ENGINEERING Information Technology & Electronic Engineering
https://www.academax.com/EITEE; https://www.engineering.org.cn
https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=11344065
E-mail: eng.itee@zju.edu.cn

Engineering Journals

Available on IEEE Xplore®

Research Article
https://doi.org/10.1631/ENG.ITEE.2025.0104

RetryTrigger: intelligent inference duplication for enhancing LLM resilience to hardware transient faults

Jiajia JIAO✉, Yixu YU

College of Information Engineering, Shanghai Maritime University, Shanghai 201306, China

Abstract: Large language models (LLMs) have exhibited outstanding performance across a wide range of natural language processing (NLP) tasks. However, the rising prevalence of hardware transient faults has made silent data corruptions (SDCs) in LLMs increasingly problematic, severely degrading output quality and user experience. State-of-the-art protection schemes primarily rely on hardware-assisted algorithm-based fault tolerance (ABFT) or boundary-setting-driven online fault tolerance (FT2) for selective layers, yet these solutions suffer from strict hardware dependencies, substantial overhead, or incomplete coverage. To address these limitations, we propose RetryTrigger, a novel hardware-free fault-aware inference methodology capable of handling all potential faults. During LLM inference, RetryTrigger dynamically collects runtime output features (e.g., maximum probability, top-k probability gaps, output entropy, logits statistics, and inference latency), which are used to train a LightGBM meta-model. This meta-model accurately predicts whether duplicate inference should be performed, thereby effectively mitigating faults while preserving efficiency without additional hardware dependence. Extensive experiments on seven representative LLMs (including T5-Small, RoBERTa, BioMedBERT, Qwen2.5-Coder-0.5B/7B, MiniMind, and Opt) demonstrate that RetryTrigger reduces SDC rates by up to 95.33% (on average 92.97%) and achieves a minimal performance overhead of 2.4012% (on average 4.1167%), offering a superior balance between reliability and efficiency compared to state-of-the-art solutions.

Key words: Large language models; System resilience; Intelligent fault detection; Inference duplication; Transient faults

1 Introduction

Large language models (LLMs), primarily built upon Transformer-based architectures, excel at modeling long-range dependencies through self-attention mechanisms. Leveraging these capabilities, LLMs have achieved remarkable success across a diverse spectrum of natural language processing (NLP) tasks, ranging from sentiment analysis and text generation to language translation and code synthesis (Radford et al., 2019; Zhang et al., 2023; Jiang et al., 2024). The massive-scale pre-training of LLMs enables them to rapidly learn complex linguistic patterns, thereby facilitating their widespread adoption in both academic research and industrial applications.

However, the rapid growth in model size, coupled with prolonged computational workloads, has made LLMs increasingly vulnerable to hardware transient faults. Such faults can be triggered by cosmic ray strikes, voltage fluctuations, or thermal instability, and have become one of the predominant threats to system reliability (Baumann, 2005). Unlike permanent hardware defects, transient faults cause temporary bit upsets that may corrupt intermediate computations and even lead to system crashes. When occurring within LLM architectures during the inference phase, these faults can result in silent data corruptions (SDCs), thereby degrading output quality and diminishing user experience. The impact is even more critical in safety-sensitive domains, such as medical diagnosis (Zhou et al., 2025) and legal document analysis (Battaglini-Fischer et al., 2025), which necessitate non-negotiable resilience against hardware transient faults.

Fault-tolerance research for LLMs spans the training and inference phases. Training-stage defenses often include system-level solutions such as optimized checkpointing and elastic scheduling (Li et al., 2025; Wan et al., 2025), as well as algorithm-based fault tolerance (ABFT) and fault-aware training (Cavagnero et al., 2022; Liang et al., 2025). While effective for long-running distributed training, these methods

✉ Jiajia JIAO, jiaojiajia@shmtu.edu.cn
Jiajia JIAO, https://orcid.org/0000-0003-3680-787X
Yixu YU, https://orcid.org/0009-0003-3257-1393
CLC number: TP391

Received: Oct. 28, 2025; Revision accepted: Mar. 4, 2026;
Crosschecked: Mar. 17, 2026; Published online: Apr. 10, 2026

© The Authors 2026. Published by Zhejiang University Press Co., Ltd. This is an open access article distributed under the terms of the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0/)

are rarely tailored for the strict latency constraints of inference. Consequently, inference-stage protections have received increasing attention to mitigate SDCs caused by fault amplification. These works are often classified into the following three categories: (1) Boundary setting policy. Activation clipping and range restriction (Hoang et al., 2019; Chen ZT et al., 2021; Mousavi et al., 2024; Roquet et al., 2024) limit intermediate activation magnitudes to suppress fault propagation. Although effective for small deep neural networks (DNNs), in LLMs, they often require costly offline profiling (Sun et al., 2025) and still leave partial coverage gaps. An illustrative extension of activation clipping for LLMs is first-token-inspired online fault tolerance (FT2) (Sun et al., 2025), which achieves online fault tolerance via a token-boundary setting mechanism. These bounds are derived from the first token's statistics and applied to subsequent tokens, avoiding costly offline profiling. But the FT2 coverage is limited to selected layers and depends on the representativeness of early-token activations, leaving potential faults undetected in unprotected regions. (2) Inference-time ABFT. To achieve cost-effective design, a diversity of ABFT variants are proposed to inject checksums, statistical detectors, or adaptive recomputation into attention and normalization kernels (Dai et al., 2025; Xie et al., 2025; Xue et al., 2025), mitigating fault impacts with the modest overhead. However, these techniques demand targeted instrumentation and incur persistent runtime costs. (3) Redundancy-based schemes (Venkatesha and Parthasarathi, 2024) replicate critical computations and compare outputs. Despite their robustness, even selective duplication significantly increases computational and energy consumption, limiting their applicability in large-scale deployments.

Existing LLM inference-stage protections either rely on offline profiling, suffer from incomplete coverage, or impose sustained runtime overhead. Furthermore, training-stage defenses are not suited for protecting live inference. To bridge this gap, we introduce RetryTrigger, a novel fault-aware inference methodology that leverages lightweight runtime features and a learned meta-model to intelligently trigger duplication only when necessary. This design achieves comprehensive coverage and high accuracy in detecting potential faults, while keeping the average overhead in the low single-digit percentage range and requiring no hardware modifications or time-consuming profiling. The key contributions of this work are summarized as follows:

1. Fault-aware detection via Runtime feature exploitation. We design a lightweight fault-aware classifier that uses runtime features such as maximum probability, entropy, and logits statistics, trained with LightGBM. It achieves over 95% detection accuracy while incurring minimal runtime overhead and offering strong interpretability.

2. Flexible plug-and-play re-execution framework at the software level. Our pure-software two-stage RetryTrigger framework dynamically flags suspicious inferences using the offline pre-trained classifier, and selectively triggers online automatic recomputation. This plug-and-play architecture requires zero hardware modifications or model retraining, ensuring broad applicability.

3. Proven resilience-overhead tradeoff superiority. Extensive experiments demonstrate that RetryTrigger reduces SDC rates by an average of 92.97% and up to 95.33% under low fault rates, surpassing state-of-the-art approaches. The framework maintains a mean runtime overhead of only 4.1167% and as low as 2.4012% in the best case, offering a practical solution for safety-critical LLM applications.

We make the artifact of RetryTrigger publicly available on GitHub (https://github.com/ltbz/RetryTrigger).

2 Background

This section establishes the essential background for our work. We first review the architecture of LLMs with a focus on the inference phase. We then define the hardware fault model considered in this study and detail the fault injection methodology employed to rigorously validate the effectiveness of RetryTrigger.

2.1 LLM inference architecture

Transformer-based LLMs generate sequences autoregressively, predicting one token at a time, conditioned on all previously generated tokens. During each decoding step, the input token embedding is processed through a stack of Transformer blocks, each consisting of the following key components:

1. Multi-head self-attention (MHSA). Calculate attention weights from query (Q), key (K), and value (V) projections to integrate contextual information across all tokens.

2. Feed-forward network (FFN). Apply nonlinear transformations to the attention output.

3. Residual connections and layer normalization. Facilitate gradient flow and stabilize numerical scaling.

Upon traversing all blocks, the final hidden state is projected into output logits via a linear projection, followed by SoftMax to produce the next-token probability distribution. This cycle repeats iteratively until the end-of-sequence token is produced.

Due to the sequential dependency, a transient fault occurring in any single decoding step does not remain isolated; instead, it can propagate and accumulate throughout subsequent iterations. This fact significantly magnifies the potential impact of transient faults on LLM output quality.

2.2 Fault model and injection

We focus explicitly on transient bit flips in intermediate activation tensors produced during a model forward pass. We exclude persistent storage faults (e.g., permanent DRAM/cached parameter corruption or persistent register faults). Instead, our fault model targets ephemeral errors that are likely to originate from arithmetic logic units (ALUs) or on-chip transient buffers during inference.

Concretely, faults are modeled as bit-flips in the 16-bit bfloat16 representation of activation values. Two fault severity levels are considered: single-bit upset (SBU), where one randomly selected bit of an activation element is inverted, and double-bit upset (DBU), where two distinct bits within the same element are flipped. To emulate the sporadic nature of transient upsets, each injection experiment introduces at most one fault event.

Fault injection is performed at the model level

by corrupting the output tensors of randomly selected torch.nn.Linear layers (e.g., $Q/K/V$ projections and FFN outputs). For each experimental run, we conduct the following three points:

1. Choose a linear layer uniformly at random.

2. Pick a random spatial coordinate within that layer's output activation tensor via sequence position and hidden index.

3. Trigger the bit-flip at a randomly chosen temporal point in execution. For autoregressive models, this is a random decoding step; for single-pass models, it is within the single forward pass.

Our implementation uses PyTorch forward hooks (register_forward_hook) to intercept and modify the target tensor in-place. Hooks are registered and removed per run to ensure independence across trials.

Faulty outputs are compared to a golden, fault-free execution to identify SDCs. Key metrics include SDC rate, cosine similarity to the golden output, and the detection/mitigation performance of RetryTrigger. This methodology provides precise control over fault location, timing, and severity while remaining faithful to the transient, non-persistent fault model.

3 Proposed method: RetryTrigger

3.1 Motivation

Let f_θ denote LLMs with parameters θ and vocabulary $\mathcal{V}$, generating tokens $y_{1:T}$ via autoregressive decoding. At each decoding step t, the model produces logits $\boldsymbol{z}_t \in \mathbb{R}^{|\mathcal{V}|}$, and a probability distribution $\boldsymbol{p}_t = \mathrm{SoftMax}(\boldsymbol{z}_t)$ is derived. Transient faults can momentarily corrupt intermediate arithmetic operations or memory accesses. These faults alter the logits $\boldsymbol{z}_t$ and subsequent probability $\boldsymbol{p}_t$, leading to SDC in the generated sequence $y_{1:T}$ without raising any crash or exception.

For example, given the prompt "What is deep learning?", a faulty decoding step may yield "Deep learning is a branch of machine learning, specialized in ? royal engineering performance.". Similarly, in an English-to-French translation task, "What is your name?" may incorrectly be rendered as "Qu' est emergency votre nom?". Such outputs are semantically wrong yet syntactically plausible, making detection challenging in production environments.

A straightforward mitigation is to retry every decoding step, but this approach is incompatible with latency and cost budgets in large-scale deployments. A more common heuristic accepts a token only when the maximum probability $p_{t,\max} = \max_i p_{t,i}$ exceeds a threshold. However, transient faults can produce miscalibrated distributions—e.g., inflated logits yielding a deceptively high $p_{t,\max}$ while the token is incorrect.

Our key insight is that SDC-indicative patterns are often subtle yet multi-faceted, manifesting as abnormal probability entropy, atypical top-k gaps, skewed logits statistics, and runtime latency spikes. This motivates the design of a lightweight meta-model that consumes a compact set of runtime features and predicts whether a retry is warranted at step t. Such adaptive retry preserves latency and computational budgets while requiring no retraining or structural modifications to the underlying LLM.

3.2 Problem formulation

1. Preliminaries. Given an LLM f_θ with parameters θ and vocabulary $\mathcal{V}$; an autoregressive inference process producing $y_{1:T}$, with per-step probabilities $\boldsymbol{p}_t \in \mathbb{R}^{|\mathcal{V}|}$ derived from logits $\boldsymbol{z}_t$; a transient fault model $\mathcal{F}$ that injects bit-flips or arithmetic errors during inference.

2. Decision strategy. Let $\boldsymbol{\phi}_t \in \mathbb{R}^d$ be a feature vector extracted at step t, where features may include statistical measures of $\boldsymbol{p}_t$ and $\boldsymbol{z}_t$ (entropy, top-k gaps, and logits moments) and system-level signals (step latency). We define a decision function: $g: \mathbb{R}^d \rightarrow \{\mathrm{retry}, \mathrm{accept}\}$, which determines whether decoding step t should be re-executed.

3. Objective. The goal is to minimize the probability of producing an SDC under the fault model $\mathcal{F}$, subject to a strict latency constraint: $\min_g \Pr_{\mathcal{F}}[\mathrm{SDC}]$ s.t. $\mathrm{Latency}(g) \leq L_{\max}$, where $L_{\max}$ is the maximum tolerable inference latency for the system.

This formulation frames the selective retry problem as a lightweight binary classification task. It aims to intercept fault-induced errors at the earliest possible decoding step with minimal computational overhead, thereby balancing reliability and performance in the LLM reference.

3.3 RetryTrigger framework

The proposed RetryTrigger adopts a two-stage design as shown in Fig. 1: offline fault characterization and online dynamic mitigation.

1. Offline stage

We first curate a diverse set of random inputs tailored to the specific application domains of target LLMs (e.g., QA for MiniMind, translation for T5-Small, sentiment analysis for RoBERTa, biomedical fill-mask for BioMedBERT, code completion for Qwen2.5-Coder, and text completion for Opt). For each input, a fault-free inference pass without fault injection is processed to obtain the golden reference output sequence. Subsequently, we perform faulty inference runs by injecting an SBU or DBU into the output activation tensor of a randomly selected linear layer. PyTorch's forward hook mechanism is adopted for this injection, targeting these layers due to the dense computation and high vulnerability to transient hardware faults.

During each faulty run, we extract eleven runtime features characterizing the model's internal state, including probability distribution metrics (e.g., entropy and max probability), top-k probability gaps, and step latency. Labels retry $\in \{0, 1\}$ are first assigned programmatically by comparing the faulty output with the fault-free reference (retry $= 1$ denotes a deviation and regeneration and retry $= 0$ indicates a match), and subsequently manually verified on a subset of samples to ensure semantic coherence and avoid SDC rate amplification. The resulting tabular dataset is cleaned and split (80% for training/20% for validation) to train a LightGBM classifier, chosen for its efficiency and superior performance on structured features. The trained meta-model is then serialized (.pkl) for deployment.

2. Online stage

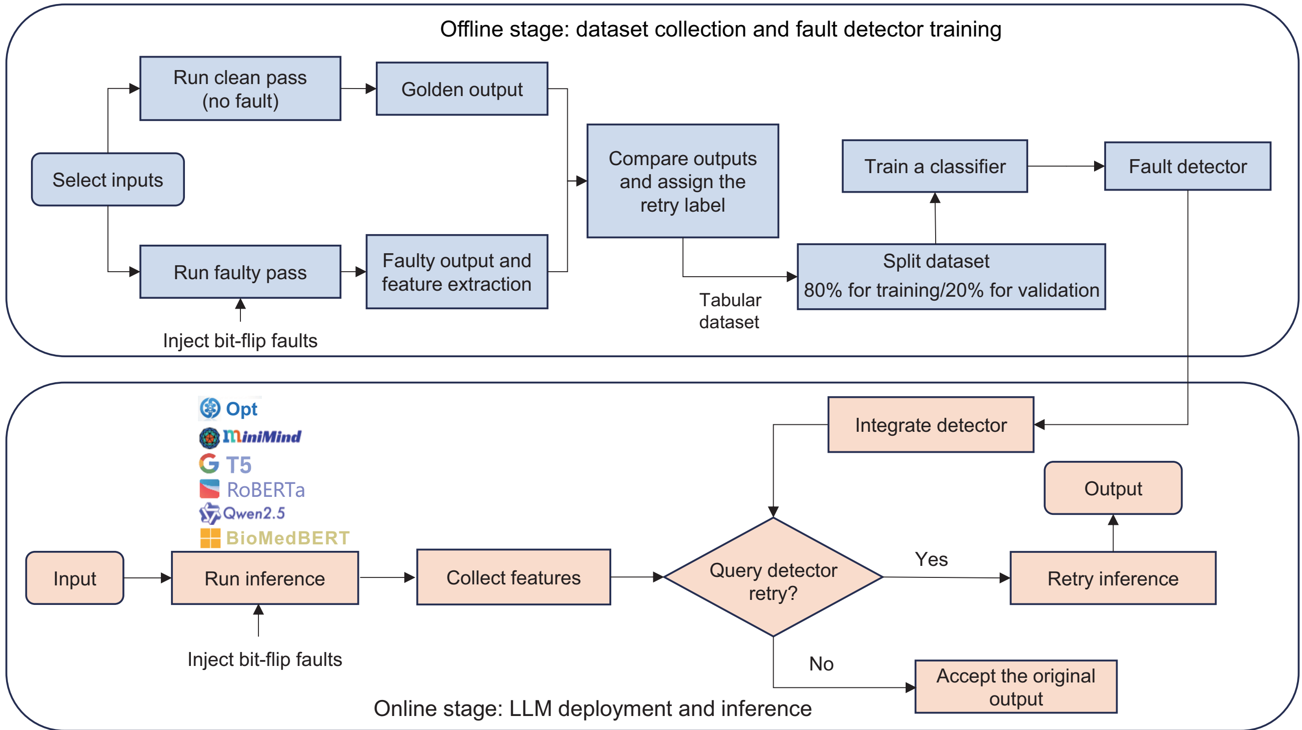

Fig. 1 The RetryTrigger pipeline, which comprises two main stages: (1) offline data curation, which involves collecting runtime features from the model under controlled fault injection to build a dataset for the LightGBM detector; (2) online deployment, where the trained detector analyzes an LLM's runtime features in real time to predict whether a retry is necessary

During inference, the same runtime features are monitored at each decoding step. The pre-trained LightGBM classifier is queried in real time, and if retry = 1, the current step is immediately re-executed. As transient faults are non-persistent, a single re-computation is typically sufficient to recover the correct state. Otherwise, if retry = 0, the original output is accepted without re-execution. This fault-aware selective retry mechanism introduces negligible latency, ensuring practicality in real-world deployment scenarios.

3.4 Feature selection

1. Feature groups

In the offline phase of RetryTrigger, we extract a compact vector of 11 lightweight runtime features for each decoding step or generated sequence in Table 1. These features are computed from tensors already available during LLM inference, such as logits $\boldsymbol{z}$ and probability vector $\boldsymbol{p}$ from per-step timing measurements, incurring negligible computational overhead. We categorize these features into three groups based on the specific fault manifestations they are designed to capture:

(1) Confidence and universal features (max_prob, top2_prob, top3_prob, top2_gap)

These features quantify the model's confidence and the separability between candidate tokens. For example, max_prob represents the peak probability, while top2_gap measures the margin between the leading candidate and the runner-up. In benign reference, these values are stable. However, transient faults often disrupt this stability, causing probability distributions to either collapse (uncertainty) or artificially inflate (overconfidence). Confidence features should be universally across models and tasks because they relate to fundamental properties of a well-formed output distribution (low uncertainty and clear preference).

(2) Distributional shape and model-specific features (entropy, logits_mean, logits_std, skewness, kurtosis, top10_prob_mass)

These features characterize the global geometry of the logits and probability vectors:

(a) entropy measures the overall uncertainty; faults can flatten distributions (increasing entropy) or spuriously sharpen them (lowering entropy).

(b) logits_mean and logits_std summarize the first two moments of logits, detecting the baseline shift or inflated dispersion by bit flips.

(c) skewness and kurtosis capture higher-order effects such as asymmetry and peakedness, revealing faults that symmetrize or flatten the distribution.

(d) top10_prob_mass quantifies the concentration of probability among top-ranked tokens. Benign outputs are concentrated, while faults "leak" probability to irrelevant candidates. Some shape features (e.g., entropy and logits_std) may be universally useful, while others (logits_mean, skewness, kurtosis, and top10_prob_mass) are model-specific or task-dependent due to natural variation in logits scale and shape

Table 1 Summary of the 11 runtime features used in RetryTrigger. Each feature is computed from tensors readily available during inference (logits z, probabilities p, runtime Δt) and incurs negligible extra cost. Features are grouped into three categories with corresponding a priori hypotheses regarding their universality across models and environments

Feature	Definition/Formula	Rationale	Group	Hypothesis
max_prob	$p_{(1)} = \max_i p_i$	Proxy for local confidence; faults can spuriously inflate or depress confidence	Confidence	Universal
top2_gap	$p_{(1)} - p_{(2)}$	Measures separation between the best and the second-best tokens; faults distort typical gap patterns	Confidence	Universal
entropy	$-\sum_i p_i \log p_i$	Quantifies global uncertainty; faults either flatten or sharpen the distribution abnormally	Distributional shape	Universal
top2_prob	$p_{(2)}$	Complements $p_{(1)}$, capturing secondary probability peaks	Confidence	Universal
top3_prob	$p_{(3)}$	Further complements $p_{(1)}$, revealing abnormal tail behaviors	Confidence	Universal
logits_mean	$\mu = \frac{1}{V}\sum_i z_i$	First moment of logits; faults can shift the baseline mean	Distributional shape	Model-specific
logits_std	$\sigma = \sqrt{\frac{1}{V}\sum_i (z_i - \mu)^2}$	Dispersion of logits; extreme values from faults often inflate the standard deviation	Distributional shape	Universal
runtime	Δt per forward pass	Behavioral side-channel; faults may co-occur with micro-stalls or hardware hiccups	Behavioral	Environment-specific
top10_prob_mass	$\sum_{i=1}^{10} p_{(i)}$	Concentration of probability mass; flattening from faults leaks mass to irrelevant tokens	Distributional shape	Model-specific/Task-dependent
skewness	$\mathbb{E}\left[\left(\frac{z_i-\mu}{\sigma}\right)^3\right]$	Asymmetry of logits; confident outputs are right-skewed, and faults reduce skewness	Distributional shape	Model-specific
kurtosis	$\mathbb{E}\left[\left(\frac{z_i-\mu}{\sigma}\right)^4\right] - 3$	Peakedness of logits; faults flatten peaks, lowering kurtosis	Distributional shape	Model-specific

across architectures.

(3) Behavioral and environment-specific features (runtime)

The feature records per-step latency, serving as a side-channel indicator. Transient faults can cause micro-stalls, cache misses, or memory delays, subtly increasing runtime. This feature is environment-specific. Its reliability depends strongly on the hardware/software stack and fault manifestation, making it less consistently robust.

2. Feature significance analysis

To quantify the contribution of each runtime feature for different LLMs, we conduct SHapley Additive exPlanation (SHAP) analysis on the full feature set to evaluate their role in fault detection. Our analysis reveals a key insight that the optimal detector for each model often relies on a specialized subset of features rather than the entire set. This finding provides important evidence of the model-specific nature of fault signatures, bridging the gap between the apparent feature importance shown in the SHAP summaries (Fig. 2) and the actual performance observed on the validation set.

(1) Overcoming feature dominance for robustness. An unexpected finding is that removing a feature ranked as highly influential in the SHAP summary can improve detection performance. A prominent case is BioMedBERT, where excluding logits_mean, a dominant contributor in the full-model SHAP profile, yielded the highest F1-score. This suggests that an over-reliance on a single dominant feature may "mask" other subtle but informative signals. By removing logits_mean, the model is compelled to exploit a broader set of distributional features, such as skewness and kurtosis, leading to a more balanced and resilient decision boundary.

(2) Shift from confidence metrics to distributional morphology. For models such as RoBERTa and MiniMind, the optimal feature subsets notably omit conventional confidence-based metrics, including max_prob and top2_gap. Instead, peak performance is achieved by leveraging features that characterize the statistical moments of the logits distribution: logits_mean (for RoBERTa), logits_std, skewness, and kurtosis. This shift suggests that for these models, the global shape and higher-order statistical structure of the logits vector provide a more reliable fault indicator than the confidence value assigned to a single top-ranked token.

(3) Model-specific reliance on side-channel signals. The ablation studies confirm the model-dependent utility of certain behavioral features. For example, runtime emerged as a critical signal for T5-Small, consistent with its high SHAP ranking. In contrast, excluding runtime yielded better results for RoBERTa and Qwen2.5-Coder. This divergence suggests that latency-based side-channel indicators can be valuable but are not universally reliable, with their effectiveness strongly influenced by a specific LLM architecture (e.g., T5-Small's encoder–decoder design) and its interaction with transient hardware faults.

Experimental results across seven LLMs demonstrate that a lightweight detector, trained on our selected eleven features, can accurately and efficiently identify transient faults. Moreover, the consistently high recall for the retry=1 class demonstrates its practical utility as a mechanism for enhancing the reliability of LLM inference in real-world deployments.

3.5 Offline self-built dataset

We constructed a dataset of 11 runtime features under fault injections. For each task–LLM pair, we first performed a fault-free pass to produce a deterministic golden output. The binary label retry $\in \{0,1\}$ is determined through a

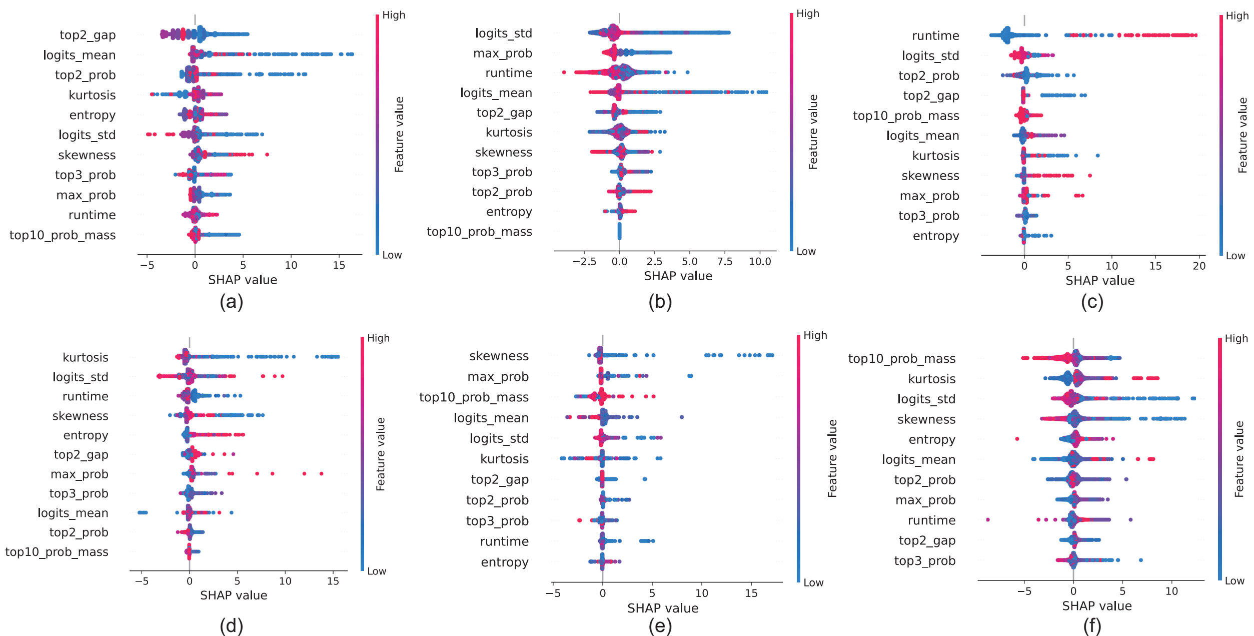

Fig. 2 SHAP summary plots for the trained RetryTrigger detector on each of the seven LLMs, based on the models trained with the full feature set. For each feature, the plot shows the impact of its value (color-coded from low/blue to high/red) on the model's prediction for the positive class (SDC=1). Positive SHAP values push the prediction toward "retry", while negative values push it toward "no-retry". Subfigures (a–f) correspond to the seven LLMs: (a) BioMedBERT; (b) RoBERTa; (c) T5-Small; (d) Qwen2.5-Coder-0.5B/7B; (e) MiniMind; (f) Opt

hybrid approach of initial programmatic assignment followed by manual verification to distinguish between benign linguistic variations and verifiable SDC. This label serves as the target variable for the meta-model and is excluded from the input feature set. Subsequently, for each of the R faulty runs, we injected a single transient fault (either SBU or DBU) by a PyTorch forward hook on a randomly selected linear layer, corrupting one element of the output activation at a random step.

1. Encoder-only architectures (e.g., RoBERTa and BioMedBERT). As shown in Table 2, these LLMs produce a single final logits vector $\boldsymbol{z}_{\text{final}}$ per inference run. Specifically, for sentiment analysis tasks (RoBERTa), $\boldsymbol{z}_{\text{final}}$ corresponds to the sequence-level class logits; for biomedical masked language modeling (fill-mask) tasks (BioMedBERT), $\boldsymbol{z}_{\text{final}}$ corresponds to the token-level logits at the [MASK] position. The feature vector $\boldsymbol{\phi}$ is extracted directly from this $\boldsymbol{z}_{\text{final}}$, yielding one training sample per run.

2. Encoder–decoder and decoder-only architectures (e.g., T5-Small and Qwen2.5-Coder). These LLMs, also detailed in Table 2, generate sequences token-by-token. During each faulty run, we collected the logits $\{\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_T\}$ at every decoding step using a custom LogitsProcessor. Runtime features are computed at each step and aggregated—via arithmetic mean—into a single summary feature vector $\boldsymbol{\phi}_{\text{agg}}$ representing the entire generation process. For this category, runtime is measured as the total wall-clock time to generate the full output sequence.

As Algorithm 1 shows, the offline dataset is collected and then preprocessed to train a fault-aware detector.

1. Dataset statistics

We constructed a comprehensive dataset comprising a total of 122 659 samples derived from seven distinct LLMs and diverse tasks, as summarized in Table 2: MiniMind (text QA), T5-Small (translation), RoBERTa (sentiment analysis), BioMedBERT (biomedical fill-mask), Qwen2.5-Coder-0.5B/7B (code completion), and Opt (text completion). For BioMedBERT and RoBERTa, we selected 50 random input instances for each task, performing 200 runs per instance to generate 10 000 samples per model. For MiniMind, T5-Small, Qwen2.5-Coder, and Opt, we selected 10 random input instances for each task, with 100 runs per instance. Given that these models generate sequences token-by-token and each token constitutes an individual sample for our detector, the total sample counts for these models are significantly higher: 44 033 for MiniMind, 11 616 for T5-Small, 20 229 for Qwen2.5-Coder, and 26 781 for Opt. Each entry in the dataset consists of 11 runtime features accompanied by a binary retry label.

2. Preprocessing

Before training, data cleaning is applied by capping extreme feature values (e.g., to $\pm 10^{10}$) to mitigate infinities and outliers, followed by filling any remaining NaN (short for not a number) values with zeros.

The final dataset is partitioned into training (80%) and validation (20%) subsets via a stratified split. This stratification is conditioned on both the LLMs and the retry label, ensuring distributionally representative sets.

Table 2 Statistics of the Offline Self-built Dataset across different LLM architectures

Architecture	Model	Task	Granularity	Number of instances	Number of total samples
Encoder-only	RoBERTa	Sentiment analysis	Sequence (logits)	50	10 000
	BioMedBERT	Biomedical fill-mask	Token ([MASK] position, logits)	50	10 000
Encoder–decoder	T5-Small	Translation	Token (logits)	10	11 616
Decoder-only	MiniMind	Text Q&A	Token (logits)	10	44 033
	Qwen2.5-Coder-0.5B/7B	Code completion	Token (logits)	10	20 229
	Opt	Text completion	Token (logits)	10	26 781
Total	Seven models			140	122 659

Q&A: question and answering

Algorithm 1 Self-built dataset for fault-aware detector

Require: Task-matched input $\mathcal{X}$; a set of LLMs $\{f^{(m)}\}$; the number of injection runs R.
Ensure: The preprocessed dataset $\mathcal{D}$ and its training/validation subsets.
1: Initialize an empty dataset for records, $\mathcal{D} \leftarrow \varnothing$
2: **for** each model $f^{(m)}$ and input $x \in \mathcal{X}$ **do**
3: **Clean pass:** Decode x with $f^{(m)}$ to obtain the golden output $y^\star$
4: **for** $r_{\text{run}} = 1$ **to** R **do**
5: Randomly configure a fault (SBU or DBU) at a spatio-temporal location (layer, coordinate, and step)
6: Register a forward hook on the target linear layer to inject the fault during inference
7: **Faulty pass:**
8: **if** $f^{(m)}$ is generative **then**
9: Decode x to obtain the faulty output $\hat{y}$ and collect per-step logits $\{\boldsymbol{z}_t\}$
10: Extract features from each $\boldsymbol{z}_t$ and aggregate them into a summary vector $\boldsymbol{\phi}_{\text{agg}}$
11: **else**
12: $f^{(m)}$ is classificational
13: Perform forward pass on x to obtain the final logits $\boldsymbol{z}_{\text{final}}$ and faulty prediction $\hat{y}$
14: Extract features $\boldsymbol{\phi}$ from $\boldsymbol{z}_{\text{final}}$
15: **end if**
16: Assign label $y_{\text{label}} = \mathbf{1}\{\hat{y} \neq y^\star\}$ and append the record $\{\boldsymbol{\phi}_{\text{features}}, y_{\text{label}}\}$ to $\mathcal{D}$
17: Remove the forward hook
18: **end for**
19: **end for**
20: Preprocess dataset $\mathcal{D}$: cap extreme values (e.g., to $\pm 10^{10}$) and handle NaN/inf /* inf is short for infinity */
21: Split $\mathcal{D}$ into training (80%) and validation (20%) subsets

3.6 Fault-aware detector selection and training

1. Classifier selection. Our feature space is low-dimensional, tabular in nature, and exhibits nonlinear feature interactions.

From a theoretical standpoint, gradient-boosted decision trees (GBDTs) are intrinsically well-suited to this regime: (1) the ability to model nonlinear decision boundaries effectively without manual feature transformation; (2) robustness to heterogeneous feature scales; (3) native support for weighted loss functions to mitigate class imbalance; (4) intrinsic interpretability via feature importance scores. Among GBDT implementations, LightGBM (Ke et al., 2017) takes advantage of its histogram-based algorithms and leaf-wise growth strategy for high-throughput, low-latency inference, making it uniquely suitable for integration into real-time fault detection and protection with negligible computational overhead.

From an empirical perspective, we benchmarked multiple candidate models, including logistic regression (Hosmer et al., 2013) and XGBoost (Chen TQ and Guestrin C, 2016) on the collected dataset. As detailed in Section 4, LightGBM consistently achieved the most favorable performance and efficiency trade-off. It delivered the highest recall on the positive class (retry=1), and competitive F1-scores, while maintaining sub-millisecond inference latency. This combination of theoretical suitability and empirical validation justifies our adoption of LightGBM as the meta-classifier for fault-aware retry detection.

2. Training setup. We implemented a binary classifier using the LGBMClassifier implementation from the lightGBM library. To address the inherent class imbalance, where correct outputs (retry = 0) dominate, we set the scale_pos_weight parameter to the ratio of negative to positive samples in the training set. To identify the optimal feature subset for each LLM, we performed extensive ablation studies in Section 4, which guided the final feature selection. Subsequently, the most effective model configuration was trained separately for each LLM. The ensemble of trained fault-aware detectors was serialized and consolidated into a single .pkl artifact.

3.7 Online retry integration for recovery

RetryTrigger integrates the trained detector to determine whether an LLM's output should be accepted or re-executed. The integration mechanism is tailored to the operational characteristics of each LLM architecture.

1. Initial inference pass. The LLM executes a standard inference to produce an initial output $\hat{y}$, which may be affected by transient faults.

2. Feature extraction and decision making. During the initial inference, we extracted runtime features and queried the RetryTrigger classifier to obtain a decision. The procedure adapts to the model type:

(1) Encoder-only models (e.g., RoBERTa and BioMedBERT). A feature vector $\boldsymbol{\phi}$ is directly extracted from the final output logits $\boldsymbol{z}_{\text{final}}$. The RetryTrigger computes a binary decision $\hat{r} \in \{0, 1\}$ from $\boldsymbol{\phi}$.

(2) Encoder–decoder and decoder-only models (e.g., T5-Small and Qwen2.5-Coder). We implemented two distinct detection strategies to evaluate the impact of detection granularity: (a) Per-token decision. Per-step logits $\{\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_T\}$ are collected via a LogitsProcessor. For each decoding step t, a feature vector $\boldsymbol{\phi}_t$ is extracted and passed to the RetryTrigger

to yield a per-token decision $\hat{r}_t$. (b) Post-hoc decision. All per-step features are aggregated into a summary vector $\boldsymbol{\phi}_{\text{agg}}$, from which a sequence-level decision $\hat{r}_{\text{agg}}$ is obtained. The overall decision is computed as $\hat{r} = \max\left(\{\hat{r}_t\} \cup \{\hat{r}_{\text{agg}}\}\right)$. A retry is triggered if any token is flagged or if the aggregated features indicate an anomaly. These two strategies offer different trade-offs between sensitivity and context awareness, which are explicitly compared in the following Section 4.

3. Decision-driven action. (1) If $\hat{r} = 0$ (no retry): the initial output $\hat{y}$ is accepted as final. (2) If $\hat{r} = 1$ (retry): the initial output $\hat{y}$ is discarded, and a single full re-execution is performed with the same input and configuration. To bound latency, the output from this second pass is accepted unconditionally.

By enforcing a strict single-retry budget, we ensured deterministic latency bounds. The effectiveness of the proposed strategies in maximizing fault detection coverage is validated empirically in the following section.

4 Results and analysis

This section describes the experimental setup and a comprehensive evaluation of RetryTrigger across seven representative LLMs using fault injection. The comparative SDC rate, cosine similarity, and the corresponding computational overhead are used to verify the effectiveness of RetryTrigger. Finally, the limitations of the proposed framework are discussed.

4.1 Experimental configuration

1. Experimental configuration. To accommodate different model scales, our experiments were conducted on two distinct hardware configurations. For smaller models, we used an environment equipped with an NVIDIA GeForce RTX 3080, 20 GB, 12 virtual CPU (vCPU) Intel® Xeon® Platinum 8352V CPU @ 2.10 GHz, 48 GB RAM. For larger models (Qwen2.5-Coder-7B), we employed a higher-performance setup featuring an NVIDIA GeForce RTX 4080 Super, 32 GB. All experiments were implemented using PyTorch 2.4.1 with CUDA 12.1 support, integrating single and multibit-fault injection tools for neurons.

2. Diverse LLMs for different tasks. To ensure the broad applicability of RetryTrigger, we constructed training datasets for seven distinct LLMs, covering a wide spectrum of architectures (encoder-only, encoder–decoder, and decoder-only) and parameter scales. Notably, in addition to lightweight models, we explicitly incorporated Qwen2.5-Coder-7B to rigorously verify the framework's scalability to larger-parameter models and its robustness in handling complex generation tasks. This diverse selection allows us to evaluate whether our fault detection mechanism generalizes effectively across different model sizes and operational contexts.

Each LLM was evaluated on a designated canonical task. For full reproducibility, we explicitly defined the evaluation dataset(s) and the canonical prompt format for each model. In the reliability assessments (Figs. 3 and 4), 10 distinct and entirely unseen input instances were selected for each LLM. These prompts were randomly sampled from their respective raw datasets (e.g., magpie-SFT, WMT, SST-2, and PubMedQA), with strict verification to exclude any text strings used during the offline dataset collection phase. Evaluating on these independent prompts ensures that our reported SDC reduction results accurately reflect the true generalization capability of the RetryTrigger framework.

(1) MiniMind (text QA). Task: knowledge-based or generative question answering. Dataset(s): e.g., deepctrl-SFT or magpie-SFT (https://www.modelscope.cn/datasets/gongjy/minimind_dataset/files). Example prompt: "User: {Which country is Paris the capital of?} Assistant:".

(2) T5-Small (translation). Task: machine translation (source → target). Dataset(s): e.g., WMT dataset (https://huggingface.co/datasets/wmt/wmt14) or custom parallel corpora. Example prompt: "Translate to {fr}: {Renewable energy sources are crucial for a sustainable future.}".

(3) RoBERTa (sentiment classification). Task: sentence-level sentiment classification (e.g., 3-way). Dataset(s): e.g., SST-2 (https://huggingface.co/datasets/SetFit/sst2) or TweetEval sentiment dataset (https://huggingface.co/datasets/cardiffnlp/tweet_eval/viewer/sentiment). Example prompt: "Classify sentiment: {Today's movie has a wonderful plot!}".

(4) BioMedBERT (biomedical fill-mask). Task: masked token prediction/biomedical cloze. Dataset(s): e.g., PubMedQA dataset (https://huggingface.co/datasets/qiaojin/PubMedQA). Example prompt: "Complete the blank: {Enzymes act as biological [MASK] to accelerate reactions.}".

(5) Qwen2.5-Coder (0.5B and 7B) (code completion). Task: code autocompletion/snippet generation. Dataset(s): e.g., HumanEval, MBPP (https://huggingface.co/datasets/google-research-datasets/mbpp), or in-house code tasks. Example prompt: "Complete code: {def add_numbers(a, b): Returns the sum of two numbers.}".

(6) Opt (text completion). Task: autoregressive text generation and sequence completion. Dataset(s): e.g., Wikitext-103 (https://huggingface.co/datasets/Salesforce/wikitext) or custom writing completion set. Example prompt: "Complete the text: {The universe is vast and contains billions of galaxies. Each galaxy is made of}".

3. Classification metrics. The classification results on the validation split are reported with a focus on precision, recall, and F1-score for both class 0 (no-retry) and class 1 (retry). Given the reliability requirements to prevent SDCs from propagating to the final output, minimizing false negatives is paramount. Consequently, the recall of class 1 is prioritized as the primary performance metric.

4. Resilience metrics. We focused on two complementary metrics that quantify the presence and severity of SDC, alongside a reduction metric to measure efficacy.

(1) SDC rate. SDC rate is the fraction of evaluated requests for which the faulty output differs from the golden output obtained in a fault-free run:

$$\text{SDC_rate} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\{\text{output}_i^{\text{faulty}} \neq \text{output}_i^{\text{gold}}\} \times 100\%, \quad (1)$$

where the difference is measured based on criteria tailored to the specific tasks suited for particular LLMs. For encoder-only models, an SDC is identified by a top-1 prediction mismatch with respect to the task-specific output. Specifically,

for biomedical masked language modeling (fill-mask) tasks, an SDC occurs when the predicted token at the [MASK] position differs from the fault-free baseline; for sentiment analysis tasks, an SDC is detected when the top-1 predicted class label deviates from the baseline prediction.

For encoder–decoder and decoder-only models, we established a deterministic, fault-free baseline output by disabling sampling (e.g., employing greedy decoding or beam search). While any divergence from this baseline indicates that an injected fault has perturbed the model's computational path, our experimental observations reveal that certain deviations remain semantically and syntactically correct despite being string-dissimilar to the golden output.

Consequently, we do not categorize every string mismatch as an SDC; instead, we implemented a hybrid verification process to determine the final SDC status. Initially, labels (retry $\in \{0, 1\}$) are assigned programmatically by comparing the faulty output with the fault-free reference, where retry=1 denotes a deviation requiring regeneration and retry=0 indicates an exact match. To prevent SDC rate amplification, we subsequently conducted an exhaustive manual verification of 100% of the samples programmatically flagged as divergent (retry=1). During this human-in-the-loop inspection, the labeling criteria were strictly tailored to the task semantics. For natural language tasks such as translation (T5-Small), text QA (MiniMind), and text completion (Opt), a flagged output was manually reassigned to retry=0 if it was semantically equivalent to the baseline, grammatically correct, and free of garbled text, despite superficial lexical differences. Conversely, for code generation tasks (Qwen2.5-Coder), strict syntactic requirements rendered programmatic string matching highly accurate, necessitating minimal manual overrides. Ultimately, a fault is recorded as an SDC only if this analysis confirms a verifiable corruption in the output—such as a loss of original meaning, logical inconsistency, or structural breakdown—rather than a benign alternative expression. This rigorous methodology ensures that our reported SDC rates precisely reflect faults that genuinely compromise the model's utility.

(2) Cosine similarity. To quantify the severity of the deviation, we computed the cosine similarity between the vector representations of the faulty and golden outputs. The construction of vectors differs across models as follows:

(a) BioMedBERT (masked token logits). Vector: the logits at the masked position $\boldsymbol{z} \in \mathbb{R}^{|\mathcal{V}|}$, i.e., $\boldsymbol{v} = \text{logits}[0, \text{mask_idx}, :]$, where $|\mathcal{V}|$ is the vocabulary size and each element of $\boldsymbol{v}$ denotes the unnormalized prediction score for one vocabulary token. Interpretation: captures distributional shift in token prediction scores.

(b) RoBERTa (classification/sentiment). Vector: classification logits $\boldsymbol{v} \in \mathbb{R}^{C}$ ($C = \text{number of classes}$), i.e., $\boldsymbol{v} = \text{logits}[0]$. Interpretation: measures changes in label scores/confidence.

(c) Qwen2.5-Coder (code generation, hidden-state pooling). Vector: mean-pooled hidden states of the newly generated tokens at the last layer, $\boldsymbol{v} = \frac{1}{L}\sum_{i=1}^{L} \boldsymbol{h}_i^{(L)}$, where L denotes the number of newly generated tokens. Interpretation: captures semantic similarity between generated sequences.

(d) MiniMind (text QA; FastText averaging). Vector: mean of FastText word embeddings from valid tokens, $\boldsymbol{v} = \frac{1}{N}\sum_{w \in \mathcal{W}} \text{ft}(w)$, after segmentation and filtering. Zero vectors are returned if too few valid tokens remain. w denotes an individual valid token retained after segmentation and filtering, $\mathcal{W}$ denotes the set of all such valid tokens, $\text{ft}(w)$ denotes the FastText embedding of token w, and N denotes the number of valid tokens in $\mathcal{W}$. Therefore, N has the same meaning as the denominator in the formula, i.e., $N = |\mathcal{W}|$. Interpretation: reflects lexical-level similarity or shallow semantics.

(e) T5-Small (translation; FastText averaging for target language). Vector: mean of FastText embeddings for words in the translated sentence (e.g., French), $\boldsymbol{v} \in \mathbb{R}^{300}$. Interpretation: captures lexical/phrase-level similarity in the target language.

(f) Opt (text completion; FastText averaging). Vector: mean of FastText word embeddings from valid English tokens in the generated sequence, $\boldsymbol{v} = \frac{1}{N}\sum_{w \in \mathcal{W}} \text{ft}(w)$, after English-specific segmentation and filtering. Interpretation: reflects lexical-level semantic similarity of the generated English text.

(3) SDC reduction. To evaluate the effectiveness of RetryTrigger, we measure the relative reduction in the SDC rate, defined as

$$\text{SDC reduction} = \frac{\text{SDC}_{\text{before}} - \text{SDC}_{\text{after}}}{\text{SDC}_{\text{before}}} \times 100\%, \tag{2}$$

where $\text{SDC}_{\text{before}}$ is the baseline SDC rate without any protection, and $\text{SDC}_{\text{after}}$ is the residual SDC rate when RetryTrigger is active.

4.2 Comparison with state-of-the-art methods

Table 3 compares our proposed RetryTrigger with three representative state-of-the-art fault mitigation methods: ReaLM, FT2, and ALBERTA. Due to the unavailability of official source code or reproducible artifacts for these baselines, the reported values are extracted directly from their respective original publications. Therefore, the listed results represent aggregate or averaged metrics across different LLMs rather than exact one-to-one comparisons.

ReaLM (Xie et al., 2025) adopts a statistical ABFT approach combined with hardware–software co-design, performing runtime validation through embedded checksum logic. It achieves reliable inference with less than 0.29 perplexity degradation while maintaining under 2.00% additional circuit area and power overhead, making it suitable for hardware-level resilience enhancement. FT2 (Sun et al., 2025) introduces an online bound-setting mechanism that protects critical Transformer layers based on statistics from the first generated token. This method achieves approximately 92.92% SDC reduction but incurs a runtime overhead of about 3.42%, mainly due to the layer-wise dynamic range monitoring. ALBERTA (Liu et al., 2025) focuses on algorithm-based error resilience in Transformer architectures, employing checksum-based ABFT within GEMM kernels and selective replay to reach nearly 99.00% fault coverage. It achieves this with a very low computational overhead (below 0.20%), but its protection granularity is confined to matrix-multiplication submodules.

In contrast, RetryTrigger operates as a lightweight, pretrained fault detector, which uses runtime confidence and logits distribution features to decide whether to re-execute inference. It achieves about 86%–95% SDC reduction across

Table 3 Comparison of RetryTrigger with existing works. Note that since these methods use different metrics, models, and datasets, the reported values serve as a reference for their respective performance envelopes rather than a direct head-to-head comparison

Method	LLM(s) used	Core innovation	Reliability improvement	Overhead
ReaLM (Xie et al., 2025)	Opt-1.3B, LLaMA-2-7B, and LLaMA-3-8B	Statistical ABFT+circuit co-design	<0.29 perplexity degradation	<2.00% circuit area/power
FT2 (Sun et al., 2025)	Seven models (Opts, LLaMA, Qwens, ...)	Critical layers+online bounds (1st token)	~92.92%	~3.42% runtime
ALBERTA (Liu et al., 2025)	Vision Transformers (ViT and DeiTs)	Checksum ABFT for GEMM (FP aware)+replay	~99.00% coverage	<0.20% computational overhead
RetryTrigger (ours)	RoBERTa, BioMedBERT, T5-Small, Qwen2.5-Coder-0.5B/7B, MiniMind, and Opt	Lightweight runtime feature detection+flexible retry policy	~92.97%	~4.12% retry rate (computational overhead)/ 0.0090–1.0470 s absolute latency overhead

seven distinct LLMs (ranging from RoBERTa, BioMedBERT, T5-Small, MiniMind, Opt, and Qwen2.5-Coder-0.5B to larger Qwen2.5-Coder-7B) while incurring less than 5% latency overhead, making it practical for deployment in latency-sensitive systems.

4.3 Reliability improvements under variable configurations

This subsection presents a comprehensive analysis of the experimental results, characterizing the vulnerability of seven distinct LLMs to transient faults and validating the effectiveness of the proposed RetryTrigger.

1. Reliability improvements under different LLM inputs

Our evaluation considers distinct input samples to assess how different inputs affect the reliability of our method. As shown in Fig. 3, the baseline reliability without protection exhibits significant volatility across different LLMs and inputs.

Our results reveal that reliability is not a static model attribute. It is deeply intertwined with input semantics. First, the Opt model exemplifies sensitivity. Its baseline SDC rate

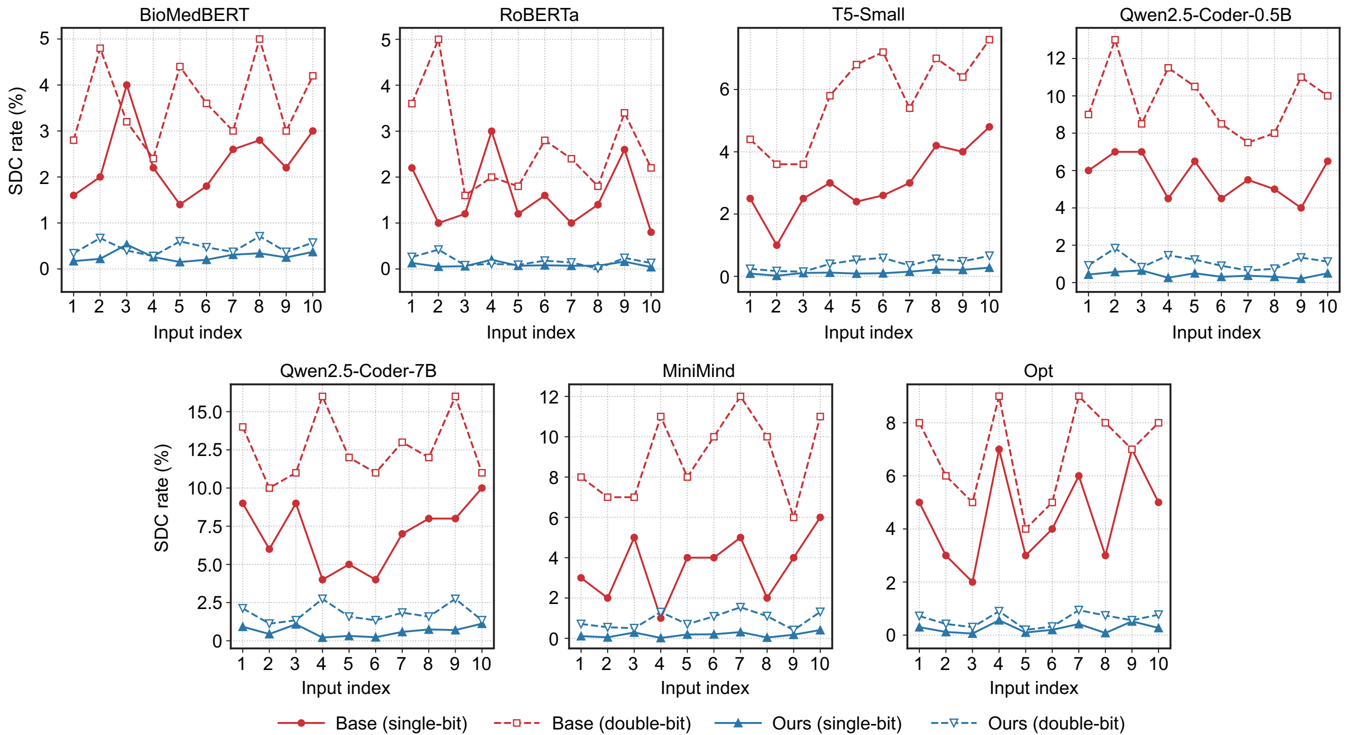

Fig. 3 Comparison of SDC rates across seven LLMs under SBU and DBU transient fault injections. Each model is tested with 10 distinct input samples. The red lines represent the baseline method without protection, showing high volatility and SDC rates. The blue lines represent our proposed RetryTrigger method. Solid lines denote SBU fault scenarios, while dashed lines denote DBU fault scenarios. The visualization highlights that RetryTrigger effectively mitigates SDC rates under different fault models compared to the baseline

under DBU fluctuates considerably, more than doubling from a minimum of 4.00% (input 5) to peaks of 9.00% (inputs 4 and 7). Second, this instability is even more pronounced in large-scale models like Qwen2.5-Coder-7B, where the vulnerability surges from 10.00% (input 2) to a maximum of 16.00% (input 4). This suggests that specific prompts trigger highly fragile neuron pathways. Third, even intrinsically robust encoder-only models such as BioMedBERT are not immune, showing a variance where the SDC rate rises from 2.40% (input 4) to 5.00% (input 8).

Significantly, RetryTrigger mitigates this input-dependent volatility, demonstrating robust and consistent protection regardless of the input complexity. As indicated by the flattened blue curves in Fig. 3, our method effectively suppresses the baseline SDC spikes. Taking the Opt model as an example, despite the baseline SDC rate spiking to 9.00% at input 7, RetryTrigger successfully constrains the effective SDC rate to just 0.94%, achieving an order-of-magnitude reduction. Similarly, for the most vulnerable case (Qwen2.5-Coder-7B), the protection remains stable, reducing the 16.00% baseline SDC rate to 2.74%. This confirms that our fault detector in RetryTrigger identifies intrinsic feature anomalies rather than being biased by specific input tokens.

2. Reliability improvements under different LLM tasks

Our evaluation covers diverse architectures, ranging from code generation (Qwen) to text classification (RoBERTa). As shown in Fig. 4, the baseline without protection resilience varies significantly. The Qwen2.5-Coder models are the most vulnerable, with SDC rates reaching 12.6% under DBU. This is likely due to the strict syntactic precision required for coding tasks. In contrast, encoder-only models like RoBERTa show higher intrinsic robustness with 2.66% SDC rate under DBU. The T5-Small model, an encoder–decoder architecture, exhibits a moderate level of resilience, positioning it between the vulnerable decoder-only models and the robust encoder-only models.

Crucially, RetryTrigger demonstrates consistent efficacy across this spectrum. Despite the high baseline vulnerability of decoder-only models, our method successfully suppresses the majority of errors. It achieves an impressive average SDC reduction of 92.97% under SBU and 89.90% under DBU. This universality confirms that RetryTrigger captures fundamental error patterns, regardless of the specific model architecture or task.

3. Reliability improvements under different fault models

A consistent trend observed across all LLMs in Fig. 3 is that DBU is significantly more disruptive than SBU. Comparing the average SDC rate in Fig. 4, every model exhibits a substantially higher baseline SDC rate under DBU conditions. For instance, the SDC rate for Qwen2.5-Coder-0.5B surges from 5.65% (SBU) to 9.75% (DBU), an increase of over 70%. Similarly, T5-Small nearly doubles from 3.00% to 5.78%. This aligns with physical intuition that multi-bit corruption induces larger numerical deviations.

However, RetryTrigger remains highly resilient against this increased severity. While DBUs cause more frequent errors, they also generate distinct feature anomalies, such as extreme logit shifts. Our detector effectively leverages these traces. Consequently, RetryTrigger achieves a peak SDC reduction of 95.33% under SBU and 93.61% under DBU, respectively. This proves that our protective capability does not degrade, even when facing more destructive DBU fault models.

4. Performance comparison of different intelligent fault detectors

To identify the optimal meta-classifier for RetryTrigger, we compared three representative methods: LightGBM (Ke et al., 2017), XGBoost (Chen TQ and Guestrin C, 2016), and logistic regression (Hosmer et al., 2013). Fig. 5 visualizes the

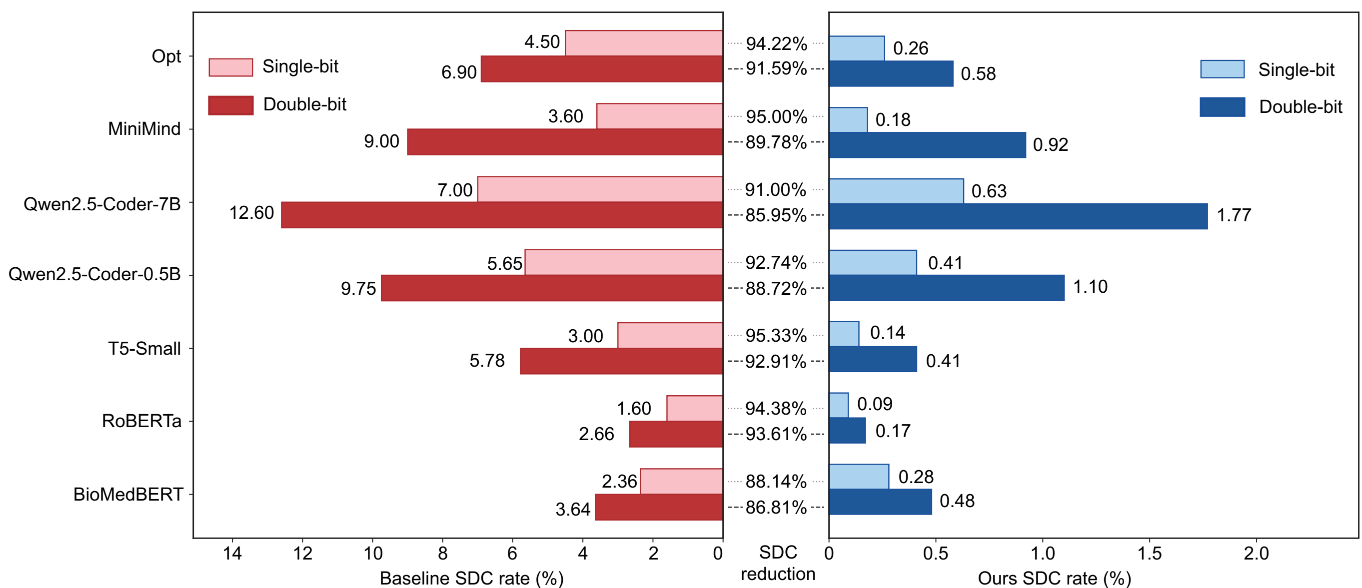

Fig. 4 Comparison of SDC rates and SDC reduction percentages between the baseline and our proposed method. The left panel shows baseline performance, where light red and dark red bars represent SBU and DBU faults, respectively. The right panel shows the performance of our method, using light blue and dark blue bars for SBU and DBU faults. The values in the center represent the SDC reduction, calculated as the percentage decrease in SDC rate relative to the baseline

key metric recall for class 1 (retry=1).

LightGBM demonstrates clear superiority, attaining the highest average recall (0.9745) and average F1-score (0.9626) among all candidates. Given that recall for the positive class is critical for minimizing missed SDCs, this consistently high score underscores LightGBM's effectiveness. XGBoost achieves a strong average recall of 0.9728 and performs competitively on several models, but is slightly outperformed by LightGBM for most LLMs. Logistic regression lags noticeably with an average recall of 0.9182, confirming that the decision boundary is highly nonlinear and better captured by tree-based ensemble methods.

Based on these results, LightGBM is preferred as the final meta-model for RetryTrigger, leveraging its superior detection coverage and balanced precision–recall performance.

5. Reliability improvements under different retry policies

We evaluated two distinct policies for triggering a retry. One is a per-token policy (online detection), which stops current inference generation and initiates a retry as soon as any single token is classified as anomalous. The other is a post-hoc policy, which waits for the entire sequence to be generated and then makes a single decision based on the aggregated (averaged) features of all generated tokens.

Notably, our experiments reveal that both policies are highly effective at achieving SDC reduction rates in Fig. 4. This is primarily because a significant majority of faults manifest as catastrophic numerical corruptions, resulting in NaN or Inf values within the feature space (e.g., entropy, logits_std, skewness, and kurtosis). Crucially, the preprocessing applied to these values—encompassing both capping and filling—is not intended to eliminate the fault signals; instead, it standardizes them into a high-magnitude representation (e.g., $\pm 10^{10}$) that remains a distinct outlier. Such standardized extreme values are readily identified by the LightGBM classifier, whether on an individual token basis (in the per-token policy) or after aggregation (as the mean of a set containing NaN is also NaN).

To ensure a rigorous evaluation, all 10 distinct input samples selected for each LLM (as reported in Tables 4 and 5) remained strictly independent of the training set. These inputs represent unseen data for the LightGBM meta-model, thereby ensuring that our SDC reduction results accurately reflect RetryTrigger's generalization capability in authentic, real-world scenarios.

While both policies demonstrate comparable detection effectiveness, they introduce different performance characteristics, specifically in terms of latency. A detailed quantitative comparison of their respective extra latency is therefore presented in Section 4.5.

In Tables 4 and 5, the bolded values represent cosine similarity achieved through RetryTrigger, highlighting its performance improvement over the unprotected baseline. These bold results are closer to 1.0 than the baseline outcomes. Through a second inference pass, the model restores corrupted outputs to a state that is nearly identical to the fault-free "golden" reference in both semantic and lexical terms. Thus, these bold values demonstrate that RetryTrigger effectively mitigates the severity of bias caused by SDC, even in severe fault scenarios like DBU, while maintaining exceptionally high output fidelity.

4.4 Fault detector analysis

This subsection validates the core engine of RetryTrigger. We analyzed the trained detector on a held-out set. This analysis is crucial. It confirms that our detection mechanism is the fundamental driver behind the system's effectiveness.

1. Overall performance across diverse models. Table 6 presents the classification performance of RetryTrigger when trained individually for each of the seven LLMs. The

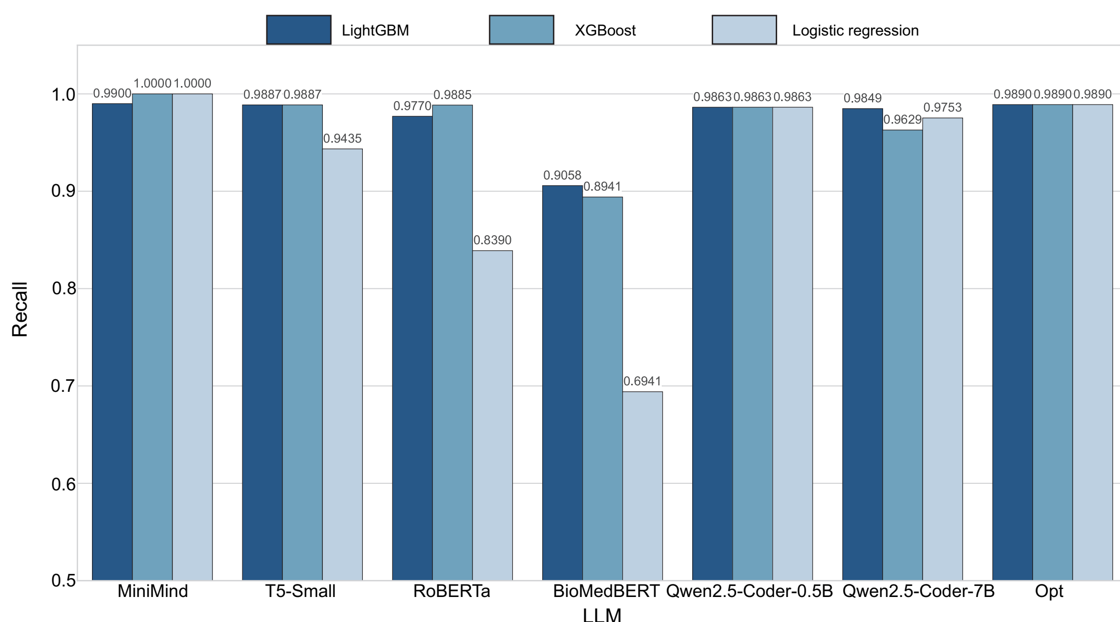

Fig. 5 Comparison of meta-classifier recall for the positive class (retry=1) across seven LLMs

Table 4 Comparison of cosine similarity across different LLMs under SBU fault injection. Each model is tested with 10 distinct input samples. The table reports the cosine similarity before and after applying our proposed RetryTrigger method

Input	Method	Cosine similarity						
		BioMedBERT	RoBERTa	T5-Small	Qwen2.5-Coder-0.5B	Qwen2.5-Coder-7B	MiniMind	Opt
Input 1	Base	0.9892 ± 0.0078	0.9669 ± 0.0137	0.9907 ± 0.0114	0.9430 ± 0.0075	0.9035 ± 0.0025	0.9885 ± 0.0155	0.9998 ± 0.0010
	Ours	0.9992 ± 0.0002	0.9985 ± 0.0013	0.9992 ± 0.0008	0.9850 ± 0.0025	0.9993 ± 0.0006	0.9990 ± 0.0008	0.9999 ± 0.0009
Input 2	Base	0.9840 ± 0.0092	0.9801 ± 0.0121	0.9973 ± 0.0037	0.9220 ± 0.0150	0.9231 ± 0.0037	0.9898 ± 0.0130	0.9999 ± 0.0006
	Ours	0.9984 ± 0.0011	0.9998 ± 0.0001	0.9999 ± 0.0003	0.9800 ± 0.0030	0.9998 ± 0.0002	0.9995 ± 0.0005	0.9999 ± 0.0005
Input 3	Base	0.9589 ± 0.0101	0.9771 ± 0.0102	0.9913 ± 0.0101	0.9210 ± 0.0105	0.8927 ± 0.0036	0.9861 ± 0.0180	0.9992 ± 0.0067
	Ours	0.9965 ± 0.0005	0.9997 ± 0.0008	0.9993 ± 0.0007	0.9780 ± 0.0032	0.9920 ± 0.0004	0.9985 ± 0.0010	0.9998 ± 0.0007
Input 4	Base	0.9844 ± 0.0112	0.9677 ± 0.0098	0.9873 ± 0.0130	0.9760 ± 0.0110	0.9728 ± 0.0045	0.9910 ± 0.0120	0.9980 ± 0.0166
	Ours	0.9973 ± 0.0002	0.9978 ± 0.0011	0.9988 ± 0.0009	0.9900 ± 0.0020	0.9921 ± 0.0002	0.9998 ± 0.0003	0.9995 ± 0.0012
Input 5	Base	0.9899 ± 0.0105	0.9745 ± 0.0098	0.9958 ± 0.0026	0.9350 ± 0.0130	0.9621 ± 0.0056	0.9873 ± 0.0197	0.9980 ± 0.0169
	Ours	0.9996 ± 0.0001	0.9996 ± 0.0009	0.9991 ± 0.0009	0.9820 ± 0.0028	0.9998 ± 0.0002	0.9988 ± 0.0009	0.9996 ± 0.0008
Input 6	Base	0.9882 ± 0.0088	0.9711 ± 0.0085	0.9942 ± 0.0032	0.9745 ± 0.0080	0.9811 ± 0.0049	0.9875 ± 0.0160	0.9980 ± 0.0125
	Ours	0.9982 ± 0.0014	0.9995 ± 0.0010	0.9990 ± 0.0008	0.9880 ± 0.0022	0.9998 ± 0.0002	0.9986 ± 0.0011	0.9994 ± 0.0010
Input 7	Base	0.9779 ± 0.0109	0.9743 ± 0.0109	0.9866 ± 0.0123	0.9580 ± 0.0140	0.9378 ± 0.0064	0.9859 ± 0.0175	0.9953 ± 0.0216
	Ours	0.9970 ± 0.0024	0.9995 ± 0.0005	0.9985 ± 0.0011	0.9860 ± 0.0024	0.9988 ± 0.0007	0.9983 ± 0.0012	0.9992 ± 0.0013
Input 8	Base	0.9712 ± 0.0135	0.9775 ± 0.0078	0.9718 ± 0.0114	0.9650 ± 0.0125	0.9110 ± 0.0024	0.9901 ± 0.0140	0.9965 ± 0.0336
	Ours	0.9969 ± 0.0016	0.9996 ± 0.0010	0.9978 ± 0.0013	0.9870 ± 0.0023	0.9983 ± 0.0006	0.9996 ± 0.0004	0.9999 ± 0.0006
Input 9	Base	0.9841 ± 0.0098	0.9691 ± 0.0134	0.9752 ± 0.0108	0.9885 ± 0.0095	0.9284 ± 0.0012	0.9870 ± 0.0185	0.9995 ± 0.0020
	Ours	0.9980 ± 0.0007	0.9982 ± 0.0016	0.9980 ± 0.0012	0.9920 ± 0.0018	0.9996 ± 0.0004	0.9987 ± 0.0009	0.9999 ± 0.0014
Input 10	Base	0.9635 ± 0.0158	0.9824 ± 0.0074	0.9727 ± 0.0109	0.9335 ± 0.0090	0.8997 ± 0.0086	0.9848 ± 0.0210	0.9984 ± 0.0139
	Ours	0.9966 ± 0.0018	0.9999 ± 0.0004	0.9975 ± 0.0015	0.9830 ± 0.0029	0.9995 ± 0.0005	0.9980 ± 0.0013	0.9997 ± 0.0011
Average	Base	0.9791 ± 0.0108	0.9741 ± 0.0104	0.9863 ± 0.0089	0.9517 ± 0.0110	0.9312 ± 0.0044	0.9878 ± 0.0165	0.9983 ± 0.0125
	Ours	$\mathbf{0.9978 \pm 0.0010}$	$\mathbf{0.9992 \pm 0.0009}$	$\mathbf{0.9987 \pm 0.0010}$	$\mathbf{0.9851 \pm 0.0025}$	$\mathbf{0.9979 \pm 0.0004}$	$\mathbf{0.9989 \pm 0.0008}$	$\mathbf{0.9997 \pm 0.0009}$

Table 5 Comparison of cosine similarities across different LLMs under DBU fault injection. Each model is tested with 10 distinct input samples. The table reports the cosine similarity before and after applying our proposed RetryTrigger method

Input	Method	Cosine similarity						
		BioMedBERT	RoBERTa	T5-Small	Qwen2.5-Coder-0.5B	Qwen2.5-Coder-7B	MiniMind	Opt
Input 1	Base	0.9763 ± 0.0115	0.9529 ± 0.0143	0.9850 ± 0.0074	0.8950 ± 0.0145	0.8635 ± 0.0067	0.9780 ± 0.0090	0.9989 ± 0.0078
	Ours	0.9975 ± 0.0012	0.9972 ± 0.0012	0.9985 ± 0.0012	0.9720 ± 0.0035	0.9991 ± 0.0007	0.9985 ± 0.0010	0.9995 ± 0.0010
Input 2	Base	0.9641 ± 0.0157	0.9468 ± 0.0150	0.9831 ± 0.0093	0.8520 ± 0.0155	0.8992 ± 0.0045	0.9850 ± 0.0105	0.9996 ± 0.0021
	Ours	0.9945 ± 0.0018	0.9965 ± 0.0015	0.9990 ± 0.0009	0.9550 ± 0.0050	0.9997 ± 0.0003	0.9990 ± 0.0008	0.9998 ± 0.0006
Input 3	Base	0.9739 ± 0.0130	0.9741 ± 0.0125	0.9908 ± 0.0109	0.9015 ± 0.0090	0.8895 ± 0.0021	0.9845 ± 0.0120	0.9979 ± 0.0181
	Ours	0.9970 ± 0.0010	0.9995 ± 0.0006	0.9992 ± 0.0008	0.9750 ± 0.0032	0.9923 ± 0.0005	0.9992 ± 0.0007	0.9999 ± 0.0005
Input 4	Base	0.9788 ± 0.0105	0.9720 ± 0.0110	0.9848 ± 0.0073	0.8680 ± 0.0170	0.8437 ± 0.0125	0.9580 ± 0.0150	0.9991 ± 0.0040
	Ours	0.9982 ± 0.0008	0.9992 ± 0.0007	0.9978 ± 0.0015	0.9600 ± 0.0045	0.9919 ± 0.0019	0.9970 ± 0.0015	0.9994 ± 0.0012
Input 5	Base	0.9665 ± 0.0165	0.9731 ± 0.0120	0.9828 ± 0.0075	0.8815 ± 0.0160	0.8645 ± 0.0062	0.9775 ± 0.0095	0.9993 ± 0.0060
	Ours	0.9950 ± 0.0015	0.9993 ± 0.0008	0.9975 ± 0.0016	0.9650 ± 0.0040	0.9996 ± 0.0002	0.9986 ± 0.0011	0.9997 ± 0.0007
Input 6	Base	0.9714 ± 0.0140	0.9678 ± 0.0035	0.9763 ± 0.0099	0.9000 ± 0.0120	0.8811 ± 0.0109	0.9650 ± 0.0130	0.9982 ± 0.0119
	Ours	0.9972 ± 0.0010	0.9980 ± 0.0010	0.9968 ± 0.0018	0.9730 ± 0.0034	0.9997 ± 0.0002	0.9975 ± 0.0013	0.9998 ± 0.0008
Input 7	Base	0.9746 ± 0.0109	0.9699 ± 0.0080	0.9856 ± 0.0069	0.9150 ± 0.0110	0.8322 ± 0.0029	0.9510 ± 0.0160	0.9938 ± 0.0231
	Ours	0.9978 ± 0.0002	0.9985 ± 0.0009	0.9988 ± 0.0010	0.9800 ± 0.0028	0.9911 ± 0.0008	0.9965 ± 0.0017	0.9991 ± 0.0013
Input 8	Base	0.9629 ± 0.0170	0.9728 ± 0.0115	0.9599 ± 0.0132	0.9080 ± 0.0135	0.8543 ± 0.0038	0.9640 ± 0.0140	0.9974 ± 0.0121
	Ours	0.9940 ± 0.0020	0.9994 ± 0.0007	0.9965 ± 0.0017	0.9780 ± 0.0030	0.9916 ± 0.0006	0.9973 ± 0.0014	0.9994 ± 0.0011
Input 9	Base	0.9756 ± 0.0126	0.9566 ± 0.0130	0.9650 ± 0.0120	0.8750 ± 0.0085	0.8326 ± 0.0056	0.9890 ± 0.0110	0.9990 ± 0.0029
	Ours	0.9973 ± 0.0012	0.9975 ± 0.0011	0.9970 ± 0.0016	0.9620 ± 0.0042	0.9994 ± 0.0004	0.9995 ± 0.0006	0.9996 ± 0.0009
Input 10	Base	0.9678 ± 0.0150	0.9710 ± 0.0095	0.9577 ± 0.0133	0.8870 ± 0.0100	0.8935 ± 0.0083	0.9570 ± 0.0145	0.9965 ± 0.0193
	Ours	0.9968 ± 0.0008	0.9988 ± 0.0008	0.9962 ± 0.0020	0.9680 ± 0.0038	0.9995 ± 0.0005	0.9968 ± 0.0016	0.9993 ± 0.0012
Average	Base	0.9712 ± 0.0137	0.9657 ± 0.0110	0.9771 ± 0.0098	0.8883 ± 0.0127	0.8654 ± 0.0064	0.9709 ± 0.0125	0.9980 ± 0.0107
	Ours	$\mathbf{0.9965 \pm 0.0012}$	$\mathbf{0.9984 \pm 0.0009}$	$\mathbf{0.9977 \pm 0.0014}$	$\mathbf{0.9688 \pm 0.0037}$	$\mathbf{0.9964 \pm 0.0006}$	$\mathbf{0.9980 \pm 0.0012}$	$\mathbf{0.9996 \pm 0.0009}$

Table 6 Detailed comparison of meta-classifier performance on the validation set for detecting SDCs. For each LLM, we reported precision, recall, and F1-score for both class 0 (no-retry) and class 1 (retry)

LLM	Classifier	Class 0 (no-retry)			Class 1 (retry)		
		Precision	Recall	F1-score	Precision	Recall	F1-score
MiniMind	LightGBM	0.9998	1.0000	0.9999	1.0000	0.9900	0.9898
	XGBoost	1.0000	0.9998	0.9999	0.9803	1.0000	0.9900
	Logistic regression	1.0000	0.9970	0.9985	0.6578	1.0000	0.7936
T5-Small	LightGBM	0.9990	0.9995	0.9993	0.9943	0.9887	0.9915
	XGBoost	0.9990	0.9995	0.9993	0.9943	0.9887	0.9915
	Logistic regression	0.9950	0.9347	0.9639	0.5439	0.9435	0.6900
RoBERTa	LightGBM	0.9989	0.9864	0.9926	0.7657	0.9770	0.8585
	XGBoost	0.9994	0.9837	0.9915	0.7350	0.9885	0.8431
	Logistic regression	0.9921	0.9210	0.9552	0.3258	0.8390	0.4694
BioMedBERT	LightGBM	0.9958	0.9973	0.9966	0.9390	0.9058	0.9221
	XGBoost	0.9953	0.9958	0.9955	0.9047	0.8941	0.8994
	Logistic regression	0.9850	0.8971	0.9390	0.2304	0.6941	0.3460
Qwen2.5-Coder-0.5B	LightGBM	0.9997	1.0000	0.9998	1.0000	0.9863	0.9931
	XGBoost	0.9997	0.9997	0.9997	0.9863	0.9863	0.9863
	Logistic regression	0.9997	0.9949	0.9973	0.7826	0.9863	0.8727
Qwen2.5-Coder-7B	LightGBM	0.9996	0.9998	0.9997	0.9924	0.9849	0.9886
	XGBoost	0.9990	0.9990	0.9990	0.9629	0.9629	0.9629
	Logistic regression	0.9995	0.9950	0.9972	0.7824	0.9753	0.8682
Opt	LightGBM	0.9998	1.0000	0.9999	1.0000	0.9890	0.9944
	XGBoost	0.9998	1.0000	0.9999	1.0000	0.9890	0.9944
	Logistic regression	0.9998	0.9840	0.9918	0.5172	0.9890	0.6792
Average	LightGBM	**0.9989**	**0.9976**	**0.9983**	**0.9559**	**0.9745**	**0.9626**
	XGBoost	**0.9989**	0.9968	0.9978	0.9376	0.9728	0.9525
	Logistic regression	0.9959	0.9605	0.9776	0.5486	0.9182	0.6742

The best result for each metric is highlighted in bold

results demonstrate the strong generalization capability of our feature-based approach across diverse architectures and tasks. For the negative class (retry = 0), the detector achieves near-perfect precision and recall across all models, indicating an extremely low false positive (FP) rate and thus minimal overhead from unnecessary retries.

2. High sensitivity to faults. The primary goal of RetryTrigger is to reliably identify faulty outputs. This is measured by the recall of the positive class (retry=1). As shown in Table 6, our detector achieves excellent recall for this critical class across all models, ranging from 0.9058 for BioMedBERT to 0.9900 for MiniMind. This high sensitivity ensures that the vast majority of SDCs are successfully caught before reaching the user. For instance, with the Opt model, the detector successfully identifies over 0.9890 of the faulty outputs.

3. Precision-recall trade-off. While prioritizing high recall, we also observed the classic precision-recall trade-off. For RoBERTa, achieving a 0.9770 recall on SDCs comes at the cost of a lower precision (0.7657), meaning that some correct outputs are flagged for retry. However, this strategy is highly favorable. Since a retry has a near-certain probability of producing a correct output, a slight increase in redundancy is a small price to pay. It guarantees significantly higher overall system reliability.

4. Why does SDC rate remain non-zero? Although our detector achieves near-perfect detection performance, this does not imply that the final SDC rate of the RetryTrigger mechanism converges to zero. The observed residual SDC rate is the result of multiple interacting factors beyond detection accuracy alone. There are three cases: (1) retry failure is a dominant contributor. Even when a transient fault is correctly detected (true positive (TP)) and a retry is triggered, the re-inference is executed on the same faulty hardware platform. As a result, the retry itself is still subject to transient faults and may again produce an erroneous output. (2) Detection leakage (false negatives) directly contributes to SDCs. Faulty outputs that escape detection are accepted by the system without retry, leading to SDC. (3) False-positive-induced retries can also introduce new error opportunities. When a correct output is mistakenly flagged as faulty, an unnecessary retry is triggered, exposing an otherwise correct execution to additional transient fault risks that would not have occurred without intervention.

4.5 Overhead analysis

1. Relative overhead (retry rate)

To evaluate the practicality and efficiency of our proposed method, we analyzed its computational overhead. We defined overhead as the "retry rate", the percentage of initial runs that trigger a retry action. A retry is initiated in two cases: (1) a TP, where a genuine error is correctly detected, or (2) an FP, where a correct execution is incorrectly flagged as faulty. Therefore, the total overhead is defined as the sum of the TP-induced overhead and the FP-induced overhead. The TP-induced overhead is the probability of correctly detecting a real error, and the FP-induced overhead is the probability of a false alarm on a correct execution.

Tables 7 and 8 present a detailed breakdown of this overhead analysis for each model. The baseline SDC rate

Table 7 Relative overhead analysis of the RetryTrigger method under SBU fault injection. The total overhead is the percentage of runs that trigger a retry, composed of TP-induced and FP-induced retries

Model	Baseline SDC (P_{err})	Recall_1 (TPR)	$1-\text{Recall}_0$ (FPR)	TP-induced overhead (%)	FP-induced overhead (%)	Total overhead (retry rate (%))
BioMedBERT	0.0236	0.9058	0.0027	2.1376	0.2636	**2.4012**
RoBERTa	0.0160	0.9770	0.0136	1.5632	1.3382	**2.9014**
T5-Small	0.0300	0.9887	0.0005	2.9661	0.0485	**3.0146**
Qwen2.5-Coder-0.5B	0.0565	0.9863	0.0000	5.5725	0.0000	**5.5725**
Qwen2.5-Coder-7B	0.0700	0.9849	0.0002	6.8943	0.0186	**6.9129**
MiniMind	0.0360	0.9900	0.0000	3.5640	0.0000	**3.5640**
Opt	0.0450	0.9890	0.0000	4.4505	0.0000	**4.4505**
Average	0.0396	0.9745	0.0024	3.8783	0.2384	**4.1167**

Table 8 Relative overhead analysis of the RetryTrigger method under DBU fault injection

Model	Baseline SDC (P_{err})	Recall_1 (TPR)	$1-\text{Recall}_0$ (FPR)	TP-induced overhead (%)	FP-induced overhead (%)	Total overhead (retry rate (%))
BioMedBERT	0.0364	0.9058	0.0027	3.2971	0.2601	**3.5572**
RoBERTa	0.0266	0.9770	0.0136	2.5988	1.3238	**3.9226**
T5-Small	0.0578	0.9887	0.0005	5.7146	0.0471	**5.7617**
Qwen2.5-Coder-0.5B	0.0975	0.9863	0.0000	9.6164	0.0000	**9.6164**
Qwen2.5-Coder-7B	0.1260	0.9849	0.0002	12.4097	0.0174	**12.4271**
MiniMind	0.0900	0.9900	0.0000	8.9100	0.0000	**8.9100**
Opt	0.0690	0.9890	0.0000	6.8241	0.0000	**6.8241**
Average	0.0719	0.9745	0.0024	7.0530	0.2355	**7.2884**

(P_{err}) is taken from the average of 10 runs in Fig. 4. The true positive rate (TPR) (Recall_1) and false positive rate (FPR) (calculated as $1-\text{Recall}_0$) are derived from our meta-model's performance in Table 6.

In both tables, the bolded values highlight the final total overhead (retry rate) for each LLM and their overall averages. These values are emphasized to demonstrate the minor efficiency penalty introduced by the RetryTrigger framework. Highlighting these metrics underscores the framework's practical engineering value: it achieves substantial reliability improvements without imposing prohibitive computational costs, thereby ensuring its viability for large-scale and latency-sensitive LLM deployments.

The results confirm high efficiency. Under SBU injection, the average total overhead is merely 4.1167%. More importantly, the decomposition of this overhead reveals that the vast majority (3.8783%) is TP-induced, meaning that these retries are essential for preventing errors. In contrast, the FP-induced overhead, which represents the true "efficiency penalty", is negligible, averaging only 0.2384%. Notably, three out of the seven models (Qwen2.5-Coder-0.5B, MiniMind, and Opt) exhibit zero FPs. This indicates an ideal operational scenario where the detector achieves perfect precision (zero FPs), ensuring that absolutely no computational resources are wasted on re-executing correct queries. Even for the RoBERTa, which has the highest FP-induced overhead (1.3382%), the trade-off remains highly favorable given its 0.9770 fault recall.

Under the more severe DBU injection, the average total overhead rises to 7.2884%. However, this increase is primarily driven by the higher prevalence of faults (higher baseline SDC), not by a degradation in detector efficiency. For instance, Qwen2.5-Coder-7B shows the highest overhead of 12.4271%, but this is mathematically inevitable due to its high baseline SDC rate of 0.1260. However, its FP-induced overhead remains minimal (0.0174%). This shows that RetryTrigger scales gracefully. The cost of protection increases linearly with the fault rate, while the cost of false alarms remains constantly low.

In summary, relative overhead analysis confirms a decisive advantage. RetryTrigger secures a substantial reduction in the SDC rate (>0.90 on average). Yet, it demands only a modest computational cost (4.1167% retry rate). This highly favorable trade-off validates its practicality as a reliability enhancement technique.

2. Absolute overhead (extra latency)

Beyond the retry rate, we analyzed the extra latency introduced by RetryTrigger. We defined this as the increase in end-to-end wall-clock time required to complete an inference task when the mechanism is active. This overhead arises from two sources: (1) the intrinsic cost of executing the detection logic within each run ($T_{\text{faulty_avg}}-T_{\text{baseline}}$) and (2) the amortized time cost of potential retry operations ($T_{\text{baseline}}\times P_{\text{retry}}$). We compared this latency overhead for seven models under both the per-token and post-hoc detection policies, using greedy decoding across all experiments for fair comparison.

Table 9 presents the measured average latencies (T_{baseline} for fault-free runs and $T_{\text{faulty_avg}}$ for runs with detection enabled), the observed retry rate (P_{retry}), the calculated final average latency ($T_{\text{avg_total}}$), and the resulting absolute latency overhead (in seconds) for each configuration. The results presented in Table 9 clearly demonstrate that the optimal retry policy, in terms of absolute extra latency in seconds, is highly dependent on the specific LLM characteristics.

Fig. 6 illustrates the relationship between the relative overhead (retry rate) and absolute overhead (extra latency). The curves reveal a positive yet nonlinear correlation. While higher retry rates generally increase latency, the gradient of this increase is strictly governed by the architecture, scale, and detection policy. For models with large vocabularies, such

Table 9 Extra latency comparison for encoder–decoder and decoder-only models

Model	Policy	T_{baseline} (s)	$T_{\text{faulty_avg}}$ (s)	P_{retry} (%)	$T_{\text{avg_total}}$ (s)	Absolute overhead (s)
BioMedBERT	Per-token					
	Post-hoc	0.0067	0.0171	2.4012	0.0172	0.0105
RoBERTa	Per-token					
	Post-hoc	0.0068	0.0157	2.9014	0.0158	0.0090
Qwen2.5-Coder-0.5B	Per-token	0.5451	0.7594	5.5725	0.7897	0.2446
	Post-hoc	0.6172	1.5662	5.5725	1.6005	0.9833
Qwen2.5-Coder-7B	Per-token	0.7814	1.0228	6.9129	1.0768	0.2954
	Post-hoc	0.6506	1.6527	6.9129	1.6976	1.0470
Opt	Per-token	0.1678	0.4624	4.4505	0.4698	0.3020
	Post-hoc	0.1560	0.8637	4.4505	0.8706	0.7146
T5-Small	Per-token	0.0482	0.0966	3.0146	0.0980	0.0498
	Post-hoc	0.0519	0.0706	3.0146	0.0721	0.0202
MiniMind	Per-token	0.7408	0.8043	3.5640	0.8307	0.0899
	Post-hoc	0.6990	0.8001	3.5640	0.8250	0.1260

T_{baseline} and $T_{\text{faulty_avg}}$ are the measured average time in second; P_{retry} is the total overhead (retry rate) for each model, as reported in Table 7; $T_{\text{avg_total}}$ is the calculated final average latency considering retries; absolute overhead is calculated as $T_{\text{avg_total}} - T_{\text{baseline}}$

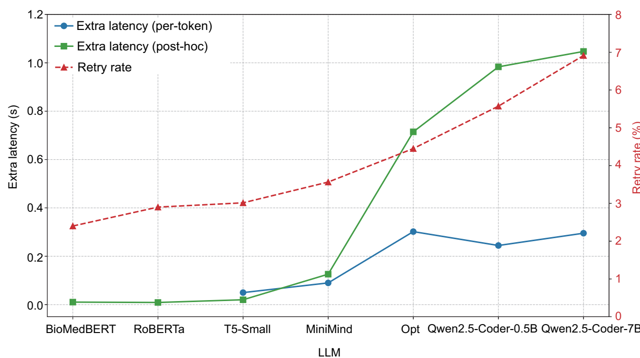

Fig. 6 Correlation analysis between retry rate and extra latency across seven LLMs. Models on the x axis are sorted by their retry rate (red dashed line, right axis). The solid lines (left axis) represent the extra latency overhead for per-token and post-hoc policies

as Qwen2.5-Coder (1.52×10^5 tokens) and Opt, the post-hoc policy (green line) exhibits a steep latency spike. This is attributed to the computational bottleneck of processing logits for the entire sequence in a single batch, where the vocabulary size significantly amplifies the feature extraction cost. In contrast, the per-token policy (blue line) maintains a significantly flatter profile, capitalizing on the "fail-fast" mechanism to terminate faulty runs early. However, this trend reverses for extremely lightweight models like T5-Small. Due to its rapid baseline inference speed, the cumulative cost of executing detection logic at every step in the per-token policy becomes disproportionately expensive compared to a single post-hoc analysis. This divergence confirms that latency impact is a multi-dimensional trade-off involving inference speed, vocabulary size, and policy mechanics.

3. Absolute overhead comparison between two retry polices

We compared the absolute overhead in Table 9 to determine the optimal configuration for each LLM scenario. For the Qwen2.5-Coder series and Opt, the per-token policy proves substantially more efficient, reducing the absolute overhead by approximately 3× to 4× compared to the post-hoc policy (e.g., 0.2446 s vs. 0.9833 s for Qwen2.5-Coder-0.5B). This confirms that avoiding the bulk processing bottleneck is crucial for standard decoder-only architectures with large vocabularies. Conversely, for T5-Small, the post-hoc policy emerges as the clear winner, introducing a negligible 0.0202 s overhead compared to 0.0498 s for the per-token policy, as the one-time analysis is more economical relative to the fast baseline. For MiniMind, the per-token policy holds a moderate advantage (0.0899 s vs. 0.1260 s) as the slower baseline masks the step-wise detection cost.

These results underscore that optimal latency requires architectural adaptation. Consequently, RetryTrigger adopts an architecture-aware deployment strategy: prioritizing the per-token policy for large-vocabulary models to exploit early termination, while reserving the post-hoc policy for lightweight models to amortize detection costs. This flexibility ensures robust reliability with minimal end-to-end latency penalty.

4.6 Discussion of limitations

While RetryTrigger achieves consistent SDC reductions across diverse LLMs, two key limitations should be noted:

1. Dependence on the fault model. Our evaluation leverages a software-level fault injection framework built on PyTorch's forward hooks to inject statistically-controlled transient hardware faults. Real-world faults may exhibit different spatial-temporal correlations, microarchitectural signatures, or vendor-specific behaviors not fully captured by such a fault model. Thus, SDC reductions should be interpreted relative to this fault model, and validation with hardware fault traces or field-programmable gate array (FPGA)-based fault emulation would improve the physical fidelity of the assessment.

2. Dependence on runtime internal observability. The meta-model relies on access to logits, per-step probabilities, timing, and other intermediate signals. In proprietary black-box application programming interfaces (APIs) or constrained inference environments where such data are hidden or costly to obtain, applicability may be limited. Moreover, aggressive kernel fusion or extreme quantization can alter distributional

patterns, potentially reducing detection accuracy.

These limitations highlight opportunities for future work, such as integrating hardware performance counters for broader fault coverage or developing black-box variants suitable for environments with restricted observability.

5 Related works

1. Hardware protection mechanisms. Hardware-level protection serves as the foundational defense against transient faults. Extensive research has been dedicated to characterizing the vulnerability of modern accelerators. For instance, recent studies have evaluated the instruction-level error characteristics of graphics processing units (GPUs) under low supply voltages (Tan et al., 2025a) and developed frameworks for fast, precise error resilience assessment of GPU microarchitectures (Tan et al., 2025b). Beyond general-purpose GPUs, specialized neural network accelerators, particularly those based on systolic arrays, have also been scrutinized through microarchitecture-level fault injection frameworks (Tan et al., 2023b) and quantitative analysis of architectural vulnerability factors (Tan et al., 2023a). Building on these reliability insights, error-correcting codes (ECCs), such as the single-error correction, double-error detection (SEC–DED) scheme proposed by Hamming (1950), have been widely deployed in modern GPUs and memory systems to mitigate soft errors. However, ECCs are primarily designed for storage elements and provide negligible protection against combinational logic faults occurring within arithmetic units or accumulators (Baumann, 2005). To further enhance resilience, redundancy-based techniques, such as dual or triple modular redundancy, have been explored by Venkatesha and Parthasarathi (2024), but these approaches incur high energy and latency overhead, making them impractical for LLM inference. Consequently, recent efforts have shifted toward hybrid redundancy and lightweight algorithm-based checks (Liang et al., 2025; Liu et al., 2025; Titopoulos et al., 2025), attempting to extend protection into computation layers, particularly within attention and matrix multiplication units.

2. Software-based fault tolerance. At the software and algorithmic level, resilience strategies generally fall into static hardening or dynamic monitoring. Early works such as FT-ClipAct (Hoang et al., 2019) and Ranger (Chen ZT et al., 2021) improved DNN resilience through activation clipping and fine-grained post-training adjustments. Subsequent designs, including ProAct (Mousavi et al., 2024) and FitAct (Ghavami et al., 2022), further reduce error propagation by dynamically restricting activation ranges during inference. For Transformer-based models, Roquet et al. (2024) and Sha et al. (2024) investigated layer-wise range restriction and clipping to improve reliability in vision and language Transformers. While these methods offer low overhead, most require offline profiling or fine-tuning, which limits their applicability to real-time LLM inference.

Significantly, recent studies have proposed online detection and correction mechanisms for LLMs. FT2 (Sun et al., 2025) introduces a first-token-inspired bound-setting strategy that eliminates the need for offline calibration while maintaining high detection accuracy, achieving over 90% SDC reduction. Similarly, ALBERTA (Liu et al., 2025) incorporates checksum-based error resilience directly into Transformer kernels, achieving near-complete coverage with minimal runtime overhead. These software-based solutions improve model resilience without hardware modification, although they still require kernel instrumentation or retraining.

3. Hardware–software co-design approaches. To balance protection strength and deployment cost, cross-layer strategies have gained traction. ReaLM (Xie et al., 2025) employs statistical ABFT with hardware support to detect and mitigate transient errors during LLM inference. Similarly, Dai et al. (2025) propose FT-Transformer, an end-to-end fault-tolerant Transformer architecture that combines ABFT with hardware-assisted checksum validation. Although these hybrid schemes achieve superior efficiency, their reliance on specialized hardware modifications or non-standard accelerator features creates a significant barrier to adoption on commodity GPUs.

In contrast, our proposed RetryTrigger operates purely in the inference phase and does not modify model weights or inference kernels. It leverages runtime output statistics and a lightweight LightGBM-based classifier (Ke et al., 2017) to adaptively trigger re-execution only for suspicious outputs. This design provides a practical, low-overhead, and framework-agnostic software solution that complements existing hardware and co-design resilience methods.

6 Conclusions

RetryTrigger is proposed for enhancing LLM resilience via dynamically collecting runtime features and leveraging a LightGBM meta-model to accurately predict whether inference outputs should be retried. Extensive evaluations on seven representative LLMs demonstrated that RetryTrigger achieves up to 95.33% and an average of 92.97% SDC reduction, with minimal performance overhead averaging only 4.1167%. This good balance between robustness and efficiency makes RetryTrigger a practical solution for deploying robust LLMs in resource-constrained or real-time environments.

Future work will focus on enhancing compatibility with black-box APIs via logit-agnostic variants, and validating with real hardware fault traces to further strengthen ecological validity.

Acknowledgments

This work was supported by the Shanghai Oriental Talent Program.

Author contributions

Jiajia JIAO designed the research. Yixu YU processed the data. Yixu YU drafted the paper. Jiajia JIAO helped organize the paper. Jiajia JIAO and Yixu YU revised and finalized the paper.

Conflict of interest

Both authors declare that they have no conflict of interest.

Data availability

The data and code are publicly available on GitHub at https://github.com/ltbz/RetryTrigger.

Declaration on the use of generative AI tools

The authors used ChatGPT to improve the language of the manuscript, reviewed and edited the content as needed, and take full responsibility for the content of the published article.

References

Battaglini-Fischer S, Srinivasan N, Szarvas BL, et al., 2025. FAILS: a framework for automated collection and analysis of LLM service incidents. 16th ACM/SPEC Int Conf on Performance Engineering, p.187-194. https://doi.org/10.1145/3680256.3721320

Baumann RC, 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans Dev Mater Reliab*, 5(3):305-316. https://doi.org/10.1109/TDMR.2005.853449

Cavagnero N, Dos Santos F, Ciccone M, et al., 2022. Transient-fault-aware design and training to enhance DNNs reliability with zero-overhead. IEEE 28th Int Symp on On-Line Testing and Robust System Design, p.1-7. https://doi.org/10.1109/IOLTS56730.2022.9897813

Chen TQ, Guestrin C, 2016. XGBoost: a scalable tree boosting system. Proc 22nd ACM SIGKDD Int Conf on Knowledge Discovery and Data Mining, p.785-794. https://doi.org/10.1145/2939672.2939785

Chen ZT, Li GP, Pattabiraman K, 2021. A low-cost fault corrector for deep neural networks through range restriction. 51st Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.1-13. https://doi.org/10.1109/DSN48987.2021.00018

Dai HL, Wu SX, Huang JJ, et al., 2025. FT-Transformer: resilient and reliable Transformer with end-to-end fault tolerant attention. https://doi.org/10.48550/arXiv.2504.02211

Ghavami B, Sadati M, Fang ZM, et al., 2022. FitAct: error resilient deep neural networks via fine-grained post-trainable activation functions. Design, Automation & Test in Europe Conf & Exhibition, p.1239-1244. https://doi.org/10.23919/date54114.2022.9774635

Hamming RW, 1950. Error detecting and error correcting codes. *Bell Syst Tech J*, 29(2):147-160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

Hoang LH, Hanif MA, Shafique M, 2019. FT-ClipAct: resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. https://doi.org/10.48550/arXiv.1912.00941

Hosmer DWJr, Lemeshow S, Sturdivant RX, 2013. Applied Logistic Regression. John Wiley & Sons, New York, USA.

Jiang JY, Wang F, Shen JS, et al., 2024. A survey on large language models for code generation. https://doi.org/10.48550/arXiv.2406.00515

Ke GL, Meng Q, Finley T, et al., 2017. LightGBM: a highly efficient gradient boosting decision tree. Proc 31st Int Conf on Neural Information Processing Systems, p.3149-3157.

Li Y, Yang SL, Liu CC, et al., 2025. Resilio: an elastic fault-tolerant training system for large language models. *J Comput Res Dev*, 62(6):1380-1395 (in Chinese). https://doi.org/10.7544/issn1000-1239.202550147

Liang YH, Li XY, Ren J, et al., 2025. ATTNChecker: highly-optimized fault tolerant attention for large language model training. Proc 30th ACM SIGPLAN Annual Symp on Principles and Practice of Parallel Programming, p.252-266. https://doi.org/10.1145/3710848.3710870

Liu HX, Singh V, Filipiuk M, et al., 2025. ALBERTA: algorithm-based error resilience in transformer architectures. *IEEE Open J Comput Soc*, 6:85-96. https://doi.org/10.1109/ojcs.2024.3400696

Mousavi S, Ahmadilivani MH, Raik J, et al., 2024. ProAct: progressive training for hybrid clipped activation function to enhance resilience of DNNs. https://doi.org/10.48550/arXiv.2406.06313

Radford A, Wu J, Child R, et al., 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9.

Roquet L, dos Santos FF, Rech P, et al., 2024. Cross-layer reliability evaluation and efficient hardening of large vision Transformers models. Proc 61st ACM/IEEE Design Automation Conf, Article 291. https://doi.org/10.1145/3649329.3655688

Sha QS, Paulitsch M, Pattabiraman K, et al., 2024. Global Clipper: enhancing safety and reliability of transformer-based object detection models. https://doi.org/10.48550/arXiv.2406.03229

Sun Y, Zhu Z, Mulpuru C, et al., 2025. FT2: first-token-inspired online fault tolerance on critical layers for generative large language models. Proc 34th Int Symp on High-Performance Parallel and Distributed Computing, Article 7. https://doi.org/10.1145/3731545.3731570

Tan JWJ, Ping LQ, Wang QX, et al., 2023a. Saca-AVF: a quantitative approach to analyze the architectural vulnerability factors of CNN accelerators. *IEEE Trans Comput*, 72(11):3042-3056. https://doi.org/10.1109/TC.2023.3283685

Tan JWJ, Wang QX, Yan KG, et al., 2023b. Saca-FI: a microarchitecture-level fault injection framework for reliability analysis of systolic array based CNN accelerator. *Future Gener Comput Syst*, 147:251-264. https://doi.org/10.1016/j.future.2023.05.009

Tan JWJ, Wang JS, Yan KG, et al., 2025a. Evaluating GPU's instruction-level error characteristics under low supply voltages. *IEEE Trans Comput*, 74(2):555-568. https://doi.org/10.1109/TC.2024.3500366

Tan JWJ, Li XR, Zhong A, et al., 2025b. GEREM: fast and precise error resilience assessment for GPU microarchitectures. *IEEE Trans Parall Distrib Syst*, 36(5):1011-1024. https://doi.org/10.1109/TPDS.2025.3552679

Titopoulos V, Alexandridis K, Dimitrakopoulos G, 2025. Custom algorithm-based fault tolerance for attention layers in transformers. https://doi.org/10.48550/arXiv.2507.16676

Venkatesha S, Parthasarathi R, 2024. Survey on redundancy based-fault tolerance methods for processors and hardware accelerators—trends in quantum computing, heterogeneous systems and reliability. *ACM Comput Surv*, 56(11):275. https://doi.org/10.1145/3663672

Wan BR, Han MJ, Sheng YY, et al., 2025. ByteCheckpoint: a unified checkpointing system for large foundation model development. 22nd USENIX Symp on Networked Systems Design and Implementation, p.559-578.

Xie T, Zhao JW, Wan ZS, et al., 2025. ReaLM: reliable and efficient large language model inference with statistical algorithm-based fault tolerance. https://doi.org/10.48550/arXiv.2503.24053

Xue XH, Liu C, Min F, et al., 2025. ApproxABFT: approximate algorithm-based fault tolerance for neural network processing. https://doi.org/10.48550/arXiv.2302.10469

Zhang WX, Deng Y, Liu B, et al., 2023. Sentiment analysis in the era of large language models: a reality check. https://doi.org/10.48550/arXiv.2305.15005

Zhou S, Xu ZD, Zhang M, et al., 2025. Large language models for disease diagnosis: a scoping review. https://arxiv.org/abs/2409.00097