Comprehensive Technical Analysis and Reproduction Guide for Gemini (SOSP 2023)

Executive Summary

This report provides a comprehensive technical analysis of the paper "Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints," presented at SOSP 2023. The Gemini system addresses the critical problem of fault tolerance in large-scale distributed training of AI models. Failures in these massive, long-running computations are frequent and lead to substantial economic losses from wasted GPU-hours. Traditional fault tolerance, which relies on checkpointing to slow persistent storage, imposes a harsh trade-off between recovery time and training overhead.

Gemini's core innovation is to utilize the high-bandwidth, aggregated CPU memory of the compute cluster as a fast, distributed, first-tier storage for checkpoints.[1] This enables extremely frequent, per-iteration checkpointing with minimal latency. To overcome the challenges of this approach, Gemini introduces two key contributions: 1) a provably near-optimal checkpoint placement algorithm that maximizes the probability of successful recovery from in-memory checkpoints, even in the event of multiple node failures, and 2) an interference-aware traffic scheduling algorithm that pipelines checkpointing data transfers to minimize their impact on training throughput.[1] The system guarantees 100% recovery by falling back to a conventional persistent storage layer if necessary.

A six-week reproduction of Gemini for a graduate-level distributed systems course is assessed as **feasible but challenging**. The project's success is contingent upon two primary factors: securing access to adequate hardware resources (a minimum of four multi-GPU nodes with high-speed networking) and carefully defining a Minimal Viable Reproduction (MVR) scope. The availability of an official "Artifact Evaluation" code repository is a significant asset, providing a concrete starting point.[4] However, this also presents a key challenge, as such research artifacts are often prototypes tightly coupled to a specific execution environment (in this case, Amazon Web Services) and may require substantial effort to adapt and debug. Other critical challenges include the inherent complexity of integrating with a sophisticated framework like DeepSpeed and the need to replicate a realistic failure-injection mechanism to validate the system's core functionality.

SECTION 1: Paper Details & Availability

1.1 Core Paper Information

A successful project proposal begins with accurate and complete bibliographic details. The core information for the Gemini paper has been verified as follows.

Field	Details
Full Title	Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints [1]
Authors	Zhuang Wang (Rice University), Zhen Jia (Amazon Web Services, Inc.), Shuai Zheng (Amazon Web Services), Zhen Zhang (Amazon Web Services), Xinwei Fu (Amazon Web Services), T. S. Eugene Ng (Rice University), Yida Wang (Amazon) [2]
Conference	29th ACM Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany [1]
DOI	10.1145/3600006.3613145 [1]
Paper Link	https://dl.acm.org/doi/10.1145/3600006.3613145 (Official ACM)
Preprint	(https://zhuangwang93.github.io/docs/Gemini_SOSP23.pdf) (Author's Site)

1.2 Code and Artifact Availability

The availability of source code is a critical factor for a reproduction project. The authors have provided an artifact for evaluation purposes.

Field	Details & Assessment
GitHub Repository	(https://github.com/zhuangwang93/Gemini_SOSP23) [6]
Artifact Badges	No official ACM artifact badges (e.g., Available, Functional, Reproduced) were found associated with the paper in the available materials.[6] SOSP 2023 did have an artifact

	evaluation process.[7]
Documentation	*To be assessed directly from the repository.* Initial inspection is required. Artifact evaluation packages often have minimal documentation focused on reproducing specific figures from the paper.
Prerequisites	*To be assessed directly from the repository.* The paper specifies PyTorch, DeepSpeed, CUDA, NCCL, and etcd versions, but the repository may have additional dependencies.[2]
License	*To be assessed directly from the repository.* The absence of a standard open-source license (e.g., MIT, Apache 2.0) could pose a constraint on modification and use.
Installation Complexity	**Rated: Hard (Anticipated).** The repository is explicitly for "Artifact Evaluation".[4] Such artifacts are typically research prototypes, not production-ready software. They are often brittle and hard-coded for the authors' specific hardware and software environment (AWS EC2), making adaptation to a different cluster a significant challenge.

The "Artifact Evaluation" designation is a double-edged sword. On one hand, it confirms the existence of functional code that was reviewed by a committee, which is a substantial advantage over projects with no public code. On the other hand, the code's primary purpose was to validate the paper's claims under specific conditions, not to be a general-purpose, portable, or well-documented tool. The project team should budget significant time in the initial phase for "software archaeology"—understanding, adapting, and stabilizing this codebase for their target environment.

1.3 Reproducibility Assessment

- **Data Availability**: The paper uses the public **Wikipedia-en corpus** for training.[2] While the dataset is accessible, the specific preprocessing pipeline used by the authors is not detailed, which may require minor reverse-engineering to ensure a comparable data input.
- **Experimental Setup**: The hardware environment is meticulously detailed, centered on **Amazon Web Services (AWS)**, specifically p4d.24xlarge instances with NVIDIA A100 GPUs and p3dn.24xlarge instances with V100 GPUs.[2] The networking (400Gbps EFA) and persistent storage (FSx) are also AWS-specific. This strong coupling to the AWS ecosystem is a key consideration. The system's performance, particularly the traffic

scheduling algorithm, may be implicitly optimized for the latency and topology characteristics of the AWS environment. Reproducing the exact performance figures on a non-AWS cluster (e.g., an academic HPC center) will be challenging and may reveal sensitivities to the underlying infrastructure.
- **Runtime Estimates**: The paper does not provide total runtimes for its experiments. However, it motivates its work by noting that a single checkpoint of a large model to persistent storage can take over 40 minutes.[9] This implies that full-scale experiments are extremely time-consuming and will need to be scaled down significantly for a 6-week project.
- **Known Issues**: A direct inspection of the GitHub repository's "Issues" tab is required to identify any community-reported problems with reproducibility. None were found in the initial research.

SECTION 2: Technical Deep Dive

2.1 Problem Statement & Motivation

- **The Core Problem**: Modern large-scale AI model training involves thousands of GPUs running for weeks or months. At this scale, component failures are not an exception but a certainty.[10] The standard mechanism for fault tolerance is checkpoint/restart, where the entire training state (model weights, optimizer states) is periodically saved to durable storage. The central challenge is that this durable storage (e.g., a network file system or cloud object store) has orders of magnitude lower bandwidth than the internal cluster interconnect.[2]
- **Economic Importance**: This bandwidth disparity creates a costly dilemma. Infrequent checkpoints (e.g., every 3-4 hours) minimize training overhead but mean that a failure can erase hours of expensive computation. For context, the training of OPT-175B lost approximately 178,000 GPU-hours to failures, and LLaMA 3.1 training failed on average every three hours.[2] Conversely, frequent checkpoints to slow storage would introduce so much overhead that training would grind to a halt.[12]
- **Limitations of Existing Approaches**: Existing systems are fundamentally bottlenecked by the I/O performance of their persistent storage tier. This leads to long checkpoint save times, long checkpoint retrieval times (during which GPUs are idle), and consequently, a high "wasted time" for each failure event.[2]
- **Relevance to Spot/Preemptible Instances**: The problem is exacerbated by the use of cheaper spot or preemptible cloud instances. These instances offer large cost savings but can be revoked with little notice. A system like Gemini, which can recover from a failure in seconds rather than minutes or hours, dramatically improves the economic viability of using these less reliable but more affordable resources for large-scale

training.

2.2 Gemini's Core Algorithm & Design

- **High-Level Approach**: Gemini introduces a hierarchical storage system that uses the aggregated CPU RAM of the training nodes as a high-speed, distributed cache for checkpoints.[1] It decouples short-term, high-frequency checkpoints for failure recovery (stored in RAM) from long-term, low-frequency checkpoints for archival purposes (stored on persistent disk). This design leverages the fact that the network bandwidth between nodes is vastly higher than the bandwidth to external storage, enabling per-iteration checkpointing.[1] This approach reflects a classic systems design pattern: using a faster, more expensive, but volatile storage tier (RAM) to mask the latency of a slower, cheaper, and durable one (disk). The economic calculus is that the cost of saved GPU time from near-instant recovery far outweighs the risk and complexity of managing an in-memory checkpointing system.
- **Key Innovations**:
    1. **Near-Optimal Checkpoint Placement**: To mitigate the risk of losing in-memory checkpoints due to node failures, Gemini replicates checkpoint shards across multiple nodes. It proposes a placement algorithm that maximizes the probability of a full checkpoint being recoverable. For N nodes and a replication factor of m, if N is divisible by m, nodes are divided into disjoint groups of size m, and replication occurs within the group. This is proven to be optimal. If not divisible, a hybrid "mixed" strategy using group and ring placements provides a near-optimal guarantee.[2]
    2. **Interference-Aware Traffic Scheduling**: To achieve its "zero overhead" claim, Gemini cannot simply pause training to save a checkpoint. Instead, it employs a scheduling algorithm that pipelines the transfer of checkpoint data from GPU to remote CPU memory, interleaving it with the primary training communication (e.g., all-reduce for gradients). This effectively hides the checkpointing latency by utilizing network and CPU resources during moments they are not the bottleneck for the main training loop.[2]
- **Recovery Mechanism**:
    - **Failure Detection**: A designated Root Agent monitors node health via heartbeats sent to a distributed key-value store. Hardware failures are detected and handled in coordination with the underlying cloud platform's infrastructure management service (e.g., AWS Auto Scaling Groups), which is responsible for provisioning replacement nodes.[2]
    - **Recovery Workflow**: Upon detecting a failure, the Root Agent orchestrates the recovery. It identifies the latest globally consistent checkpoint, directs all active and newly provisioned nodes to load it, and signals the training framework to resume execution from that state.[2]

    - **Tiered Restoration**: The system greedily restores from the fastest available tier. It first checks for a checkpoint in the node's local RAM, then in the RAM of remote peers, and only as a last resort does it fall back to retrieving the latest checkpoint from slow persistent storage. This ensures the fastest possible recovery while maintaining a 100% success guarantee.[1]

2.3 Architecture & Components

The system is composed of a control plane and a data plane operating on each node.

- **System Components**:
    - **Worker Agent**: A process on each training node responsible for executing the data plane tasks: capturing its local model state shard, sending it to peers according to the placement strategy, and receiving shards from others. It also periodically updates its health status in the key-value store.[2]
    - **Root Agent**: A single, leader-elected process that forms the control plane. It monitors the cluster's health, coordinates with the cloud provider to replace failed nodes, and orchestrates the recovery process by instructing workers which checkpoint to load.[2] The use of a centralized coordinator is a pragmatic design choice that simplifies implementation, though it could present a scalability bottleneck in extremely large clusters. The leader election mechanism provides resilience against the Root Agent's own failure.[2]
    - **Distributed Key-Value Store**: An external service like **etcd** is used for service discovery, health monitoring (heartbeating), and leader election for the Root Agent.[2]
    - **Cloud Operator**: An abstraction representing the API of the cloud infrastructure (e.g., AWS), used by the Root Agent to request the replacement of failed physical or virtual machines.[2]
- **Data Flow**: During a checkpoint operation, model state data flows from GPU memory to local CPU memory (RAM). From there, it is transmitted over the high-speed cluster network to the CPU memory of peer nodes designated by the placement algorithm.[2]
- **State Management**: The state being checkpointed is comprehensive, including not just the model parameters but also the optimizer states (e.g., momentum and variance buffers for Adam), which are essential for correctly resuming training.[13] The current training step or iteration number is also saved to manage the training data pipeline.

2.4 Implementation Details

- **Programming Language**: While not explicitly stated, the use of PyTorch and DeepSpeed makes **Python** the certain primary implementation language.[2]
- **ML & Distributed Frameworks**: The system is implemented as a fault-tolerance layer

on top of **PyTorch** and is tightly integrated with the **DeepSpeed** library, specifically using the **ZeRO-3** parallelism strategy, which partitions model and optimizer states across all GPUs.[2]

- **Storage & Networking Backends**:
  - In-memory storage is host CPU RAM.
  - Persistent storage in the experiments was **AWS FSx**.[2]
  - The coordination backend was **etcd** v3.5.[2]
  - Networking relies on high-performance interconnects like **AWS Elastic Fabric Adapter (EFA)**, with GPU-to-GPU communication likely handled by **NCCL** as is standard in DeepSpeed.[2]
- **Serialization**: The paper does not specify the serialization format, but standard PyTorch mechanisms (torch.save) are the most probable choice.

SECTION 3: Evaluation & Benchmarks

3.1 Experimental Setup

- **Models and Datasets**: The evaluation was performed by training large language models, including **GPT-2**, **BERT**, and **RoBERTa**, with parameter counts scaled up to 100 billion. The training data was the **Wikipedia-en corpus**.[2]
- **Cluster Configuration**: The primary testbed consisted of 16 AWS p4d.24xlarge instances, for a total of 128 NVIDIA A100 (40GB) GPUs. These nodes were connected via a 400Gbps EFA network. The baseline persistent storage was an AWS FSx file system with an aggregated bandwidth of 20Gbps.[2]
- **Baseline Comparisons**: Gemini was compared against two practical baselines:
  1. **Strawman**: Mimics the configuration of the BLOOM-176B training, checkpointing to persistent storage every 100 iterations.[2]
  2. **HighFreq**: A more aggressive baseline that checkpoints to persistent storage every 12 iterations, testing the limits of that approach.[2]

It is important to contextualize these baselines. Both rely on a persistent storage backend with a fixed, relatively low bandwidth. In high-performance computing (HPC) environments, it is common to use a parallel file system (e.g., Lustre) whose bandwidth scales with the number of compute nodes. The choice of a fixed-bandwidth baseline, while representative of many cloud setups, may present Gemini's relative performance gains in a particularly favorable light.[9]

3.2 Key Results

- **Primary Claim**: The headline result is that Gemini achieves **more than 13x faster failure recovery** than existing solutions, measured by the total "wasted time" (recomputation time + checkpoint retrieval time).[2]
- **Performance Metrics**:
    - **Recovery & Retrieval Time**: Gemini reduces checkpoint retrieval time by up to **250x** (seconds vs. minutes) and total wasted time by over **92%** compared to the best-performing baseline.[2]
    - **Training Throughput Impact**: The paper makes the strong claim of **"no overhead on training throughput"**.[1] This is a critical result, suggesting that the interference-aware traffic scheduling algorithm is highly effective at hiding checkpointing latency. This claim is the most ambitious and potentially the most sensitive to the specific hardware and workload balance. Replicating it successfully is the acid test for any reproduction effort.
    - **Checkpoint Frequency**: The system successfully enables checkpointing at the optimal frequency of **every training iteration**.[1]
- **Figures to Reproduce**: The two most important results to reproduce to validate the paper's core claims are:
    1. **Iteration Time Comparison (related to Figure 7)**: A plot showing that the per-iteration training time with Gemini enabled is statistically indistinguishable from a run with no checkpointing at all. This would validate the "zero overhead" claim.
    2. **Average Wasted Time Comparison**: A bar chart comparing the total wasted time per failure for Gemini, the Strawman baseline, and the HighFreq baseline. This would validate the ">13x faster recovery" claim.

3.3 Evaluation Scenarios

- **Failure Injection**: Failures were realistically simulated by using the **AWS Auto Scaling Group (ASG)** service to terminate running EC2 instances, triggering the full detection, replacement, and recovery cycle.[2]
- **Workload Characteristics**: The experiments used large models with standard configurations (e.g., micro-batch size of 8, Adam optimizer) and relied on the ZeRO-3 optimization in DeepSpeed, which is a common setup for training models that do not fit on a single GPU.[2]

SECTION 4: Systems Traits Analysis

4.1 Reliability

- **Fault Tolerance**: Gemini provides fault tolerance through a multi-layered checkpoint/restart mechanism. The primary innovation is using replicated, in-memory checkpoints for fast recovery from common failures. The system guarantees eventual recovery by falling back to durable, persistent storage, thus handling the case where an entire in-memory checkpoint is lost.[2]
- **Failure Model**: The system is designed to handle fail-stop failures of entire nodes, which can be caused by hardware faults, network partitions, or software crashes. It can tolerate multiple simultaneous node failures, with the probability of successful in-memory recovery being a function of the replication factor and the placement strategy.[2]
- **Recovery Guarantees**: Gemini provides a **100% recovery guarantee** due to its fallback to persistent storage. Its probabilistic guarantee relates to the *speed* of recovery; the placement algorithm is designed to maximize the probability of a fast, in-memory recovery.[1]

4.2 Scalability

- **Horizontal Scalability**: The system is designed to scale horizontally with the training cluster. The checkpoint placement algorithm is parameterized by the number of nodes N and replicas m, allowing it to adapt to different cluster sizes.[2]
- **Scalability Bottlenecks**:
    1. **Network Bandwidth**: While Gemini leverages high-speed interconnects, the checkpointing traffic still consumes network bandwidth. The traffic scheduling algorithm is the key mitigation here, but in a network-bound training job, adding checkpoint traffic could still create a bottleneck.
    2. **Root Agent**: The centralized Root Agent, which polls a key-value store for health status, could become a coordination bottleneck at extreme scales (thousands of nodes). The paper's evaluation scale (128 GPUs) is large but not at the frontier where this would likely become a major issue.
    3. **CPU Memory Capacity**: Each node must have sufficient RAM to store its own checkpoint shard and potentially shards from its peers. This requirement scales with the model size and could become a constraint.

4.3 Correctness/Guarantees

- **Consistency Guarantees**: Gemini operates in the context of **synchronous data-parallel training**. This model requires that all workers operate on the same model

version at each step. To maintain correctness, upon a failure, the entire system must roll back to the same, globally consistent state. Gemini ensures this by having the Root Agent designate a single, valid checkpoint version from which all nodes must recover.[2]

- **Correct Recovery**: Recovery is correct because each checkpoint represents a complete, consistent snapshot of the training state at a specific iteration. By restoring this state across all nodes, the training is guaranteed to resume from a valid point in its execution history.[2]
- **Ordering Guarantees**: The system ensures that training resumes from the *latest available complete checkpoint*. All progress made in iterations after that checkpoint was saved is discarded, which is the standard trade-off in any checkpoint/restart system.[2]

SECTION 5: Feasibility for 6-Week Reproduction

5.1 Scope Estimation

A full-scale reproduction is infeasible in six weeks. A successful project must focus on a Minimal Viable Reproduction (MVR).

- **Essential (Must-Have)**:
    1. Set up a multi-node (at least 4 nodes) distributed training environment using PyTorch and DeepSpeed.
    2. Implement the core in-memory checkpointing mechanism: saving and loading model state shards to and from local RAM.
    3. Implement a simplified version of the checkpoint placement strategy (e.g., the group placement for N divisible by m=2).
    4. Create a basic failure injection script that can kill a worker process or node.
    5. Implement the recovery workflow: detect the failure, select the latest in-memory checkpoint, and restart the training job.
    6. Reproduce a scaled-down version of the "Average Wasted Time" result, showing a significant improvement over a baseline that writes to a network file system.
- **Nice-to-Have**:
    1. Implement the more complex "mixed" placement strategy.
    2. Attempt to reproduce the "zero overhead" claim by implementing a simplified version of the traffic scheduling algorithm. This is significantly more complex.
    3. Integrate with a cluster manager for automated node replacement.
    4. Explore Gemini's performance under different failure scenarios (e.g., correlated failures).

5.2 Resource Requirements

The paper's hardware is beyond the scope of a typical academic project. The minimum requirements for a meaningful MVR are estimated below.

Resource	Minimum Requirement (for MVR)	Justification & Notes
Compute Nodes	4 nodes	To test distributed checkpointing and recovery with at least one failure.
GPUs per Node	2-4 NVIDIA GPUs (e.g., V100, A100)	Required for DeepSpeed ZeRO-3 and to have a non-trivial computation load.
GPU Memory	32 GB+	To train a moderately sized model that necessitates distributed storage (e.g., a GPT-2 variant).
CPU Memory (RAM)	128 GB+ per node	Must be large enough to hold the sharded model/optimizer state. A 1B parameter model can require ~16GB of state, so ample headroom is needed.
Network	100 Gbps Ethernet with RDMA (RoCE/InfiniBand)	High-speed, low-latency networking is critical for both DeepSpeed performance and for Gemini's in-memory transfers. Standard 10Gbps Ethernet will likely be a bottleneck.
Persistent Storage	Standard NFS mount	A baseline network file system is needed to compare against.
Estimated GPU-Hours	200-400 GPU-hours	For development, debugging, and running final experiments on a scaled-down model.

Suitability for IBEX (KAUST Cluster): The IBEX cluster likely has nodes that meet these specifications. The key will be securing a reservation of at least 4 suitable nodes simultaneously for interactive development and final runs.
Cloud Alternatives: Using AWS/GCP/Azure spot instances is a viable alternative and would more closely mimic the paper's environment. A budget for ~200 GPU-hours on instances like AWS p3.8xlarge or p4d.24xlarge would be required.

5.3 Technical Prerequisites

- **Skills Needed**:
    - **PyTorch**: Advanced proficiency is required, including familiarity with its distributed package (torch.distributed).
    - **Distributed Systems**: Strong understanding of concepts like consensus, leader election, failure detection, and consistency models.
    - **Systems Programming**: Proficiency in Python and shell scripting. Experience with frameworks like DeepSpeed is a major plus but can be learned.
- **Learning Curve**: The primary learning curve will be understanding the internals of DeepSpeed and how to modify its training loop and state management to integrate Gemini's logic. This is non-trivial and will likely consume the first 1-2 weeks of the project.

SECTION 6: Implementation Strategy

6.1 Minimal Viable Reproduction (MVR)

- **Core Functionality**: The MVR must demonstrate the core trade-off: fast, in-memory checkpointing provides significantly lower recovery time than slow, persistent checkpointing. This requires implementing: (1) a mechanism to save/load state to RAM, (2) a replication strategy between nodes, and (3) a failure recovery coordinator.
- **Simplifications**:
    - **No Traffic Scheduling**: For the MVR, assume synchronous checkpointing (pause-and-save). This sacrifices the "zero overhead" goal but makes the implementation vastly simpler and still allows for testing the primary claim of faster recovery.
    - **Manual Coordination**: The Root Agent can be simplified to a script that is manually run after a failure is detected, rather than a fully autonomous, leader-elected service.
    - **Fixed Replication Strategy**: Implement only the simpler group placement strategy with a replication factor of 2.

6.2 Week-by-Week Plan Suggestion

- **Week 1: Setup & Baseline**:
    - Provision a 4-node cluster environment (IBEX or cloud).
    - Set up PyTorch, DeepSpeed, and all dependencies.
    - Successfully run a distributed training job for a baseline GPT-2 model.

    - Establish a baseline performance metric: training throughput and recovery time from a checkpoint on NFS.
- **Week 2: In-Memory Checkpointing (Local)**:
    - Modify the DeepSpeed training script to add a checkpointing hook.
    - Implement the logic to save the model and optimizer state shards to a RAM disk (or directly in memory) on each local node.
    - Implement the corresponding logic to load from this in-memory checkpoint.
- **Week 3: Distributed Checkpointing & Replication**:
    - Implement the communication logic for workers to send their in-memory checkpoint shards to a designated peer (implementing the group placement strategy for m=2).
    - Ensure each node stores its own shard and one replica shard.
- **Week 4: Failure Recovery Logic**:
    - Develop a simple coordinator script (the MVR Root Agent).
    - Implement a failure injection script to kill a random worker process.
    - Implement the recovery logic: the coordinator detects the dead worker, identifies the latest valid checkpoint version, and instructs all live workers to reload from their local or replicated in-memory copies.
- **Week 5: Testing & Evaluation**:
    - Run experiments to measure the key MVR metric: "wasted time" for Gemini vs. the NFS baseline.
    - Debug and stabilize the system.
    - Generate plots comparing the recovery times.
- **Week 6: Final Experiments & Report Writing**:
    - Run final, clean experiments.
    - Analyze results and write the final project report and proposal.

6.3 Testing Strategy

- **Unit Tests**: Key components like the serialization/deserialization logic and the placement algorithm's node mapping should have unit tests.
- **Integration Tests**: Test the full checkpoint-and-restore cycle on a multi-node setup without failures to ensure correctness.
- **Failure Injection**: The core of the testing strategy. A script should be able to:
    - Kill a random worker process (simulating a software crash).
    - Use cluster commands to halt or reboot a full node (simulating hardware failure).
    - Test recovery from these scenarios repeatedly to ensure robustness.

SECTION 7: Potential Extensions & Bonus Work

7.1 Limitations to Explore

- **Correlated Failures**: Gemini's placement strategy assumes independent node failures. An interesting extension would be to analyze its resilience to correlated failures, such as a rack-level power or network failure that takes down an entire group of nodes simultaneously. The probability of successful in-memory recovery would plummet in this scenario.
- **Scalability of the Coordinator**: The centralized Root Agent is a potential bottleneck. One could analyze the latency of the recovery coordination process as the number of nodes increases to identify at what scale it becomes a problem.
- **Network Contention**: The "zero overhead" claim relies on the ability to hide checkpoint traffic. An extension could be to test workloads with different computation-to-communication ratios to find the breaking point where checkpointing traffic begins to visibly impact training throughput.
- **Scope**: The paper explicitly states that elastic and asynchronous training are out of scope.[2] Exploring the challenges of adapting Gemini to an elastic training scenario (where the number of workers can change) would be a significant research direction.

7.2 Proposed Improvements

- **Decentralized Coordination**: Replace the Root Agent and etcd dependency with a peer-to-peer gossip protocol for health monitoring and a decentralized consensus algorithm for agreeing on which checkpoint to restore. This would be much more scalable and resilient but also far more complex.
- **Adaptive Replication**: Instead of a fixed replication factor, a more advanced system could adapt the number of replicas based on observed failure rates or the criticality of the training job.
- **Differential/Incremental Checkpointing**: Gemini checkpoints the full state every time. Combining its in-memory approach with differential checkpointing (saving only what has changed) could further reduce the amount of data transferred over the network for each checkpoint.

7.3 Alternative Approaches

- **Gradient-Based Checkpointing (e.g., Checkmate)**: Systems like Checkmate propose an alternative where checkpoints are not explicit state snapshots but are reconstructed by streaming gradients to a "shadow" cluster that applies them to a copy of the model. This avoids pausing the primary cluster but requires dedicated hardware for the shadow

cluster.[10]

- **Elastic/Adaptive Recovery (e.g., Odyssey)**: Instead of just restarting from a checkpoint, systems like Odyssey attempt to dynamically reconfigure the training job to run on the remaining healthy nodes, avoiding the delay of waiting for replacement nodes. This trades off some post-recovery performance for higher availability.[10]

SECTION 8: Risk Assessment

8.1 Technical Risks & Mitigation Strategies

A proactive risk management plan is essential for a time-constrained project.

Risk	Likelihood	Impact	Mitigation Strategy
Artifact Code is Intractable	High	High	**Plan A**: Time-box the effort to adapt the artifact code to a maximum of 1.5 weeks. **Plan B (Fallback)**: If adaptation fails, pivot to a clean-room implementation of the MVR from scratch, guided by the paper's algorithms. This is feasible because the core ideas are well-described.
Insufficient Hardware Access	Medium	High	**Plan A**: Start discussions with the KAUST IBEX administrators immediately to understand reservation policies and availability. **Plan B (Fallback)**: Secure a small cloud computing grant or use student credits on AWS/GCP. Scale down the model

			and node count to fit the budget.
DeepSpeed Integration Complexity	High	Medium	Focus the implementation on hooks and callbacks provided by DeepSpeed rather than deep modification of its core engine. Isolate the Gemini logic in a separate module that interacts with DeepSpeed through a minimal, well-defined API.
Network Performance Issues	Medium	Medium	Run network benchmarks (e.g., iperf, ib_write_bw) between cluster nodes early in Week 1. If performance is subpar, adjust expectations for the "zero overhead" claim and focus solely on the recovery time improvements, which are less sensitive to network latency.

SECTION 9: Related Work & Context

9.1 Background on Distributed Training Fault Tolerance

Fault tolerance in large-scale computing has traditionally been addressed by checkpoint/restart. In the context of distributed AI training, this involves periodically saving the model parameters and optimizer states to a distributed file system.[14] The primary challenge has always been the overhead of this process. As models grew from megabytes to

terabytes, the time to pause training, serialize the state, and write it to disk became a significant fraction of the total training time, leading to the development of more sophisticated strategies.[10]

9.2 Related Systems

- **Traditional Checkpoint/Restart Libraries**: Libraries like Berkeley Lab Checkpoint/Restart (BLCR) and DMTCP have long existed for general HPC applications. However, they are often application-agnostic and may not be optimized for the specific structure of ML training workloads.[15]
- **Checkmate (MLSys '21)**: This system takes a different approach by avoiding explicit state snapshots. It forwards gradients to a secondary "shadow" cluster that maintains a replica of the model. This can achieve per-iteration checkpointing with very low overhead on the primary cluster but requires provisioning a separate set of machines, increasing hardware costs.[10]
- **Odyssey (OSDI '23)**: Odyssey focuses on adaptive recovery. When a failure occurs, instead of waiting for a replacement node, it intelligently resizes and reconfigures the training job to continue on the remaining resources. This prioritizes minimizing idle time over maintaining peak performance.[10]
- **LowDiff (SoCC '22)**: This system focuses on reducing checkpointing overhead by using differential checkpointing, saving only the changes (gradients) since the last full checkpoint. This is effective but can complicate the recovery process, which may need to "replay" several differential checkpoints.[10]

Gemini's contribution is unique in its focus on leveraging a resource already present in the cluster (CPU RAM) to create a fast recovery path, combining this with a theoretically-grounded placement strategy to manage the risk of using volatile storage.

9.3 Recent Developments

The field of fault-tolerant and efficient large-scale training is highly active. Research continues to explore the trade-offs between different strategies: checkpointing vs. redundancy, synchronous vs. asynchronous recovery, and centralized vs. decentralized control. The increasing scale of models and the economic pressures of training continue to make systems like Gemini highly relevant. There has been no major follow-up work yet that directly cites and supersedes Gemini's specific approach, given its recent publication date.

SECTION 10: Practical Considerations

10.1 Development Environment

- **Recommended Setup**: Use **Docker containers** to manage dependencies and ensure a consistent environment across all cluster nodes. Create a Dockerfile that installs the specific versions of CUDA, PyTorch, DeepSpeed, and other libraries mentioned in the paper.[2] This will dramatically simplify deployment and reduce "works on my machine" issues.
- **IDE/Tools**: For development, use an IDE with remote development capabilities (e.g., VS Code with Remote-SSH) to edit code directly on the cluster nodes. Use a terminal multiplexer like tmux or screen to manage multiple windows and keep processes running after disconnecting.
- **Debugging**: Debugging distributed systems is notoriously difficult. Rely heavily on **structured logging**. Each process on each node should log events with timestamps, its rank, and clear messages. For collective operations that hang, use tools like torch.distributed.monitored_barrier or attach debuggers like pdb or ipdb before and after the suspected call.

10.2 Datasets

- **Source**: The **Wikipedia-en corpus** is available from multiple sources. The Hugging Face datasets library provides an easy way to download and preprocess it (datasets.load_dataset('wikipedia', '20220301.en')).
- **Size and Preprocessing**: The full dataset is large. For development and testing, use a small subset (e.g., the first 10,000 samples) to enable rapid iteration. Preprocessing (tokenization) can be done once and the result saved to disk to avoid repeating this step in every training run.
- **Synthetic Data**: For initial systems-level testing (e.g., verifying checkpoint transfer and recovery), using synthetic, randomly generated data is highly recommended. This decouples the systems-level logic from the data loading pipeline, simplifying debugging.

10.3 Demo/Visualization

- **Key Plots**: The most effective visualizations will directly address the paper's main claims:
    1. A line graph of **training throughput (samples/sec)** over time, comparing a baseline run with a Gemini run. This should show the lines are nearly identical, proving the "zero overhead" claim.
    2. A bar chart of **average recovery time** comparing Gemini to an NFS-based

checkpointing baseline. This should show a dramatic reduction, proving the core value proposition.
3. A timeline visualization of a failure event, showing the point of failure, the short time to recovery with Gemini, and the long idle time for the baseline.

- **Dashboard**: For a more advanced demonstration, a simple web-based dashboard using a tool like streamlit or dash could show the status of each worker node in real-time (e.g., training, checkpointing, failed, recovering) and plot live throughput.

Recommendation

This project is **recommended as a challenging but highly rewarding** endeavor for a two-student team in CS240. Its success hinges on disciplined project management, aggressive scoping, and early engagement with the required hardware infrastructure.

- **Feasibility**: The project is feasible within the 6-week timeframe provided the scope is strictly limited to the **Minimal Viable Reproduction (MVR)** outlined in Section 6.1. Attempting to reproduce the advanced traffic scheduling for the "zero overhead" claim is a high-risk stretch goal that should only be attempted if the MVR is completed ahead of schedule.
- **Recommended Scope**:
  - **Must-Haves**:
    - Multi-node in-memory checkpointing and replication.
    - A functional failure detection and recovery workflow.
    - Experimental results demonstrating a significant recovery time advantage over a persistent storage baseline for a moderately sized model.
  - **Nice-to-Haves**:
    - Implementation of the "mixed" placement strategy for non-divisible node counts.
    - A basic implementation of pipelined checkpointing to reduce overhead.
    - Analysis of system performance with more than one simultaneous failure.

This project offers an excellent opportunity to engage deeply with critical distributed systems concepts—including fault tolerance, replication, consistency, and coordination—within the modern, high-impact context of large-scale AI infrastructure.

Quick Reference Sheet

Category	Information
Paper	**Title**: Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints (SOSP '23) **DOI**: 10.1145/3600006.3613145

Code	**GitHub Repo**:(https://github.com/zhuangwang93/Gemini_SOSP23)
Dataset	**Name**: Wikipedia-en Corpus **Source**: Hugging Face datasets library ('wikipedia', '20220301.en')
Key Dependencies	**PyTorch**: 1.13 **DeepSpeed**: v0.7.3 **CUDA**: 11.6 **NCCL**: v2.14.3 **etcd**: v3.5
Minimum Hardware	**Nodes**: 4 **GPUs/Node**: 2+ (32GB+ VRAM) **CPU RAM/Node**: 128GB+ **Network**: 100Gbps with RDMA
Primary Claim	>13x faster failure recovery with zero training overhead.
MVR Goal	Demonstrate significantly faster recovery from in-memory checkpoints vs. NFS baseline after injecting a failure.

Works cited

1. Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints - Rice University, accessed October 29, 2025, https://www.cs.rice.edu/~eugeneng/papers/SOSP23.pdf
2. Gemini: Fast Failure Recovery in Distributed ... - Zhuang Wang, accessed October 29, 2025, https://zhuangwang93.github.io/docs/Gemini_SOSP23.pdf
3. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints, accessed October 29, 2025, https://openreview.net/forum?id=Jdtrg3Gthy
4. Zhuang Wang zhuangwang93 - GitHub, accessed October 29, 2025, https://github.com/zhuangwang93
5. Accepted Papers - SOSP 2023 - Symposium on Operating Systems Principles, accessed October 29, 2025, https://sosp2023.mpi-sws.org/accepted.html
6. Zhuang Wang, accessed October 29, 2025, https://zhuangwang93.github.io/
7. Artifact Evaluation, accessed October 29, 2025, https://sysartifacts.github.io/sosp2023/index
8. Call for Papers - SOSP 2023 - Symposium on Operating Systems Principles, accessed October 29, 2025, https://sosp2023.mpi-sws.org/cfp.html
9. Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints [pdf], accessed October 29, 2025, https://news.ycombinator.com/item?id=39074646
10. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints | Request PDF - ResearchGate, accessed October 29, 2025, https://www.researchgate.net/publication/374920379_GEMINI_Fast_Failure_Recovery_in_Distributed_Training_with_In-Memory_Checkpoints
11. A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN

Training with Reconfigurable Parallelism - USENIX, accessed October 29, 2025, https://www.usenix.org/system/files/atc25-lian.pdf
12. More-efficient recovery from failures during large-ML-model training - Amazon Science, accessed October 29, 2025, https://www.amazon.science/blog/more-efficient-recovery-from-failures-during-large-ml-model-training
13. Rice CS and Amazon researchers reveal GEMINI at SOSP in Germany | Computer Science, accessed October 29, 2025, https://csweb.rice.edu/news/rice-cs-and-amazon-researchers-reveal-gemini-sosp-germany
14. Fault tolerance in distributed systems using deep learning approaches - PMC - NIH, accessed October 29, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC11706390/
15. Application checkpointing - Wikipedia, accessed October 29, 2025, https://en.wikipedia.org/wiki/Application_checkpointing