Dixit, Rik Van Riel, Gautham Vunnam, and Sriram Sankar. 2025. Hardware Sentinel: Protecting Software Applications from Hardware Silent Data Corruptions. In Proceedings of the 30th ACM International Conference on Architectural Support

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 482–497. https://doi.org/10.1145/3676641.3716258

[18] Hochschild, P.H., Turner, P., Mogul, J.C., Govindaraju, R., Ranganathan, P., Culler, D.E. and Vahdat, A., 2021, June. Cores that don't count. In Proceedings of the Workshop on Hot Topics in Operating Systems (pp. 9-16).

[19] Ma, D., Lin, F., Desmaison, A., Coburn, J., Moore, D., Sankar, S. and Jiao, X., 2024, April. Dr. DNA: Combating silent data corruptions in deep learning using distribution of neuron activations. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (pp. 239-252).

[20] Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A. and Yang, A., 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783.

[21] Li, S., Huang, J., Tang, P.T.P., Khudia, D., Park, J., Dixit, H.D. and Chen, Z., 2022, December. Efficient soft-error detection for low-precision deep learning recommendation models. In 2022 IEEE International Conference on Big Data (Big Data) (pp. 1556-1563). IEEE.

[22] Wang, S., Zhang, G., Wei, J., Wang, Y., Wu, J. and Luo, Q., 2023, October. Understanding silent data corruptions in a large production cpu population. In Proceedings of the 29th Symposium on Operating Systems Principles (pp. 216-230).

[23] Serebryany et al., “SiliFuzz: Fuzzing CPUs by proxy”, https://arxiv.org/abs/2110.11519

[24] Datacenter GPU Manager, NVIDIA, https://github.com/nvidia/dcgm

[25] Open Field Health Check, AMD, https://github.com/amd/Open-Field-Health-Check

[26] OpenDCDiag, Intel, https://github.com/opendcdiag/opendcdiag

[27] Wongpanich et al., https://arxiv.org/pdf/2502.06982

[28] N. Karystinos et al., “Harpocrates: breaking the silence of CPU faults through hardware-in-the-loop program generation.” in Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), 2024, pp. 516--531.

[29] M. Shamsa et al., “Improved Silent Data Error Detection Through Test Optimization Using Reinforcement Learning” in Proceedings of the IEEE International Reliability Physics Symposium (IRPS), 2025.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

[30] T. Macieira et al., “Silent data corruptions: understand and quantify" in Proceedings of the IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2024, pp. 1--7.

[31] O. Chatzopoulos, N. Karystinos, G. Papadimitriou, D. Gizopoulos, H. D. Dixit and S. Sankar, "Veritas – Demystifying Silent Data Corruptions: μArch-Level Modeling and Fleet Data of Modern x86 CPUs," 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA), Las Vegas, NV, USA, 2025, pp. 1-14, doi: 10.1109/HPCA61900.2025.00012.

[32] O. Chatzopoulos, M. Trakosa and D. Gizopoulos, "Accurate Analysis of Silent Data Corruptions in Programmable AI Accelerator Microarchitectures," 2025 IEEE 31st International Symposium on On-Line Testing and Robust System Design (IOLTS), Ischia, Italy, 2025, pp. 1-4, doi: 10.1109/IOLTS65288.2025.11117132.

[33] O. Chatzopoulos, G. Papadimitriou, D. Gizopoulos, H. D. Dixit and S. Sankar, "From Gates to SDCs: Understanding Fault Propagation Through the Compute Stack," 2025 Design, Automation & Test in Europe Conference (DATE), Lyon, France, 2025, pp. 1-7, doi: 10.23919/DATE64628.2025.10992727.

[34] J. Ma et al., “Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach,” ASPLOS '24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, Pages 220 - 235, https://doi.org/10.1145/3622781.3674182.

[35] W. Li, C. Nigh, D. Duvalsaint, S. Mitra and R. D. Blanton, "PEPR: Pseudo-Exhaustive Physically-Aware Region Testing," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 314-323, doi: 10.1109/ITC50671.2022.00083.

[36] F. Angione, P. Bernardi and A. Sinha, "From Structural Test Escapes to Silent Data Errors: A Preliminary Analysis," 2025 IEEE 9th International Test Conference India (ITC India), Bangalore, India, 2025, pp. 1-6, doi: 10.1109/ITCIndia66078.2025.11141623.

[37] G. Papadimitriou, D. Gizopoulos, H. D. Dixit and S. Sankar, "Silent Data Corruptions: The Stealthy Saboteurs of Digital Integrity," 2023 IEEE 29th International Symposium on On-Line

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Testing and Robust System Design (IOLTS), Crete, Greece, 2023, pp. 1-7, doi: 10.1109/IOLTS59296.2023.10224870.

[38] D. Gizopoulos, G. Papadimitriou, O. Chatzopoulos, N. Karystinos, H. D. Dixit and S. Sankar, "Silent Data Corruptions in Computing Systems: Early Predictions and Large-Scale Measurements," 2024 IEEE European Test Symposium (ETS), The Hague, Netherlands, 2024, pp. 1-10, doi: 10.1109/ETS61313.2024.10567770.

9.0 License

OCP encourages participants to share their proposals, specifications and designs with the community. This is to promote openness and encourage continuous and open feedback. It is important to remember that by providing feedback for any such documents, whether in written or verbal form, that the contributor or the contributor's organization grants OCP and its members irrevocable right to use this feedback for any purpose without any further obligation.

It is acknowledged that any such documentation and any ancillary materials that are provided to OCP in connection with this document, including without limitation any white papers, articles, photographs, studies, diagrams, contact information (together, “Materials”) are made available under the Creative Commons Attribution-ShareAlike 4.0 International License found here: https://creativecommons.org/licenses/by-sa/4.0/, or any later version, and without limiting the foregoing, OCP may make the Materials available under such terms.

As a contributor to this document, all members represent that they have the authority to grant the rights and licenses herein. They further represent and warrant that the Materials do not and will not violate the copyrights or misappropriate the trade secret rights of any third party, including without limitation rights in intellectual property. The contributor(s) also represent that, to the extent the Materials include materials protected by copyright or trade secret rights that are owned or created by any third-party, they have obtained permission for its use consistent with the foregoing. They will provide OCP evidence of such permission upon OCP’s request. This document and any "Materials" are published on the respective project's wiki page and are open to the public in accordance with OCP's Bylaws and IP Policy. This can be found at http://www.opencompute.org/participate/legal-documents/. If you have any questions please contact OCP.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.