REANCHOR: INPUT-CONDITIONED POST-BACKBONE REPRESENTATION RECOVERY UNDER SOFT ERRORS

Anonymous authors
Paper under double-blind review

ABSTRACT

Transient soft errors in hardware can corrupt DNN weights or intermediate activations during inference and propagate to incorrect predictions. Many algorithm-level mitigation methods require fault-aware retraining or modifications to the backbone, limiting their use with already-trained models. Range-based mitigation is less intrusive, yet residual errors at the backbone–head interface may remain within calibrated ranges or persist after range projection. This motivates recovering task-relevant representations directly at the post-backbone interface. We propose ReAnchor for this setting. It first applies a fixed range projection to stabilize the terminal backbone feature and then combines the projected feature with sample-aligned multi-scale context extracted from the current input to recover a task-relevant representation for the original task head. Only the ReAnchor module is post-trained; the backbone and task head remain fixed, and inference requires no knowledge of the fault location, corrupted bit index, or fault rate. Across three common datasets and three different model architectures, ReAnchor largely preserves clean accuracy while its advantage becomes more apparent as the injected fault rate increases. At a fault rate of 10^{-5}, ReAnchor achieves the highest mean top-1 accuracy in all nine dataset–backbone settings. Compared with the best competing method in each setting, including the unmitigated baseline and the four range-based methods, the average improvement is 31.4 percentage points.

1 INTRODUCTION

Deep neural networks (DNNs) in reliability-sensitive systems are vulnerable to transient soft errors, such as bit flips in weights or intermediate activations, that may propagate and alter predictions (Ibrahim et al., 2020; Ahmadilivani et al., 2024; Roquet et al., 2024). We study fault mitigation for already-trained models under the practical constraint that the deployed backbone architecture and parameters remain fixed.

Hardware redundancy, error-correcting codes, and protected execution can improve reliability but often incur substantial overhead (Yu et al., 2024; Dos Santos et al., 2025; Bolchini et al., 2024). Algorithm-level methods reduce this dependence, although many require fault-aware retraining or backbone modification (Hari et al., 2022; Amarnath et al., 2024; Wang et al., 2024). Range-based methods are less intrusive, constraining activations within calibrated or learned ranges without retraining the entire backbone (Chen et al., 2021; Hoang et al., 2020; Ghavami et al., 2022; Mousavi et al., 2024).

Range-based mitigation can be applied throughout the backbone to constrain abnormal activations during fault propagation (Chen et al., 2021). We focus on the terminal backbone feature because fault effects that remain at this stage are passed directly to the task head. At this interface, range projection can only modify out-of-range values: if a fault-induced prediction error occurs while all terminal activations remain finite and within their calibrated bounds, the representation is left unchanged; if range violations occur, clipping may reduce extreme deviations but does not necessarily restore a representation that yields the correct prediction. Section 3 shows that both cases occur non-negligibly under fine-grained bounds. These results indicate that range restriction alone is insufficient for recovering fault-induced errors at the terminal interface.

These observations motivate a different intervention point: rather than attempting to identify individual faults or repeatedly constrain intermediate activations, we ask whether the representation

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

delivered to a fixed task head can be recovered after backbone execution. We refer to this setting as post-backbone representation recovery. The objective is task-oriented rather than reconstructive: a recovered feature need not reproduce the clean feature exactly, but it should restore a representation that supports the original, fixed task head while avoiding degradation on fault-free inputs.

Based on this design, we propose ReAnchor, an input-conditioned framework for post-backbone representation recovery. ReAnchor first applies a fixed range projection to suppress extreme numerical deviations and then combines the projected terminal feature with information extracted from the current input to recover the representation passed to the original task head. During post-training, only ReAnchor is optimized, while the already-trained backbone and task head remain fixed.

Our main contributions are summarized as follows: (1) We characterize the limitations of range-based soft-error mitigation at the backbone-head interface, showing that fault-induced errors can remain within calibrated ranges or persist after projection. (2) We study task-oriented post-backbone representation recovery for fault-induced prediction errors and develop ReAnchor to perform recovery without fault-specific information at inference. (3) We evaluate ReAnchor across three datasets, three backbone architectures, and multiple fault rates. At 10^{-5}, ReAnchor achieves the highest mean top-1 accuracy in all nine dataset–backbone settings, with an average gain of 31.4 percentage points over the best competing method in each setting. On CIFAR-10, sample-level analysis further shows that ReAnchor corrects 42.22–61.44% of fault-induced prediction errors across the three backbones and evaluated fault rates.

2 Problem Setup and Threat Model

2.1 Problem Setup

Let x and y denote an input and its label, and let a trained model consist of a backbone B_θ and task head H_ψ connected by the original interface $\mathcal{Q}$. Clean and faulty inference are defined as

$$\begin{aligned} Z^{\mathrm{cln}} &= B_\theta(x), \qquad \hat{y}^{\mathrm{cln}} = \arg\max H_\psi\big(\mathcal{Q}(Z^{\mathrm{cln}})\big), \\ Z^{\delta} &= \widetilde{B}_\theta(x;\delta), \qquad \hat{y}^{\delta} = \arg\max H_\psi\big(\mathcal{Q}(Z^{\delta})\big), \end{aligned} \tag{1}$$

where $\widetilde{B}_\theta(\cdot;\delta)$ denotes backbone execution under fault realization δ.

We study post-training mitigation under a frozen-model deployment setting, where the backbone B_θ and task head H_ψ remain unchanged. At inference, the mitigation mechanism operates after a single backbone execution; it neither updates nor re-executes the backbone and uses no fault-specific information such as fault locations, corrupted bit indices, or fault rates. It may access only the input x and the terminal backbone representation. Under these constraints, our objective is to reduce fault-induced prediction errors while preserving fault-free accuracy.

2.2 Threat Model

Transient soft errors can originate from different hardware components and affect multiple states involved in DNN inference. We do not attempt to model every hardware-level failure mode explicitly; instead, we approximate their model-level effects through transient bit-level corruptions of backbone weights and intermediate activations. Faults are sampled over this combined exposed domain rather than under separate weight-only or activation-only assumptions. Perturbed weights are restored after the affected execution, while corrupted activations affect only that execution (Bolchini et al., 2023; Huang et al., 2023).

We assume a selectively protected deployment consisting of an *exposed fault domain* and a *protected region*. The backbone B_θ resides in the exposed fault domain, whereas the original interface $\mathcal{Q}$ and task head H_ψ reside in the protected region. The ReAnchor recovery module R_ϕ is evaluated under both protected and exposed settings. In the default protected setting, R_ϕ resides in the protected region and is isolated from injected faults. In the exposed setting, R_ϕ is subjected to the same soft-error model as B_θ, while $\mathcal{Q}$ and H_ψ remain protected. This selective-protection setting is consistent with prior studies on reliable DNN inference (Chen et al., 2019; Breier et al., 2022; Shin et al., 2023; Zheng et al., 2025). Detailed fault-domain definitions and bit-sampling rules are provided in Appendix B.

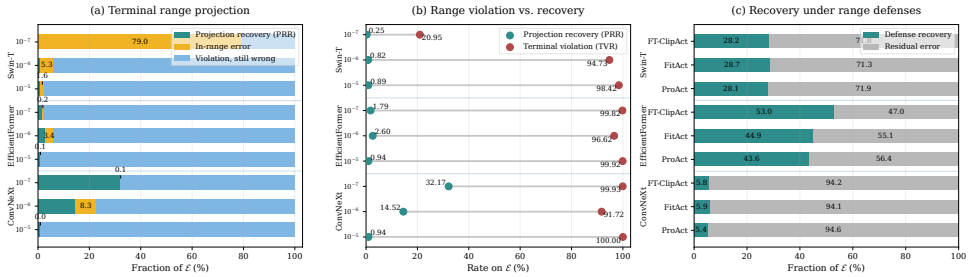

Figure 1: Limitations of range-based mitigation on CIFAR-100. (a) Terminal projection partitions $\mathcal{E}$ into recovered, in-range, and unresolved range-violating errors. (b) TVR is much higher than PRR, indicating that most terminal range violations remain uncorrected after projection. (c) FT-ClipAct, FitAct, and ProAct at 10^{-6} on the same baseline error set $\mathcal{E}$. All values are percentages of $\mathcal{E}$.

3 Range-Based Activation Mitigation and Its Limitations

Range-based mitigation methods mitigate abnormal activations by constraining them within calibrated or learned ranges at different granularities (Xue et al., 2023). FT-ClipAct (Hoang et al., 2020) applies layer-wise activation clipping, FitAct (Ghavami et al., 2022) learns fine-grained neuron-wise activation bounds during post-training, and ProAct (Mousavi et al., 2024) combines coarse- and fine-grained range constraints through progressively optimized clipping thresholds. Despite these differences in range granularity and calibration strategy, these methods share a common correction principle: activations are modified when they violate the prescribed ranges.

The terminal backbone feature is the final representation passed to the task head, and fault effects that remain at this stage can directly affect the prediction. We therefore analyze range-based mitigation at the backbone–head interface from two complementary perspectives. First, we use fine-grained neuron-wise bounds to test whether fault-induced prediction errors produce terminal range violations and whether projecting the violating values back into range can recover the correct prediction. The projection is applied to the same faulty feature Z^δ, which is then evaluated through the original interface $\mathcal{Q}$ and the fixed task head H_ψ without re-executing the backbone. Second, we evaluate existing range-based mitigation methods applied across multiple backbone layers to determine how many fault-induced errors remain after their protection.

For both analyses, we consider samples that are correctly classified under fault-free inference but become misclassified under a fault realization. We define the corresponding error set as

$$\mathcal{E} = \left\{(x, \delta) \;\middle|\; \hat{y}^{\text{cln}} = y,\ \hat{y}^{\delta} \neq y\right\}, \tag{2}$$

which isolates prediction errors introduced by the sampled fault realization δ.

Terminal range analysis. Neuron-wise bounds are obtained from fault-free training features using the empirical minimum and maximum at each channel and spatial location over the training set. These fine-grained bounds reduce the influence of range granularity on the analysis.

For each $(x, \delta) \in \mathcal{E}$, we apply the projection directly to the *same* faulty terminal feature Z^δ and pass the projected feature through the original backbone–head interface and fixed task head; the backbone is not re-executed. A terminal violation is defined as the presence of at least one non-finite or out-of-range entry. This partitions $\mathcal{E}$ into three disjoint subsets: $\mathcal{E}_{\text{in}}$, where Z^δ is finite and entirely within range and projection is therefore the identity; $\mathcal{E}_{\text{rec}}$, where a violation is present and projection restores the correct prediction; and $\mathcal{E}_{\text{vio-unres}}$, where a violation is present but the projected prediction remains incorrect. Thus,

$$\mathcal{E} = \mathcal{E}_{\text{in}} \cup \mathcal{E}_{\text{rec}} \cup \mathcal{E}_{\text{vio-unres}}, \qquad \mathcal{E}_{\text{in}}, \mathcal{E}_{\text{rec}}, \mathcal{E}_{\text{vio-unres}} \text{ pairwise disjoint.} \tag{3}$$

We use two metrics to characterize these outcomes: the projection recovery rate (PRR) and the terminal violation rate (TVR):

$$\text{PRR} = \frac{|\mathcal{E}_{\text{rec}}|}{|\mathcal{E}|}, \qquad \text{TVR} = \frac{|\mathcal{E}_{\text{rec}}| + |\mathcal{E}_{\text{vio-unres}}|}{|\mathcal{E}|}, \tag{4}$$

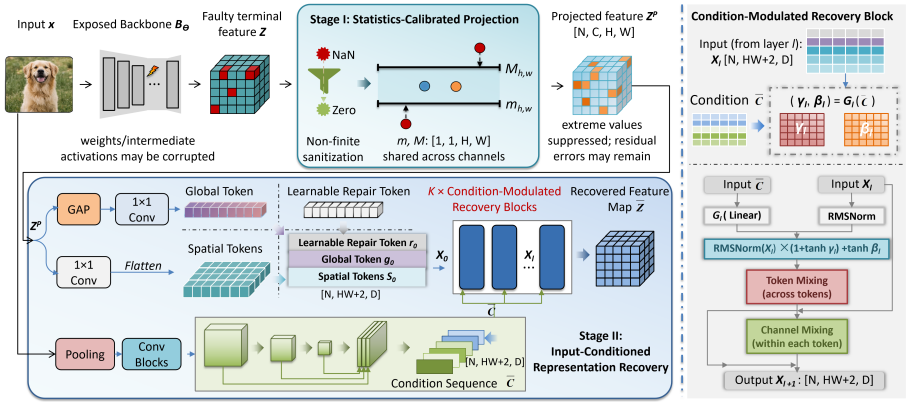

Figure 2: Overview of ReAnchor. Stage I projects the terminal feature into calibrated spatial ranges and Stage II recovers the representation using input-derived multi-scale context before the protected task head.

Results. We evaluate CIFAR-100 (Krizhevsky et al., 2009) with Swin-Transformer (Liu et al., 2021), EfficientFormerV2 (Li et al., 2023), and ConvNeXt (Liu et al., 2022) using paired fault realizations; implementation details are provided in Appendix B. As shown in Figure 1(a), fault-induced errors need not produce terminal range violations: at 10^{-7}, 79.0% of Swin-Tiny errors remain within the calibrated neuron-wise ranges and are therefore unchanged by projection. Figure 1(b) further shows that range violations are not necessarily recoverable. At 10^{-5}, TVR reaches 98.42%, 99.92%, and 100.00% for Swin-Transformer, EfficientFormerV2, and ConvNeXt, while PRR is only 0.89%, 0.94%, and 0.94%, respectively. Figure 1(c) shows the same limitation for FT-ClipAct (Hoang et al., 2020), FitAct (Ghavami et al., 2022), and ProAct (Mousavi et al., 2024), which recover only a subset of the baseline errors at 10^{-6}. Overall, range restriction alone leaves substantial residual fault-induced errors at the terminal interface.

4 REANCHOR

Range projection is effective at suppressing extreme numerical deviations, but it cannot reconstruct information that has already been distorted during backbone execution (Wasim et al., 2023). ReAnchor therefore treats projection as a fixed stabilization step and places a lightweight, input-conditioned recovery module at the backbone–head interface, as illustrated in Figure 2.

Let B_θ denote the already-trained backbone, R_ϕ the ReAnchor recovery module, $\mathcal{Q}$ the original backbone–head interface, and H_ψ the fixed task head. For an input x with terminal feature $Z \in \mathbb{R}^{N\times C\times H\times W}$, ReAnchor computes

$$Z^p = \Pi_\Omega(\mathcal{S}_\Omega(Z)), \qquad \widehat{Z} = R_\phi(Z^p, x), \qquad \widehat{y} = \arg\max H_\psi\Big(\mathcal{Q}(\widehat{Z})\Big). \tag{5}$$

Here, $\mathcal{S}_\Omega$ sanitizes non-finite values and Π_Ω applies a fixed range projection. The interface $\mathcal{Q}$ is unchanged from the original model.

Following Section 2, the backbone belongs to the exposed fault domain, whereas $\mathcal{Q}$ and H_ψ are protected. ReAnchor is protected by default and is separately evaluated under injected faults in Section 5.4. During post-training, B_θ and H_ψ remain frozen and only ϕ is optimized.

4.1 STATISTICS-CALIBRATED PROJECTION.

Stage I provides lightweight numerical stabilization before learned representation recovery, rather than fine-grained feature reconstruction. Accordingly, ReAnchor uses a channel-shared lower and upper bound at each spatial position (h, w), calibrated offline from fault-free training features:

$$m_{h,w} = \min_{\substack{x_j\in\mathcal{D}_{\text{cal}}\\ 1\le k\le C}} \big[Z_j^{\text{cln}}\big]_{k,h,w}, \qquad M_{h,w} = \max_{\substack{x_j\in\mathcal{D}_{\text{cal}}\\ 1\le k\le C}} \big[Z_j^{\text{cln}}\big]_{k,h,w}, \qquad \Omega = \{[m_{h,w}, M_{h,w}]\}_{h,w}. \tag{6}$$

Sharing the calibrated bounds across channels reduces the required calibration metadata from $2CHW$ values for neuron-wise bounds to $2HW$. A sensitivity analysis of alternative range granularities is provided in Appendix C.1.

As shown in Stage I of Figure 2, at inference time, `NaN` values are replaced with zero, while $+\infty$ and $-\infty$ are mapped to the corresponding calibrated upper and lower bounds, respectively. The sanitized feature is then clipped element-wise to Ω before being passed to the learned recovery stage.

4.2 Input-Conditioned Representation Recovery

To provide the recovery module with complementary information for correcting a corrupted terminal representation, ReAnchor combines a structured tokenization of Z^p with sample-specific context derived from the current input. Specifically, a global token g_0 is obtained by global average pooling followed by a learned 1×1 projection, spatial tokens S_0 are produced by a separate 1×1 projection and flattening over the $H \times W$ grid, and r_0 is introduced as a learnable auxiliary token. These components form $X_0 = [g_0; r_0; S_0] \in \mathbb{R}^{N \times (HW+2) \times D}$ which provides the sequence representation required by the subsequent condition-modulated recovery blocks.

Since errors propagated through the backbone may leave residual distortions that cannot be resolved from Z^p alone, ReAnchor further derives sample-specific guidance directly from the current input. As illustrated in Figure 2, a lightweight convolutional encoder extracts multi-scale input context and aligns it with the token sequence to condition the recovery process.

Each recovery block refines the conditioned representation along the two dimensions of the token sequence: across token positions and along the hidden feature dimension. Token mixing allows information to propagate among the global, repair, and spatial tokens, while channel mixing updates the feature representation within each token. Before these two operations, the input-derived condition sequence $\overline{C}$ modulates the current token state, allowing the recovery process to adapt to the current input. For block l,

$$\begin{aligned} (\gamma_l, \beta_l) &= G_l(\overline{C}), \\ Y_l &= \mathrm{RMSNorm}(X_l) \odot (1 + \tanh(\gamma_l)) + \tanh(\beta_l), \\ U_l &= X_l + \mathcal{M}_l^{\mathrm{tok}}(Y_l), \\ X_{l+1} &= U_l + \mathcal{M}_l^{\mathrm{ch}}\left(\mathrm{RMSNorm}(U_l)\right). \end{aligned} \tag{7}$$

Here, $G_l(C)$ produces sample-dependent modulation parameters, while M_l^{tok} and M_l^{ch} propagate information across token positions and feature dimensions, respectively.

After K recovery blocks, the updated spatial tokens are projected from the hidden dimension D back to the terminal feature dimension C and reshaped to the original $H \times W$ layout, yielding $\widehat{Z} \in \mathbb{R}^{N \times C \times H \times W}$. The recovered terminal feature is subsequently passed through the original interface $\mathcal{Q}$ and the fixed task head H_ψ to obtain the final prediction, following equation 5.

4.3 Post-Training Objective

ReAnchor is post-trained with the backbone B_θ and task head H_ψ fixed. For each minibatch, we evaluate the clean branch and sample a faulty branch with probability p_f, where $\delta \sim p_{\mathrm{FI}}^{\mathrm{train}}$ denotes the sampled transient fault realization.

Only the recovery parameters ϕ are optimized:

$$\mathcal{L} = \lambda_{\mathrm{cln}}\mathcal{L}_{\mathrm{task}}^{\mathrm{cln}} + b\big(\lambda_{\mathrm{f}}\mathcal{L}_{\mathrm{task}}^{\mathrm{f}} + \lambda_{\mathrm{feat}}\mathcal{L}_{\mathrm{feat}} + \lambda_{\mathrm{int}}\mathcal{L}_{\mathrm{int}}\big), \qquad b \sim \mathrm{Bernoulli}(p_f). \tag{8}$$

Here, $\mathcal{L}_{\mathrm{task}}^{\mathrm{cln}}$ and $\mathcal{L}_{\mathrm{task}}^{\mathrm{f}}$ provide task supervision for the clean and faulty branches, respectively. $\mathcal{L}_{\mathrm{feat}}$ combines SmoothL1 and cosine distance to regularize the recovered terminal representation toward its fault-free counterpart, while $\mathcal{L}_{\mathrm{int}}$ applies SmoothL1 alignment after the fixed interface $\mathcal{Q}$. These terms encourage task-relevant representation recovery rather than exact feature reconstruction. The loss weights, fault-path sampling probability, optimizer, and post-training schedule are detailed in Appendix A.2.

5 EXPERIMENTS

We evaluate ReAnchor across datasets, backbones, and fault rates, followed by analyses of its recovery mechanism, component contributions, recovery-module fault exposure, and deployment overhead.

5.1 EXPERIMENTAL SETUP

Datasets, backbones, and baselines. We evaluate CIFAR-10 (Krizhevsky et al., 2009), CIFAR-100 (Krizhevsky et al., 2009), and Tiny ImageNet-200 (Le et al., 2015) with three representative modern backbone families: ConvNeXt (Liu et al., 2022), Swin-Transformer (Liu et al., 2021), and EfficientFormerV2 (Li et al., 2023), covering convolutional, hierarchical Transformer, and efficiency-oriented architectures. Baselines include the undefended baseline model and four range-based methods: Ranger (Chen et al., 2021), FT-ClipAct (Hoang et al., 2020), FitAct (Ghavami et al., 2022), and ProAct (Mousavi et al., 2024). All methods use the same backbone and task head for each dataset–backbone pair.

Fault injection and training. Using PyTorchFI (Mahmoud et al., 2020), we inject transient bit flips into backbone weights and intermediate activations at fault rates from 10^{-7} to 10^{-5}. Each affected FP32 value is independently selected according to the fault rate, with one uniformly sampled bit flipped upon corruption. Unless otherwise stated, only the backbone is exposed to faults, while ReAnchor and the task head remain protected.

Metrics and protocol. We report clean, faulty, and mitigated top-1 accuracy, together with the paired accuracy change (ΔAcc), error correction rate (ECR), and mitigation-induced harm rate (MIHR). Model complexity and inference efficiency are measured by parameter count, FLOPs, and inference latency. Unless otherwise stated, each robustness analysis is conducted over 100 paired full-test-set evaluation runs. Within each paired run, all compared inference paths replay the same sequence of per-forward fault realizations. Different analyses use independently sampled sets of 100 paired runs. For the ablation analysis, we additionally report 95% confidence intervals (CIs) for selected paired accuracy differences.

Additional details on the datasets, backbone and baseline implementations, fault-injection and training procedures, and evaluation metrics and protocol are provided in Appendix B.

5.2 ROBUSTNESS ANALYSIS

We evaluate ReAnchor on CIFAR-10, CIFAR-100, and Tiny ImageNet-200 using ConvNeXt, EfficientFormerV2, and Swin-Transformer under fault-free inference and multiple fault rates. We report clean accuracy to measure the impact of the mitigation on nominal performance, and fault-mitigated accuracy to assess robustness under increasing fault severity. We further report ECR and MIHR on CIFAR-10 to quantify corrected fault-induced errors and newly introduced prediction errors.

Figure 3 summarizes mean top-1 accuracy across the nine dataset-backbone settings. Under fault-free inference, ReAnchor changes nominal accuracy only slightly. At a fault rate of 10^{-7}, ReAnchor is generally comparable to the evaluated range-based methods, with a larger advantage in the Tiny ImageNet-200/ConvNeXt setting. The gap becomes more pronounced at higher fault rates on average across the nine dataset–backbone settings. At 10^{-6}, ReAnchor achieves the highest mean accuracy in seven of the nine dataset–backbone settings, and at 10^{-5} it ranks first in all nine. At 10^{-5}, its improvement over the best competing method in each setting ranges from 7.3 to 50.6 percentage points, where the comparison includes the unmitigated baseline and the four range-based methods. Averaged over the nine settings, the improvement is 31.4 percentage points. Complete numerical results for all dataset–backbone settings and fault rates are provided in Appendix C.2.

Table 1 provides a complementary sample-level view of this behavior on CIFAR-10. Across the three backbones and evaluated fault rates, ReAnchor corrects 42.22–61.44% of the prediction errors introduced by the sampled faults. At 10^{-7}, the MIHR remains between 0.70% and 3.30%, indicating that relatively few originally correct predictions become incorrect after mitigation. At higher fault rates, ReAnchor continues to recover a substantial fraction of fault-induced errors, while the

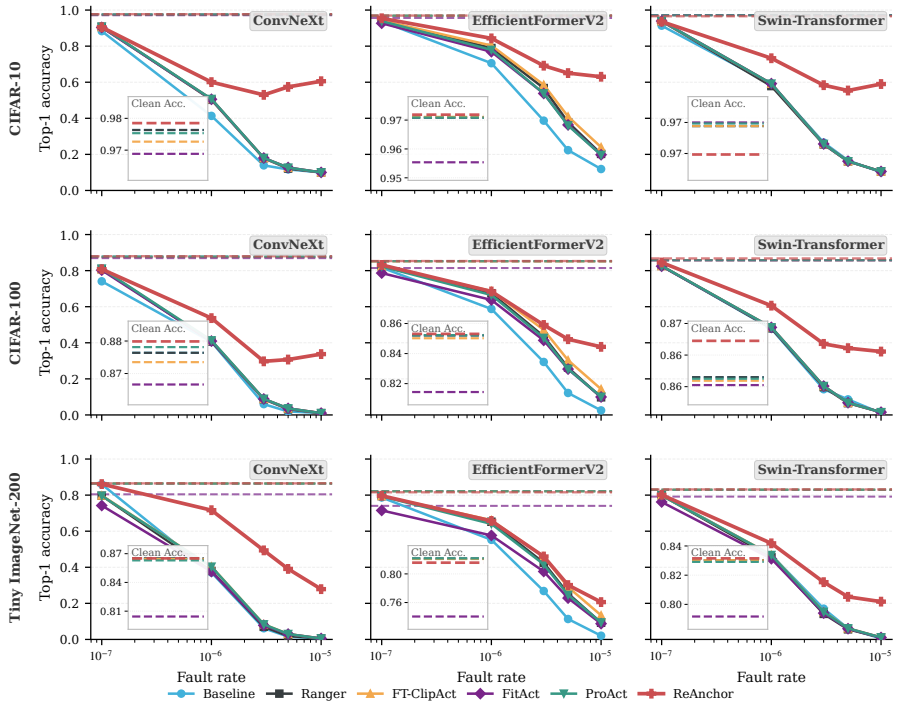

Figure 3: Top-1 accuracy versus fault rate on CIFAR-10 (top row), CIFAR-100 (middle row) and Tiny ImageNet-200 (bottom row) for ConvNeXt (left), EfficientFormerV2 (center), and Swin-Transformer (right). Dashed horizontal lines denote the clean accuracy of each method.

Table 1: Error correction rate (ECR) and mitigation-induced harm rate (MIHR) of ReAnchor on CIFAR-10.

	ConvNeXt			EfficientFormerV2			Swin-Transformer		
Metric	10^{-7}	10^{-6}	10^{-5}	10^{-7}	10^{-6}	10^{-5}	10^{-7}	10^{-6}	10^{-5}
ECR (%)	53.36	42.22	59.69	56.03	55.93	61.44	55.67	45.05	57.52
MIHR (%)	1.68	8.97	31.48	0.70	1.73	20.52	3.30	8.52	30.00

increase in MIHR indicates that more originally correct predictions are changed to incorrect ones after mitigation.

These results show that ReAnchor provides greater robustness than the evaluated range-based methods, particularly under higher fault rates, while largely preserving clean accuracy. The improvement is observed across different datasets and backbone architectures. To further characterize where these gains arise, we analyze the residual errors that remain after ReAnchor's Stage I and evaluate the extent to which they can be corrected by Stage II. Additional results are provided in Appendix C.3.

5.3 Ablation Analysis

We conduct ablations and dependency controls to identify which components account for ReAnchor's robustness gain and whether the gain depends on jointly using the terminal backbone representation and sample-aligned input context. *Projection only* isolates Stage I, whereas the *matched feature-only mixer* controls for additional post-backbone capacity without input conditioning. *Shuffled conditioning* breaks the sample-wise correspondence between the input condition and the terminal feature, while *w/o multi-scale conditioning* removes the multi-scale input context. The *input-conditioned affine* variant provides a parameter-matched but structurally simpler conditioning mechanism. We further ablate the *global token* and *repair token* to assess their individual contributions

Table 2: Ablation and dependency-control results on CIFAR-10 with EfficientFormerV2 at a fault rate of 10^{-6}.

Method	Clean	Faulty Top-1	Δ Acc. (pp)	95% CI (pp)	ECR (%)	MIHR (%)
Baseline	0.969	0.696	–	–	–	–
Core ablations						
Projection only	0.969	0.699	+0.34	[0.28, 0.40]	11.3	4.3
Matched feature-only mixer	0.970	0.701	+0.48	[0.41, 0.56]	11.8	4.7
Shuffled conditioning	0.969	0.700	+0.42	[0.36, 0.49]	10.8	4.2
Matched input-conditioned affine	0.969	0.799	+10.30	[7.66, 12.95]	41.5	2.3
w/o multi-scale conditioning	0.970	0.701	+0.50	[0.42, 0.57]	12.0	4.7
w/o global token	0.970	0.838	+14.25	[10.09, 18.41]	55.0	1.9
w/o repair token	0.970	0.839	+14.32	[10.15, 18.49]	54.9	1.9
Input/backbone dependency controls						
Matched input-only control	0.625	0.625	-7.06	[−13.88, −0.24]	63.0	36.3
Aligned condition + shuffled Z^p †	0.098	0.243	-45.28	[−56.53, −34.03]	37.7	80.6
Full ReAnchor	0.970	**0.844**	**+14.84**	[**10.46**, **19.22**]	57.0	**1.8**

† Z^p is shuffled across samples only at inference time while the input condition remains sample-aligned; Clean is evaluated under the same shuffle with a fault-free backbone.

to recovery. Finally, a separately trained *matched input-only control* and an inference-time *aligned condition + shuffled* Z^p diagnostic test whether the observed gain can be explained by an input-only pathway or by reduced dependence on the terminal backbone representation.

Projection alone and additional feature-side capacity provide only marginal improvements over the faulty-backbone baseline: *Projection only* and the *matched feature-only mixer* reach 0.699 and 0.701, respectively, compared with 0.696 for the baseline. Breaking input–feature alignment or removing multi-scale conditioning similarly limits the gain to at most 0.50 pp. In contrast, the *input-conditioned affine* variant reaches 0.799, showing that sample-aligned conditioning itself provides a substantial benefit, but it remains below Full ReAnchor at 0.844; the paired difference is $+4.54$ pp (95% CI: $[2.80, 6.28]$). Removing the global or repair token yields 0.838 and 0.839, respectively, with ECR decreasing from 57.0% to 55.0% and 54.9%, while MIHR increases only slightly from 1.8% to 1.9%. These results indicate that both tokens contribute to the final recovery performance, but the dominant gain comes from the broader mechanism that integrates sample-aligned input context with the terminal backbone representation.

The dependency controls provide further evidence against an input-only explanation of the robustness gain. The *matched input-only control* reaches only 0.625 clean accuracy and has an MIHR of 36.3%, despite its relatively high ECR of 63.0%, indicating that many apparent fault corrections are accompanied by substantial degradation on originally correct predictions. Likewise, shuffling Z^p while keeping the input condition aligned reduces clean and faulty accuracy to 0.098 and 0.243, respectively, and raises MIHR to 80.6%. Full ReAnchor, by comparison, maintains clean accuracy at 0.970, achieves 0.844 faulty Top-1 accuracy and a 57.0% ECR, and limits MIHR to 1.8%. Together, these controls support the conclusion that ReAnchor relies on the coordinated use of the terminal backbone representation and sample-aligned input context, rather than on range projection, additional post-backbone capacity, or an independent input pathway alone.

5.4 Robustness under Joint Backbone and Recovery-Module Faults

To evaluate whether ReAnchor remains effective when the recovery module is also exposed to faults, we compare three paired inference paths: the faulty-backbone baseline, protected ReAnchor, and faulty ReAnchor. We conduct this comparison across multiple backbones, datasets, and fault rates, while sharing the same backbone fault realization within each paired trial.

Figure 4 shows that exposing the recovery module to the same nominal fault model reduces its benefit in some settings, but Faulty ReAnchor remains above the faulty-backbone baseline across all evaluated backbones, datasets, and fault rates. At 10^{-6} on CIFAR-100, the baseline/protected/faulty accuracies are 0.40/0.54/0.48 for ConvNeXt, 0.58/0.68/0.66 for EfficientFormerV2, and 0.48/0.60/0.60 for Swin-Transformer. Even at the rate of 10^{-5}, faulty ReAnchor still reaches 0.31, 0.37, and 0.32, compared with baseline accuracies of only 0.01, 0.03, and 0.01. Protection provides an additional robustness margin, while ReAnchor continues to outperform the baseline even when the recovery module is included in the fault domain.

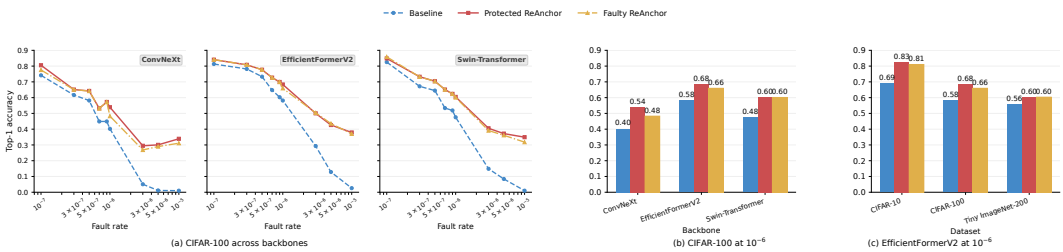

Figure 4: Reliability of ReAnchor under recovery-module faults. (a) Top-1 accuracy versus fault rate on CIFAR-100 across three backbones. (b) Backbone comparison on CIFAR-100 at 10^{-6}. (c) Cross-dataset comparison with EfficientFormerV2 at 10^{-6}. Baseline uses only the backbone; Protected ReAnchor keeps the recovery module fault-free; Faulty ReAnchor exposes it to the same nominal fault rate as the backbone.

Table 3: Average computational and runtime efficiency across ConvNeXt, EfficientFormerV2, and Swin-Transformer on CIFAR-10.

Method	Clean Δ (pp)	Params (M)	Added Params (M)	GFLOPs (G)	Latency (ms)
Baseline	0.00	20.364	0.000	6.400	6.837
Ranger	0.00	20.364	0.000 (0.00%)	6.400 (+0.00%)	7.174 (+4.93%)
FT-ClipAct	−0.07	20.364	0.000 (0.00%)	6.400 (+0.00%)	7.961 (+16.44%)
FitAct	−0.54	27.174	6.810 (+33.44%)	6.429 (+0.45%)	9.953 (+45.68%)
ProAct	0.00	20.479	0.115 (+0.56%)	6.422 (+0.34%)	8.326 (+29.09%)
ReAnchor	−0.11	20.953	0.589 (+2.89%)	6.455 (+0.86%)	8.230 (+20.37%)

5.5 Computational and Runtime Overhead

We evaluate the computational and runtime overhead of ReAnchor on ConvNeXt, EfficientFormerV2, and Swin-Transformer. All methods are profiled on CIFAR-10 under the same RTX 4090 setting.

Table 3 shows that ReAnchor adds little parameter and FLOP overhead, with a millisecond-scale increase in runtime across the three backbones. Averaged over the three backbones, it adds 2.89% parameters and only 0.86% FLOPs, with a small clean-accuracy change of -0.11 percentage points. The average latency overhead is 20.37%, while the absolute increase remains between 1.35 and 1.42 ms across the three architectures. Detailed per-model overhead of ReAnchor is reported in Appendix C.4. Compared with the evaluated range-based methods, ReAnchor avoids the large parameter expansion of FitAct and the higher runtime overhead of FitAct and ProAct, while delivering stronger fault-recovery performance.

6 Conclusion

We studied how to improve soft-error robustness without modifying or retraining the already-trained backbone and task head, focusing on residual fault-induced errors at the backbone–head interface. In the residual-error regime considered here, our analysis shows that range restriction alone is insufficient: some erroneous terminal representations remain within calibrated ranges, while others remain incorrect after projection. ReAnchor addresses this regime by combining fixed numerical stabilization with recovery conditioned on sample-aligned input context, while keeping the backbone and task head unchanged. Across three evaluated datasets and three backbone families, ReAnchor recovers a substantial fraction of these residual errors while largely preserving clean accuracy and provides larger robustness gains than the evaluated range-based methods, particularly at higher fault rates. The exposed-recovery experiments further show that ReAnchor remains above the baseline when the recovery module itself is included in the fault domain. These results support input-conditioned post-backbone representation recovery as a low-intrusion strategy for the residual-error regime considered here under the evaluated fault model. Future work will investigate device-level fault measurements and deployment on physical edge platforms.

AI USE STATEMENT

In this work, we used generative AI tools only to assist with polishing the manuscript, including improvements to grammar, wording, clarity, and readability. We did not use generative AI tools for other tasks requiring disclosure, and the remaining disclosure categories are not applicable to this work. All AI-assisted edits were reviewed and revised by the authors to ensure technical accuracy and consistency with the methods and results presented in the paper. The authors take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide the implementation and evaluation details needed to reproduce ReAnchor and the reported results. Appendix A specifies the ReAnchor configuration, post-training objective, training algorithm, and optimization settings. Appendix B describes the datasets, backbone and baseline implementations, fault-injection procedure, paired-evaluation protocol, metrics, statistical reporting, and runtime measurement. Appendix C provides the complete robustness results and additional analyses, including terminal-range sensitivity and post-range recovery. Together with the method and threat-model definitions in the main text, these materials specify the model configuration, fault model, training procedure, and evaluation protocol used in our experiments.

REFERENCES

Mohammad Hasan Ahmadilivani, Mahdi Taheri, Jaan Raik, Masoud Daneshtalab, and Maksim Jenihhin. A systematic literature review on hardware reliability assessment methods for deep neural networks. *ACM Computing Surveys*, 56(6):1–39, 2024.

Chandramouli Amarnath, Mohamed Mejri, Kwondo Ma, and Abhijit Chatterjee. Error resilience in deep neural networks using neuron gradient statistics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1149–1162, 2024.

Cristiana Bolchini, Luca Cassano, Antonio Miele, and Alessandro Toschi. Fast and accurate error simulation for cnns against soft errors. *IEEE Transactions on Computers*, 72(4):984–997, 2023.

Cristiana Bolchini, Luca Cassano, and Antonio Miele. Resilience of deep learning applications: A systematic literature review of analysis and hardening techniques. *Computer Science Review*, 54: 100682, 2024.

Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. Sniff: Reverse engineering of neural networks with fault attacks. *IEEE Transactions on Reliability*, 71(4):1527–1539, 2022.

Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. Binfi: An efficient fault injector for safety-critical machine learning systems. In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23, 2019.

Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. A low-cost fault corrector for deep neural networks through range restriction. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–13, 2021.

Fernando Fernandes Dos Santos, Niccolò Cavagnero, Marco Ciccone, Giuseppe Averta, Angeliki Kritikakou, Olivier Sentieys, Paolo Rech, and Tatiana Tommasi. Improving deep neural network reliability via transient-fault-aware design and training. *IEEE Transactions on Emerging Topics in Computing*, 13(3):829–840, 2025.

Behnam Ghavami, Mani Sadati, Zhenman Fang, and Lesley Shannon. Fitact: Error resilient deep neural networks via fine-grained post-trainable activation functions. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1239–1244. IEEE, 2022.

Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Making convolutions resilient via algorithm-based error detection techniques. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2546–2558, 2022.

Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1241–1246, 2020.

Haitong Huang, Xinghua Xue, Cheng Liu, Ying Wang, Tao Luo, Long Cheng, Huawei Li, and Xiaowei Li. Statistical modeling of soft error influence on neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):4152–4163, 2023.

Younis Ibrahim, Haibin Wang, Junyang Liu, Jinghe Wei, Li Chen, Paolo Rech, Khalid Adam, and Gang Guo. Soft errors in dnn accelerators: A comprehensive review. *Microelectronics Reliability*, 115:113969, 2020.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Yann Le, Xuan Yang, et al. Tiny imagenet visual recognition challenge. *CS 231N*, 7(7):3, 2015.

Yanyu Li, Ju Hu, Yang Wen, Georgios Evangelidis, Kamyar Salahi, Yanzhi Wang, Sergey Tulyakov, and Jian Ren. Rethinking vision transformers for mobilenet size and speed. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 16889–16900, 2023.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11966–11976, 2022.

Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Vicarte, Sarita Adve, Christopher Fletcher, Iuri Frosio, and Siva Hari. PyTorchFI: A runtime perturbation tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pp. 25–31, 2020.

Seyedhamidreza Mousavi, Mohammad Hasan Ahmadilivani, Jaan Raik, Maksim Jenihhin, and Masoud Daneshtalab. Proact: Progressive training for hybrid clipped activation function to enhance resilience of dnns. *arXiv preprint arXiv:2406.06313*, 2024.

Lucas Roquet, Fernando Fernandes dos Santos, Paolo Rech, Marcello Traiola, Olivier Sentieys, and Angeliki Kritikakou. Cross-layer reliability evaluation and efficient hardening of large vision transformers models. In *2024 61st ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2024.

Hyein Shin, Myeonggu Kang, and Lee-Sup Kim. Fault-free: A framework for analysis and mitigation of stuck-at-fault on realistic reram-based dnn accelerators. *IEEE Transactions on Computers*, 72(7):2011–2024, 2023.

Jing Wang, Jinbin Zhu, Xin Fu, Di Zang, Keyao Li, and Weigong Zhang. Enhancing neural network reliability: Insights from hardware/software collaboration with neuron vulnerability quantization. *IEEE Transactions on Computers*, 73(8):1953–1966, 2024.

Syed Talal Wasim, Kabila Haile Soboka, Abdulrahman Mahmoud, Salman Khan, David Brooks, and Gu-Yeon Wei. Hardware resilience properties of text-guided image classifiers. In *Advances in Neural Information Processing Systems*, volume 36, pp. 67886–67899, 2023.

Xinghua Xue, Cheng Liu, Ying Wang, Bing Yang, Tao Luo, Lei Zhang, Huawei Li, and Xiaowei Li. Soft error reliability analysis of vision transformers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(12):2126–2136, 2023.

Anlan Yu, Shusen Jing, Ning Lyu, Wujie Wen, and Zhiyuan Yan. Error correction output codes for robust neural networks against weight-errors: A neural tangent kernel point of view. In *Advances in Neural Information Processing Systems*, volume 37, pp. 82493–82513, 2024.

Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen. SAVE: Software-implemented fault tolerance for model inference against GPU memory bit flips. In *2025 USENIX Annual Technical Conference*, pp. 1585–1604, 2025.

A Implementation Details

This section summarizes the ReAnchor configuration, post-training procedure, and optimization settings. The already-trained backbone and task head remain fixed, and only ReAnchor is optimized. ReAnchor operates at the backbone–head interface and receives no fault-specific information during inference.

A.1 ReAnchor Configuration

ReAnchor uses the same recovery architecture across all evaluated backbones, with only the hidden dimension adapted to the terminal channel width. Let C denote the terminal channel dimension. We set

$$D = \min\left(\frac{C}{4},\, 128\right), \tag{9}$$

which gives $D = 56$ for EfficientFormerV2 ($C = 224$) and $D = 128$ for ConvNeXt and Swin-Transformer ($C = 768$).

ReAnchor operates on a spatial terminal feature $Z \in \mathbb{R}^{B\times C\times H\times W}$. ConvNeXt and EfficientFormerV2 provide the terminal representation in this spatial form directly. The Swin-Transformer output is instead arranged as a token sequence of shape $[B, HW, C]$; before applying ReAnchor, we transpose and reshape it to the common spatial layout,

$$[B, HW, C] \;\rightarrow\; [B, C, HW] \;\rightarrow\; [B, C, H, W]. \tag{10}$$

In our experiments, $H = W = 7$. This operation only changes the tensor layout and introduces no additional learnable parameters.

The recovery module contains $K = 3$ blocks and operates on 51 tokens: one global token, one repair token, and 49 spatial tokens. The input-condition encoder uses 12 channels and extracts context at 7×7, 3×3, and 1×1 resolutions. Within each recovery block, the token-mixing MLP follows $51 \to 25 \to 51$, while the channel-mixing MLP uses an expansion ratio of four, $D \to 4D \to D$. The condition features are projected through a lightweight $12 \to D/2 \to D$ mapping before modulating the recovery blocks. The recovered terminal feature is then passed through the original interface $\mathcal{Q}$ and the frozen task head.

A.2 Post-Training Objective and Optimization

The backbone B_θ and task head H_ψ remain frozen during post-training, and only the ReAnchor parameters ϕ are optimized. We use the post-training objective defined in equation 8, with $\lambda_{\text{cln}} = 1$, $\lambda_f = 1$, $\lambda_{\text{feat}} = 0.5$, and $\lambda_{\text{int}} = 1$. The two task terms are cross-entropy losses on the clean and faulted paths. $\mathcal{L}_{\text{feat}}$ aligns the recovered fault feature with the corresponding clean terminal reference using Smooth-L_1 and cosine penalties, while $\mathcal{L}_{\text{int}}$ aligns their pooled representations. Clean reference features are detached from the computation graph.

Algorithm 1 summarizes the post-training procedure. For every mini-batch, ReAnchor is first evaluated on the clean path. A faulted path is then sampled with probability p_f. When the faulted path is active, a transient fault realization $\delta \sim p_{\text{FI}}^{\text{train}}$ is applied to a temporary copy of the frozen backbone and discarded after the current forward pass. The sampled fault information is used only by the injector and is not provided to ReAnchor. The backbone B_θ, the interface $\mathcal{Q}$, the task head H_ψ, and the calibrated projection ranges Ω remain fixed; only ϕ is updated. AdamW is initialized once before post-training and retained across epochs.

We use batch size 128, a fixed learning rate of 10^{-4}, weight decay 10^{-4}, and the default AdamW coefficients $(\beta_1, \beta_2) = (0.9, 0.999)$ with $\epsilon = 10^{-8}$. ConvNeXt, EfficientFormerV2 and Swin-Transformer are post-trained for 300 epochs. The fault path is sampled with $p_f = 0.8$, and the supplied configuration uses a fixed training fault rate $r_{\text{tr}} = 10^{-5}$. The training fault rate is used only by the injector.

Algorithm 1 Post-training of ReAnchor

Require: Frozen backbone B_θ; fixed interface $\mathcal{Q}$; frozen task head H_ψ; ReAnchor R_ϕ; calibrated ranges Ω; training set $\mathcal{D}$; fault-path probability p_f; training fault distribution $p_{\mathrm{FI}}^{\mathrm{train}}$; loss weights $\lambda_{\mathrm{cln}}, \lambda_f, \lambda_{\mathrm{feat}}, \lambda_{\mathrm{int}}$

Ensure: Post-trained ReAnchor parameters ϕ

Initialize AdamW on ϕ with learning rate 10^{-4} and weight decay 10^{-4}
for epoch $= 1, \ldots, E$ **do**
  for mini-batch $(x, y) \sim \mathcal{D}$ **do**
    Zero gradients of R_ϕ
    $Z^{\mathrm{cln}} \leftarrow B_\theta(x)$
    $Z^{p,\mathrm{cln}} \leftarrow \Pi_\Omega\big(S_\Omega(Z^{\mathrm{cln}})\big)$
    $\widehat{Z}^{\mathrm{cln}} \leftarrow R_\phi(Z^{p,\mathrm{cln}}, x)$
    $\mathcal{L}_{\mathrm{task}}^{\mathrm{cln}} \leftarrow \mathrm{CE}\Big(H_\psi\Big(\mathcal{Q}(\widehat{Z}^{\mathrm{cln}})\Big), y\Big)$
    Detach $Z^{p,\mathrm{cln}}$ and $\mathcal{Q}(Z^{p,\mathrm{cln}})$ as clean references
    Sample $b \sim \mathrm{Bernoulli}(p_f)$
    $\mathcal{L}_{\mathrm{task}}^{f} \leftarrow 0, \quad \mathcal{L}_{\mathrm{feat}} \leftarrow 0, \quad \mathcal{L}_{\mathrm{int}} \leftarrow 0$
    if $b = 1$ **then**
      Sample $\delta \sim p_{\mathrm{FI}}^{\mathrm{train}}$
      $\widetilde{B}_\theta \leftarrow \mathrm{InjectFaults}(\mathrm{Copy}(B_\theta), \delta)$
      $Z^f \leftarrow \widetilde{B}_\theta(x; \delta)$
      $Z^{p,f} \leftarrow \Pi_\Omega\big(S_\Omega(Z^f)\big)$
      $\widehat{Z}^f \leftarrow R_\phi(Z^{p,f}, x)$
      $\mathcal{L}_{\mathrm{task}}^{f} \leftarrow \mathrm{CE}\Big(H_\psi\Big(\mathcal{Q}(\widehat{Z}^f)\Big), y\Big)$
      $\mathcal{L}_{\mathrm{feat}} \leftarrow \mathrm{SmoothL1}\Big(\widehat{Z}^f, \mathrm{sg}(Z^{p,\mathrm{cln}})\Big) + 0.25\Big[1 - \cos\Big(\mathrm{vec}(\widehat{Z}^f), \mathrm{vec}(\mathrm{sg}(Z^{p,\mathrm{cln}}))\Big)\Big]$
      $\mathcal{L}_{\mathrm{int}} \leftarrow \mathrm{SmoothL1}\Big(\mathcal{Q}(\widehat{Z}^f), \mathrm{sg}\big(\mathcal{Q}(Z^{p,\mathrm{cln}})\big)\Big)$
      Release $\widetilde{B}_\theta$ after the faulted forward pass
    end if
    $\mathcal{L} \leftarrow \lambda_{\mathrm{cln}}\mathcal{L}_{\mathrm{task}}^{\mathrm{cln}} + b\Big(\lambda_f\mathcal{L}_{\mathrm{task}}^{f} + \lambda_{\mathrm{feat}}\mathcal{L}_{\mathrm{feat}} + \lambda_{\mathrm{int}}\mathcal{L}_{\mathrm{int}}\Big)$
    Backpropagate $\mathcal{L}$ through R_ϕ only
    Update ϕ with AdamW
  end for
end for
return ϕ

B Experimental Configuration and Evaluation Protocol

This section provides the experimental configuration and evaluation protocol used throughout the paper. We describe the datasets, backbone and baseline implementations, fault-injection and paired-evaluation procedure, and the metrics and statistical reporting used for robustness and runtime evaluation.

B.1 Datasets, Models, and Baseline Implementation

We evaluate ConvNeXt, EfficientFormerV2, and Swin-Transformer with an input resolution of 224×224 on CIFAR-10, CIFAR-100, and Tiny ImageNet-200. Their backbone configurations and terminal interfaces are summarized in Table 4. In all cases, ReAnchor operates on the final spatial representation, while the original backbone–head interface and head are retained.

Clean training images are used for model training, ReAnchor post-training, and the calibration statistics required by range-based methods. All reported clean and fault-injected accuracies are measured on the corresponding test/validation split and are not used to optimize ReAnchor or any evaluation-time threshold. For every backbone–dataset pair, the same trained backbone checkpoint is used

Table 4: Backbone configurations and terminal interfaces used in our experiments.

Configuration	ConvNeXt	EfficientFormerV2	Swin-Transformer
Stage depths	$[3, 3, 9, 3]$	$[3, 3, 9, 6]$	$[2, 2, 6, 2]$
Stage widths	$[96, 192, 384, 768]$	$[32, 48, 120, 224]$	$[96, 192, 384, 768]$
Patch size	–	–	4
Attention heads	–	–	$[3, 6, 12, 24]$
Window size	–	–	7
Terminal attention blocks	–	2	–
Terminal feature	$768 \times 7 \times 7$	$224 \times 7 \times 7$	$768 \times 7 \times 7$
Interface $\mathcal{Q}$	Global average pooling	Global average pooling	Global average pooling
Task head	Original linear classifier	Original linear classifier	Original linear classifier

Table 5: Key implementation settings for the range-based baselines.

Method	Applied location	Granularity	Key setting
Ranger	Activation-module outputs	Layer-wise	Clean profiled bounds
FT-ClipAct	Activation-module outputs	Layer-wise	$N = 4$, $M = 3$, $\Delta_{\mathrm{AUC}} = 0.2$
FitAct	Activation-module outputs	Neuron-wise	150 epochs, LR 10^{-3}
ProAct	Activation-module outputs	Layer-/neuron-wise	50/100 epochs, LR 10^{-2}

for the undefended baseline models, range-based methods, and ReAnchor so that the comparison isolates the effect of the recovery method.

Adaptation and Implementation of Range-Based Baselines. Several range-based baselines were originally developed and evaluated primarily on CNNs with ReLU-family activations. To enable a consistent comparison on ConvNeXt, EfficientFormerV2, and Swin-Transformer, we adapt their activation profiling and restriction wrappers to the activation modules used by each backbone, including GELU where applicable. This adaptation changes only the insertion of the activation restriction; the threshold granularity, calibration rule, and optimization procedure of each baseline are otherwise retained.

All methods use clean activation statistics for calibration. The main implementation choices are summarized in Table 5.

Here, "activation-module outputs" denotes the outputs of the activation operators detected during profiling, where the corresponding range restriction is inserted. For FT-ClipAct, N is the maximum number of interval-search iterations, M is the minimum number of iterations before early termination is allowed, and Δ_{AUC} is the stopping threshold on the change in the fault-robustness AUC used by the interval search. For ProAct, "50/100 epochs" denotes 50 optimization epochs for the layer-wise stages and 100 epochs for the final neuron-wise refinement stage.

FT-ClipAct remains layer-wise in our implementation. FitAct optimizes neuron-wise clipping thresholds, whereas ProAct follows progressive threshold optimization with layer-wise restrictions followed by neuron-wise refinement in the final activation layer. For FitAct and ProAct, only the clipping thresholds are optimized; the already-trained backbone weights remain frozen.

B.2 Fault Injection and Paired Evaluation

We implement fault injection with PyTorchFI (Mahmoud et al., 2020) following the threat model in the main text. Unless otherwise stated, the backbone B_θ is the only exposed component, while the original interface $\mathcal{Q}$, task head H_ψ, and ReAnchor module R_ϕ remain protected. Section 5.4 additionally exposes R_ϕ to faults while keeping $\mathcal{Q}$ and H_ψ protected.

Bit-level fault model. For each evaluated fault rate, weight and activation faults are sampled independently at the same nominal rate unless otherwise stated. Each FP32 value in the exposed fault domain is independently selected according to the specified fault rate, and one of its 32 bit positions is sampled uniformly and flipped upon corruption. A fault realization for one forward execution is denoted by δ. Each realization is transient: weight perturbations are reverted immediately after that forward execution, and activation corruptions affect only the current forward. A new realization is sampled for the next forward execution.

Transient execution. Fault injection is performed independently for each forward execution. Before a faulted forward, the corresponding weight faults are applied to a clean backbone state and PyTorchFI hooks generate transient activation corruptions during execution. After the forward completes, the perturbed weights are restored and the activation faults are discarded. Consequently, no sampled fault realization is carried over to the next test forward.

Paired evaluation. We use paired fault realizations only to reduce comparison variance between inference paths. For each test forward, we first sample a transient realization δ. The faulty-backbone baseline and ReAnchor are then evaluated under the same realization: the identical weight corruption is replayed for the two corresponding forwards, and the Python, NumPy, PyTorch CPU/CUDA random-number-generator states together with the PyTorchFI runtime state are restored so that the activation-fault locations and bit-flip sequence are matched. Each path starts from the same fault-free backbone state, and the injected weight faults are removed after its forward execution. The paired executions should therefore be interpreted as two counterfactual evaluations of the same transient inference event, rather than as consecutive inference passes over which a fault persists. A new realization is sampled for the next test forward.

B.3 Metrics and Statistical Reporting

We report top-1 accuracy using the standard definition. The additional metrics used in the main text are defined below.

Accuracy change. For a mitigation path m, the paired accuracy change relative to the faulty-backbone baseline is reported in percentage points:

$$\Delta\mathrm{Acc} = 100 * (\mathrm{Acc}_{\mathrm{m}} - \mathrm{Acc}_{\mathrm{Faulty}}) . \tag{11}$$

Error correction and mitigation-induced harm. Let

$$\mathcal{E} = \left\{ i \,\middle|\, \hat{y}_i^{\mathrm{cln}} = y_i,\ \hat{y}_i^{f} \neq y_i \right\} \tag{12}$$

denote the samples that are correct under clean inference but incorrect under the faulty-backbone baseline. For a mitigated inference path with prediction $\hat{y}_i^m$, the error correction rate (ECR) measures the fraction of these fault-induced errors for which the correct prediction is restored:

$$\mathrm{ECR} = \frac{\#\left\{ i \in \mathcal{E} \,|\, \hat{y}_i^m = y_i \right\}}{|\mathcal{E}|}. \tag{13}$$

The mitigation-induced harm rate (MIHR) measures the fraction of predictions that are correct under the backbone baseline but become incorrect after mitigation:

$$\mathrm{MIHR} = \frac{\#\left\{ i \,\middle|\, \hat{y}_i^{f} = y_i,\ \hat{y}_i^m \neq y_i \right\}}{\#\left\{ i \,\middle|\, \hat{y}_i^{f} = y_i \right\}}. \tag{14}$$

Statistical reporting. Fault-injected accuracy and $\Delta\mathrm{Acc}$ are reported. The corresponding two-sided 95% confidence interval is

$$\bar{x} \pm t_{0.975,\, n-1} \frac{s}{\sqrt{n}}, \tag{15}$$

where $\bar{x}$ and s are the sample mean and standard deviation over n paired trials.

Inference latency. Inference latency is measured under the same hardware and software environment for all compared methods, using identical input resolution, batch size, numerical precision, and test samples. All models are evaluated in inference mode with gradient computation disabled. GPU warm-up iterations are excluded, and CUDA synchronization is performed immediately before and after each timed forward pass.

Only model forward time is included; data loading, host-to-device transfer, offline calibration, and post-training are excluded. We report the average per-image latency as

$$\mathrm{Latency} = \frac{\text{total timed inference time}}{\text{number of evaluated images}}. \tag{16}$$

Table 6: Sensitivity to terminal-range granularity on CIFAR-10 with EfficientFormerV2 at a fault rate of 10^{-6}. Metadata counts include both lower and upper bounds.

Granularity	Metadata	Clean	Faulty	ReAnchor	Δ Acc. vs. Faulty (pp)
$2HW$ (spatial)	98	0.970	0.698	**0.842**	**+14.4**
$2CHW$ (neuron-wise)	21,952	0.969	0.699	0.832	+13.3
$2C$ (channel-wise)	448	0.969	0.699	0.834	+13.5
2 (global)	2	0.969	0.698	0.832	+13.4

The same timing protocol and number of measured forward passes are used across all methods to ensure a consistent runtime comparison.

C Additional Robustness Results and Analysis

This section provides supplementary results that extend the robustness evaluation in the main text. We first analyze the sensitivity of ReAnchor to terminal-range granularity, then report the complete numerical results across datasets, backbones, and fault rates, and finally examine the recovery of fault-induced errors that remain unresolved after range correction.

C.1 Terminal-Range Fine-Grained Sensitivity Analysis

ReAnchor uses channel-shared spatial bounds as a deployment-oriented design choice. To study how the bound granularity affects recovery performance, we compare several terminal-range settings.

We evaluate CIFAR-10 with EfficientFormerV2 at a fault rate of 10^{-6}. The terminal feature has size $C \times H \times W = 224 \times 7 \times 7$. We compare four empirical min/max parameterizations: channel-shared spatial bounds ($2HW$), neuron-wise bounds ($2CHW$), channel-wise bounds ($2C$), and a global lower/upper pair (2). The four choices require 98, 21,952, 448, and 2 calibration values, respectively.

As shown in Table 6, all four settings preserve clean accuracy, while the corresponding faulty accuracies remain nearly identical at 0.698–0.699. The channel-shared spatial design reaches a mean ReAnchor accuracy of 0.842, compared with 0.832–0.834 for the alternative granularities, and provides the largest mean paired gain at $+14.4$ pp. The main distinction is the calibration cost. Neuron-wise bounds increase the metadata from 98 to 21,952 values, a $224\times$ expansion, without improving mean recovery accuracy. Channel-wise and global bounds are more compact, but discard spatially varying range information and do not improve the recovery result. These results support the $2HW$ channel-shared spatial design used in the main method as a practical balance: it retains location-specific terminal statistics, substantially reduces calibration metadata relative to $2CHW$, and achieves the highest mean recovery accuracy among the evaluated granularities.

C.2 Full Robustness Results

Tables 7–9 report the complete numerical results corresponding to the main robustness comparison. We evaluate all methods at five shared fault rates, 10^{-7}, 10^{-6}, 3×10^{-6}, 5×10^{-6}, and 10^{-5}, using 100 paired full-test-set evaluation runs for each setting. Within each paired trial, the compared inference paths are evaluated under the same sampled fault realization. We report mean top-1 accuracy over the 100 paired fault realizations.

The complete results are consistent with the trends reported in Section 5.2. ReAnchor achieves the highest mean accuracy in seven of the nine dataset–backbone settings at 10^{-6}, and in all nine settings at each of 3×10^{-6}, 5×10^{-6}, and 10^{-5}. At 10^{-5}, relative to the best-performing comparator in each setting, selected from the unmitigated model and the four range-based methods, ReAnchor improves top-1 accuracy by 7.3–50.6 percentage points, with an average gain of 31.4 percentage points across the nine dataset–backbone settings.

Table 7: Mean top-1 accuracy for CIFAR-10 at the five fault rates.

Backbone	Fault rate	Baseline	Ranger	FT-ClipAct	FitAct	ProAct	ReAnchor
ConvNeXt	10^{-7}	0.883	0.907	0.906	0.904	0.907	0.905
	10^{-6}	0.413	0.503	0.506	0.506	0.507	0.601
	3×10^{-6}	0.139	0.179	0.180	0.180	0.180	0.530
	5×10^{-6}	0.117	0.122	0.127	0.127	0.127	0.574
	10^{-5}	0.100	0.100	0.100	0.100	0.100	0.606
EfficientFormerV2	10^{-7}	0.937	0.941	0.942	0.927	0.939	0.955
	10^{-6}	0.706	0.790	0.803	0.769	0.779	0.843
	3×10^{-6}	0.387	0.568	0.586	0.538	0.538	0.692
	5×10^{-6}	0.223	0.378	0.410	0.363	0.360	0.651
	10^{-5}	0.119	0.203	0.240	0.198	0.197	0.630
Swin-Transformer	10^{-7}	0.913	0.938	0.938	0.939	0.939	0.936
	10^{-6}	0.589	0.579	0.592	0.592	0.592	0.733
	3×10^{-6}	0.266	0.255	0.258	0.258	0.258	0.584
	5×10^{-6}	0.166	0.161	0.162	0.162	0.162	0.554
	10^{-5}	0.101	0.105	0.105	0.105	0.105	0.590

Table 8: Mean top-1 accuracy for CIFAR-100 at the five fault rates.

Backbone	Fault rate	Baseline	Ranger	FT-ClipAct	FitAct	ProAct	ReAnchor
ConvNeXt	10^{-7}	0.741	0.808	0.808	0.803	0.811	0.808
	10^{-6}	0.405	0.407	0.412	0.409	0.413	0.537
	3×10^{-6}	0.060	0.087	0.089	0.089	0.089	0.297
	5×10^{-6}	0.019	0.032	0.036	0.036	0.036	0.307
	10^{-5}	0.011	0.010	0.010	0.010	0.010	0.337
EfficientFormerV2	10^{-7}	0.817	0.825	0.824	0.787	0.823	0.832
	10^{-6}	0.588	0.675	0.687	0.638	0.665	0.685
	3×10^{-6}	0.294	0.436	0.472	0.413	0.427	0.499
	5×10^{-6}	0.122	0.262	0.303	0.254	0.255	0.420
	10^{-5}	0.025	0.098	0.143	0.100	0.101	0.378
Swin-Transformer	10^{-7}	0.826	0.826	0.825	0.824	0.826	0.845
	10^{-6}	0.478	0.481	0.486	0.485	0.486	0.606
	3×10^{-6}	0.143	0.154	0.160	0.160	0.160	0.394
	5×10^{-6}	0.086	0.065	0.068	0.068	0.068	0.370
	10^{-5}	0.010	0.015	0.015	0.015	0.015	0.351

C.3 Recovery of Errors Unresolved by Range Correction

We further examine the errors that remain unresolved after range mitigation and whether they can be recovered by ReAnchor. For each fault realization, the same faulty terminal feature is used to obtain the range-corrected and full ReAnchor predictions, allowing us to directly evaluate the recovery of these remaining errors. Specifically, $\mathcal{E}_{\text{in}}^{\text{RA}}$ contains fault-induced errors whose faulty terminal features remain within the calibrated range, $\mathcal{E}_{\text{vio-rec}}^{\text{RA}}$ contains range-violating errors whose predictions become correct after range projection, and $\mathcal{E}_{\text{vio-unres}}^{\text{RA}}$ contains range-violating errors whose predictions remain incorrect after our range projection. We denote the proportions of these three subsets among all fault-induced errors as ρ_{in}, $\rho_{\text{vio-rec}}$, and $\rho_{\text{vio-unres}}$, respectively, with $\rho_{\text{in}} + \rho_{\text{vio-rec}} + \rho_{\text{vio-unres}} = 1$. We further report R_{in} and R_{vio} to measure the fractions of $\mathcal{E}_{\text{in}}^{\text{RA}}$ and $\mathcal{E}_{\text{vio-unres}}^{\text{RA}}$, respectively, that are subsequently recovered by full ReAnchor. In addition, $\mathrm{PRRpost}$ measures the overall recovery rate over all errors left unresolved by the range stage, i.e., $\mathcal{E}_{\text{in}}^{\text{RA}} \cup \mathcal{E}_{\text{vio-unres}}^{\text{RA}}$.

The decomposition reveals different fault modes across architectures. Across all settings, range projection directly corrects only a small subset of fault-induced errors, while the learned repair module contributes the larger share of recovery, with $\mathrm{PRR}_{\text{post}}$ ranging from 0.404 to 0.634 for the remaining errors. At 10^{-6}, range projection directly corrects 0.081, 0.111, and 0.067 of the fault-induced errors for ConvNeXt, EfficientFormerV2, and Swin-Transformer, respectively. For ConvNeXt and EfficientFormerV2, the remaining errors are dominated by unresolved range violations, with $\rho_{\text{vio-unres}} = 0.850$ and 0.887. Although ρ_{in} reaches 0.395 in Swin-Transformer, full ReAnchor still recovers 0.213 of these within-range errors and 0.619 of the unresolved range-violating errors.

Table 9: Mean top-1 accuracy for Tiny ImageNet-200 at the five fault rates.

Backbone	Fault rate	Baseline	Ranger	FT-ClipAct	FitAct	ProAct	ReAnchor
ConvNeXt	10^{-7}	0.860	0.797	0.798	0.742	0.796	0.861
	10^{-6}	0.371	0.380	0.404	0.376	0.403	0.716
	3×10^{-6}	0.062	0.070	0.084	0.078	0.084	0.494
	5×10^{-6}	0.016	0.019	0.030	0.030	0.030	0.391
	10^{-5}	0.005	0.005	0.005	0.005	0.005	0.279
EfficientFormerV2	10^{-7}	0.787	0.795	0.795	0.715	0.793	0.797
	10^{-6}	0.552	0.653	0.661	0.577	0.640	0.657
	3×10^{-6}	0.268	0.425	0.459	0.377	0.416	0.460
	5×10^{-6}	0.113	0.255	0.290	0.229	0.247	0.301
	10^{-5}	0.021	0.093	0.134	0.087	0.094	0.207
Swin-Transformer	10^{-7}	0.801	0.799	0.799	0.762	0.798	0.802
	10^{-6}	0.467	0.456	0.469	0.447	0.468	0.533
	3×10^{-6}	0.171	0.142	0.153	0.146	0.153	0.318
	5×10^{-6}	0.061	0.056	0.062	0.060	0.062	0.236
	10^{-5}	0.005	0.011	0.011	0.011	0.011	0.210

Table 10: Error decomposition and post-range recovery on CIFAR-10. ρ_{in}, $\rho_{\text{vio-rec}}$, and $\rho_{\text{vio-unres}}$ denote the fractions of fault-induced errors in $\mathcal{E}^{\text{RA}}_{\text{in}}$, $\mathcal{E}^{\text{RA}}_{\text{vio-rec}}$, and $\mathcal{E}^{\text{RA}}_{\text{vio-unres}}$, respectively. R_{in} and R_{vio} denote full-ReAnchor recovery within the corresponding unresolved subsets. Ratios are rounded to three decimals; "–" indicates an empty subset.

Backbone	Fault rate	ρ_{in}	$\rho_{\text{vio-rec}}$	$\rho_{\text{vio-unres}}$	R_{in}	R_{vio}	PRR_{post}
ConvNeXt	10^{-7}	< 0.001	0.207	0.793	0.300	0.461	0.461
	10^{-6}	0.069	0.081	0.850	0.002	0.436	0.404
	10^{-5}	< 0.001	0.001	0.999	–	0.597	0.597
EfficientFormerV2	10^{-7}	0.001	0.111	0.888	0.429	0.571	0.571
	10^{-6}	0.002	0.111	0.887	0.295	0.570	0.569
	10^{-5}	0.001	0.108	0.890	0.155	0.634	0.634
Swin-Transformer	10^{-7}	0.131	0.098	0.771	0.254	0.616	0.564
	10^{-6}	0.395	0.067	0.538	0.213	0.619	0.447
	10^{-5}	0.080	0.102	0.818	0.245	0.619	0.585

These results are consistent with the design of ReAnchor, where range projection provides an initial correction and the learned repair module accounts for most of the subsequent recovery.

C.4 Runtime Overhead of ReAnchor

We provide the per-model computational and runtime costs of ReAnchor in Table 11. For parameter count, the Backbone and Head columns report the parameter counts of the original model components, while $+\Delta$ ReAnchor denotes the additional parameters introduced by the recovery module. For FLOPs and latency, Baseline reports the cost of the original inference path, and $+\Delta$ ReAnchor reports the additional cost introduced by ReAnchor. Across the three evaluated architectures, ReAnchor adds 0.145–0.810M parameters, 0.01–0.07 GFLOPs, and 1.35–1.42 ms of inference latency per image.

Across the three architectures, the absolute overhead introduced by ReAnchor remains small in terms of model size and arithmetic complexity. Specifically, ReAnchor adds 0.810M parameters to ConvNeXt and Swin-Transformer, corresponding to approximately 2.9% of their original model sizes, while only 0.145M additional parameters are required for EfficientFormerV2 (approximately 2.5%). The additional computation is similarly limited: ReAnchor introduces only 0.01–0.07 GFLOPs, which is below 1% of the baseline FLOPs for all three architectures. In contrast, the relative latency overhead is more noticeable and varies with the runtime characteristics of the underlying backbone. The absolute latency increase is highly consistent across architectures, ranging only from 1.35 to 1.42 ms per image, despite substantial differences in their baseline latency. Consequently, the relative increase is larger for the faster ConvNeXt baseline (approximately 35.0%), compared with approximately 13.9% for EfficientFormerV2 and 21.0% for Swin-Transformer. This discrepancy

Table 11: Per-model absolute computational and runtime overhead of ReAnchor.

	Parameters (M)			FLOPs (G)		Latency (ms)	
Model	Backbone	Head	$+\Delta$ ReAnchor	Baseline	$+\Delta$ ReAnchor	Baseline	$+\Delta$ ReAnchor
ConvNeXt	27.820	0.008	0.810	8.91	0.07	4.03	1.41
EfficientFormerV2	5.736	0.002	0.145	1.30	0.01	9.71	1.35
Swin-Transformer	27.519	0.008	0.810	8.99	0.07	6.77	1.42

between the small FLOP increase and the more visible latency increase indicates that the runtime cost of the recovery module is not fully captured by arithmetic complexity alone. Nevertheless, the nearly architecture-independent absolute latency increment suggests that ReAnchor introduces a relatively stable post-backbone processing cost rather than an overhead that scales proportionally with the computational complexity of the backbone.