LayerCheck: Adaptive Layer-wise Checkpointing for Large Language Model Post-training

Minqiu Sun*, Xin Huang†, Luanzheng Guo‡, Nathan R. Tallent‡, Kento Sato†, Dong Dai*
* University of Delaware, Newark, United States
Email: mqsun@udel.edu, dai@udel.edu
† RIKEN Center for Computational Science, Kobe, Japan
Email: xin.huang@riken.jp, kento.sato@riken.jp
‡ Pacific Northwest National Laboratory, Richland, United States
Email: lenny.guo@pnnl.gov, Nathan.Tallent@pnnl.gov

***Abstract*—With the rising computational and monetary costs of training large language models (LLMs), checkpointing—periodically storing model states for recovery—becomes essential for fault tolerance. Conventional checkpointing entails a severe trade-off between checkpoint frequency (I/O overhead) and computational recovery (recovery time). State-of-the-art approaches mitigate this cost through pipelining checkpoint I/Os, differential checkpointing, or in-memory persistence, yet none leverage the distinct characteristics of LLM training dynamics, where model weight updates are non-uniformly distributed across transformer layers. This observation implies that saving all weights each time might not be efficient. Inspired by this observation, we present `LayerCheck`, a layer-wise adaptive checkpointing framework that selectively persists layers whose updates exceed a threshold. This design avoids periodic I/O bursts by distributing layer-wise checkpoint writes over time, resulting in smoother and more balanced I/O profiles. Upon recovery, `LayerCheck` reconstructs a mixed-timestamp composite model state by aggregating the most recently persisted versions of each layer together with their matching optimizer states. Under a bounded per-layer staleness guard, this introduces a controlled perturbation: under standard Adam assumptions it adds a bounded staleness term, and empirically the post-restart loss deviates from the failure-free trajectory by at most 0.54%. Empirical results on multiple open-source LLMs with different datasets further demonstrate that recovered models preserve the original convergence behavior and accuracy while substantially reducing checkpoint overheads. Specifically, `LayerCheck` achieves up to 22.6× reduction in total checkpoint size and 1.31× reduction in end-to-end training time compared to state-of-the-art systems, significantly lowering the cost of checkpointing.**

***Index Terms*—Checkpointing, Large language models, Fault-tolerant computing.**

I. Introduction

Large language models (LLMs) have undergone rapid and unprecedented advancements in recent years. Training these models, including both pre-training and post-training, has become increasingly time-consuming. It's well known that pre-training, which establishes the model's foundational capabilities, can require thousands of GPUs running continuously for weeks or months [1]. Post-training, which is the main focus of this study, still demands hundreds of GPUs over days or weeks [2], [3]. Such prolonged runtimes render failures unavoidable. Gandhi et al. [4] report that failures can occur as frequently as every 45 minutes in large-scale training.

Checkpointing, originally developed in HPC to protect long-running scientific applications [5]–[9], is the standard fault-tolerance mechanism for large-scale LLM training [10], [11]. Mainstream frameworks such as DeepSpeed [12] and FSDP [13] typically use fixed-interval checkpointing, periodically saving the full training state, including model weights, optimizer states, and RNG/scheduler states, to persistent storage for failure recovery [4].

However, fixed-interval checkpointing introduces substantial overhead in I/O, storage, and, most importantly, end-to-end training time [14]. It fundamentally trades checkpoint overhead for recovery cost: infrequent checkpoints reduce runtime overhead but increase recomputation after failures, whereas frequent checkpoints shorten rollback distance but incur higher I/O and synchronization cost. This trade-off worsens at scale as job-level MTBF decreases, forcing shorter checkpoint intervals. Prior work reports that checkpointing can consume up to 12% of total training time, and as much as 43% in extreme cases [15].

Recent work on LLM checkpointing overhead falls into three groups. (1) **Pipeline optimization.** GEMINI [16] uses in-memory checkpointing to overlap GPU–storage transfers with training, enabling high-frequency snapshots. DataState-LLM [17] further decouples checkpointing from computation via lazy synchronization. These approaches, however, struggle as model sizes grow and the GPU–storage gap widens. (2) **Offline compression.** Delta-DNN [18] and ExCP [19] apply lossy differential compression across consecutive checkpoints to cut storage cost. However, full checkpoints must still be materialized before compression, leaving I/O volume unchanged. (3) **Selective persistence.** Online differential checkpointing persists only changes: Check-N-Run [20] targets recommendation models, while LowDiff [21] extends the idea to LLMs by saving only per-iteration gradients, at the cost of recomputation and lineage tracking. Amber [22] further explores selective incremental checkpointing for LLM training by using a bitmap to identify changed parameters and persist only those parameters together with their optimizer state. While this can reduce written bytes, it also requires fine-grained per-parameter tracking and bitmap maintenance, whose metadata and runtime overhead grows with model size. All of these still

stall training and inflate recovery cost.

Existing selective/differential persistence schemes typically decide what to write based on a global rule (e.g., fixed frequency, fixed partitions, or explicit deltas). In contrast, we exploit an empirical property that is repeatedly observed in LLM post-training: *different transformer layers evolve at different paces across steps* [23]. This layer-wise update asynchrony suggests that writing the *entire* model at every checkpoint is often redundant: at many steps, only a subset of layers experiences meaningful drift.

Motivated by this observation, we propose `LayerCheck`, a *layer-wise adaptive checkpointing* mechanism. `LayerCheck` tracks the update significance of each layer during training and persists only those layers whose accumulated drift is sufficiently large. By writing partial model states in most checkpoints, `LayerCheck` substantially reduces checkpoint I/O volume while keeping recovery practical: reconstruction simply merges the most recently persisted state of each layer into a composite checkpoint.

Selective persistence introduces an inherent trade-off at recovery time: layers not written in the most recent checkpoint are restored from earlier iterations, yielding a *mixed-timestamp* model state. Following the perturbation-based view of partial recovery [24], we interpret this mismatch as a bounded perturbation from the ideal failure-free trajectory. Let t_i denote the last iteration at which layer i is persisted, and let τ be the failure iteration. We define the *layer age* (staleness) of layer i at recovery as $\tau - t_i$. To prevent any layer from becoming arbitrarily stale, `LayerCheck` enforces a staleness guard with a user-specified budget $S_{\max}$: if $\tau - t_i \geq S_{\max}$, the system forces persistence of layer i even when its drift is below the update threshold. Under this bounded-staleness invariant, we show that recovery does not change the asymptotic convergence rate (§III), and we further validate empirically that post-recovery training closely matches full-checkpoint baselines (§IV).

`LayerCheck` is guided by three practical design principles. *First,* it makes persistence decisions based on *accumulated* layer drift over multiple iterations, so that gradual but meaningful changes are not missed. *Second,* instead of tracking per-parameter updates, it maintains a lightweight *per-layer* update statistic computed from already-available training signals, reducing the tracking cost to a small vector of layer-level summaries. *Third,* it uses a threshold K tied to the magnitude of training updates, optionally in a learning-rate-aware manner, while the staleness guard $S_{\max}$ provides an explicit safeguard independent of threshold tuning.

We evaluate `LayerCheck` on multiple open-source LLMs and measure checkpoint I/O volume, storage footprint, end-to-end training time, and recovery time under failure injection. Compared to state-of-the-art checkpointing systems, `LayerCheck` reduces the *total* checkpoint size by up to **22.6×** and improves total training time by up to **1.31×** without measurable degradation in model quality in our evaluated settings. Recovery is also faster: by reconstructing the composite state from the most recently persisted layers, `LayerCheck` reduces recovery cost by up to **3.4×**. In addition, we further study projected behavior under more failure-prone scale-out regimes using a simulation parameterized by measured system components. These results indicate that layer-wise selective persistence can deliver substantial system-level savings without sacrificing training outcomes. Our contributions are threefold:

- **Layer-wise selectivity from training dynamics.** We identify *layer-wise update asynchrony* as an exploitable property of LLM post-training and propose `LayerCheck`, a layer-wise selective checkpointing mechanism that persists full layer states only when their accumulated updates are significant, with negligible overhead.
- **Recovery fidelity under mixed-timestamp checkpoints.** We provide theoretical and empirical evidence that LayerCheck's mixed-timestamp checkpoints preserve training fidelity: post-restart loss deviates from the failure-free trajectory by at most **0.54%**, and downstream benchmark scores match failure-free fine-tuning.
- **Fast, practical recovery without replay.** We design a per-layer checkpointing and reconstruction workflow that restores the composite, full checkpoint effectively, avoiding the retraining and replay overheads common in fixed-frequency checkpointing.

The rest of the paper is organized as follows. §II introduces the necessary background. In §III we describe the design and implementation of `LayerCheck`. We present the extensive experimental results in §IV. §V discusses the closely related work. §VI concludes this study and discusses future work.

The source code of `LayerCheck` is publicly available at https://github.com/DIR-LAB/LayerCheck.

II. Background

Large language models (LLMs) have demonstrated remarkable power in various domains [25]–[29]. LLM training, like other neural networks, involves a forward pass, loss computation, a backward pass, and parameter update. Here, we review the key fundamentals that motivate our layer-wise checkpointing: (i) the structural components of LLMs, (ii) the unbalanced updating characteristics intrinsic to LLM training.

A. Problem Setup and Goals

We consider large language model (LLM) post-training under a fail-stop failure model, where the job may terminate unexpectedly and must be restored from persisted checkpoints. A checkpoint includes the model weights, any persistent optimizer state associated with them, when present (e.g., Adam/AdamW momentum), and some other configuration files of the model's architecture, training arguments, and random number generator, which are necessary to resume training. Unlike traditional checkpointing that snapshots a globally consistent state at a single iteration, `LayerCheck` allows *mixed-timestamp* restoration, where each layer may be restored from its most recent available checkpoint. Our goal is to reduce end-to-end training time under failures by reducing checkpoint I/O overhead while preserving post-recovery training behavior and final model quality.

B. Layer Structure of LLMs

In Figure 1, we show the layer structure of Qwen2.5-7B model [30]. Here, we partition trainable parameters into N layer groups at the granularity of modules: `embed_tokens`, each `transformer block`$_i$ ($i = 1, \ldots, B$), `normalization`, and `lm_head`, yielding $N = B + 3$. Tokens are first embedded to a vector, then propagated through 28 consecutive transformer layers. Then, after layer normalization to eliminate internal covariate shift, the hidden representations are projected back to the vocabulary space to produce logits in the `lm_head` layer; finally, a softmax yields output probabilities. Note that in smaller models, the `lm_head` layer may be weight-tied to `embed_tokens` to reduce parameter count [31].

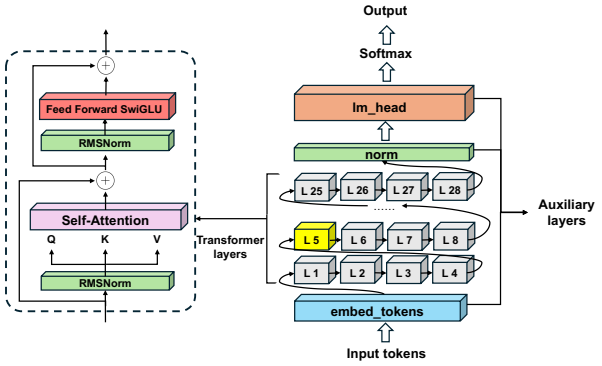


Fig. 1: The layer-wise structure in a Qwen2.5-7B model.

C. Layer-wise Unbalanced Updating

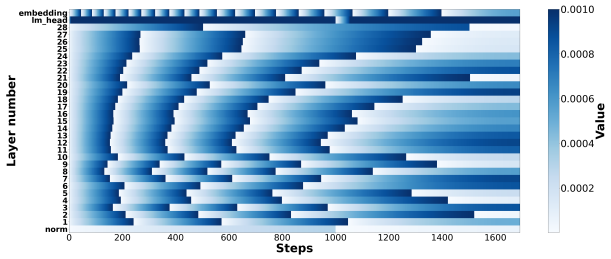


Fig. 2: Temporal and layer-wise heterogeneity of model updates.

Many existing studies have observed the unbalanced updates to different layers of the LLM models at different training phases [23], [32], [33]. To better showcase that, we visualize the per-layer weight updates for an open-source Qwen2.5-7B model in Figure 2. Here, the x-axis represents training steps, while the y-axis enumerates the model's all 31 layers: 28 transformer layers and 3 auxiliary layers. The heatmap color intensity corresponds to the weight updates: darker regions indicate larger accumulated weight updates.

Here, each layer's accumulated weight update is reset to 0 once it exceeds the threshold, as if the layer had been checkpointed. The wave-like visualization clearly highlights the variation in update magnitudes across layers and steps, underscoring the non-uniformity of layer-wise learning dynamics during training.

Despite the substantial variability in actual weight updates across layers and steps, current checkpointing strategies uniformly save all layers. This uniform approach could be inefficient, as the most significant model updates may happen asynchronously across layers. Such unbalanced updates motivate our layer-wise checkpointing strategy.

III. Design and Implementation

This section presents the design and implementation of `LayerCheck`, a layer-wise selective checkpointing mechanism for LLM post-training. `LayerCheck` is motivated by a repeatedly observed training-system property: different Transformer layers drift at different paces, making full-model snapshots frequently redundant.

At a high level, `LayerCheck` operates in three stages. (1) *Online monitoring:* during training, it maintains a lightweight per-layer drift statistic and decides which layers to persist at each checkpoint opportunity. (2) *Selective persistence:* when a layer is selected, `LayerCheck` writes the *full layer state*, including both weights and optimizer tensors, so that recovery does not require delta replay. (3) *Composite recovery:* after a failure, it reconstructs a complete training state by merging the most recently persisted version of each layer, which may originate from different iterations.

The central fidelity concern is *mixed-timestamp recovery*: selective persistence yields a checkpoint where different layers (and their Adam moments) can be restored from different steps. To control this effect, we enforce a simple bounded-staleness invariant ($S_{\max}$), which also underpins the convergence argument in §III-F.

A. Overall Workflow

Figure 3 illustrates the workflow along the training timeline. The x-axis is the iteration index. Each block corresponds to the full training state at that iteration, while each box within the block represents a layer state consisting of (i) the layer parameters and (ii) the optimizer tensors associated with those parameters (e.g., Adam first/second moments under AdamW).

From full snapshots to per-layer persistence. Conventional checkpointing persists the entire model and optimizer state at each checkpoint. In contrast, `LayerCheck` evaluates every layer at runtime and persists only a subset of layers at most checkpoints. A layer is selected when its *accumulated normalized drift* since its last persistence exceeds a threshold. Intuitively, fast-changing layers are refreshed more frequently, while slow-changing layers are skipped, reducing checkpoint I/O and storage.

Composite checkpoint at recovery (no rollback). When a failure occurs after iteration t_{fail} completes, `LayerCheck` recovers by selecting, for each layer ℓ, the most recent successfully persisted layer checkpoint at time $t_\ell \le t_{\text{fail}}$ and assembling these layer states into a full training checkpoint. This composite checkpoint can contain mixed timestamps across layers; however, because each layer is restored together with

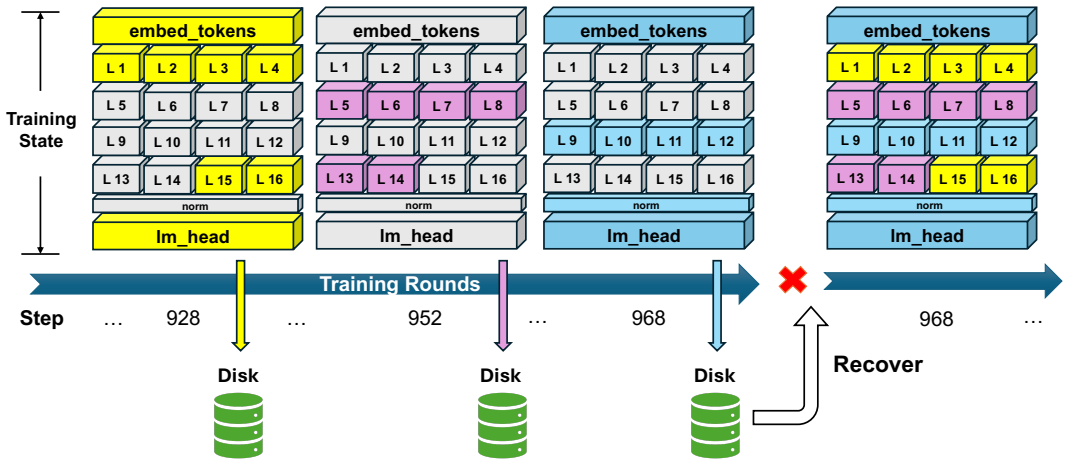


Fig. 3: `LayerCheck` workflow. Layers are persisted at different iterations based on accumulated drift; recovery assembles the most recent persisted state of each layer into a composite checkpoint.

its corresponding optimizer tensors, the recovered optimizer state remains *internally consistent within each layer*. Training then resumes directly at iteration $t_{\text{fail}}+1$ without rolling back to an older full-model snapshot.

B. Layer-wise Monitoring and Checkpointing

This subsection details how `LayerCheck` quantifies per-layer update significance and triggers layer persistence online.

Accumulated Per-layer Updates. Checkpointing serves fault tolerance, so our goal is to ensure that resuming from a recovered composite checkpoint yields statistically indistinguishable model quality compared to a failure-free run. Identifying, at checkpoint time, which individual parameter updates matter most is generally infeasible. Moreover, tracking per-parameter changes would require auxiliary state comparable to the model size. `LayerCheck` therefore tracks drift at the granularity of layers.

For each layer ℓ, we maintain an accumulator Δ_ℓ initialized to 0. At iteration t, after the optimizer computes the parameter update, we estimate a normalized update magnitude:

$$u_\ell(t) = \frac{\text{mean}\big(|W_\ell(t) - W_\ell(t-1)|\big)}{\text{mean}\big(|W_\ell(t-1)|\big) + \varepsilon_w}, \tag{1}$$

where $W_\ell(t)$ denotes the parameter tensor(s) of layer ℓ at iteration t, $\text{mean}(\cdot)$ is taken element-wise across that layer's parameters, and ε_w is a small constant for numerical stability. We accumulate drift over time:

$$\Delta_\ell \leftarrow \Delta_\ell + u_\ell(t). \tag{2}$$

We use the absolute value to obtain an upper-bound style estimate (conservative counting): it avoids cancellation across coordinates and biases toward persisting slightly more layers rather than missing meaningful drift. Using the mean reduces sensitivity to outliers and yields a stable layer-level signal.

Thresholding with learning-rate adaptation. A layer is persisted when its accumulated drift exceeds a threshold:

$$\Delta_\ell \geq K(t). \tag{3}$$

Instead of a single fixed threshold, `LayerCheck` uses a base threshold K_0 and scales it with the current learning rate to account for common LR schedules (warmup/decay) under AdamW. Concretely, we set $K(t) = K_0 \cdot s(\text{lr}(t))$, where $s(\cdot)$ is a lightweight scaling function derived from the LR schedule (e.g., proportional to the current LR relative to the initial LR). When a layer is persisted, `LayerCheck` writes the full layer state and resets the accumulator:

$$\Delta_\ell \leftarrow 0. \tag{4}$$

C. Bounding Mixed-Timestamp Staleness

Selective persistence implies that at recovery time, different layers may be restored from different iterations. We quantify this effect via *staleness*. If a failure occurs at iteration τ, and layer ℓ was last persisted at iteration $t_\ell \leq \tau$, then the layer staleness is $\tau - t_\ell$. We additionally define an overall (size-weighted) checkpoint staleness metric for reporting, where larger layers contribute proportionally more.

While threshold-based persistence typically keeps most layers fresh, some layers (e.g., normalization-related tensors) can drift very slowly and may otherwise remain unpersisted for long periods. To prevent pathological cases and to make recovery robust, `LayerCheck` enforces a staleness guard:

$$\max_\ell(\tau - t_\ell) \leq S_{\max}. \tag{5}$$

If a layer reaches staleness $S_{\max}$, `LayerCheck` forces persistence of that layer at the next checkpoint opportunity even if its drift accumulator has not crossed the threshold, and then resets its accumulator Δ_ℓ to 0. This guard applies to the *maximum per-layer staleness* and serves as a worst-case safety bound. In practice, however, the checkpoint-wide mismatch is better reflected by a parameter-size-weighted average staleness, since a large maximum can be caused by a very small tensor (e.g., normalization layer) whose contribution to the full checkpoint is negligible. In our experiments, with $K_0 = 10^{-3}$, most layers are naturally refreshed roughly every ~ 200 iterations,

the parameter-weighted average staleness remains low, and the bound $S_{\max} = 1000$ acts as a rarely triggered safety backstop. It is shown in Figure 4 that the maximum (blue line) is shaped by the normalization layer mainly from step 200 to step 1000, but the parameter-size-weighted average staleness (red line) reflects the mean staleness of all the layers better. This invariant is also the key system property used in the convergence analysis (§III-F).

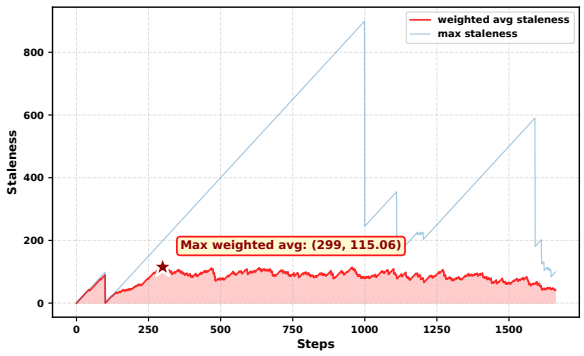


Fig. 4: Aggregate staleness during Qwen2.5-7B fine-tuning with $K_0 = 10^{-3}$. The annotation marks the maximum parameter-weighted average staleness.

D. Implementation on PyTorch/DeepSpeed

We implement `LayerCheck` on top of PyTorch and DeepSpeed ZeRO-3 [12]. The design goal is *non-intrusive integration*: between failures, training executes the standard AdamW update; `LayerCheck` only augments the optimizer with lightweight drift accounting and selectively filters what is serialized during checkpointing.

1) Layer-aligned optimizer state layout.: A practical challenge is that optimizer states are stored as flattened tensors and parameter groups, which do not directly expose a layer boundary. To enable per-layer persistence and recovery, we reconstruct the optimizer parameter groups so that they mirror the model's layer-wise hierarchy while preserving the original weight-decay policy. In our implementation, each Transformer layer is split into two parameter groups (with-weight-decay and without-weight-decay), while auxiliary layers that contain only one type are mapped to a single group. Because this grouping order is consistent given the model configuration (number of Transformer blocks and weight-decay settings), we can locate the optimizer shard corresponding to a layer during serialization and reconstruction.

2) Reusing the optimizer update to compute drift.: Computing $W(t) - W(t-1)$ naively would introduce extra tensor copies. However, we piggyback drift computation on the optimizer update path to avoid extra tensor copies and keep monitoring overhead low.

3) Threshold defaults and special-casing embeddings.: With LR-adaptive thresholding, we set the base threshold to $K_0 = 10^{-3}$ for most layers. This scale is consistent with prior observations that 10^{-3}-level relative parameter drift is a meaningful unit during fine-tuning [22], [34]. Embedding layers often exhibit smaller updates; to avoid forcing frequent persistence of embeddings, we use a larger base threshold for embeddings (e.g., 10^{-2}).

We also disable selective persistence during an initial warm-up window (first 200 steps) and take a full base checkpoint that serves as a fallback for rarely-updated layers.

E. Recovery from `LayerCheck` Checkpoints

When a failure occurs at step t_{fail}, `LayerCheck` reconstructs a full checkpoint in three steps.

1) For each layer ℓ, find the most recent persisted layer checkpoint timestamp $t_\ell \le t_{\text{fail}}$.
2) Load the corresponding layer state $(W_\ell(t_\ell), O_\ell(t_\ell))$, where O_ℓ denotes the optimizer tensors associated with W_ℓ (e.g., Adam moments). If a layer has no prior layer checkpoint, fall back to the base full checkpoint at t_{base}.
3) Assemble a complete training state by placing each loaded layer state back into its original layout.

Training then resumes from the reconstructed composite checkpoint.

Metadata-driven reconstruction. During training, `LayerCheck` records the persistence history of each layer in each step. Upon failure, recovery derives a per-layer reconstruction plan from this metadata and assembles the most recent valid layer states into a DeepSpeed-compatible full checkpoint.

F. Convergence Analysis of `LayerCheck`

Recovery replaces $(\theta_\tau, m_\tau, v_\tau)$ with a mixed-timestamp composite state. Following ExCP [19], we provide a proof-of-concept analysis showing that bounded staleness adds a bounded term to Adam's standard regret bound without changing its asymptotic average-regret rate.

1) Setup and assumptions: We follow the standard Adam regret analysis of Kingma & Ba [35], which assumes a convex per-iteration loss. Let $\theta_t \in \mathbb{R}^d$ be the parameter vector, $f_t(\cdot)$ be the per-iteration loss, and $g_t = \nabla f_t(\theta_t)$. Adam maintains m_t and v_t using $\beta_1, \beta_2 \in (0,1)$. We also adopt standard boundedness assumptions used in Adam analyses [35]:

$$\|g_t\|_2 \le G, \qquad \|g_t\|_\infty \le G_\infty, \quad \forall t, \tag{6}$$

$$\|\theta_n - \theta_m\|_2 \le D, \quad \|\theta_n - \theta_m\|_\infty \le D_\infty, \quad \forall m, n \in \{1, \dots, T\}. \tag{7}$$

We assume that the recovered composite state $\tilde{\theta}_\tau$ lies in the same bounded domain and use the standard Adam initialization $m_0 = v_0 = 0$.

Following ExCP [19], we use $\alpha_t = \alpha/\sqrt{t}$, $\beta_{1,t} = \beta_1 \lambda^{t-1}$ for $\lambda \in (0,1)$, and assume $\beta_1^2/\sqrt{\beta_2} < 1$. For simplicity, we omit hats and absorb Adam's bounded bias-correction factors into the constants.

2) Composite recovery model: Suppose a failure occurs at iteration τ. For each layer $\ell \in \{1, \dots, L\}$, let $t_\ell \le \tau$ be the iteration of its most recent persisted layer checkpoint. Because

LayerCheck stores weights together with their layer-local optimizer tensors, recovery forms

$$\tilde{\theta}_\tau \triangleq \bigoplus_{\ell=1}^{L} \theta_{t_\ell}^{(\ell)}, \quad \tilde{m}_\tau \triangleq \bigoplus_{\ell=1}^{L} m_{t_\ell}^{(\ell)}, \quad \tilde{v}_\tau \triangleq \bigoplus_{\ell=1}^{L} v_{t_\ell}^{(\ell)}. \quad (8)$$

Let $t(i)$ denote the timestamp of the layer containing coordinate i. Under the staleness guard, $\tau - t(i) \le S_{\max}$ for all i.

Lemma 1 (Second-moment deviation over $S_{\max}$ steps)**.** *Assume* (6) *and* $v_0 = 0$*. For any* $s \in [\tau - S_{\max}, \tau]$ *and every coordinate* i*,*

$$|v_{\tau,i} - v_{s,i}| \le 2(1 - \beta_2^{S_{\max}})G_\infty^2. \quad (9)$$

Proof. Let $k = \tau - s \le S_{\max}$. Unrolling $v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2)g_{t,i}^2$ and using $0 \le v_{s,i} \le G_\infty^2$ gives

$$|v_{\tau,i} - v_{s,i}| \le 2(1-\beta_2^k)G_\infty^2 \le 2(1-\beta_2^{S_{\max}})G_\infty^2.$$

□

The remaining recovered states are also bounded:

$$\|\tilde{\theta}_\tau - \theta_\tau\|_\infty \le D_\infty, \qquad \|\tilde{m}_\tau - m_\tau\|_\infty \le 2G_\infty. \quad (10)$$

Theorem 1 (Proof-of-concept regret bound under bounded staleness)**.** *Assume* (6)–(7) *and the staleness guard* (5)*. Consider one recovery at iteration* τ *and denote the regret of the recovered run by* $\tilde{R}(T)$*. Following the ExCP-style proof-of-concept decomposition, we retain the standard bounded-domain and moment terms for* θ *and* m *and isolate the additional second-moment term:*

$$\tilde{R}(T) \le R_{Adam}(T) + \Delta_{\text{stale}}(T), \quad (11)$$

where $R_{Adam}(T) = O(\sqrt{T})$*. Applying Lemma 1 and* $|\sqrt{a} - \sqrt{b}| \le \sqrt{|a-b|}$ *gives*

$$\begin{aligned}\Delta_{\text{stale}}(T) &\triangleq \frac{D^2\sqrt{T}}{2\alpha(1-\beta_1)} \sum_{i=1}^{d} \left|\sqrt{v_{\tau,i}} - \sqrt{v_{t(i),i}}\right| \\ &\le \frac{D^2 d G_\infty}{2\alpha(1-\beta_1)} \sqrt{2T(1-\beta_2^{S_{\max}})} = O(\sqrt{T}).\end{aligned} \quad (12)$$

Consequently, the average regret satisfies $\tilde{R}(T)/T = O(1/\sqrt{T})$.

3) Multiple failures.: If recoveries occur at iterations $\tau_1 < \cdots < \tau_K \le T$, the corresponding staleness terms add up. When K is fixed independently of T, their sum remains $O(\sqrt{T})$, so the asymptotic average-regret rate is unchanged.

IV. Evaluation

This section evaluates LayerCheck against the paper's central claim: *layer-wise selective persistence can reduce checkpoint cost while maintaining safe and effective recovery*, even though the recovered training state is assembled from layers persisted at different timestamps. We therefore first examine whether LayerCheck's mixed-timestamp recovery preserves stable post-failure training behavior. At system-level, checkpoint-based fault tolerance incurs two kinds of overhead: checkpoint overhead during training and recovery overhead after a failure. The former affects steady-state training efficiency, while the latter determines how much time is lost before useful training resumes. Accordingly, we evaluate LayerCheck from two perspectives: the checkpoint cost it introduces during training and the recovery cost it incurs after failures. We focus on end-to-end training impact under I/O pressure and on recovery behavior under comparable recovery quality. We organize experiments around three questions.

- First, when a failure occurs and we restart from a composite (mixed-timestamp) checkpoint, does training remain stable and converge to the same quality as failure-free training?
- Second, in steady state, how much wall-clock time and storage can we save by avoiding unnecessary writes of slow-changing layers?
- Third, after a failure, how much time do we save in recovery once we control for the amount of rollback/staleness each method permits?

A. Experimental Setup

1) Testbed: All experiments run on a single 8-GPU server with NVIDIA A100 40GB GPUs and two AMD EPYC 7713 CPUs (2,048 GB DRAM). The storage backend is a 100 TB Lustre parallel file system. And our evaluated node uses one Mellanox ConnectX-6 InfiniBand HCA, connected at HDR 200 Gbps.

We enable DeepSpeed ZeRO Stage-3 so that model states and optimizer tensors are sharded across GPUs. All methods use AdamW [36]. For uniform post-restart evaluation, each checkpoint additionally stores a BF16 copy of the model weights. This copy does not change LayerCheck's selective persistence decisions; it simply ensures that evaluation reads the same representation across methods.

We evaluate three open-source LLMs spanning small to medium scale: Llama-3.2-1B [1], Qwen-2.5-3B, and Qwen-2.5-7B [30]. We consider two supervised fine-tuning (SFT) workloads: MedQA [37] (medical domain) and OpenThoughts [38] (reasoning). Unless otherwise stated, each run trains for 1 epoch with maximum gradient norm 1 and initial learning rate 10^{-5}. Due to GPU memory constraints, we set sequence length to 2048, micro-batch size to 1, and gradient accumulation steps to 2 across all methods.

We compare LayerCheck against CheckFreq [39], GEMINI [16], and LowDiff [21]. These baselines reflect three distinct approaches: overlap checkpoint process with model training(CheckFreq), overlapped checkpoint and in-memory checkpoint (GEMINI), and reducing bytes written via differential information (LowDiff). Together with LayerCheck, they cover different points in the checkpointing design space and are therefore compared as representative standalone systems for frequent checkpointing/recovery settings. Unless otherwise stated, we follow default configurations in the original papers.

In all experiments, "checkpoint time" refers to wall-clock time spent on the checkpointing path and cannot be overlapped with other processes, including serialization, storage I/O, and

any synchronization required by the framework to produce a consistent checkpoint. This quantity directly affects training throughput, because time spent in checkpointing cannot be used for model training.

2) Evaluation Methodology: Under DeepSpeed ZeRO-3, model weights and optimizer states are sharded across ranks, and each rank performs checkpointing independently. The core behavior of LayerCheck — per-layer drift tracking, selective persistence, and composite recovery — is therefore determined by per-rank activity, enabling it to be faithfully evaluated in a single-node sharded setting where each rank's state mirrors the per-rank size in larger-scale parallel training. We adopt this methodology following recent LLM checkpointing systems [22] and extend to scale-out regimes via a data-driven simulation in IV-E, consistent with GEMINI [16].

We evaluate recovery fidelity using both post-restart loss trajectories and final downstream benchmark scores.

B. Recovery Fidelity: Training After Mixed-Timestamp Recovery

The most distinctive aspect of `LayerCheck` is composite recovery: after a failure, the recovered training state is assembled by choosing, for each layer, the most recent persisted version of that layer's weights and associated optimizer tensors. Because layers are persisted at different iterations, the reconstructed checkpoint inevitably contains mixed timestamps. This raises a concrete risk: even if each individual layer is internally consistent (weights and optimizer moments originate from the same time), the overall model state may represent a configuration that never existed during failure-free training.

`LayerCheck` addresses this risk with an explicit system invariant. Rather than attempting to force all layers to share the same timestamp at recovery, it enforces a bounded-staleness guard $S_{\max}$ that ensures no layer can be arbitrarily stale relative to the failure point. This invariant is inspectable during runtime and is also the key assumption used in our convergence discussion (§III-F).

To evaluate recovery fidelity, we examine the training loss trajectories after restart. A mixed-timestamp checkpoint can perturb both parameters and optimizer moments; if this perturbation were large or adversarial, we would expect an instability divergence, or a widening gap indicating that recovery pushes training into a different regime.

For Qwen2.5-7B fine-tuning on OpenThoughts, we inject failures at the point of *maximum parameter-weighted average staleness* for each configuration.

This is the most globally challenging moment for composite recovery because it maximizes the checkpoint-wide mismatch across layers while the maximum per-layer staleness still remains bounded by $S_{\max}$. For $K{=}10^{-3}$, 10^{-2}, and 10^{-4}, these points occur at steps 299, 931, and 992, respectively. We use the failure-free run as the visual reference, since exact full-state (lossless) recovery reproduces the failure-free trajectory by construction.

Figure 5a first examines a fixed operating point ($K{=}10^{-3}$) under cascading composite recoveries. The first failure is injected at step 299, followed by two additional failures at steps 799 and 1299, with training resuming from the reconstructed composite checkpoint after each one. The recovered trajectory remains close to the failure-free baseline throughout, with no visible sign of compounding instability or progressive drift. We then measure the post-restart training's final loss: the maximum absolute deviation from the failure-free trajectory is 0.0024 (**0.54%** relative), well within step-to-step training noise. This result suggests that the perturbation introduced by mixed-timestamp recovery does not accumulate into runaway optimization error across repeated failures, consistent with the multi-failure analysis in §III-F.

Figure 5b next shows how recovery fidelity changes with the drift threshold K under single-failure recovery. Across all three thresholds, the resumed runs closely track the baseline after restart. The patterns also match the intended trade-off. Smaller K refreshes layers more aggressively, so the post-recovery deviation is smaller. Larger K skips more writes, so the recovered state can be more stale and may exhibit a slightly higher loss band immediately after restart; however, this deviation remains bounded rather than amplifying with continued optimization.

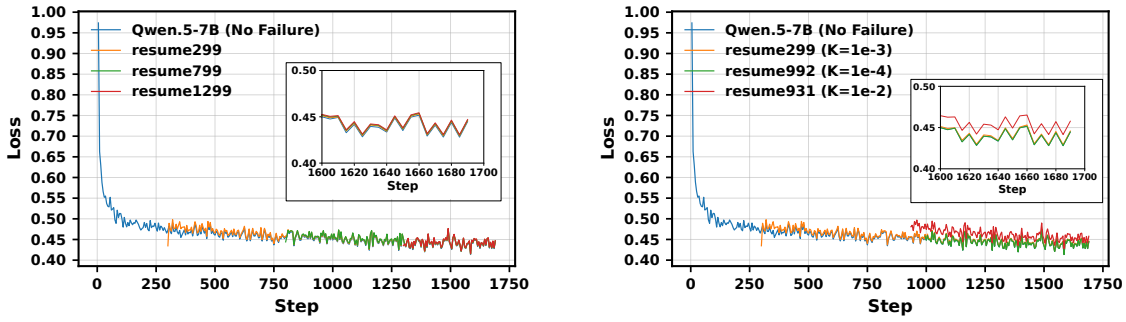
Taken together, Figure 5 shows that LayerCheck recovery is robust in two complementary senses: it remains stable under repeated failures at a fixed operating point, and it remains well behaved across the freshness–efficiency trade-off induced by different K values. This is exactly the behavior the $S_{\max}$ guard is designed to enable.

C. Checkpoint Efficiency: End-to-End Time and Storage

We next quantify steady-state checkpoint cost during model training. Because `LayerCheck` targets redundant persistence of slow-changing layers, we evaluate an intentionally I/O-stressful checkpoint-per-iteration regime, as also used by prior checkpointing systems [16], [17], [21]. We use this setting to evaluate whether each system can sustain high-frequency checkpointing and to expose differences in written bytes, serialization work, and overlap behavior, rather than modeling every production workload.

1) Time overhead: Figure 6 reports end-to-end runtime for 1 epoch on OpenThoughts. `LayerCheck` achieves the shortest training time across all three models, with overhead of only 11.7–20.0% relative to checkpoint-free training (t_{net}), **1.31**× faster than LowDiff on Llama3.2-1B and 3.76× faster than GEMINI on Qwen2.5-7B, with larger gaps for CheckFreq and DeepSpeed default.

Checkpoint fraction alone does not explain this. `LayerCheck`'s checkpoint fraction (10.7–12.0%) is comparable to LowDiff's (9.3–15.9%), yet end-to-end times differ substantially. The reason is that under per-iteration checkpointing, the baselines must drain full-model-scale checkpoint work each step. Once this work exceeds one training iteration, overlap breaks down and visible stalls emerge. This is exactly where `LayerCheck` gains its systems advantage: under checkpoint pressure, it is not



(a) Loss trajectories under cascading composite recoveries (fixed $K = 10^{-3}$).

(b) Loss trajectories under varying drift thresholds K (single failure).

Fig. 5: Recovery fidelity of LayerCheck on Qwen2.5-7B (OpenThoughts dataset).

enough to pipeline checkpointing better; reducing the amount of state and serialization work itself is crucial.

Crucially, this overhead is dominated by checkpoint I/O (the hatched portion of each LayerCheck bar). Subtracting the checkpoint time, the remaining training time essentially matches t_{net} across all three models, indicating that LayerCheck's per-iteration drift tracking adds negligible overhead to the optimizer update path itself. This is consistent with our piggyback design (§III-D), where drift computation reuses tensors already materialized during the AdamW update rather than introducing additional passes over model parameters.

The shared Lustre backend further amplifies these effects, but the qualitative trend is not specific to Lustre. Methods that write less state and avoid repeated full-model rewrites are less exposed to storage variability and I/O backpressure.

Beyond reducing steady-state training stalls, this result also has an important implication for failures. By lowering the cost of frequent persistence, `LayerCheck` makes it more practical to keep restart points close to the failure point. We therefore examine recovery overhead under comparable recovery quality in the next subsection.

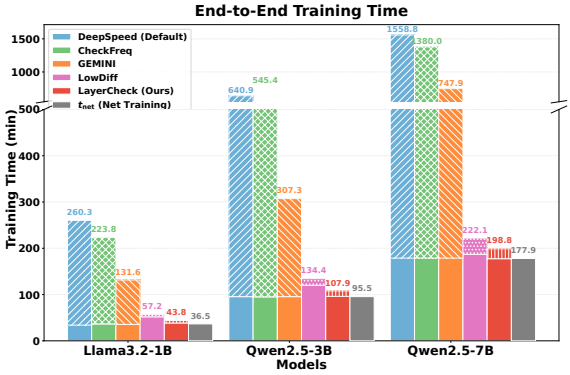


Fig. 6: End-to-end training time for 1 epoch on OpenThoughts dataset with checkpoint-per-iteration. Hatched portions denote non-overlapped checkpointing time in training.

2) I/O volume: We further quantify I/O volume using checkpoint storage footprint. To make this concrete, we report total checkpoint size over the run, the number of checkpoint events (directories), and the mean checkpoint size. These capture complementary aspects of storage pressure: total footprint reflects cumulative write volume, while mean checkpoint size relates to per-event burstiness that affects latency and overlap.

Table I reports checkpoint statistics on OpenThoughts for $K = 10^{-3}$. The "#Ckpt Layers" column counts the total number of *layer writes* across the run, highlighting the key distinction from full-state checkpointing where each checkpoint necessarily rewrites every layer. As expected, `LayerCheck` produces many checkpoint events that contain only a small subset of layers.

The reductions in written bytes are substantial. `LayerCheck` reduces total checkpoint storage by up to **22.6×** (average $\sim 17.1\times$) relative to LowDiff, and reduces mean per-checkpoint size by up to 6.6× (average $\sim 5.0\times$). This result directly supports the paper's main systems argument: layer-level sparsity in "meaningful change" can be exploited to avoid rewriting most of the model most of the time. In turn, fewer bytes translates to fewer stalls in two ways. It reduces raw write time, and it increases the likelihood that persistence work can be pipelined without blocking the training loop, especially on storage backends with high variance in throughput.

Finally, we examine how the threshold K controls the efficiency–freshness trade-off. Table II reports the footprint across datasets for Qwen2.5-7B. A larger K persists fewer layers and yields fewer checkpoints, while a smaller K refreshes layers more aggressively and increases total footprint. We use $K = 10^{-3}$ as the default operating point because it provides a practical balance between recovery fidelity and checkpoint cost in our evaluated workloads.

D. Recovery Overhead

Recovery overhead is meaningful only when recovery quality is controlled. A method can appear "fast" simply by checkpointing much more frequently, while infrequent checkpointing can appear "slow" because it must replay more lost iterations after a failure. Per-iteration checkpointing imposes prohibitive overhead on the baselines, and even setting that

TABLE I: Checkpoint statistics on OpenThoughts ($K = 10^{-3}$). For each model, our method is listed last and highlighted in bold for readability.

Model (Layers)	Method	Ckpt Layers	Ckpt Dirs	Total Size (GB)	Mean Size (GB)
Qwen2.5-7B (31)	Default	52452	1692	180367.20	106.60
	LowDiff	–	930	19258.68	20.71
	LayerCheck	**319**	**277**	**1205.38**	**4.35**
Qwen2.5-3B (38)	Default	64296	1692	73111.32	43.21
	LowDiff	–	930	8512.85	9.15
	LayerCheck	**340**	**273**	**376.21**	**1.38**
Llama3.2-1B (18)	Default	30456	1692	29305.44	17.32
	LowDiff	–	930	3837.05	4.12
	LayerCheck	**266**	**242**	**264.82**	**1.02**

TABLE II: `LayerCheck` ablation across datasets for Qwen2.5-7B (31 layers).

Dataset	K	Ckpt Layers	Ckpts	Total Size (GB)	Mean Size (GB)
OpenThoughts	10^{-2}	94	40	393.20	9.83
	10^{-4}	2398	1208	9375.56	7.76
	10^{-3}	319	277	1205.38	4.35
MedQA	10^{-2}	96	52	355.43	6.84
	10^{-4}	2821	1378	11388.00	8.26
	10^{-3}	373	327	1452.00	4.44

aside, all methods would replay at most one step after a failure, making the recovery comparison uninformative. We therefore use a matched-freshness regime instead. LowDiff and `LayerCheck` retain their native per-step persistence semantics, since both mechanisms are defined around continual step-level state preservation. For rollback-based baselines (DeepSpeed default, CheckFreq, and GEMINI), we set the checkpoint interval to match the average staleness observed under `LayerCheck`, which is 106 training steps on Qwen2.5-7B. Assuming failures are uniformly distributed within a checkpoint interval, this corresponds to an average rollback distance of 53 steps after a failure.

We decompose the total recovery time for a single failure as

$$t_{\text{recovery}} = t_{\text{load}} + t_{\text{retrain}} + t_{\text{reconstruct}}, \tag{13}$$

where t_{load} is the time to read checkpoint state from storage, t_{retrain} is the time to replay lost iterations after rollback, and $t_{\text{reconstruct}}$ is method-specific reconstruction cost (e.g., layer merging or decompression).

In Figure 7, we report the aligned recovery overhead

$$t^{\star}_{\text{recovery}} = t_{\text{load}} + t_{\text{retrain}}, \tag{14}$$

We exclude $t_{\text{reconstruct}}$ from the cross-method comparison because GEMINI and LowDiff do not provide complete public recovery implementations, and reconstruction cost is highly implementation-dependent. This narrows the scope of the comparison but preserves comparability across frameworks.

Figure 7 reports the aligned recovery overhead across all three models under this matched-freshness setting. `LayerCheck` achieves the lowest recovery overhead at 10s, compared to 34s for LowDiff (**3.4×** faster) for Qwen2.5-7B. Under this protocol, the rollback-based full-state methods naturally become similar, because they replay the same expected rollback window and differ mainly in where these state loaded. `LayerCheck` achieves the lowest recovery overhead among the compared methods. This improvement follows the same logic as the steady-state results: by writing less state during training, `LayerCheck` has less state to load after failures. In addition, composite recovery reduces the need to roll back and replay iterations; it is designed to restart near the failure point under the bounded-staleness invariant. LowDiff also benefits from per-step recovery semantics, but `LayerCheck` remains lower because selective persistence reduces the amount of state that must be materialized at restart than the differential checkpoint.

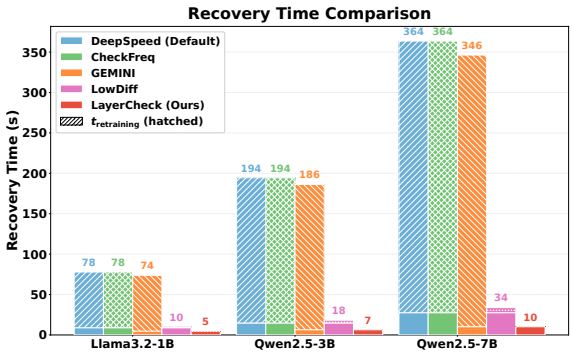


Fig. 7: Aligned recovery overhead of a single failure under matched restart freshness across three models. We report $t_{\text{load}} + t_{\text{retrain}}$ as $t^{\star}_{\text{recovery}}$.

E. Scalability under Repeated Failures

As training scales out, failures become more frequent [16], and steady-state checkpoint overhead alone no longer predicts overall system efficiency. We therefore study projected behavior under repeated failures using a data-driven simulation parameterized by measured system components in the previous sections, with mean time between failures (MTBF) from 0.01 to 3 hours. All method-specific time components used in the simulation (e.g., per-iteration training time, checkpoint overhead, and recovery time terms in (13)) are measured on Qwen2.5-7B using the same OpenThoughts dataset under the same storage stack, and then plugged into the MTBF-driven failure injection model.

Following GEMINI [16], we report effective training time ratio (ETTR), defined as

$$\text{ETTR} \triangleq \frac{t_{\text{train}}}{t_{\text{train}} + t_{\text{ckpt}} + t_{\text{recovery}}}. \tag{15}$$

ETTR can be read as utilization: the fraction of wall-clock time spent doing useful forward/backward/optimizer work rather than checkpointing and recovery.

The simulation in Figure 8 shows that, `LayerCheck` consistently achieves the highest ETTR across all MTBF values. For example, at MTBF=0.5 hours, `LayerCheck` attains **92.3%**, outperforming LowDiff (88.8%) and GEMINI (84.5%). This result ties together the earlier sections in a single metric: when failures are frequent enough to matter, the system that minimizes both steady-state I/O stalls and failure-time recovery costs yields the highest effective throughput.

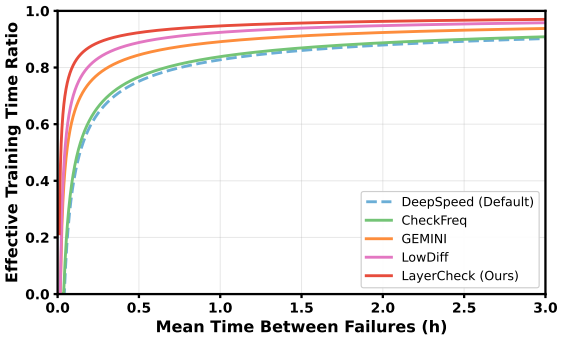


Fig. 8: Effective training time ratio under repeated failures with varying MTBF. Results are computed from Eq. 15 with timing parameters measured on Qwen2.5-7B (OpenThoughts).

F. Recovered Model Quality

Loss trajectories establish that training remains stable after composite recovery, but they do not fully answer whether the final model behaves the same on downstream evaluation tasks. This matters because LLM fine-tuning losses can correlate imperfectly with benchmark scores.

Table III therefore compares final benchmark scores for (i) failure-free fine-tuning and (ii) fine-tuning with an injected failure and recovery followed by continued training. Across datasets and model sizes, recovered runs match the failure-free results closely and do not exhibit systematic degradation. Combined with Figure 5, these results support the paper's main claim: `LayerCheck` reduces checkpoint and recovery cost while preserving the end-to-end outcome of training.

G. Summary and Limitations

Taken together, these experiments support a consistent picture. Composite recovery does not destabilize optimization, which is the key stability concern raised by mixed timestamps. Selective persistence substantially reduces written bytes and, in an I/O-stressful regime, translates to end-to-end time savings. Under an aligned staleness budget, `LayerCheck` also reduces recovery overhead, and these benefits compound under repeated failures as reflected in ETTR.

Two practical limitations remain. First, the absolute magnitude of improvements depends on the storage backend [40]. Faster local storage reduces checkpoint cost for all methods, but selective persistence still reduces bytes written and thus reduces the sensitivity of training throughput to storage variability [41], [42].

Second, we exclude method-specific reconstruction time $t_{\text{reconstruct}}$ from the cross-method comparison. This term is primarily implementation-dependent and is not reported uniformly across prior systems; moreover, several baselines do not provide complete public recovery code. We therefore compare only the recovery components that can be aligned consistently across methods. `LayerCheck` nevertheless includes an explicit reconstruction path together with the $S_{\max}$ fidelity guard, making its recovery semantics explicit.

Finally, our implementation, evaluation, and analysis focus on AdamW. LayerCheck's persistence decision is based on realized layer-wise parameter changes rather than Adam-specific statistics, so the same principle can be applied conceptually to vanilla SGD. In this case, LayerCheck would persist the selected layer weights without per-parameter optimizer state. Because the optimizer affects the scale and temporal pattern of parameter updates, the base threshold K_0 should be calibrated accordingly. A smaller K_0 triggers more frequent persistence, reducing layer staleness and improving recovery fidelity at the cost of higher I/O overhead, whereas a larger K_0 reduces checkpoint writes but may increase staleness and recovery deviation.

V. Related Work

Checkpointing for Deep Learning and LLM Training. Checkpointing is the standard fault-tolerance mechanism in major deep learning frameworks such as PyTorch [43] and TensorFlow [44], but it exposes the familiar trade-off between checkpoint overhead and recovery cost. Prior work can be grouped into several lines. *Pipeline and concurrent checkpointing* reduce training stalls by overlapping snapshotting, and persistence with computation, as in CheckFreq [39], GEMINI [16], PCcheck [45], Datastate-LLM [17], and FlowCheck [46]. *Elastic or redundancy-aware recovery* improves resilience under failures or changing resources, as in CPR [15], Just-in-Time checkpointing [14], Elastor [47], and Checkmate [48]. *Partial, incremental, and differential checkpointing* reduce what must be written or reconstructed, including SCAR [24], Check-N-Run [20], LowDiff [21], LLMTailor [49] and Amber [22]. Among these systems, LowDiff [21] is a recent online differential-checkpointing baseline and is therefore our primary comparison on the online-I/O axis. LLMTailor is closest in spirit: it assembles a resumable checkpoint from existing ones via fixed, rule-based layer-merging recipes, with no online layer selection or convergence analysis—its filtered variants can even degrade model quality.

In contrast, LayerCheck makes online, per-layer, per-step persistence decisions via accumulated drift with LR-aware thresholding, provides a bounded-staleness convergence guarantee for mixed-timestamp recovery, and avoids replay-based recovery while still reducing checkpoint I/O.

Checkpoint Compression and Model Reduction. Some prior work reduces checkpoint size through compression. ExCP [19] combines quantization and compression to shrink checkpoints, while Inshrinkerator [50] exploits weight sensitivity to compress checkpointed states. QD-Compressor [51]

TABLE III: Benchmark on OpenThoughts and MedQA (After fine-tuning vs. recovered checkpoints).

Dataset	Model	Status	ARC-easy	HellaSwag	Lambada	PIQA	MedMCQA	MMLU-Med	PubMedQA
MedQA	Qwen2.5-7B	After FT (No failure)	81.44	60.59	67.07	79.43	60.39	92.00	75.20
		Recovered (step=888)	**81.78**	60.36	**67.30**	**80.03**	**60.41**	**93.00**	**75.60**
	Qwen2.5-3B	After FT (No failure)	77.65	54.82	63.50	77.20	52.12	82.00	74.60
		Recovered (step=867)	**77.95**	**54.90**	**63.56**	**77.31**	**52.26**	81.00	74.60
	Llama3.2-1B	After FT (No failure)	64.02	46.73	53.64	73.12	31.77	33.00	55.80
		Recovered (step=1374)	**64.14**	**46.91**	53.60	72.96	**32.44**	**34.00**	55.80
OpenThoughts	Qwen2.5-7B	After FT (No failure)	79.92	59.70	68.87	77.97	59.86	85.00	75.20
		Recovered (step=299)	**80.35**	59.63	68.37	**78.24**	59.74	85.00	75.00
	Qwen2.5-3B	After FT (No failure)	77.31	54.46	66.45	78.07	51.78	74.00	73.00
		Recovered (step=775)	77.15	54.38	**66.56**	**78.18**	51.66	74.00	72.60
	Llama3.2-1B	After FT (No failure)	64.77	47.39	60.22	74.54	31.44	28.00	58.00
		Recovered (step=1254)	64.69	47.39	**60.43**	74.54	**31.46**	28.00	58.00

further applies layer-wise adaptive quantization to model snapshots in federated learning. AutoCheck [7] and ADVICE [52] propose an LLVM-based compiler augmentation to identify variables necessary to checkpointing that significantly reduces checkpoint states. ZOCheck [53] designs a lightweight checkpointing inspired by LLM Zeroth-order (ZO) execution semantics, representing training progress with compressed states and recovers via deterministic replay. These methods reduce storage footprint, but they are primarily designed for local checkpoints or smaller federated settings rather than online checkpointing in large-scale distributed LLM training.

More broadly, model reduction techniques such as quantization, pruning, and structural compression have been used to lower the storage and compute cost of large models. Recent quantization studies [54], [55] show that low-bit representations can preserve a favorable accuracy–efficiency tradeoff at inference time. Structural approaches including LLM-Streamline [56], LLM-Pruner [57], LaCo [58], and ShortGPT [59] reduce model size by pruning, or merging redundant layers, building on the broader pruning literature [60]. Although these methods are not checkpointing systems, they suggest that LLM layers differ in redundancy and sensitivity. `LayerCheck` is motivated by a related observation, but applies it to fault tolerance: instead of compressing the deployed model itself, it selectively persists full layer states online during training to reduce checkpoint I/O and recovery cost.

VI. Conclusion

In this paper, we introduce `LayerCheck`, an adaptive layer-wise system for reducing checkpoint overhead in LLM post-training. By selectively persisting only layers with significant accumulated drift, `LayerCheck` reduces checkpoint I/O and shortens end-to-end training time while preserving final model quality. By reusing already-available training signals, `LayerCheck`'s layer-drift tracking adds negligible per-step overhead, while selective persistence substantially reduces the checkpoint cost. Experiments show that our approach reduces the total checkpoint size by up to 22.6× and the checkpointing time by up to 1.31× compared with existing SOTA, while preserving final model quality. A key next step is to integrate `LayerCheck` with overlap-oriented checkpoint pipelines, so that our reduced checkpoint work can also benefit from end-to-end overlap across computation, communication, and I/O.

Acknowledgments

We sincerely thank the reviewers for their valuable feedback. This work was supported in part by the National Science Foundation (NSF) under grants CCF-2521613 and CCF-2412345. This research is in part supported by the U.S. Department of Energy (DOE) through the Office of Advanced Scientific Computing Research's "Orchestration for Distributed & Data-Intensive Scientific Exploration" (77765) and Pacific Northwest National Laboratory's AT SCALE LDRD "Decentralized data mesh for autonomous materials synthesis". PNNL is operated by Battelle for the DOE under Contract DE-AC05-76RL01830. The Authors acknowledge the National Artificial Intelligence Research Resource (NAIRR) Pilot for contributing to this research result. This work was partially supported by also the RIKEN TRIP Initiative (AGIS / Foundational Software Development) and JSPS KAKENHI Grant Number JP24K14974.

References

[1] AI@Meta, "Llama 3 model card," 2024.

[2] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, *et al.*, "Scaling instruction-finetuned language models," *Journal of Machine Learning Research*, vol. 25, no. 70, pp. 1–53, 2024.

[3] B. Li, L. Guo, B. She, N. R. Tallent, V. Adetola, and R. Ge, "Powermorph: Shaping llm training for data center demand response," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1204–1218, 2026.

[4] S. Gandhi and C. Kozyrakis, "Moetion: Efficient and reliable sparse checkpointing for mixture-of-experts models at scale," 2025.

[5] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2010.

[6] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, pp. 494–499, 2006.

[7] X. Fu, W. Zhang, S. Meng, X. Huang, W. Xu, L. Guo, and K. Sato, "Autocheck: Automatically identifying variables for checkpointing by data dependency analysis," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, IEEE, 2024.

[8] W. Xu, X. Huang, S. Meng, W. Zhang, L. Guo, and K. Sato, "An efficient checkpointing system for large machine learning model training," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 896–900, IEEE, 2024.

[9] X. Fu, X. Huang, W. Xu, W. Zhang, S. Meng, L. Guo, and K. Sato, "Benchmarking variables for checkpointing in hpc applications," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 406–413, IEEE, 2024.

[10] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1181–1190, 2014.

[11] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pp. 615–626, 2012.

[12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, (New York, NY, USA), p. 3505–3506, Association for Computing Machinery, 2020.

[13] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li, "Pytorch fsdp: Experiences on scaling fully sharded data parallel," 2023.

[14] T. Gupta, S. Krishnan, R. Kumar, A. Vijeev, B. Gulavani, N. Kwatra, R. Ramjee, and M. Sivathanu, "Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures," in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1110–1125, ACM, 04 2024.

[15] K. Maeng, S. Bharuka, I. Gao, M. C. Jeffrey, V. Saraph, B.-Y. Su, C. Trippel, J. Yang, M. Rabbat, B. Lucia, and C.-J. Wu, "Cpr: Understanding and improving failure tolerant training for deep learning recommendation with partial recovery," 2020.

[16] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. S. E. Ng, and Y. Wang, "GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 364–381, ACM, 10 2023.

[17] A. Maurya, R. Underwood, M. M. Rafique, F. Cappello, and B. Nicolae, "Datastates-llm: Lazy asynchronous checkpointing for large language models," in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, (New York, NY, USA), p. 227–239, Association for Computing Machinery, 2024.

[18] Z. Hu, X. Zou, W. Xia, S. Jin, D. Tao, Y. Liu, W. Zhang, and Z. Zhang, "Delta-dnn: Efficiently compressing deep neural networks via exploiting floats similarity," in *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[19] W. Li, X. Chen, H. Shu, Y. Tang, and Y. Wang, "ExCP: Extreme LLM checkpoint compression via weight-momentum joint shrinking," in *Proceedings of the 41st International Conference on Machine Learning* (R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, eds.), vol. 235 of *Proceedings of Machine Learning Research*, pp. 27575–27588, PMLR, 21–27 Jul 2024.

[20] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "{Check-N-Run}: A checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 929–943, 2022.

[21] C. Yao, Y. Hu, F. Liu, Z. Liu, and D. Feng, "Lowdiff: Efficient frequent checkpointing via low-cost differential for high-performance distributed training systems," 2025.

[22] Z. Wang, W. Zhu, Z. Zhang, C. Yan, F. Guo, Y. Li, and Y. Xu, "Amber: Towards fast and space-efficient incremental checkpointing in large language model training," in *Proceedings of the 54th International Conference on Parallel Processing*, ICPP '25, (New York, NY, USA), p. 678–688, Association for Computing Machinery, 2025.

[23] Y. Zhou, T. Pang, K. Liu, C. H. Martin, M. W. Mahoney, and Y. Yang, "Temperature balancing, layer-wise weight analysis, and neural network training," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, (Red Hook, NY, USA), Curran Associates Inc., 2023.

[24] A. Qiao, B. Aragam, B. Zhang, and E. Xing, "Fault tolerance in iterative-convergent machine learning," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 5220–5230, PMLR, 09–15 Jun 2019.

[25] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *ACM Transactions on Intelligent Systems and Technology*, vol. 16, no. 5, pp. 1–72, 2025.

[26] C. Egersdoerfer, P. Carns, S. Snyder, R. Ross, and D. Dai, "Stellar: Storage tuning engine leveraging llm autonomous reasoning for high performance parallel file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'25)*, 2025.

[27] C. Egersdoerfer, A. Sareen, J. L. Bez, S. Byna, D. Xu, and D. Dai, "Ioagent: Democratizing trustworthy hpc i/o performance diagnosis capability via llms," in *Proceedings of the 39th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2025.

[28] C. Egersdoerfer, A. Sareen, J. L. Bez, S. Byna, and D. Dai, "Ion: Navigating the hpc i/o optimization journey using large language models," in *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 86–92, 2024.

[29] B. She, B. Chen, L. Guo, and F. Li, "Pfagent: A tractable and self-evolving power-flow agent for interactive grid analysis," 2026.

[30] Qwen, :, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," 2025.

[31] O. Press and L. Wolf, "Using the output embedding to improve language models," 2017.

[32] G. Jawahar, B. Sagot, and D. Seddah, "What does BERT learn about the structure of language?," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (A. Korhonen, D. Traum, and L. Màrquez, eds.), (Florence, Italy), pp. 3651–3657, Association for Computational Linguistics, July 2019.

[33] J. Phang, H. Liu, and S. R. Bowman, "Fine-tuned transformers show clusters of similar representations across layers," in *Proceedings of the Fourth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP* (J. Bastings, Y. Belinkov, E. Dupoux, M. Giulianelli, D. Hupkes, Y. Pinter, and H. Sajjad, eds.), (Punta Cana, Dominican Republic), pp. 529–538, Association for Computational Linguistics, Nov. 2021.

[34] Y. Kim, S. Lee, A. Jung, B. Ryu, and S. Hong, "Task vector quantization for memory-efficient model merging," 2025.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[36] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[37] D. Jin, E. Pan, N. Oufattole, W.-H. Weng, H. Fang, and P. Szolovits, "What disease does this patient have? a large-scale open domain question answering dataset from medical exams," *Applied Sciences*, vol. 11, no. 14, p. 6421, 2021.

[38] E. Guha, R. Marten, S. Keh, N. Raoof, G. Smyrnis, H. Bansal, M. Nezhurina, J. Mercat, T. Vu, Z. Sprague, A. Suvarna, B. Feuer, L. Chen, Z. Khan, E. Frankel, S. Grover, C. Choi, N. Muennighoff, S. Su, W. Zhao, J. Yang, S. Pimpalgaonkar, K. Sharma, C. C.-J. Ji, Y. Deng, S. Pratt, V. Ramanujan, J. Saad-Falcon, J. Li, A. Dave, A. Albalak, K. Arora, B. Wulfe, C. Hegde, G. Durrett, S. Oh, M. Bansal, S. Gabriel, A. Grover, K.-W. Chang, V. Shankar, A. Gokaslan, M. A. Merrill, T. Hashimoto, Y. Choi, J. Jitsev, R. Heckel, M. Sathiamoorthy, A. G. Dimakis, and L. Schmidt, "Openthoughts: Data recipes for reasoning models," 2025.

[39] J. Mohan, A. Phanishayee, and V. Chidambaram, "{CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 203–216, 2021.

[40] M. H. Rashid, J. Firoz, N. R. Tallent, L. Guo, M. Tang, and D. Dai, "Qosflow: Ensuring service quality of distributed workflows using interpretable sensitivity models," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1372–1387, 2026.

[41] M. Tang, L. Guo, A. Kougkas, X.-H. Sun, and N. R. Tallent, "Characterizing dataflow for i/o-aware scheduling in hpc workflows," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 868–884, 2026.

[42] M. H. Rashid, N. R. Tallent, F. S. Bao, and D. Dai, "Carat: Client-side adaptive rpc and cache co-tuning for parallel file systems," in *2026 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1343–1357, 2026.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[44] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 265–283, USENIX Association, 11 2016.

[45] F. Strati, M. Friedman, and A. Klimovic, "Pccheck: Persistent concurrent checkpointing for ml," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, (New York, NY, USA), p. 811–827, Association for Computing Machinery, 2025.

[46] Z. Huang, H. Nie, H. Jia, B. Jiang, J. Guo, J. Lu, R. Wen, B. Lyu, S. Zhu, and X. Wang, "Flowcheck: Decoupling checkpointing and training of large-scale models," in *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, (New York, NY, USA), p. 1334–1349, Association for Computing Machinery, 2025.

[47] X. Wang, F. Fu, H. Li, H. Ge, S. Lin, J. Niu, and B. Cui, "Elastor: Elastic and efficient model partitioning and checkpointing for fault-tolerant distributed training," in *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '26, (New York, NY, USA), p. 398–412, Association for Computing Machinery, 2026.

[48] A. Bhardwaj, W. Wang, J. Carin, A. Belay, and M. Ghobadi, "Checkmate: Zero-overhead model checkpointing via network gradient replication," 2025.

[49] M. Sun, X. Huang, L. Guo, N. R. Tallent, K. Sato, and D. Dai, "Llmtailor: A layer-wise tailoring tool for efficient checkpointing of large language models," in *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, (New York, NY, USA), p. 1366–1374, Association for Computing Machinery, 2025.

[50] A. Agrawal, S. Reddy, S. Bhattamishra, V. P. S. Nookala, V. Vashishth, K. Rong, and A. Tumanov, "Inshrinkerator: Compressing deep learning training checkpoints via dynamic quantization," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, pp. 1012–1031, 2024.

[51] H. Jin, D. Wu, S. Zhang, X. Zou, S. Jin, D. Tao, Q. Liao, and W. Xia, "Design of a quantization-based dnn delta compression framework for model snapshots and federated learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 923–937, 2023.

[52] X. Huang, L. Guo, N. R. Tallent, and K. Sato, "ADVICE: Automatic identification of variables to checkpoint through compiler augmentation," in *The 26th IEEE International Symposium on Cluster, Cloud, and Internet Computing (CCGrid)*, IEEE, 2026.

[53] S. Minqiu, X. Huang, L. Guo, N. R. Tallent, K. Sato, and D. Dai., "Zocheck: Cpu-shadow checkpointing for zeroth-order llm fine-tuning," in *SC26: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2026.

[54] Y. Zhang, P. Zhang, M. Huang, J. Xiang, Y. Wang, C. Wang, Y. Zhang, L. Yu, C. Liu, and W. Lin, "Qqq: Quality quattuor-bit quantization for large language models," 2024.

[55] Y. Lin, H. Tang, S. Yang, Z. Zhang, G. Xiao, C. Gan, and S. Han, "Qserve: W4a8kv4 quantization and system co-design for efficient llm serving," 2025.

[56] X. Chen, Y. Hu, J. Zhang, Y. Wang, C. Li, and H. Chen, "Streamlining redundant layers to compress large language models," 2025.

[57] X. Ma, G. Fang, and X. Wang, "Llm-pruner: On the structural pruning of large language models," *Advances in neural information processing systems*, vol. 36, pp. 21702–21720, 2023.

[58] Y. Yang, Z. Cao, and H. Zhao, "LaCo: Large language model pruning via layer collapse," in *Findings of the Association for Computational Linguistics: EMNLP 2024* (Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, eds.), (Miami, Florida, USA), pp. 6401–6417, Association for Computational Linguistics, Nov. 2024.

[59] X. Men, M. Xu, Q. Zhang, B. Wang, H. Lin, Y. Lu, X. Han, and W. Chen, "Shortgpt: Layers in large language models are more redundant than you expect," 2024.

[60] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Network," in *Advances in Neural Information Processing Systems*, vol. 28, Curran Associates, Inc., 2015.