Hierarchical Fault Detection and Diagnosis for Transformer Architectures

SIGMA JAHAN, Dalhousie University, Canada
SAURABHSINGH RAJPUT, Dalhousie University, Canada
TUSHAR SHARMA, Dalhousie University, Canada
MOHAMMAD MASUDUR RAHMAN, Dalhousie University, Canada

Transformers now underpin critical AI systems across industry and research. Yet their faults can silently alter model behavior without runtime errors, and existing techniques offer little support for tracing these failures to their component and root cause. Such faults evade detection because loss and numerical values stay normal, and the visible symptom rarely identifies the component responsible. We present *DEFault++*, a hierarchical learning-based technique that first detects a fault, then identifies the affected component, and finally the cause within it, helping developers effectively debug transformer models. DEFault++ organizes component-level runtime measurements with a *Fault Propagation Graph (FPG)*, a structural prior over the architecture's dependency paths, and reports the evidence behind each diagnosis. To train and evaluate it, we construct *DEFault-bench*, a benchmark of 5,556 labeled runs from mutation testing across seven models, nine tasks, and both encoder and decoder architectures. DEFault++ improves fault detection over four prior techniques, reaching an F1 of 0.826–0.909, and in a developer study with 21 participants, it raises repair accuracy from 57.1% to 83.3%. These results show that transformer fault diagnosis benefits from component-level measurements and architecture-aware reasoning rather than model-level behavior alone, and DEFault-bench provides a foundation for further research on transformer fault diagnosis.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: attention mechanism, fault diagnosis, fault propagation, mutation testing, transformer

1 Introduction

Transformer models now support a wide range of applications, from natural language processing [18, 62] and code generation [8] to broader software engineering tasks [67] and medical imaging [16]. Their effectiveness comes in large part from the attention mechanism [79], which lets each token draw on information from any other token when forming its representation. Attention combines several operations [79], such as projection, masking, and score computation, and these operations interact with other transformer components such as residual connections and layer normalization, as well as with runtime settings such as kernel selection and numerical precision [61]. Small mistakes in these operations or their interactions can change model behavior without raising explicit runtime errors [11, 34, 94]. The model can lose representation quality or task performance even when training completes without exceptions or invalid values.

A fault can affect a model's behavior without producing any of the usual error symptoms. More than a third of attention-specific faults produce *silent* or latent symptoms, about twice the rate reported for traditional deep neural

Authors' Contact Information: Sigma Jahan, Dalhousie University, Halifax, Canada, sigma.jahan@dal.ca; Saurabhsingh Rajput, Dalhousie University, Halifax, Canada, saurabh@dal.ca; Tushar Sharma, Dalhousie University, Halifax, Canada, tushar@dal.ca; Mohammad Masudur Rahman, Dalhousie University, Halifax, Canada, masud.rahman@dal.ca.

Manuscript submitted to ACM

network (DNN) faults [34]. These faults survive conventional tests, reach production, and waste training resources precisely because they raise no error. Even when a fault becomes visible, the symptom rarely indicates which component is responsible, which leaves a developer without a clear starting point for debugging or repair. Weak fine-tuning performance, for example, can come from a faulty attention mask, stale key-value cache entries, or incorrect positional information [34]. Diagnosing a transformer fault therefore means identifying both the faulty component and the cause that explains its behavior.

Existing work has proposed automated debugging techniques for DNN programs. Rule-based techniques, such as AutoTrainer [97], UMLAUT [74], and DeepDiagnosis [84], check training runs for predefined symptoms such as exploding gradients, vanishing gradients, or oscillating loss. Learning-based techniques, such as DeepFD [7] and DEFault [35], use runtime features to predict DNN fault categories such as loss-function, learning-rate, or activation faults. Both lines of work remain useful for architecture-agnostic model and training faults, but they observe behavior at the model level and do not localize faults to transformer-specific components such as masking, the query-key-value (QKV) projection, positional encoding, or key-value caching. Transformer-specific debuggers inspect attention behavior more directly. ATTNChecker [45] detects `NaN` or `Inf` values during attention computation, and AtPatch [88] localizes attention-map anomalies at the patch level, but each targets a narrow failure mode rather than the full range of transformer faults. As a result, current techniques report only coarse, model-level categories such as incorrect activation or incorrect loss, without identifying which transformer component caused a fault or which cause inside that component explains it. Silent faults are the hardest case for this model-level view, since they leave the loss curves and finite numerical values these techniques monitor unchanged.

To address this gap, we propose DEFault++, a hierarchical learning-based technique for diagnosing faults in transformer models. DEFault++ produces three levels of diagnosis. It first detects whether a fault is present, then assigns the fault to a transformer component category such as the QKV projection or masking, and finally identifies the root cause within that category. Together, these three levels point a developer toward the responsible component and the cause behind the observed behavior rather than toward a generic training symptom, so a developer knows which component to inspect and why it is faulty. Even fault detection is difficult here, since a fault can shift internal attention behavior while the overall training metrics look normal. Our key idea is that each such fault still leaves a distinctive pattern inside the affected component. DEFault++ measures how each transformer component behaves during training and connects these measurements through a structural prior, the *Fault Propagation Graph (FPG)*, derived from the transformer's forward and backward equations. This prior organizes the runtime measurements so the diagnostic model can link an observed deviation to a fault category and report the measurements behind each diagnosis. Training and evaluating DEFault++ requires transformer faults labeled at the component level, which no existing benchmark provides. We therefore construct DEFault-bench, a benchmark of 5,556 labeled instances from encoder and decoder architectures, generated with a transformer-specific mutation technique. On DEFault-bench, DEFault++ improves fault detection over four prior DNN debugging techniques, reaching an F1 of up to 0.91 for fault detection. To confirm that these findings extend to faults found in practice, we further evaluate DEFault++ on real-world faults collected from open-source repositories. In a developer study with 21 participants, repair-action accuracy rises from 57.1% without assistance to 83.3% with DEFault++.

In summary, we make the following contributions:

(a) We propose *DEFault++*, a hierarchical technique that detects a fault in a transformer model, categorizes the affected component, and identifies the root cause within that component.

(b) We construct *DEFault-bench*, a benchmark of 5,556 labeled transformer instances across encoder and decoder architectures, and release the transformer-specific mutation and fault-injection framework behind it, which others can reuse to generate further transformer fault data [32].
(c) We define a *Fault Propagation Graph (FPG)*, a structural prior that encodes the dependency paths among transformer components and gives the diagnostic model the transformer's architectural structure.
(d) We design a training objective for root-cause diagnosis that combines supervised contrastive learning with prototype matching, together with an FPG-based explanation that reports which feature groups support each diagnosis.
(e) We evaluate DEFault++ on DEFault-bench, against four prior techniques, on real-world faults, and in a developer study with 21 participants.

2 Motivating Example

To show the kind of fault DEFault++ targets, consider the real-world fault in Listing 2.1, taken from the Hugging Face Diffusers library (Issue #11903[1]). The method `fuse_qkv_projections` merges the separate query, key, and value projections into one fused layer and routes the forward pass through that fused layer. The original projection modules stay in the model so that LoRA [24] can still attach adapters to them. The forward pass, however, no longer uses those modules, so LoRA updates parameters that no longer affect the model output. The run completes without exceptions, the loss decreases, and no invalid values appear. The only visible sign of the fault is that fine-tuning does not change model behavior.

Diagnosing this fault is hard for existing techniques. A static analysis that inspects the model structure or parameter list still finds valid `q_proj`, `k_proj`, and `v_proj` modules, so it cannot tell that their updates no longer reach the forward pass. Rule-based dynamic analyses that monitor training, such as AutoTrainer [97], UMLAUT [74], and DeepDiagnosis [84], do not flag the run because it shows none of their predefined symptoms, such as `NaN` or `Inf` values, exploding gradients, or oscillating loss. Learned classifiers, such as DeepFD [7] and DEFault [35], recognize predefined DNN fault categories such as activation-function faults, so they offer little help for a fault in a transformer QKV projection path. Transformer debuggers such as ATTNChecker [45] and AtPatch [88] inspect attention behavior more directly, but they target narrower symptoms such as invalid attention scores, which this example does not produce. In contrast, DEFault++ measures runtime traces such as QKV alignment, attention entropy, and projection update activity during training. These measurements characterize the faulty QKV projection path, so DEFault++ detects the fault, assigns it to the QKV category, and identifies the root cause as a stale projection update path.

3 Background and Related Work

Fault Taxonomies. Empirical studies of deep learning (DL) programs classify faults by root cause, covering faults in training procedures, hyperparameters, and layer configurations across feed-forward, convolutional, and recurrent architectures [27, 29]. These taxonomies supply the fault categories used by several DL diagnosis techniques, but they predate attention-based architectures and do not describe faults that arise from attention computation, masking, projection paths, positional encodings, key-value caching, or transformer-specific kernel behavior. In our prior work, we addressed part of this gap with a taxonomy of attention faults that classifies 555 attention faults from open-source

[1] https://github.com/huggingface/diffusers/issues/11903

```python
class Attention(nn.Module):
    def __init__(self, dim, heads=8):
        super().__init__()
        self.dim = dim
        self.q_proj = nn.Linear(dim, dim, bias=False)
        self.k_proj = nn.Linear(dim, dim, bias=False)
        self.v_proj = nn.Linear(dim, dim, bias=False)
        self.out_proj = nn.Linear(dim, dim)
        self.to_qkv = None  # populated by fuse_qkv_projections

    def fuse_qkv_projections(self):
        """Fuse Q/K/V into a single linear layer for efficiency."""
        w = torch.cat(
            [self.q_proj.weight, self.k_proj.weight, self.v_proj.weight],
            dim=0,
        )
        self.to_qkv = nn.Linear(self.dim, 3 * self.dim, bias=False)
        with torch.no_grad():
            self.to_qkv.weight.copy_(w)
        # fault: q_proj, k_proj, v_proj NOT deleted; LoRA still attaches to them

    def forward(self, x):
        # fault: q_proj/k_proj/v_proj are bypassed; forward uses fused layer
        q, k, v = self.to_qkv(x).chunk(3, dim=-1)
        # ... attention computation with q, k, v ...
        return self.out_proj(attn_output)

# Downstream: LoRA targets the stale projections
lora_config = LoraConfig(target_modules=["q_proj", "v_proj"])  # inactive in forward
model = get_peft_model(pipeline.unet, lora_config)
model.train()  # LoRA adapts stale layers; no effect on forward pass
```

Listing 2.1. A real-world transformer fault (Hugging Face Diffusers Issue #11903).

projects into seven categories and 25 root causes [33, 34]. The present work builds its fault labels on these attention-specific categories together with five non-attention categories from the DL fault taxonomies [27, 29], which cover the surrounding model and training faults such as loss, learning-rate, and activation faults (Section 6.1).

Mutation Testing for Deep Learning. Mutation testing assesses how well a test suite detects faults by introducing small, controlled changes into a program, called *mutants*, and checking whether the tests reveal them [77]. The deep learning community has adapted this idea to neural models. DeepMutation [49] and DeepMutation++ [25] introduce mutation operators that modify model structure, training code, or learned parameters, and report mutation scores to quantify test adequacy. DeepCrime [28] injects faults into the source code at training time, reruns training several times, and applies statistical tests with effect-size thresholds to decide whether a mutation changes behavior. This repetition matters because the same injected change can look harmful under one random seed and negligible under another [57]. PyTorchFI [51] injects faults at the tensor and bit levels to study the fault tolerance of PyTorch models. Mutation testing can also supply labeled training data for fault diagnosis. DeepFD [7] injects five common training faults, such as an incorrect learning rate, to create labeled samples for supervised diagnosis, and later work extends this idea to larger and more diverse sets of faulty DNN programs [35, 77, 85]. These datasets mostly cover architecture-agnostic DNN faults, such as loss misconfiguration, incorrect activation functions, or incorrect layer counts. They do not target the transformer-specific mechanisms or attention-related behavior needed to diagnose transformer faults (Table 1).

Table 1. Comparison of the DEFault-bench mutation process with prior mutation techniques. ✓ marks a supported dimension and ✗ one that is not supported.

Dimension	DeepMut.++ [25]	DeepCrime [28]	DeepFD [7]	PyTorchFI [51]	DEFault-bench
Fault granularity	Layer/param	Source (AST)	Program	Bit/tensor	Transformer unit
Mechanism count	17	24 of 35	5 types	Bit-flip	12 categories
Architecture coverage	FFNN/CNN/RNN	Generic DNN	Generic DNN	CNN (infer.)	Encoder + decoder
Attention mechanisms	✗	✗	✗	✗	✓
Behavior change	✗	✗	✗	✗	✓
Decision basis	Accuracy	GLM + Cohen's d	Accuracy	SDC rate	Accuracy + sign-flip
Noise control	✗	✓	✗	✗	✓
Scale (reported)	Varies	1,760	52 programs	10^7+ inj.	5,556
Labeled traces	✗	✗	✓	✗	✓

Debugging Techniques for Deep Learning Programs. Existing DL debugging techniques fall into three broad groups. The first group learns fault patterns from training behavior. DeepFD [7] extracts four feature types, namely loss, gradient, weight, and activation features, and classifies five common training faults. Our prior work DEFault [35] extends this idea into a hierarchical classifier that first detects whether a program is faulty and then assigns it to one of seven generic DNN fault categories. These two methods are the closest dynamic baselines to our work, since they infer fault labels from training-time measurements, and DEFault++ extends our hierarchical design from generic DNN fault categories to transformer-specific components. They observe behavior at the model level and report model-level fault categories. As a result, they cannot tell whether a symptom comes from the QKV projection, masking, positional encoding, residual connections, normalization, key-value caching, or another transformer component. DEFault++ targets exactly this finer, component-level granularity.

Table 2. Comparison of fault diagnosis techniques for DNN programs. FD, FC, and RC denote fault detection, fault categorization, and root-cause diagnosis, and a parenthetical count gives the number of categories or root causes a technique reports. DEFault++ covers 45 root causes in total, 40 for encoders and all 45 for decoders. ✓ marks a supported capability and ✗ one that is not supported.

Technique	Target	FD	FC	RC	Attn-Aware	Explanation
AutoTrainer [97]	DNN	✓	Partial (5)	✗	✗	Rule-based
UMLAUT [74]	DNN	✓	Partial	✗	✗	Rule-based
DeepDiagnosis [84]	DNN	✓	✓ (8)	✗	✗	Decision tree
DeepLocalize [86]	DNN	NaN/Inf only	✗	✗	✗	Numerical trace
DeepFD [7]	DNN	✓	✓ (5)	✗	✗	Feature attribution
DEFault [35]	DNN	✓	✓ (7)	✗	✗	SHAP
ATTNChecker [45]	Transformer	NaN/Inf only	✗	✗	✓	✗
AtPatch [88]	Transformer	Map outlier	✗	✗	✓	✗
FT-Transformer [10]	Transformer	Soft errors	✗	✗	✓	✗
DEFault++ (Ours)	Transformer	✓	✓ (12)	✓ (45)	✓	FPG+prototype

Rule-based and heuristic techniques monitor predefined failure symptoms. AutoTrainer [97], UMLAUT [74], and DeepDiagnosis [84] check training-time symptoms such as exploding gradients, vanishing gradients, and oscillating loss. DeepLocalize [86] and GRIST [93] focus on numerical instability, while Tensfa [91] and TFCheck [4] check tensor shapes and training-process invariants. Other methods work at finer granularity, including MODE [50], DeepFault [14], Apricot [95], and DeepSeer [83]. These techniques work well when a fault produces one of the symptoms or structural patterns they monitor. They have limited reach, however, for transformer faults that keep loss curves normal, tensor shapes valid, and numerical values finite while changing internal attention behavior.

A third group of techniques analyzes a model statically, before any training run, so it operates at a different level of abstraction from ours. NeuraLint [60], DEBAR [98], and NerdBug [31] check meta-model graphs, tensor abstractions, and API usage, while DeepCheck [21] and CRADLE [65] analyze trained networks or compare backend implementations. FL4Deep [56] sits between the static and dynamic views, combining both through a knowledge graph to rank candidate

root causes. These techniques find structural, symbolic, API-level, or backend-level faults, but they do not diagnose transformer-component faults from training traces. We therefore treat them as complementary to DEFault++ rather than as direct baselines.

Transformer-Specific Fault Analysis. A smaller body of work analyzes transformer-internal behavior, but each technique targets a narrower failure mode than the full transformer fault taxonomy (Fig. 3). ATTNChecker [45] detects NaN or Inf values during attention computation, AtPatch [88] localizes attention-map anomalies at the patch level, FT-Transformer [10] studies hardware soft-error tolerance for vision transformers, and Bug Attention Probe [76] applies attention probing to code-understanding tasks. None of these techniques produces a combined hierarchy of fault detection, component category, and root-cause diagnosis. A related direction perturbs inputs to study model behavior [9, 36, 71, 78, 80]. Their goal differs from ours, since these methods expose behavioral failures through modified inputs, whereas we diagnose implementation faults that arise during ordinary training. Debuggers that use large language models (LLMs), such as SoapFL [66], LLM4FL [69], ChatDBG [43], and BugReAct [30], reason over source code, execution logs, or bug reports to localize faults in general software. They are also complementary, since they operate after a developer provides software artifacts or failure reports, while DEFault++ reads training-time traces from transformer components to infer a transformer-specific fault category and root cause.

Explainability for Fault Diagnosis. Diagnostic output is more useful when it gives developers evidence for a prediction rather than only a label, because the evidence shows where to look and gives a reason to trust the diagnosis. Existing explanation methods fall into two families. Post-hoc methods compute explanations after a trained model makes a decision. SHAP [48], DiCE [58], and Anchors [70] return feature attributions, counterfactual examples, or rule-based explanations, and our prior method DEFault adopts this post-hoc approach [35]. Inherently interpretable methods instead build explanation into the prediction mechanism. Prototype networks [75] explain a prediction through similarity to class prototypes, and concept-bottleneck models [41] route predictions through human-readable intermediate concepts. DEFault++ follows this interpretable-by-design approach and reports the training-time evidence behind each transformer fault diagnosis.

4 Study Design

Our study comprises three parts. We first construct DEFault-bench, a transformer fault benchmark built through mutation testing that provides labeled examples of correct and faulty fine-tuning behavior. We then design DEFault++, a hierarchical diagnostic technique that detects a fault, categorizes the affected component, and identifies its root cause, organized around how faults propagate across transformer components. Finally, we evaluate DEFault++ on DEFault-bench, on real-world faults, and through a developer study. We organize the evaluation around four research questions that move from whether the technique works, to how it compares with prior work, to which of its design choices matter, to whether the evidence it reports can be trusted.

- **RQ_1 (Effectiveness):** How effectively does DEFault++ detect, categorize, and diagnose the root cause of transformer faults? No prior technique resolves a transformer fault to both its component and its root cause, so the first step is to establish whether the three-level design works at all. This question measures each level on the evaluated transformer models, before any comparison with prior work.

- **RQ_2 (Comparison):** How does DEFault++ compare with existing deep learning fault-detection techniques? Prior techniques target generic DNN faults and observe behavior at the model level, so it is unclear whether they detect faults that surface only inside transformer components. This question tests whether component-level measurement adds detection value over these model-level techniques.

- **RQ_3 (Design contribution):** How much do the Fault Propagation Graph and the root-cause separation training contribute to diagnostic performance? DEFault++ combines several design choices, and a strong overall result does not show which of them carry the performance. This question isolates the contribution of the propagation structure and the separation training at each diagnosis level.

- **RQ_4 (Explanation faithfulness):** Do the runtime feature groups that DEFault++ reports as evidence drive its root-cause diagnosis? A diagnosis is more useful when it comes with evidence a developer can inspect, but reported evidence is trustworthy only if it reflects the model's actual decision. This question tests whether the reported feature groups, rather than incidental ones, determine the predicted root cause.

5 Experimental Setup

This section describes the transformer models and tasks we study and the cross-validation protocol we use to evaluate generalization. The benchmark construction (Section 6) and the diagnostic technique (Section 7) both build on this setup.

5.1 Subject Models and Tasks

We select the transformer models for our study by three criteria. A model must be widely used, openly available through a standard Hugging Face Transformers implementation [26], and small enough, at most 125M parameters, to fine-tune repeatedly under matched seeds within our compute budget. Applying these criteria gives four encoder-only and three decoder-only models that together cover both architecture families, two depths, and distilled and full-size variants of the same family. The encoder models are BERT-base (110M parameters, 12 layers) [12], RoBERTa-base (125M, 12) [46], DistilBERT (66M, 6) [73], and DistilRoBERTa (82M, 6) [73]. The decoder models are GPT-2 (124M, 12) [68], GPT-Neo-125M (125M, 12) [5], and DistilGPT-2 (82M, 6) [73].

We select downstream tasks that are commonly adopted in prior transformer and deep learning studies and that come with openly released datasets. Each task is a publicly available benchmark with an established evaluation metric, and the set spans both classification and language modeling so that the injected faults are exercised under different training objectives. For the encoder models, we use five GLUE classification tasks [81], namely SST-2, QNLI, RTE, MRPC, and QQP, with accuracy as the task metric. For the decoder models, we use four language-modeling corpora, namely LAMBADA [63], PTB [53], WikiText-2 [55], and OpenWebText [19], with log-perplexity as the task metric. All models use their default tokenizers and the standard transformer-block structure that our mutation operators target.

We fine-tune every model with a single shared setup across tasks, namely AdamW [47] with linear warmup and mixed precision [12, 57], so that observed differences trace to the injected fault rather than to hyperparameter variation. We repeat each fine-tuning run under five fixed seeds $\mathcal{S} = \{42, 123, 456, 789, 101112\}$ [57] and vary the affected layers and the fault severity uniformly, so that no single layer or magnitude dominates the benchmark.

5.2 Cross-Validation Protocol

We evaluate generalization to unseen model–task pairs using nested grouped cross-validation with $k = 5$ outer folds. The grouping unit is the model–task pair, so all runs from the same pair remain in the same fold. This prevents correlated traces from the same model and task from appearing in both training and test data. The encoder subset contains 20 model–task pairs (4 models × 5 GLUE tasks), and the decoder subset contains 12 model–task pairs (3 models × 4 language-modeling tasks). Because we repeat each run across seeds, we average those per-seed measurements into one example, so the random seed is not used as a grouping variable. The outer loop evaluates held-out model–task pairs, and the inner loop uses `StratifiedGroupKFold` for model selection [87]. We fit preprocessing steps and tune hyperparameters only inside the inner training fold, and all methods use the same outer-fold assignments. Because each outer fold holds out whole model–task pairs, a fold-level standard deviation would mostly reflect which pairs are held out rather than training noise, so we report results at the model–task level.

6 DEFault-bench: Benchmark Construction

Training and evaluating a component-level diagnostic technique requires transformer faults labeled by component, which no existing benchmark provides. We therefore construct DEFault-bench, a benchmark of clean and faulty fine-tuning runs for the seven transformer models and nine tasks introduced in Section 5. We create the faulty runs with a transformer-specific mutation technique that injects documented attention and DNN faults into the components of the transformer architecture, either by changing stored parameters or by changing computations during the forward pass. We organize these faults with a fault taxonomy (Section 6.1) and realize each one as a mutation operator (Section 6.2). Each injected fault is one configuration $C = (m, t, u, f, v, \ell, \sigma)$ that fixes the model, task, target component, fault category, mutation variant, target layer, and severity. We validate each mutant in two steps. We first check that the intended structural change was applied. We then test whether the faulty run behaves differently from its matched clean run under the same seeds, following prior mutation-testing work [28, 49] (Fig. 1).

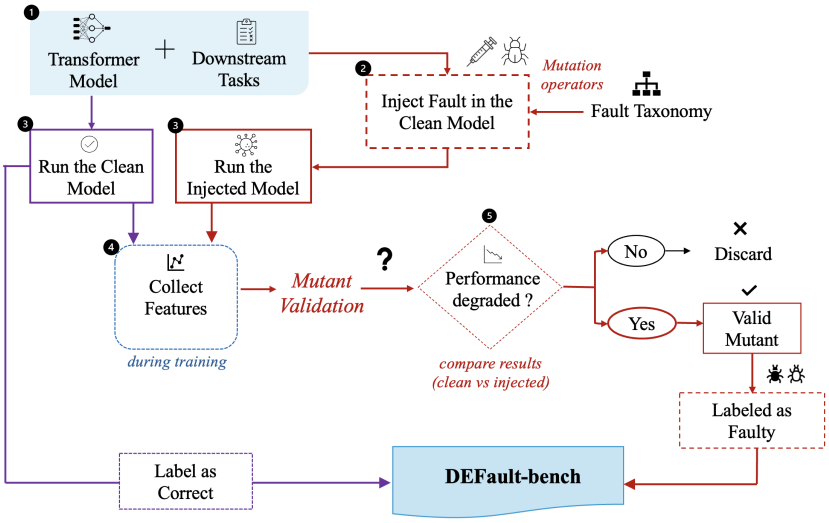


Fig. 1. Workflow for constructing DEFault-bench

6.1 Fault Taxonomy

We base the fault taxonomy on the main components of a transformer architecture (Fig. 2). A transformer model contains input embeddings, a stack of repeated transformer blocks, and an output projection head [79]. Each block contains multi-head self-attention (MHA), a position-wise feed-forward network (FFN), residual connections, and layer normalization (LayerNorm). For an input sequence $\mathbf{x} \in \mathbb{R}^{n \times d}$, a post-norm block adds each sublayer's output back to its input and then normalizes the sum, computing $\mathbf{h} = \text{LayerNorm}(\mathbf{x} + \text{MHA}(\mathbf{x}))$ for the attention sublayer and $\mathbf{y} = \text{LayerNorm}(\mathbf{h} + \text{FFN}(\mathbf{h}))$ for the feed-forward sublayer. We group faults by six components, namely embeddings, MHA, FFN, layer normalization, residual connections, and the output projection head. The same component set applies to pre-norm variants, where only the position of LayerNorm changes.

We then map these transformer components to documented fault categories and root causes (Fig. 3). The attention-related categories come from our prior fault taxonomy for attention-based neural networks (ABNNs), which we derived from 555 real-world attention faults [33, 34]. The non-attention categories come from two existing DNN fault taxonomies based on 970 faults in total [27, 29].

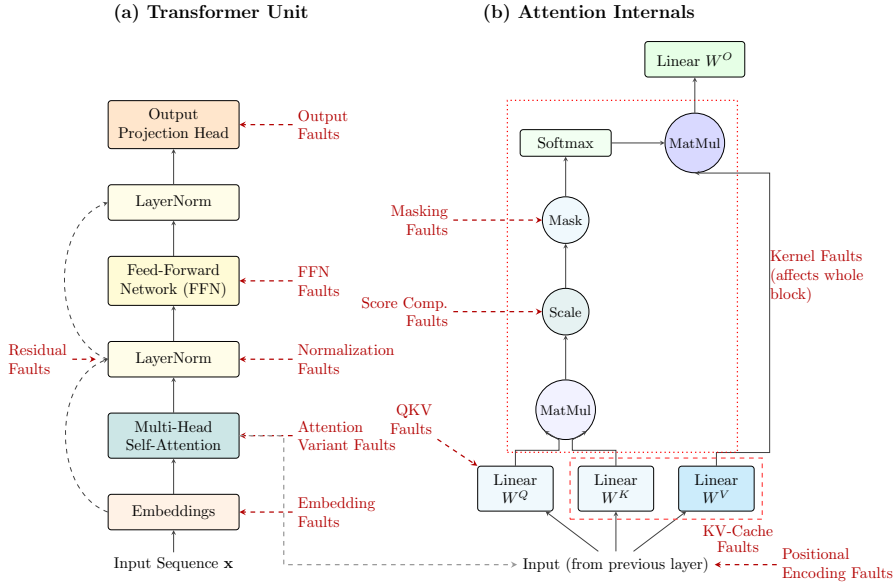


Fig. 2. Fault categories organized by transformer components

The seven attention categories map to different parts of the attention computation [34, 79], illustrated in the attention-internal view of Fig. 2,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V \tag{1}$$

where Q, K, and V are linear projections of the input, d_k is the key dimension, and M is an additive mask. *Masking* faults corrupt the mask M through padding masks, causal masks, or mask reshaping across batch and head dimensions. *QKV*, *Score*, and *Positional* faults affect the projections, the attention-score computation, and the positional information, respectively. *Kernel* faults involve the configured attention backend or numerical execution path. *Variant* faults use the wrong form of attention, such as single-head attention or causal masking in an encoder. *KV Cache* faults apply

to autoregressive decoders and affect cache-state management [2]. The remaining components each contribute one category, following prior DNN fault taxonomies [27, 29]. *Embedding* faults involve token, segment, or feature-dimension embeddings. *FFN* faults cover feed-forward weights, neurons, and activation functions. *LayerNorm* faults involve the learned scale γ, bias β, numerical-stability term ϵ, or regularization of LayerNorm parameters [57, 92]. *Residual* faults cover skip connections and residual dropout, and *Output* faults involve the prediction head or logits.

Overall, the taxonomy contains 12 fault categories and 45 root causes, including five KV Cache root causes that apply only to autoregressive decoders. Transformer models without autoregressive cache behavior, such as encoder-only models, use 11 of these categories and 40 root causes (Fig. 3).

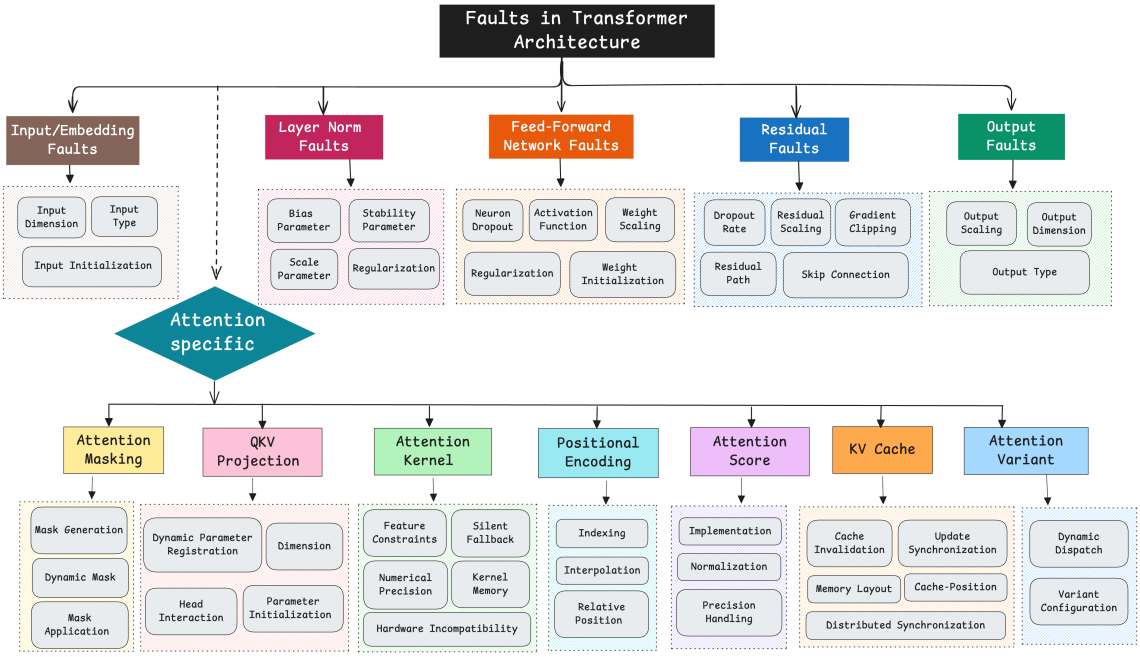


Fig. 3. Taxonomy of transformer fault categories and root-cause labels used by DEFault++

6.2 Mutation Operators

To generate faults systematically, we realize each root cause as one or more *mutation operators*, where a mutation operator is a specific code change that injects that root cause into a transformer model. We read the root causes from the taxonomy in Fig. 3 and define an operator for each distinct way a root cause can be realized as a code change, which yields 52 operators across the 12 fault categories. We present the operators in two tables that follow the two taxonomy sources. The 29 attention-specific operators come from our attention fault taxonomy and target masking, the QKV projection, score computation, positional information, the attention kernel, attention variants, and the decoder KV cache (Table 3). The 23 architecture-level operators come from the DNN fault taxonomies and target embeddings, the FFN, layer normalization, residual connections, and the output head (Table 4). This grouping reflects where a fault originates in the model, which is separate from the run-time mechanism used to inject it (Section 6.3).

We give each operator a three-letter identifier, following the naming style of prior DL mutation work [28, 49]. The first letter identifies the transformer component, and the remaining two letters form a short mnemonic for the change.

Table 3. Attention-specific mutation operators used to construct DEFault-bench

Component	Root cause	Operator	ID	Executable Mutation Parameters	ST
Masking	Mask application	Zero attention mask	MZM	Replace valid attention mask with an all-zero mask of the same shape.	B
	Mask application	Invert attention mask	MIM	Invert mask semantics while preserving expected mask shape & dtype.	B
	Mask generation	Reshape mask incorrectly	MRM	*axis*: batch or head dimension to misalign; retained only if broadcast-compatible.	EL
	Dynamic mask	Causal-mask break (decoder)	MCB	*visibility*: fraction of future keys made visible within valid mask tensor.	EU
QKV Projection	Parameter initialization	Zero Q projection	QZQ	Zero the existing Q projection weights without changing tensor shape.	B
	Parameter initialization	Zero K projection	QZK	Zero the existing K projection weights without changing tensor shape.	B
	Parameter initialization	Zero V projection	QZV	Zero the existing V projection weights without changing tensor shape.	B
	Head interaction	Swap $Q \leftrightarrow K$	QSW	Swap compatible Q and K projection tensors or outputs.	B
	Head interaction	Tie head weights	QTH	*heads*: subset of existing heads tied while preserving projection dimensionality.	EL
	Dynamic parameter registration	Freeze QKV gradients	QFG	Disable updates to existing QKV parameters while preserving the forward path.	B
	Dimension	Repartition head dimensions	QHD	*split*: alternative $(\text{heads}, d_{\text{head}})$ head factorization with $\text{heads} \cdot d_{\text{head}} = d_{\text{model}}$, retained only when broadcast-compatible.	EL
Score	Normalization	Drop $1/\sqrt{d_k}$ scaling	SDS	Remove score scaling while preserving attention-score shape.	B
	Implementation	Apply dropout pre-softmax	SPD	p: dropout probability applied to score tensors before softmax.	EU
	Precision handling	Unsafe fp16 cast	SUC	Cast score computation to fp16; configurations producing non-finite traces are discarded.	B
Positional	Indexing	Omit positional embeddings	POE	Remove or zero the positional contribution while preserving hidden-state shape.	B
	Relative position	Shift position indices	PSI	Δ: index shift restricted to valid supported positions.	EU
	Interpolation	Truncate positional support	PTL	*cutoff*: maximum retained position index; out-of-range indices are discarded.	EU
Kernel	Silent fallback	Force slow backend	KSB	Force a valid non-optimized attention backend.	B
	Feature constraints	Mismatched dropout probability	KMD	$p_{\text{cfg}}, p_{\text{kern}}$: valid training vs. kernel dropout settings.	EU
	Hardware incompatibility	Trigger valid fallback (dtype/layout)	KFT	*dtype* or *layout* mismatch that triggers a supported fallback path.	EL
	Numerical precision	Reduced-precision kernel math	KRP	*precision*: reduced-precision accumulation mode for the attention kernel. Configurations producing non-finite traces are discarded.	EL
	Kernel memory	Memory-constrained kernel path	KMC	Force the attention kernel onto a reduced-memory chunked execution path. Configurations producing non-finite traces are discarded.	B
Variant	Variant configuration	Single effective head attention	VSH	Route, tie, or mask heads so that only one effective head remains while preserving output shape.	B
	Dynamic dispatch	Causal mask in encoder	VEC	Apply a broadcast-compatible causal mask in an encoder attention layer.	B
KV Cache (decoder)	Cache invalidation	Stale cache	CST	*layers*: subset of layers using stale but shape-compatible cached K, V states.	EL
	Update synchronization	Desynchronized cache update	CDU	Withhold the current decoding step's freshly computed K, V from the cache so attention and prediction read desynchronized cached states. Cache tensor shape is preserved.	B
	Cache-position	Off-by-one indexing	COB	*shift*: cache index offset restricted to valid cache positions.	EU
	Memory layout	Truncate cache	CTR	*length*: maximum retained cache length while preserving the expected cache interface.	EU
	Distributed synchronization	Cross-request leak	CLK	Reuse compatible cached states across requests without changing cache tensor shape.	B

For example, QZQ uses **Q** for the QKV component, **Z** for zeroing, and **Q** for the query projection, while SPD uses **S** for the Score component, **P** for pre-softmax, and **D** for dropout. A fault category can hold more than one operator because one root cause can be triggered through more than one separate change. The QKV parameter-initialization root cause, for example, has three operators that zero the query, key, or value projection (QZQ, QZK, QZV), since each projection is a separate entry point for the same root cause. This is why the 52 operators map to 45 root causes rather than one operator each.

The Search Type (ST) column records how an operator selects the parameter listed in the Executable Mutation Parameters column. A **B** (binary) operator takes no parameter and applies a single fixed change. A **EU** operator has a numeric parameter swept over a predefined list of values, such as a scaling factor or a dropout probability. A **EL** operator has a categorical parameter chosen from a fixed set, such as FCA, which replaces the activation function with each function in {ReLU, GELU, Tanh, Sigmoid}. For parameterized operators, our mutation technique evaluates low,

medium, and high settings, so the benchmark spans a range of fault strengths rather than only extreme mutants. Extreme mutants carry little diagnostic information, since almost any test exposes them [28].

Table 4. Architecture-level mutation operators used to construct DEFault-bench

Component	Root cause	Operator	ID	Executable Mutation Parameters	ST
Embedding	Input initialization	Zero token embedding subset	ETZ	*percentage*: fraction of vocabulary entries zeroed while preserving embedding shape.	EU
	Input type	Swap embedding pairs	ESW	*percentage*: fraction of token pairs swapped within existing vocabulary.	EU
	Input type	Scale segment / type embedding	ESS	*factor*: multiplicative scale applied to existing segment or type embeddings.	EU
	Input dimension	Zero embedding feature dimensions	EZD	*fraction*: contiguous block of embedding feature dimensions zeroed while embedding tensor shape is preserved.	EU
FFN	Weight scaling	Scale FFN weights	FSW	*factor*: multiplicative scale on existing W_1, W_2 tensors.	EU
	Neuron dropout	Permanently drop neurons	FDN	*percentage*: fraction of hidden neurons zeroed while preserving FFN shape.	EU
	Activation function	Change activation function	FCA	*activation*: executable replacement nonlinearity (e.g., GELU → ReLU).	EL
	Regularization	Change weight regularization	FRG	*scheme*: replacement weight-decay or L_2 coefficient accepted by the optimizer.	EU
	Weight initialization	Change weight initialization	FWI	*init*: replacement initialization scheme applied to existing W_1, W_2 tensors.	EL
LayerNorm	Scale parameter	Scale γ	NSG	*factor*: multiplicative scale on existing γ parameters.	EU
	Scale parameter	Zero γ	NZG	Zero existing γ parameters while preserving LayerNorm shape.	B
	Bias parameter	Shift β	NSB	*shift*: additive offset on existing β parameters.	EU
	Stability parameter	Change ϵ	NCE	*value*: positive numerical-stability constant accepted by LayerNorm.	EU
	Regularization	Weight-decay LayerNorm parameters	NWD	*coefficient*: weight-decay coefficient applied to the LayerNorm γ, β parameters, which are normally excluded from decay.	EU
Residual	Skip connection	Remove skip connection	RRS	Zero/bypass residual branch while preserving sublayer output shape.	B
	Residual scaling	Scale residual branch	RSR	*factor*: multiplicative scale on the residual path.	EU
	Residual path	Inject Gaussian noise	RIN	σ: standard deviation of additive noise with matching tensor shape.	EU
	Gradient clipping	Change gradient clipping	RGC	*value*: replacement max-norm clip threshold accepted by the training loop.	EU
	Dropout rate	Change residual dropout rate	RDR	*p*: residual-branch dropout probability accepted by the training loop.	EU
Output	Output scaling	Scale output logits	OSL	*factor*: multiplicative scale on logits while preserving output dimension.	EU
	Output dimension	Zero rows for selected classes	OZR	*classes*: subset of existing output rows to zero.	EL
	Output type	Reinitialize output projection	ORI	*init*: replacement initialization scheme applied to existing output projection.	EL
	Output dimension	Change output interface	OOD	*dim/map*: root-cause-derived output-dimension or mapping fault; retained only when compatible with the task loss.	EU

6.3 Fault Injection Mechanism

We inject every fault programmatically, so each configuration is applied, evaluated, and reverted without manual edits to the model source, and we release this injection framework in our replication package [32]. We denote the clean model parameters by θ and the injected mutant by θ', and we evaluate the two under matched random seeds and otherwise identical fine-tuning conditions, so that any measured difference reflects the injected configuration C. Recall that $C = (m, t, u, f, v, \ell, \sigma)$ fixes the model architecture, the downstream task, the target unit, the fault category, the fault variant, the affected layer indices $\ell \subseteq \{1, \ldots, L\}$ for a model with L transformer blocks, and the severity level $\sigma \in \{\text{low}, \text{medium}, \text{high}\}$. The severity sets the magnitude of the injected change for numeric variants, such as scaling factors and dropout rates, and maps to a discrete intensity for non-numeric variants, such as the visibility ratio in causal-mask breaking. We use only single-fault configurations, with one target unit, one category, and one variant per run, so that each observed change traces to one injected fault. Transformer faults can affect either stored parameters or forward-pass computations, so injection uses two mechanisms [45, 51].

Static faults change parameter tensors at rest, before forward execution. An FFN weight-scaling fault, for example, multiplies the targeted weight matrices by a scalar,

$$\theta_i' = \begin{cases} \alpha \cdot \theta_i & \text{if } i \in \text{params}(u, \ell), \\ \theta_i & \text{otherwise} \end{cases} \tag{2}$$

where $\text{params}(u, \ell)$ selects the parameters of unit u in layers ℓ. Static faults include embedding corruption, parameter scaling, permanent neuron dropout, and LayerNorm parameter changes. *Dynamic faults* change the forward computation path through wrappers or hooks at selected module call sites. A mask-zeroing fault, for example, intercepts the attention call and replaces the mask M with zeros,

$$\texttt{forward}'(x, M) = \texttt{forward}(x, \mathbf{0}_{|M|}) \tag{3}$$

where $\mathbf{0}_{|M|}$ is a zero tensor of the same shape as M. Dynamic faults include mask manipulation, Q/K swapping, removal of score scaling, and forced kernel selection. In both mechanisms the transformer architecture stays unchanged, and only the targeted computation or parameter values change. Fig. 4 shows one representative example of each mechanism.

(a) Static fault injection (encoder FFN)

```
# static fault: mutate FFN weights at rest
layer = model.bert.encoder.layer[layer_idx]
ffn = layer
orig_w1 = ffn.intermediate.dense.weight
orig_w2 = ffn.output.dense.weight
self._backup = (orig_w1.clone(), orig_w2.clone())
with torch.no_grad():
    orig_w1.mul_(self.alpha)  # scale W1
    orig_w2.mul_(self.alpha)  # scale W2
```

(b) Dynamic fault injection (decoder causal mask)

```
# dynamic fault: wrap the same model's attention.forward to
      weaken causal mask
self._backup_fwd = attention.forward
def faulty_forward(*args, **kwargs):
    mask = _get_attention_mask(kwargs)
    if mask is not None:
        # unmask a fraction of future keys
        m = _weaken_causal_mask(mask, visibility_ratio)
        kwargs["attention_mask"] = m
    return self._backup_fwd(*args, **kwargs)
attention.forward = faulty_forward
```

Fig. 4. Two injection mechanisms on the same model: (a) static parameter mutation scales FFN weights, (b) dynamic forward wrapping weakens the causal mask at runtime. The clean run leaves all modules unmodified.

Masking faults need special handling for decoder-only models, where self-attention is causal by construction and a token at position i cannot attend to a future position $j > i$. The causal mask enforces this by placing $-\infty$ in the future positions of the additive mask M, so that softmax assigns those positions zero weight. To inject a causal violation, we weaken this constraint rather than add a new mask. We replace a configurable fraction of those $-\infty$ entries in the upper-triangular region of M with 0, controlled by the severity σ, so each query can attend to a proportion of keys beyond its own position. We treat causal violations as decoder instances of attention masking faults.

To keep clean and faulty runs comparable, we hold the main experimental settings fixed between the θ and θ' runs, such as the dataset, optimizer, and number of epochs, and we pair each faulty run with a clean run under the same seed. A context manager carries out the injection. It stores the original parameter tensors and forward methods, applies the fault specified by C, and restores the original state after evaluation. Restoring the state at the end of each configuration prevents one injected fault from carrying over into the next run.

Finally, we apply a structural verification check to each injected fault. For static faults, we verify that the parameter difference is restricted to $\text{params}(u, \ell)$ and that its magnitude matches σ within a relative tolerance of 10^{-6} to account for floating-point rounding. For dynamic faults, we verify that hooks or wrappers attach only to the intended module call sites and that the original forward function is restored after the context manager exits. We also verify execution

completeness by requiring each configuration to produce training logs, and we exclude configurations that crash or raise runtime errors.

6.4 Mutant Validation

Injecting a fault does not guarantee that it changes model behavior, so we decide which mutants to keep through a statistical mutation-killing test, following prior work [28]. For each configuration, Algorithm 1 injects the fault, verifies the structural change, fine-tunes the clean and mutant models under the shared seeds, and applies the killing test to the paired results.

Algorithm 1 Per-configuration fault injection and validation

Require: Clean model θ, configuration C, seeds $\mathcal{S}$
Ensure: Label *isKilled* and aggregated run measurements $\mathbf{x}$

$\theta' \leftarrow \text{Inject}(\theta, C)$ ▷ static or dynamic mutation
if $\text{VerifyStructural}(\theta, \theta', C) = \mathit{false}$ **then return** *discard*
end if
for $s \in \mathcal{S}$ **do**
  $a_s^c \leftarrow \text{TrainAndEvaluate}(\theta, s)$; $a_s^f \leftarrow \text{TrainAndEvaluate}(\theta', s)$
end for
$p \leftarrow \text{SignFlipPermTest}(\{a_s^c\}, \{a_s^f\})$ ▷ one-sided, paired
$\mathit{killed} \leftarrow [\, p \le \alpha \,]$ ▷ Eq. (4)
$\mathbf{x} \leftarrow \text{Aggregate}(\{a_s^c\}, \{a_s^f\})$ ▷ clean-to-faulty deltas
return $(\mathit{killed}, \mathbf{x})$

We fine-tune the clean model N (parameters θ) and the mutant M (parameters θ') each n times under matched random seeds, and we compare their performance distributions $A_N(\mathit{TestS}) = \langle A_{N_1}, \ldots, A_{N_n} \rangle$ and $A_M(\mathit{TestS}) = \langle A_{M_1}, \ldots, A_{M_n} \rangle$ on a held-out test set *TestS*. The predicate *isKilled* decides whether the mutant is killed:

$$\mathit{isKilled}(N, M, \mathit{TestS}) = \begin{cases} \mathit{true} & \text{if } p\text{-value}\big(A_N(\mathit{TestS}),\ A_M(\mathit{TestS})\big) < \alpha, \\ \mathit{false} & \text{otherwise.} \end{cases} \tag{4}$$

We use accuracy as the test-set metric for encoder classification tasks and log-perplexity for decoder language-modeling tasks [34], and we evaluate each clean and faulty configuration with the same $n = 5$ seeds at $\alpha = 0.05$. The matched seeds let us compare each faulty run directly with its clean counterpart. For each mutation, we compute the five paired differences and test whether they move consistently in the expected direction with a one-sided paired sign-flip permutation test [20]. This test asks whether the observed direction of the paired differences is unlikely under the null hypothesis that the fault has no effect. With five paired comparisons, the smallest possible one-sided exact p-value is $1/2^5 \approx 0.031$, which lies below $\alpha = 0.05$, so five matched seeds suffice for this exact test. To confirm that the test does not flag ordinary training noise as a fault, we also apply *isKilled* to pairs of clean runs that differ only in random seed, and the rate at which these clean-versus-clean pairs are flagged estimates the false-positive rate. We summarize each operator's effectiveness with the mutation score [28]:

$$MS(MO) = \frac{|\{c \in MO \,:\, \mathit{isKilled}(N, M_c, \mathit{TestS})\}|}{|MO|} \tag{5}$$

where MO is the set of injected configurations of an operator, and we report the overall mutation score as the average of $MS(MO)$ across operators. For each killed mutant we record a label $y = (u, f, v, \ell, \sigma)$ with the target unit, fault category,

fault variant, affected layer set, and severity. We omit m and t because each instance comes from one model–task context. A surviving mutant, for which *isKilled* returns false, is an injected fault whose measured effect does not pass the killing test, so its label cannot be inferred reliably. We discard surviving mutants before building the final benchmark instances.

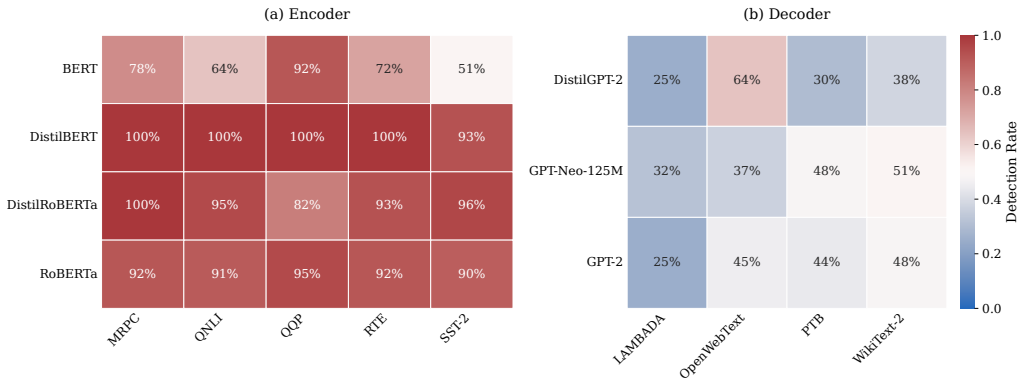


Fig. 5. Mutation scores per (model, task) pair under *isKilled* at $\alpha = 0.05$

6.5 Benchmark Statistics

After structural verification, we obtain 4,096 single-fault configurations (1,955 for encoders and 2,141 for decoders), with every category represented. Validating these mutants requires 40,960 paired clean and faulty fine-tuning runs, and constructing the balanced correct class adds 13,890 clean-variant runs, for a total of 54,850 fine-tuning runs and about 27,000 GPU-hours on NVIDIA A100 and H100 GPUs. We apply *isKilled* to all structurally valid configurations and compute the mutation score by category and architecture (Table 5, Fig. 6). Among encoder configurations, 1,598 of 1,955 mutants are killed (81.7%). Among decoder configurations, 1,180 of 2,141 mutants are killed (55.1%). Encoder mutation scores are high across most categories. Variant reaches 100%, and Score, QKV, Positional, FFN, Residual, LayerNorm, and Embedding all exceed 80%, while Kernel, Output, and Masking are the lowest. Decoder mutation scores vary more across categories and model–task pairs (Fig. 5). Masking and Kernel reach the highest decoder scores, and Score the lowest. The low Score rate is consistent with how softmax dampens score-level changes. Softmax maps the pre-softmax logits to a normalized distribution, so a moderate score change can be partly absorbed before it reaches the attention weights [13, 79, 94]. For decoder language modeling, this can keep the perplexity change below the threshold the killing test uses.

Table 5. Composition of DEFault-bench. Each value is the number of killed mutants in a fault category. Encoders contribute 1,598 faulty and 1,598 correct instances (3,196 total), and decoders 1,180 faulty and 1,180 correct (2,360 total).

Category	Variant	Score	QKV	Pos.	FFN	Residual	LayerNorm.	Emb.	Out.	Mask	Kernel	KV
Encoder	25	269	237	166	254	114	181	147	113	41	51	n/a
Decoder	23	40	78	137	190	112	159	144	74	137	68	18

After discarding surviving mutants, we build the *correct* class from clean runs and label-preserving clean variants. Using a single clean run per base model could let a classifier learn model-specific artifacts rather than fault-related behavior. We therefore generate clean variants by changing factors that should not affect task behavior, such as

the random seed, the training-data order where applicable, and hyperparameters within ranges that stay below the significance threshold. We evaluate each clean variant against its base model with the same significance test used for killed mutants, retain variants that remain statistically indistinguishable from the base model, and discard variants that meet the killed criterion. For each base model B that produces k killed mutants, we keep k clean variants from the same base model, which holds the faulty-to-correct ratio balanced within each base-model stratum. DEFault-bench contains 5,556 instances, with 3,196 encoder and 2,360 decoder instances (Table 5).

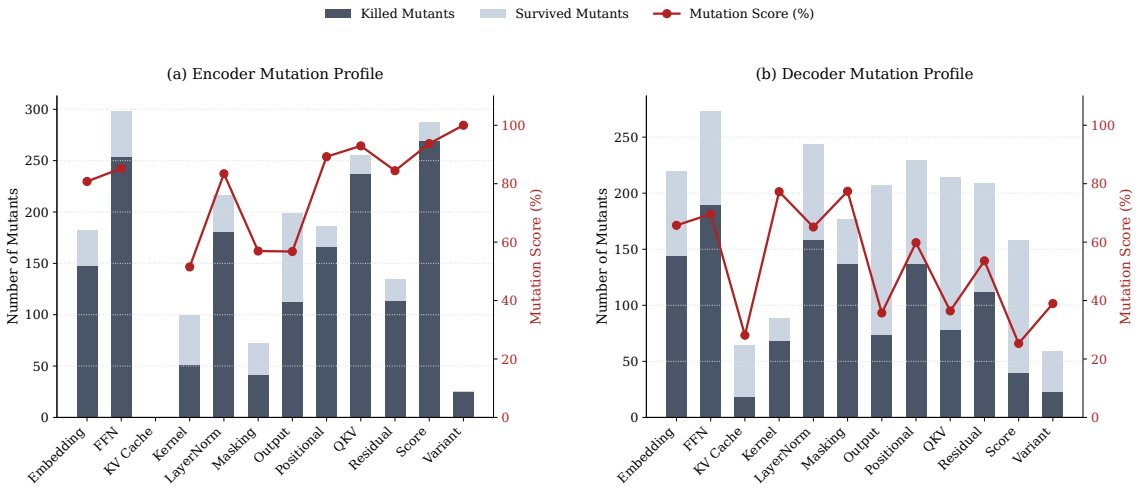


Fig. 6. Per-category mutation scores for encoder and decoder mutants in DEFault-bench

Each instance carries labels at three levels. The Level 1 label for fault detection is binary, either *faulty* or *correct*. Faulty instances also carry Level 2 and Level 3 labels. The Level 2 label is the injected fault category. Encoders use 11 categories, namely Score, Positional, FFN, QKV, LayerNorm, Residual, Masking, Embedding, Kernel, Output, and Variant, and decoders use the same 11 plus KV Cache. The Level 3 label is the injected root cause within the fault category. Each category contains 2–5 root causes, and each root cause belongs to exactly one category.

7 DEFault++ : Hierarchical Diagnostic Technique

DEFault++ takes the 5,556 labeled instances from DEFault-bench as input, each with the runtime measurements collected during training, and diagnoses them through the three-level hierarchy in Fig. 7. Level 1 uses the full feature representation to detect whether an instance is faulty. Only instances predicted as faulty pass to Level 2, which assigns a fault category. Level 3 then predicts the root cause within that category by comparing the instance to learned class prototypes, in an embedding space organized by the Fault Propagation Graph (FPG), the structural prior over transformer components defined in Section 7.1. The same embedding space produces the feature-group importance scores that explain each diagnosis.

7.1 Fault Propagation Graph (FPG)

The fault categories and root causes in our benchmark identify *where* a fault originates. They do not, however, explain how the effects of a fault can propagate to other transformer components. For example, a QKV projection fault can change the projection output and then affect attention scores, attention weights, and the residual stream. To represent

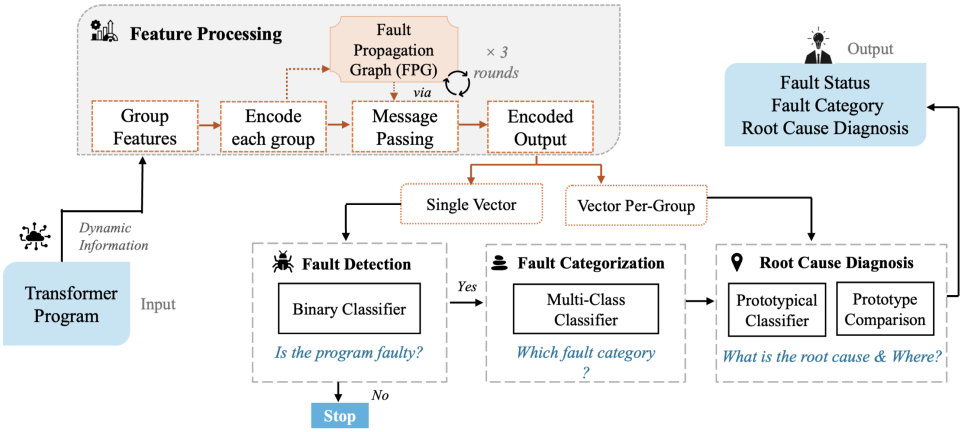


Fig. 7. Overview of DEFault++ for fault detection, categorization, and root-cause diagnosis

these dependency paths, we define the *Fault Propagation Graph (FPG)*, a directed graph whose nodes are transformer components and whose edges represent data-flow dependencies (Fig. 8). The FPG is deterministic because it follows the transformer computation graph. Each edge marks a possible path through which a fault can affect another component. The magnitude and representation of that effect can still change as it passes through learned transformations and operations such as softmax, LayerNorm, and activation functions.

We define the FPG as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ represents a transformer component. Residual connections and LayerNorm appear separately for the attention and FFN sublayers. Decoder architectures also include a KV cache node. Each directed edge $(v_i, v_j) \in \mathcal{E}$ marks a dependency derived from the forward equations, residual structure, or autoregressive state reuse.

We derive the edges by analyzing the transformer's forward and backward equations, which yield seven dependency mechanisms (Fig. 8). We write δx for a small change to a variable x, and each mechanism below describes how such a change moves from one component to another.

(1) **M1: Forward sequential propagation.** If $B = f(A)$, then a change in A can change B according to $\delta B = (\partial f / \partial A)\, \delta A$. This mechanism captures the main forward path through embeddings, attention computation, residual connections, LayerNorm, FFN, and the output head.

(2) **M2: Simultaneous propagation.** QKV projections feed both score computation (via Q, K) and attention output (via V). A single projection fault can therefore affect scores and values at the same time. In decoders, K and V also enter the cache.

(3) **M3: Residual bypass.** In $h = x + \mathrm{MHA}(x)$, the skip path carries changes in x directly to the residual output. Later normalization steps can still adjust the magnitude or representation of the propagated change.

(4) **M4: Cross-layer propagation.** The residual stream provides repeated identity paths across stacked layers [72]. A change at layer ℓ can therefore reach later representations, although intermediate normalization and nonlinear operations may reduce or reshape it.

(5) **M5: Backward pass propagation.** A fault that changes the loss $\mathcal{L}$ can change $\partial \mathcal{L} / \partial \theta_i$ for parameters whose computation contributes to that loss, which can affect weight updates across multiple components during training.

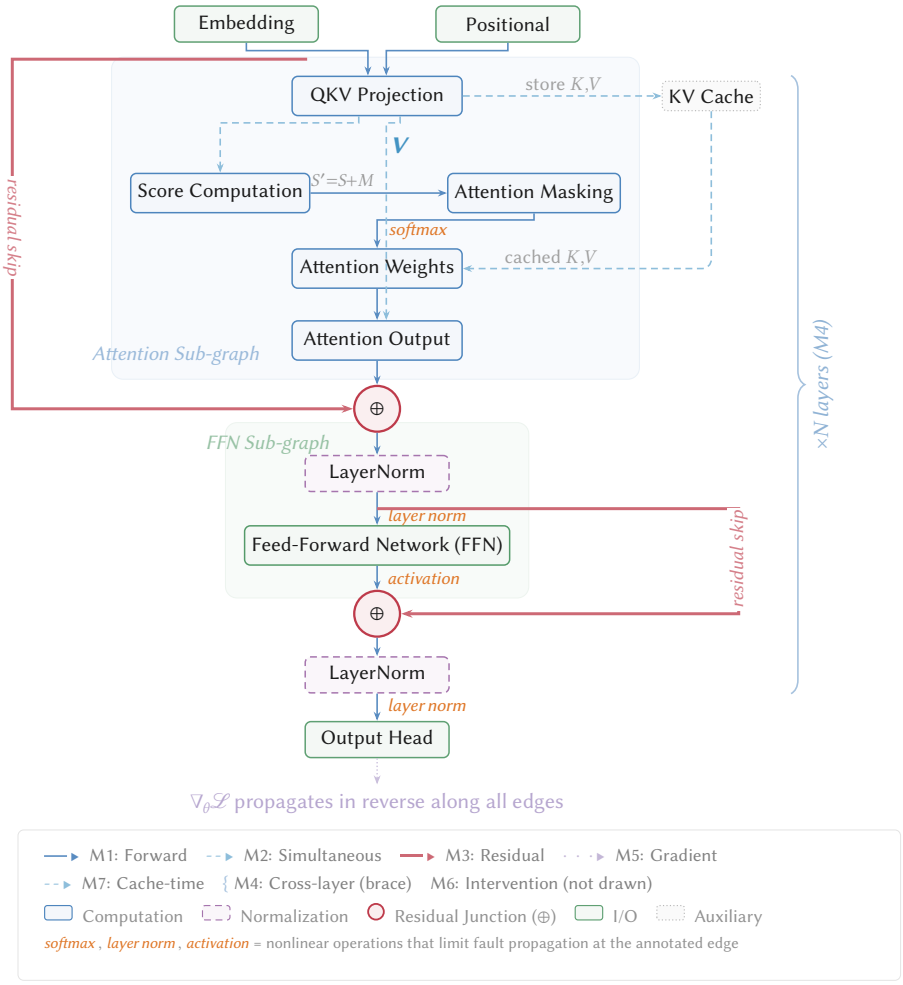


Fig. 8. Fault Propagation Graph (FPG) for transformer architectures

(6) **M6: Architecture-wide intervention.** Some fault types affect multiple components at once. A variant fault (e.g., single-head instead of multi-head attention) changes the attention computation pattern, parameter shapes, and kernel dispatch. This mechanism is a joint structural effect rather than a propagation relation.

(7) **M7: Cache propagation across time steps.** In decoder architectures, the KV cache stores K and V across generation steps. A cache fault can affect the current token and future tokens.

Three nonlinear operations limit propagation along specific edges. Softmax maps changes into a normalized distribution and reduces large shifts. LayerNorm [3] rescales activations by their mean and variance. Activation functions such as ReLU and GELU suppress or pass changes depending on their operating region [23, 82]. We annotate these limiting operations on the corresponding FPG edges. The seven mechanisms fall into three classes by how the diagnostic model uses each one, namely *propagation relations* that become graph edges, *recursive composition* that repeats across stacked layers, and a *structural intervention* that the model reads from the fault labels rather than from edges (Table 6). The propagation relations (M1, M2, M3, M5, M7) describe paths through which a change reaches another component, and

the message-passing graph (Section 7.3) uses the forward relations (M1, M2, M3, M7) as edges while representing the backward relation (M5) through gradient features. Recursive composition (M4) repeats M1 and M3 across stacked layers, and the structural intervention (M6) is a joint effect in which one architectural change alters several components at once.

Table 6. Edge derivation for the Fault Propagation Graph (FPG)

Source	Target	Mechanism Class	Scope	Details
Embedding	QKV Projection	Forward prop. (M1)	Enc/Dec	$Q, K, V = W_{Q,K,V}\, E[x]$
Positional	QKV Projection	Forward prop. (M1)	Enc/Dec	Position added to embedding before projection
QKV Proj.	Score Comp.	Simultaneous (M2)	Enc/Dec	$S = QK^\top/\sqrt{d_k}$; Q, K from projection
QKV Proj.	Attn. Output	Simultaneous (M2)	Enc/Dec	Attn = softmax$(S)\ V$; V from projection
QKV Proj.	KV Cache	Simultaneous (M2)	Dec	K, V stored in cache at each step
Score Comp.	Attn. Mask	Forward prop. (M1)	Enc/Dec	$S' = S + M$
Attn. Mask	Attn. Weights	Forward prop. (M1)	Enc/Dec	$\alpha = \text{softmax}(S')$
Attn. Weights	Attn. Output	Forward prop. (M1)	Enc/Dec	Attn $= \alpha\, V$
Attn. Output	Residual (attn)	Forward prop. (M1)	Enc/Dec	$h = x + \text{Attn}(x)$
Residual input	Residual (attn)	Residual bypass (M3)	Enc/Dec	Skip connection: δx passes with unit gain
Residual (attn)	LayerNorm (attn)	Forward prop. (M1)	Enc/Dec	$\hat{x} = \text{LN}(h)$
LayerNorm (attn)	FFN	Forward prop. (M1)	Enc/Dec	FFN input is post-norm representation
FFN	Residual (FFN)	Forward prop. (M1)	Enc/Dec	$h' = h + \text{FFN}(\hat{x})$
Residual input	Residual (FFN)	Residual bypass (M3)	Enc/Dec	Skip connection: δh passes with unit gain
Residual (FFN)	LayerNorm (FFN)	Forward prop. (M1)	Enc/Dec	$\hat{h}' = \text{LN}(h')$
LayerNorm (FFN)	Output Head	Forward prop. (M1)	Enc/Dec	Final representation enters output head
Layer ℓ residual	Layer ℓ+1	Cross-layer (M4)	Enc/Dec	Repeated M1+M3 across stacked layers
n/a	n/a	Backward prop. (M5)	Enc/Dec	Represented through gradient features
KV Cache	Attn. Weights	Cache-time (M7)	Dec	Cached K, V affect future-step attention
n/a	n/a	Intervention (M6)	Enc/Dec	Joint multi-component effect, not represented as a propagation edge

7.2 Feature Representation

We represent each training run with features that capture component-level behavior. The FPG identifies the component paths along which a fault can propagate, and we use this structure to organize the collected metrics. Architecture-agnostic DNN metrics, such as loss trajectories, accuracy, and gradient statistics, are often too broad to separate transformer-specific fault categories. A fault in a QKV projection, a mask, or a positional encoding may barely move accuracy or perplexity while clearly changing attention-level and component-level measurements. We therefore collect metrics for attention distributions, projection alignment, residual-stream behavior, optimization behavior, training dynamics, and validation behavior, and we organize them into *feature groups*, where each feature group holds the metrics for one transformer component or one model-wide aspect of training (Fig. 9, Table 7).

By how they are measured, these feature groups fall into four families, and Table 7 lists the metric types in each group, where n_g is the number of metric types in group g before aggregation. *Layer-internal* metrics capture attention and component behavior inside transformer layers, since attention distributions [9, 80], representation geometry [17, 42], normalization parameters [3, 92], and residual propagation [72] can each vary across layers. They total $C_{int} = 15$ metric types for encoders and $C_{int} = 16$ for decoders, where the decoder adds future attention mass. *Gradient* metrics total $C_{opt} = 21$ types and record gradient norm, update ratio, and update activity across parameterized components and global gradients [7, 35, 86, 97]. *Behavioral* metrics total $C_{train} = 10$ types for encoders and 12 for decoders and capture output-head behavior, positional sensitivity, step time, peak memory allocation, and decoder cache behavior. *Validation* metrics total $C_{eval} = 2$ types and capture task accuracy or perplexity and calibration error at validation checkpoints [22].

Training runs can differ in model depth, number of steps, and number of epochs. We therefore aggregate the raw measurements at the layer, epoch, and training-phase levels, as shown in Fig. 9. Layer aggregation reduces each per-layer metric to $A_\ell = 5$ statistics, preserving early, middle, and final-layer behavior while removing dependence on model

Table 7. Feature groups, metric types, and pre-aggregation feature counts n_g in DEFault++. Here n_g is the number of scalar metric types in a group before aggregation. The attention group has six metric types for encoders and seven for decoders, since future attention mass applies only to decoders, and gradient metrics are routed to their component groups.

Feature Group	Metric Types	n_g
	Layer-internal metrics (C_{int}, collected at each training step, per layer)	
Attention	Attention entropy, padding attention mass, inter-head cosine similarity, head usage, attention rank, cross-example leakage, future attention mass (decoder only)	6/7
Score	Pre-softmax score	1
FFN output	FFN output norm	1
LayerNorm	Scale parameter norm, post-norm distribution	2
Residual stream	Residual cosine similarity	1
Repr. drift	CKA layer similarity	1
QKV alignment	Q–K sim., Q–V sim., K–V sim.	3
	Gradient metrics (C_{opt}, collected at each training step)	
Optimization	Gradient and update features (gradient norm, update ratio, update activity)	21
	Behavioral metrics (C_{train}, collected at each training step)	
Embedding	Embedding norm, token-level variance	2
Positional	Positional sensitivity	1
Training dyn.	Loss trajectory, gradient norm volatility, step time, peak memory allocation	4
Output	Prediction confidence, output entropy, margin statistics	3
Cache	Cache hidden similarity, cache distribution divergence	2
	Validation metrics (C_{eval}, collected at epoch end or validation checkpoints)	
Validation perf.	Task accuracy/perplexity, calibration error	2

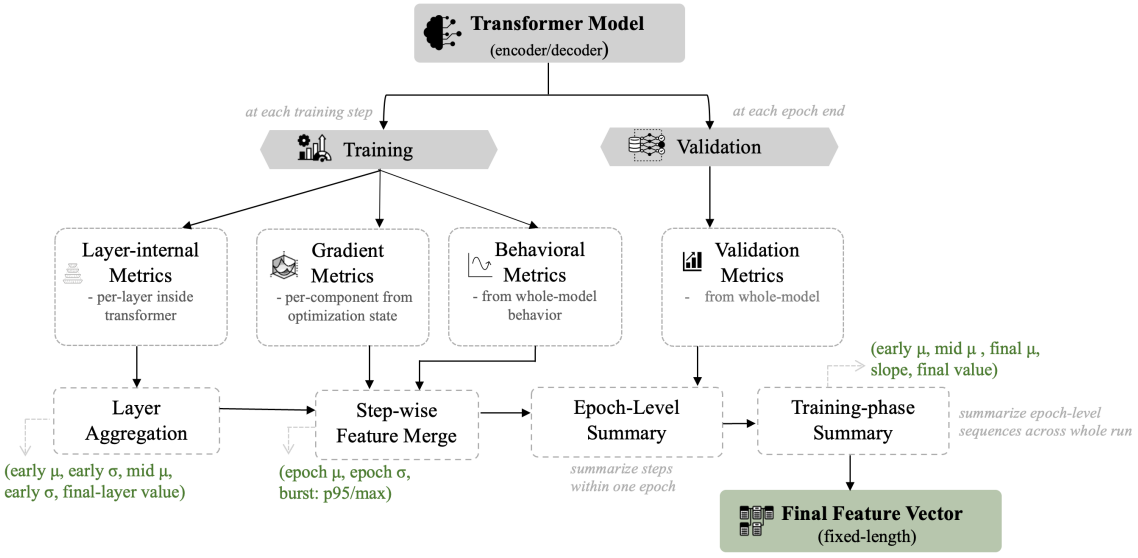


Fig. 9. DEFault++ feature construction flow.

depth. Epoch-level aggregation reduces the step-level sequence within each epoch to $A_e = 3$ statistics. Training-phase aggregation reduces the epoch sequence to $A_p = 5$ statistics. This aggregation produces one feature vector per training run while retaining information about the affected component and the training phase in which the effect appears. Combining the per-family metric-type counts with these aggregation factors gives the length of the final feature vector,

$$d_{\text{final}} = A_p \left(A_e \left(A_t \, C_{int} + C_{opt} + C_{train} \right) + C_{eval} \right), \tag{6}$$

Eq. (6) gives 1,600 features for encoders ($C_{int} = 15$, $C_{opt} = 21$, $C_{train} = 10$, $C_{eval} = 2$) and 1,705 features for decoders, where $C_{int} = 16$ adds future attention mass and $C_{train} = 12$ adds two cache metrics. We remove near-constant columns using coefficient-of-variation filtering (CV < 0.01). DEFault++ uses the filtered fixed-length feature vector as input to the diagnostic model.

7.3 Diagnostic Model

DEFault++ performs hierarchical diagnosis for one input instance at a time. Level 1 detects whether the instance is faulty. If Level 1 predicts a fault, Level 2 assigns the instance to a fault category. Level 3 then predicts the root cause within that category and computes feature-group importance scores for the diagnosis.

Feature-Group Encoding. DEFault++ first encodes each feature group (Table 7) with a dedicated multilayer perceptron (MLP), which keeps the component-level organization of the measurements. Encoders and decoders have different numbers of feature groups, $G = 12$ for encoders and $G = 13$ for decoders, since the KV cache group applies only to decoders, so we train a separate diagnostic model for each architecture.

Given a feature vector $\mathbf{z} \in \mathbb{R}^d$ partitioned into G groups $\{\mathbf{z}_{g_1}, \ldots, \mathbf{z}_{g_G}\}$ (see Table 7), each group g is encoded as

$$\mathbf{h}_g = \text{MLP}_g(\mathbf{z}_g) \in \mathbb{R}^h, \quad g = 1, \ldots, G, \tag{7}$$

where $h = 32$ is the hidden dimension per group, selected by grid search on the inner validation fold. The group embeddings form the matrix $\mathbf{H} = [\mathbf{h}_1; \ldots; \mathbf{h}_G] \in \mathbb{R}^{G \times h}$.

A learnable projection matrix W_{proj} then maps the concatenated group embeddings to a single shared representation,

$$\mathbf{z}_{\text{proj}} = W_{\text{proj}} \, \text{vec}(\mathbf{H}) \in \mathbb{R}^e, \tag{8}$$

where $e = 64$ is the embedding dimension and $\text{vec}(\mathbf{H}) \in \mathbb{R}^{Gh}$ denotes the row-major vectorization of $\mathbf{H}$. Levels 1 and 2 use this shared vector $\mathbf{z}_{\text{proj}}$ for fault detection and fault categorization, while Level 3 uses the group embedding matrix $\mathbf{H}$ to preserve group-level structure for root-cause diagnosis and explanation.

Message Passing in the Fault Propagation Graph (FPG). We use the FPG to exchange information between related feature groups. Each round updates a feature group's embedding using its current value and the embeddings of the feature groups connected to it by FPG edges. After several rounds, a group embedding includes evidence from nearby groups in the FPG. For example, the QKV alignment group can aggregate evidence from the score, attention, and residual-stream groups, which lie on the propagation paths of QKV projection faults.

The group-level graph uses the forward propagation mechanisms defined in the FPG. Backward effects are captured by gradient-based feature groups instead of graph edges. We construct the adjacency matrix $\mathbf{A} \in \mathbb{R}^{G \times G}$ by mapping FPG components to their corresponding feature groups (Table 8). A group-level edge exists when any component in the source group has an FPG edge to any component in the target group. Components mapped to the same feature group

are merged, and groups that cover multiple components inherit the edges of those components. Representation drift, Training dynamics, and Validation performance do not correspond to individual transformer components, so they use self-loops only.

Table 8. Feature-group mapping for FPG message passing in DEFault++

Feature Group	Role	Scope	Message Passing
Attention	Structural	Attention masking, weights, output	FPG edges
Score	Structural	Score computation	FPG edges
FFN output	Structural	FFN	FPG edges
LayerNorm	Structural	LayerNorm	FPG edges
Residual stream	Structural	Residual connections	FPG edges
QKV alignment	Structural	QKV projection	FPG edges
Embedding	Structural	Embedding	FPG edges
Positional	Structural	Positional encoding	FPG edges
Output	Structural	Output head	FPG edges
Cache (decoder)	Structural	KV cache	FPG edges
Representation drift	Cross-layer observation	Inter-layer residual path	Self-loop only
Training dynamics	Model-wide context	Model-wide optimization	Self-loop only
Validation performance	Model-wide context	Model-wide output quality	Self-loop only

For the decoder diagnostic model ($G = 13$), Fig. 10 shows the row-normalized adjacency matrix with self-loops, $\hat{\mathbf{A}}$. Nonzero values near the main diagonal follow the sequential data flow through the transformer block. Off-diagonal values capture branching paths such as QKV→Cache, Cache→Attention, and FFN→Residual. Representation drift, Training dynamics, and Validation performance appear as isolated self-loops, since they provide model-wide context rather than lying on a component-level propagation path.

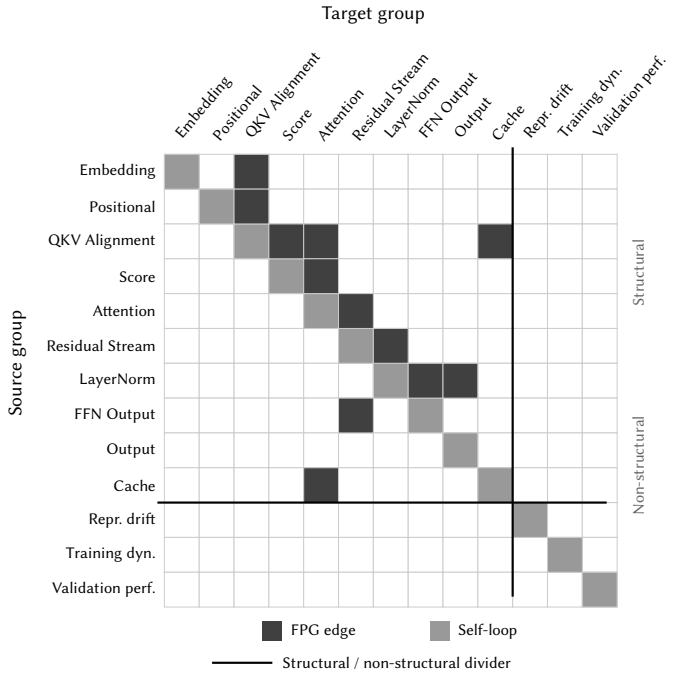


Fig. 10. Group-level adjacency matrix $\hat{\mathbf{A}}$ for the decoder diagnostic model

Each round mixes every group embedding with those of its FPG neighbors and then applies a learned transformation. Following graph convolutional networks [40], one round of message passing is

$$\mathbf{H}' = \mathrm{ReLU}\left(\hat{\mathbf{A}}\mathbf{H}W_{\mathrm{msg}}\right), \tag{9}$$

where $\hat{\mathbf{A}}$ is the row-normalized adjacency matrix with self-loops and $W_{\mathrm{msg}} \in \mathbb{R}^{h\times h}$ is a learnable weight matrix. We use row normalization because feature groups have different numbers of FPG neighbors, and it keeps the message-passing updates on a comparable scale across groups.

We apply three rounds of message passing, with a separate W_{msg} for each round. We select the number of rounds by grid search over $\{1, \ldots, 5\}$ on the inner validation fold. Three rounds cover short FPG paths, such as QKV → Score → Attention → Residual, while avoiding deeper aggregation that can oversmooth node representations [44]. DEFault++ uses the updated group embeddings in the following diagnosis stages.

Hierarchical Classification. DEFault++ uses a shared encoder across the three diagnosis levels. The encoder combines feature-group encoding, FPG message passing, and projection into a common representation. Each diagnosis level then applies its own prediction head to this representation. Sharing the encoder avoids separate representations for closely related tasks and keeps the model small, since message passing operates over only 12 feature groups for encoders and 13 for decoders.

• *Level 1: Fault detection.* The first level performs binary detection. A two-layer MLP maps $\mathbf{z}_{\mathrm{proj}}$ to logits for faulty and correct instances.

• *Level 2: Fault categorization.* Instances predicted as faulty are passed to the second level, which assigns them to a fault category. A two-layer MLP maps $\mathbf{z}_{\mathrm{proj}}$ to C category logits, where $C = 11$ for encoders and $C = 12$ for decoders since the KV Cache category applies only to decoders.

• *Level 3: Root-cause diagnosis.* The third level predicts the root cause within the fault category selected by Level 2. Since each category contains only 2–5 root causes and per-root-cause support is limited, DEFault++ uses a *prototypical classifier* [37, 75], which represents each root cause by a single point in the embedding space, its *prototype*, and classifies an instance by its distance to those points. A prototype is the mean group embedding of the training examples for a root cause, and at inference DEFault++ assigns the instance to the nearest root-cause prototype within the predicted category.

Let $\mathcal{D}_{c,r}$ denote the training samples with fault category c and root cause r. For each root cause r within category c, we compute the prototype $\boldsymbol{\pi}_{c,r}$ as the mean group embedding:

$$\boldsymbol{\pi}_{c,r} = \frac{1}{|\mathcal{D}_{c,r}|} \sum_{i\in\mathcal{D}_{c,r}} \mathbf{H}_i \in \mathbb{R}^{G\times h}. \tag{10}$$

DEFault++ compares an input instance with the prototypes from the predicted category. The squared distance to a prototype decomposes over feature groups:

$$d(\mathbf{H}, \boldsymbol{\pi}_{c,r}) = \sum_{g=1}^{G} \|\mathbf{h}_g - \boldsymbol{\pi}_{c,r,g}\|^2 = \sum_{g=1}^{G} d_g, \tag{11}$$

where $d_g = \|\mathbf{h}_g - \boldsymbol{\pi}_{c,r,g}\|^2$ is the contribution of feature group g. DEFault++ predicts the nearest root-cause prototype:

$$\hat{r} = \arg\min_{r\in\mathcal{R}_{\hat{c}}} d(\mathbf{H}, \boldsymbol{\pi}_{\hat{c},r}).$$

The group-wise decomposition of the prototype distance allows DEFault++ to estimate each feature group's contribution to the root-cause decision.

Algorithm 2 DEFault++ hierarchical diagnosis for one input instance

Input: Feature vector $\mathbf{x}$; shared encoder (group embeddings $\{\mathbf{h}_g\}_{g=1}^{G}$ and projection $\mathbf{z}_{\text{proj}}$); trained prototypes $\{\boldsymbol{\pi}_{c,r}\}$
Output: Detection label $\hat{y}$; if faulty: fault category $\hat{c}$, root cause $\hat{r}$, and per-group importance scores $\{w_g\}_{g=1}^{G}$
▷ *Level 1: Fault Detection*
Classify the instance as faulty or correct: $\hat{y} \leftarrow \text{Detect}(\mathbf{z}_{\text{proj}})$
if $\hat{y} =$ *correct* **then**
  return $\hat{y}$ ▷ correct instances exit here;
end if
▷ *Level 2: Fault Categorization*
Assign the fault to a transformer component category: $\hat{c} \leftarrow \text{Categorize}(\mathbf{z}_{\text{proj}})$
▷ *Level 3: Root-Cause Diagnosis*
for each root cause r in the predicted category $\mathcal{R}_{\hat{c}}$ **do**
  Sum squared distances to prototype r across all feature groups: $d_r \leftarrow \sum_{g=1}^{G} \|\mathbf{h}_g - \boldsymbol{\pi}_{\hat{c},r,g}\|^2$
end for
Select the nearest prototype as the predicted root cause: $\hat{r} \leftarrow \arg\min_{r \in \mathcal{R}_{\hat{c}}} d_r$
▷ *Level 3: Root-Cause Explanation*
Identify the nearest alternative root cause: $\tilde{r} \leftarrow \arg\min_{r \in \mathcal{R}_{\hat{c}} \setminus \{\hat{r}\}} d_r$
for each feature group $g = 1, \ldots, G$ **do**
  Compute how much group g favors $\hat{r}$ over $\tilde{r}$: $\Delta_g \leftarrow d_g(\boldsymbol{\pi}_{\hat{c},\tilde{r}}) - d_g(\boldsymbol{\pi}_{\hat{c},\hat{r}})$
  Normalize positive contributions into importance score: $w_g \leftarrow \max(\Delta_g, 0) \;/\; \sum_{g'} \max(\Delta_{g'}, 0)$
end for
return $\hat{y},\ \hat{c},\ \hat{r},\ \{w_g\}_{g=1}^{G}$

7.4 Training Objective

DEFault++ trains the three diagnosis levels jointly with a shared encoder. During training, the detection loss, categorization loss, and root-cause diagnosis losses all update the shared encoder.

The training objective has four components. Level 1 uses binary cross-entropy, $\mathcal{L}_{\text{detect}}$, over all samples in the batch. Level 2 uses class-weighted cross-entropy, $\mathcal{L}_{\text{cat}}$, over fault categories and applies it only to faulty samples. Level 3 uses two losses, a per-category cross-entropy loss $\mathcal{L}_{\text{rc}}$ for root-cause prediction and a separation loss $\mathcal{L}_{\text{sep}}$ for prototype-based diagnosis.

For root-cause training, DEFault++ learns root-cause distinctions separately within each fault category. For each fault category, DEFault++ uses a separate root-cause head. The head takes $\mathbf{z}_{\text{proj}}$ as input and predicts among the root causes in that category. For example, QKV root causes are compared with other QKV root causes, not with LayerNorm or FFN root causes. At inference, the model first predicts a fault category and then diagnoses the root cause within that category. We compute $\mathcal{L}_{\text{rc}}$ only over faulty samples within their ground-truth fault category.

The cross-entropy loss used for root-cause diagnosis trains the per-category heads, but prototype-based diagnosis also depends on the group embedding space $\mathbf{H}$. Root causes from the same fault category may still have similar group embeddings, which can make nearest-prototype diagnosis less reliable. DEFault++ therefore uses $\mathcal{L}_{\text{sep}}$ to separate root causes in the group embedding space. The separation loss combines supervised contrastive learning [38] with prototype matching [75]. The contrastive term separates embeddings of different root causes within the same category, while the prototype-matching term moves each sample closer to its own root-cause prototype.

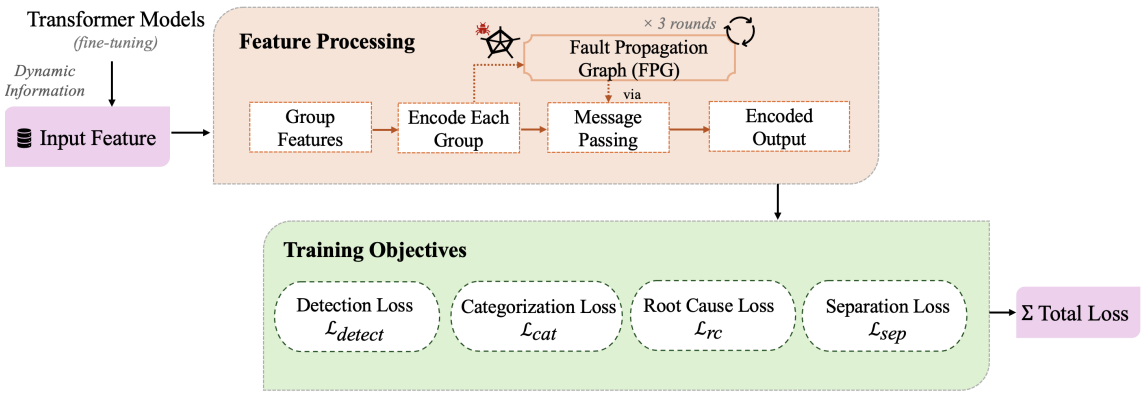


Fig. 11. Training view of DEFault++

The contrastive term uses one vector per training sample, computed by flattening the group embedding matrix: $\mathbf{u}_i = \mathrm{vec}(\mathbf{H}_i)$. For a training sample i, the comparison set is restricted to samples from the same fault category. Within this set, positives are samples with the same root cause as i, and negatives are samples with a different root cause. The contrastive term therefore pulls together embeddings from the same root cause and separates embeddings from different root causes within the same category. Following supervised contrastive learning [38], the contrastive term for category c is

$$\mathcal{L}_{\mathrm{ctr}}^{(c)} = -\frac{1}{|\mathcal{P}_c|} \sum_{(i,j)\in\mathcal{P}_c} \log \frac{\exp(\mathrm{sim}(\mathbf{u}_i, \mathbf{u}_j)/\tau)}{\sum_{\substack{k\neq i \\ c_k = c_i}} \exp(\mathrm{sim}(\mathbf{u}_i, \mathbf{u}_k)/\tau)} \tag{12}$$

where $\mathcal{P}_c$ is the set of positive pairs within category c, $\mathrm{sim}(\cdot,\cdot)$ is cosine similarity, and $\tau = 0.1$ is the temperature. The denominator excludes sample i itself and sums over the other samples in the batch that share the same fault category. We compute $\mathcal{L}_{\mathrm{ctr}}$ by averaging $\mathcal{L}_{\mathrm{ctr}}^{(c)}$ over categories with at least one positive pair in the batch.

The prototype-matching term uses the grouped squared distance from Eq. (11). It encourages each sample to lie closer to its own root-cause prototype than to other prototypes in the same category:

$$\mathcal{L}_{\mathrm{pm}} = -\frac{1}{|\mathcal{B}|} \sum_{i\in\mathcal{B}} \log \frac{\exp(-d(\mathbf{H}_i, \boldsymbol{\pi}_{c_i,r_i})/\tau_p)}{\sum_{r'\in\mathcal{R}_{c_i}} \exp(-d(\mathbf{H}_i, \boldsymbol{\pi}_{c_i,r'})/\tau_p)} \tag{13}$$

where $\mathcal{B}$ is the training batch, $\mathcal{R}_{c_i}$ is the set of root causes within category c_i, and τ_p is a temperature parameter.

The separation loss combines these two terms:

$$\mathcal{L}_{\mathrm{sep}} = \beta\,\mathcal{L}_{\mathrm{ctr}} + \gamma\,\mathcal{L}_{\mathrm{pm}}. \tag{14}$$

Finally, the full DEFault++ objective is

$$\mathcal{L}_{\mathrm{DEFault++}} = \mathcal{L}_{\mathrm{detect}} + \alpha\,\mathcal{L}_{\mathrm{cat}} + \lambda\,\mathcal{L}_{\mathrm{rc}} + \mathcal{L}_{\mathrm{sep}}. \tag{15}$$

We apply $\mathcal{L}_{\mathrm{sep}}$ only to faulty samples within their ground-truth category. We set $\alpha = 1.0$, $\lambda = 1.0$, $\beta = 0.5$, and $\gamma = 0.3$ through grid search.

7.5 FPG-based Explanation

Level 3 predicts the root cause by selecting the nearest prototype in the grouped embedding space. To explain this prediction, DEFault++ compares the predicted prototype $\boldsymbol{\pi}_{\hat{c},\hat{r}}$ with the nearest alternative prototype $\boldsymbol{\pi}_{\hat{c},\tilde{r}}$ in the predicted

fault category:

$$\tilde{r} = \arg \min_{r \in \mathcal{R}_{\hat{c}} \setminus \{\hat{r}\}} d(\mathbf{H}, \boldsymbol{\pi}_{\hat{c},r}).$$

For each feature group g, DEFault++ computes how much closer the group embedding is to the predicted prototype than to the nearest competing prototype:

$$\Delta_g = d_g(\boldsymbol{\pi}_{\hat{c},\tilde{r}}) - d_g(\boldsymbol{\pi}_{\hat{c},\hat{r}}), \quad g = 1, \ldots, G, \tag{16}$$

where $d_g(\boldsymbol{\pi}_{\hat{c},r}) = \|\mathbf{h}_g - \boldsymbol{\pi}_{\hat{c},r,g}\|^2$. A positive Δ_g means that feature group g supports the predicted root cause over the nearest competing root cause. A negative Δ_g means that the group is closer to the competing prototype. Since the full prototype distance is the sum of group-level distances, $\sum_g \Delta_g$ equals the distance gap between the predicted prototype and the nearest competing prototype.

DEFault++ normalizes the positive group contributions into importance scores:

$$w_g = \frac{\max(\Delta_g, 0)}{\sum_{g'=1}^{G} \max(\Delta_{g'}, 0)}, \quad g = 1, \ldots, G. \tag{17}$$

Groups with negative contributions receive $w_g = 0$. The importance scores sum to one and identify the feature groups that most support the predicted root cause.

For structural groups, which correspond to transformer components (e.g., attention, QKV, FFN), each w_g maps to a transformer subsystem. A high score therefore identifies the subsystem whose message-passed embedding most supports the diagnosis [90]. For the cross-layer observation group (e.g., representation drift), a high score indicates that cross-layer behavior contributes to the diagnosis. For model-wide context groups (e.g., training dynamics, validation performance), a high score means that training or validation behavior helped distinguish the predicted root cause. These non-structural groups provide supporting evidence. They should not be interpreted as the faulty component.

8 Evaluation

We evaluate DEFault++ on the benchmark (i.e., DEFault-bench) to assess its effectiveness across fault detection, fault categorization, root-cause diagnosis, and feature-group evidence. We report encoder and decoder results separately because the two architectures use different feature groups and label spaces.

8.1 Evaluation Metrics

We evaluate DEFault++ using precision, recall, and F1. At Level 1, the faulty and correct classes are balanced by construction, so we report binary precision, recall, and F1. At Levels 2 and 3, fault categories and root causes are imbalanced, so we report macro-averaged precision, recall, and F1 across classes [59]. We do not report accuracy because it can be misleading under class imbalance [6, 59]. We define true positives according to the hierarchy. Level 1 requires correct fault detection. Level 2 requires the correct fault category. Level 3 counts a prediction as correct only when both the predicted category and the predicted root cause are correct.

8.2 Baselines

We compare DEFault++ with four prior fault-detection techniques for DNNs, namely AutoTrainer [97], DeepDiagnosis [84], DeepFD [7], and DEFault [35]. These baselines distinguish faulty runs from correct runs. They do not define fault labels that match our transformer fault categories or root causes, so we limit the comparison to the fault-detection level. For the rule-based baselines, AutoTrainer and DeepDiagnosis, we use their published rules and thresholds and

map the required observables to the closest available trace features in DEFault-bench. AutoTrainer's gradient-threshold rules map to the Training dynamics group, and DeepDiagnosis includes rules that partially overlap with the Attention and Score groups. For the learning-based baselines, DeepFD and DEFault, we train the original model types from their replication packages on the fault-detection task [7, 35].

8.3 Implementation

We implement DEFault++ in PyTorch [64]. Each feature group is encoded by a separate MLP with hidden dimension $h = 32$, the shared projection maps the concatenated group embeddings to $e = 64$ dimensions, and DEFault++ applies three rounds of FPG message passing over the group-level adjacency matrix. Training uses Adam [39] with learning rate 10^{-3}, weight decay 10^{-4}, and batch size 256. We train for up to 150 epochs with early stopping on the validation split. We set the contrastive temperature to $\tau = 0.1$ and tune the loss weights by grid search. We address fault-category and root-cause imbalance using inverse-frequency weighting in the corresponding classification losses. We use a fixed set of random seeds for reproducibility. The replication package contains the implementation, hyperparameter lists, and trained models [32]. By approximately epoch 100, fault-detection F1 on the validation split shows little further improvement, while fault-categorization F1 continues to improve modestly until early stopping.

8.4 Answering RQ_1: Effectiveness of Diagnosis

To answer RQ_1, we evaluate DEFault++ at its three levels, namely fault detection, fault categorization, and root-cause diagnosis.

Fault Detection. Table 9a shows that DEFault++ achieves F1 of 0.8259 for encoders and 0.9094 for decoders. Recall is also high on both architectures, indicating that most faulty instances are detected before the later diagnosis levels.

Table 9. Overall DEFault++ performance across three levels

(a) Level 1: Fault Detection

Metric	Encoder	Decoder
F1	0.8259	0.9094
Precision	0.8014	0.9125
Recall	0.8520	0.9064

(b) Level 2: Fault Categorization

Metric	Encoder	Decoder
F1	0.8497	0.8679
Precision	0.8662	0.8794
Recall	0.8403	0.8649

(c) Level 3: Root-Cause Diagnosis

Metric	Encoder	Decoder
F1	0.8512	0.8670
Precision	0.9407	0.9475
Recall	0.7932	0.8225

Fault Categorization. Table 9b shows strong fault-categorization performance for both encoders (F1 of 0.8497) and decoders (F1 of 0.8679). Precision and recall stay close in value, which indicates that the result does not come from a strong precision-recall tradeoff. The per-category results in Table 10a show that Residual and Positional faults are among the strongest categories in both settings.

For encoder models, DEFault++ scores lowest on Variant (0.741) and Kernel (0.771). Variant differs sharply across the two families, reaching 0.948 on decoders despite a similar number of instances (25 for encoders and 23 for decoders, Table 5), so the small Variant set does not by itself explain the encoder result. One possible reason is that decoder Variant faults produce cache-related differences that the encoder-only models cannot show. Kernel faults are difficult to categorize because they affect low-level execution behavior, such as numerical precision and backend selection, rather than a specific transformer component [1].

For decoder models, DEFault++ scores lowest on FFN and KV Cache faults. FFN faults are harder to categorize because their effects can appear in downstream components, such as the residual stream and LayerNorm [79]. KV Cache

faults are challenging because the same cache fault can produce different measured behavior across token-generation steps [2, 96].

Table 10. DEFault++ performance for Level 2 and Level 3 diagnosis (fault-category-wise)

(a) Level 2: Fault Categorization (per-category)

	Encoder			Decoder		
Category	**F1**	**Prec.**	**Rec.**	**F1**	**Prec.**	**Rec.**
Embedding	0.838	0.852	0.829	0.862	0.833	0.897
FFN	0.836	0.807	0.871	0.718	0.673	0.777
Kernel	0.771	0.814	0.740	0.765	0.848	0.700
KV Cache	n/a	n/a	n/a	0.758	0.716	0.822
LayerNorm	0.790	0.775	0.820	0.843	0.937	0.766
Masking	0.917	0.952	0.886	0.890	0.927	0.857
Output	0.889	0.927	0.864	0.975	0.980	0.969
Positional	0.919	0.950	0.891	0.927	0.949	0.911
QKV	0.825	0.863	0.792	0.789	0.772	0.815
Residual	0.961	0.965	0.958	0.995	0.995	0.995
Score	0.861	0.847	0.876	0.946	0.937	0.957
Variant	0.741	0.776	0.717	0.948	0.986	0.914
Avg.	0.850	0.866	0.840	0.868	0.879	0.865

(b) Level 3: Root-Cause Diagnosis (per-category)

	Encoder			Decoder		
Category	**F1**	**Prec.**	**Rec.**	**F1**	**Prec.**	**Rec.**
Embedding	0.879	0.972	0.806	0.877	0.972	0.816
FFN	0.804	0.847	0.778	0.883	1.000	0.803
Kernel	0.814	1.000	0.706	0.747	0.988	0.676
KV Cache	n/a	n/a	n/a	0.881	1.000	0.817
LayerNorm	0.821	0.917	0.749	0.890	0.969	0.839
Masking	0.925	0.986	0.874	0.845	0.902	0.801
Output	0.882	0.958	0.822	0.972	1.000	0.947
Positional	0.930	0.992	0.886	0.975	1.000	0.953
QKV	0.765	0.868	0.694	0.646	0.727	0.592
Residual	0.908	0.927	0.895	0.958	0.961	0.955
Score	0.835	0.879	0.799	0.908	0.926	0.897
Variant	0.799	1.000	0.717	0.824	0.925	0.773
Avg.	0.851	0.941	0.793	0.867	0.948	0.823

Root-cause Diagnosis. Table 9c shows that DEFault++ achieves F1 of 0.851 on encoders and 0.867 on decoders for overall root-cause diagnosis. We observe higher precision than recall on both architectures, indicating that incorrect root-cause predictions are less common than missed root-cause cases. The per-category results (Table 10b) show that DEFault++ diagnoses root causes most accurately for Positional and Residual faults. DEFault++ performs lowest when diagnosing root causes within the QKV fault category. Several QKV root causes can change projection alignment, attention scores, and update behavior in similar ways [34]. This overlap may explain why QKV root causes are harder to distinguish. The Level 3 per-category results also indicate that the number of root causes within a fault category alone does not explain the performance. The Residual fault category includes five root causes and remains one of the strongest categories for root-cause diagnosis. QKV includes four root causes but remains the weakest. This pattern suggests that separability within each category plays a larger role than the number of root causes alone.

Summary of RQ_1: DEFault++ diagnoses faults at all three levels on the evaluated encoder and decoder models, with F1 of 0.826–0.909 for detection, 0.850–0.868 for categorization, and 0.851–0.867 for root-cause diagnosis. Decoders score slightly higher than encoders at every level, and Level 3 precision exceeds 0.94 on both.

8.5 Answering RQ_2: Comparison with Existing Techniques

To answer RQ_2, we compare DEFault++ with four existing DNN fault-detection techniques. AutoTrainer [97] and DeepDiagnosis [84] report training-symptom outputs. DeepFD [7] and DEFault [35] use DNN-level fault categories, which do not correspond to the transformer component categories or root causes used in our evaluation. We therefore restrict the comparison to fault detection (Level 1).

The baseline results (Table 11) highlight the difference between general DNN fault detection and transformer-component fault detection. AutoTrainer and DeepDiagnosis use fixed training symptoms (e.g., gradient failures, loss oscillation, overfitting). Many DEFault-bench faults do not necessarily produce these training-level symptoms. A masking, QKV, positional, or cache fault can change internal transformer behavior while the loss curve and numerical values remain within ordinary ranges. As a result, rule-based methods can miss faults that do not match their predefined

symptoms. In contrast, DeepFD and DEFault perform better than the rule-based methods because they learn from runtime features rather than relying on fixed rules. However, they still focus on general DNN training behavior and do not directly capture transformer-specific behavior (e.g., attention distributions, QKV alignment, cache behavior, or component-level propagation). These results suggest that transformer fault detection is not only a model-level prediction task. It also requires component-level measurements and the dependencies among transformer components.

Table 11. Fault-detection comparison between DEFault++ and baselines (F1)

Technique	Type	Encoder	Decoder
DEFault++	Neural hierarchical	**0.8259**	**0.9094**
DEFault [35]	Hierarchical RF	0.5660	0.7020
DeepFD [7]	Flat ensemble	0.5140	0.6550
DeepDiagnosis [84]	Rule-based (8 rules)	0.3380	0.4820
AutoTrainer [97]	Rule-based (5 rules)	0.2840	0.4270

Summary of RQ_2: DEFault++ outperforms all four baselines in detecting faults in transformer models, none of which is designed to capture transformer-component behavior. Our findings suggest that many transformer faults may not be captured by standard runtime metrics alone. Detecting these faults benefits from transformer-specific measurements and architecture-aware diagnosis.

8.6 Answering RQ_3: Ablation

To answer RQ_3, we ablate two major parts of DEFault++ and assess their impact on performance. In particular, we focus on (a) message passing in the Fault Propagation Graph (FPG) and (b) the separation loss $\mathcal{L}_{\text{sep}}$. FPG message passing applies to all three diagnosis levels, while $\mathcal{L}_{\text{sep}}$ applies only to root-cause diagnosis.

We evaluate four model variants. The full DEFault++ model uses both FPG message passing and separation loss. The −FPG variant removes message passing and keeps the remaining model unchanged. The −Sep variant removes the separation loss by setting $\beta = 0$ and $\gamma = 0$. The −FPG −Sep variant removes both components and keeps only the feature-group encoders, shared projection, and prediction heads.

We also evaluate two graph-topology variants to separate the effect of the FPG's transformer-specific structure from the influence of graph connectivity alone. The *Rewired variant* keeps the same number of nodes and edges as the FPG but randomly reassigns edge endpoints while preserving the degree distribution [54]. The *Random variant* replaces the FPG with an Erdős–Rényi graph [15] with approximately the same edge density.

Fault Detection & Categorization. For Level 1 fault detection, removing FPG message passing shows a slight performance decline on both architectures (Table 12a). This result suggests that the feature groups already contain enough information for binary fault detection, while FPG message passing adds a slight advantage by combining evidence from related components.

For Level 2 fault categorization, removing FPG message passing leads to a larger performance drop (Table 12b). Categorization requires the model to distinguish fault types that can affect related parts of the transformer. For example, QKV and Score faults both affect attention behavior, but they reach downstream components through different propagation paths. FPG message passing allows the model to combine evidence from related feature groups before predicting the fault category.

Table 12. Ablation studies for Level 1 fault detection and Level 2 fault categorization.

(a) Level 1 fault detection

Arch.	Variant	F1	Prec.	Rec.
Encoder	DEFault++	0.826	0.801	0.852
	−FPG	0.782	0.752	0.816
Decoder	DEFault++	0.909	0.912	0.906
	−FPG	0.890	0.890	0.891

(b) Level 2 fault categorization

Arch.	Variant	F1	Prec.	Rec.
Encoder	DEFault++	0.850	0.866	0.840
	−FPG	0.742	0.757	0.731
Decoder	DEFault++	0.868	0.879	0.865
	−FPG	0.788	0.797	0.783

Root-cause Diagnosis. At Level 3, removing the separation loss causes a larger performance drop than removing FPG message passing (Table 13). This pattern suggests that the main challenge at this level is distinguishing root causes within the same fault category. Root causes in the same category often affect the same transformer component, which can make their runtime measurements similar. The separation loss directly targets this problem by encouraging embeddings from different root causes in the same category to move apart.

FPG message passing also contributes to Level 3, but its effect is smaller than that of the separation loss. Removing both FPG message passing and the separation loss leads to the lowest Level 3 performance, suggesting that they provide complementary benefits.

Table 13. Ablation study of Level 3 root cause diagnosis

Architecture	Variant	F1	Precision	Recall
Encoder	DEFault++	0.851	0.941	0.793
	−FPG	0.812	0.933	0.776
	−Sep	0.756	0.910	0.752
	−FPG−Sep	0.701	0.883	0.690
Decoder	DEFault++	0.867	0.948	0.823
	−FPG	0.833	0.940	0.809
	−Sep	0.781	0.925	0.781
	−FPG−Sep	0.734	0.896	0.721

Topology Ablation. Table 14 shows that the original FPG gives the best performance at every diagnosis level for both encoders and decoders. The Rewired and Random variants degrade performance, and removing FPG message passing gives the lowest performance. The drop from the original FPG to the Rewired and Random variants shows that transformer-specific propagation links matter beyond graph connectivity alone. We observe that the topology effect is largest at the fault-categorization level. The FPG-removal ablation shows the same Level 2 pattern, with the largest performance drop appearing in fault categorization (Table 12b). Since fault categorization depends more on propagation structure, it must differentiate fault categories that can affect related transformer components. The topology differences are smaller at Level 1 and Level 3. Level 1 is a binary detection task, while Level 3 depends more on within-category root-cause separation through $\mathcal{L}_{\text{sep}}$. These ablations show that the FPG structure helps categorization, but they do not establish that its edges match the trained model's actual fault-propagation paths.

Summary of RQ$_3$: Message passing in the Fault Propagation Graph (FPG) contributes most to DEFault++'s fault-categorization performance, while the separation loss contributes most to root-cause diagnosis. The topology ablation shows that the defined FPG structure is more effective than rewired or random graph variants, especially for fault categorization.

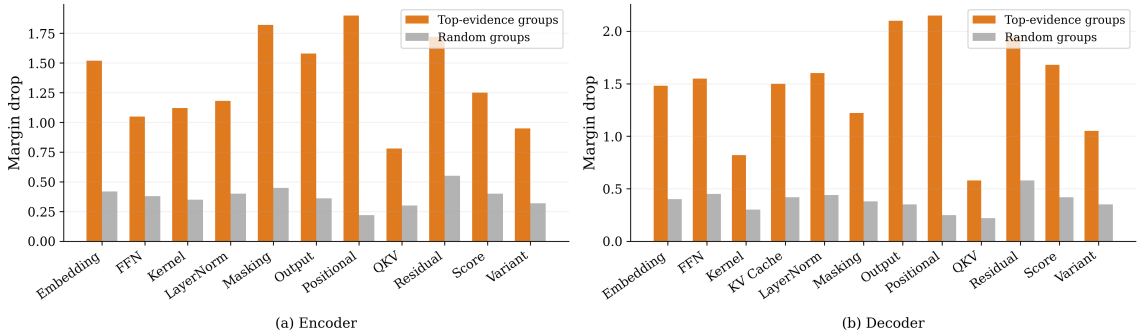
Table 14. Topology ablation results using F1 across graph variants

Level	Architecture	DEFault++	Rewired	Random	−FPG
L1	Encoder	0.826	0.790	0.786	0.782
	Decoder	0.909	0.894	0.892	0.890
L2	Encoder	0.850	0.770	0.754	0.742
	Decoder	0.868	0.807	0.796	0.788
L3	Encoder	0.851	0.820	0.816	0.812
	Decoder	0.867	0.838	0.835	0.833

8.7 Answering RQ$_4$: Explanation Faithfulness

DEFault++ uses the prototype classifier for root-cause diagnosis. The prototype distance decomposes over feature groups, so DEFault++ can measure how much each group contributes to the selected root-cause prototype. To answer RQ$_4$, we assess the faithfulness of these feature-group explanations to DEFault++'s prototype-based root-cause decisions.

We first evaluate the reported feature groups through ablation. For each prediction, we remove the two feature groups that DEFault++ ranks highest. We then recompute the prototype-distance gap between the predicted root-cause prototype and the nearest alternative prototype. The gap is computed as the distance to the nearest alternative prototype minus the distance to the predicted prototype. Larger gaps indicate stronger support for the predicted root cause. As a baseline, we repeat the same test after removing two randomly selected feature groups. Figure 12 shows that removing the top-ranked groups reduces the prototype-distance gap more than removing random groups. A similar pattern appears for both encoder and decoder models, and it applies to all fault categories.



(a) Encoder (b) Decoder

Fig. 12. Group ablation analysis for the FPG-based explanation

We next evaluate the prototype classifier used for the explanation. DEFault++ also trains a supervised root-cause classifier, which predicts root causes directly from the training labels. We compare these two classifiers by measuring how often they predict the same root cause for the same sample. Figure 13 shows that prototype accuracy increases during training, and its agreement with the supervised classifier also increases for both encoder and decoder models. Since DEFault++ computes explanations from the prototype classifier, the increasing agreement supports the use of prototype-based explanations for the learned root-cause decisions.

DEFault-bench provides the root-cause annotation for each faulty run, but it does not specify which feature groups should explain that fault. We therefore assess faithfulness by testing whether the reported groups affect DEFault++'s prototype-based decision. This evaluation supports internal faithfulness, but it does not establish the reported groups as the true causal explanation for the fault.

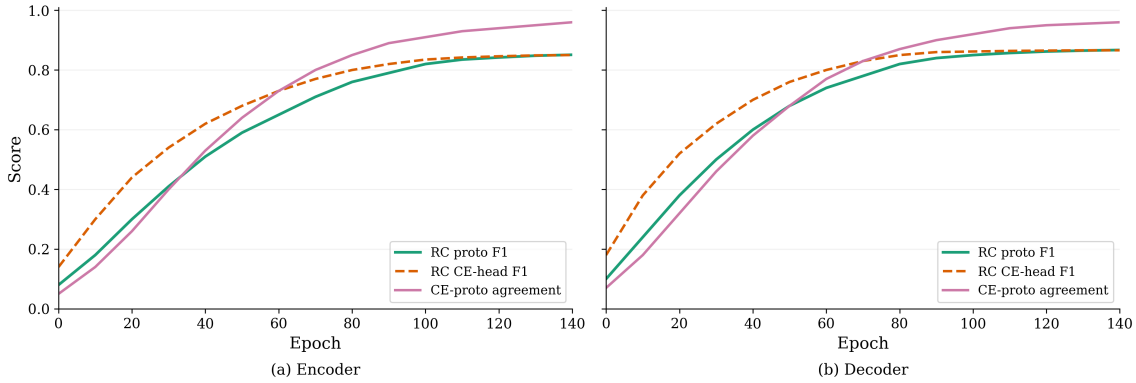


Fig. 13. Prototype-label agreement during training

Summary of RQ$_4$: DEFault++ identifies the feature groups that support its root-cause diagnoses. Our results show that higher-ranked feature groups provide stronger diagnostic evidence than randomly selected groups, and that the explanation mechanism remains aligned with the model's final prediction.

9 Real-world Fault Evaluation

We use real-world GitHub issues to evaluate whether DEFault++ generalizes beyond the synthetic faults in DEFault-bench. Following prior work [34, 56, 86], we collect 20 transformer-related bug-fix issues from open-source repositories and reproduce 11 of them. For each reproduced issue, we compare the faulty version with the fixed version using the same feature extraction procedure as in the main evaluation. DEFault++ detects 8 of the 11 faults, assigns the correct fault category for 8, and identifies the correct root cause for 4 (Table 15). Fault detection (FD) checks whether DEFault++ marks the faulty version as faulty. Fault categorization (FC) checks whether it assigns the fault to the correct transformer category. Root-cause diagnosis (RC) checks whether the predicted cause matches the cause confirmed by the corresponding bug fix.

Table 15. Real-world evaluation of DEFault++ on 11 transformer faults. FD, FC, and RC denote fault detection, fault categorization, and root-cause diagnosis. Crash* marks a run that fails before feature extraction, which we count as not applicable rather than as a diagnostic failure.

Source	Fault category	DEFault++ result			Feature Importance (top)	Relevant Scope of DEFault++
		FD	FC	RC		
jax#23349	Masking	✓	✓	✓	Attention: padding attention mass	Attention feature group
pytorch#103082	Masking	✓	✓	✓	Attention: future attention mass	Attention feature group
transformers#19045	Positional	✓	✓	✗	Positional: positional sensitivity	Forward sequential propagation (M1)
transformers#17886	Positional	✓	✓	✗	Attention: inter-head cosine similarity	Forward sequential propagation (M1)
ai-edge-torch#6	QKV	✓	✓	✓	Score: pre-softmax score	Simultaneous propagation (M2)
nsa-pytorch#20	KV Cache	✓	✓	✗	KV Cache: cache-related behavior	Generation-time cache update path
transformers#37574	KV Cache	✗	✗	✗	–	Generation-time cache behavior
transformers#36096	Score	Crash*			–	Runtime failure before feature extraction
pytorch#116333	Kernel	✗	✗	✗	–	Backend stride validation
transformers#35896	Variant	✓	✓	✗	Attention: attention entropy	Cross-layer propagation (M4)
diffusers#11903	QKV	✓	✓	✓	Optimization: projection update ratio (Dynamic parameter registration)	QKV projection/update path

Summary: FD = 8/11, FC = 8/11, RC = 4/11.

For the real-world faults that DEFault++ detects and categorizes correctly, the top-ranked feature groups generally align with the reported fault category. For the masking faults, DEFault++ assigns high importance to attention-mass features, which measure attention given to invalid tokens. For the QKV faults, the top-ranked groups include score and projection-update features, which correspond to QKV projection behavior and attention-score computation. For positional and variant faults, DEFault++ assigns high importance to position-sensitive and attention-behavior features, although these cases are not always diagnosed at the root-cause level. DEFault++ detects and categorizes several real-world faults correctly but does not always identify the correct root cause, as in the positional, KV Cache, and variant cases. This matches our benchmark results, which show that root-cause diagnosis is the most difficult level.

DEFault++ does not identify three real-world faults. In `transformers#37574`, the fault appears during autoregressive generation, where the model stores KV states from earlier decoding steps and reuses them when generating later tokens. DEFault++ extracts cache-related behavior during fine-tuning, but it does not trace how the cache changes at each generation step, which can limit its ability to diagnose faults that appear only through generation-time cache updates. In `pytorch#116333`, the fault depends on backend stride validation rather than transformer-layer behavior, so diagnosing it would require system-level information beyond the model trace DEFault++ uses. Finally, `transformers#36096` falls outside DEFault++'s scope because the fault crashes before a valid trace can be collected.

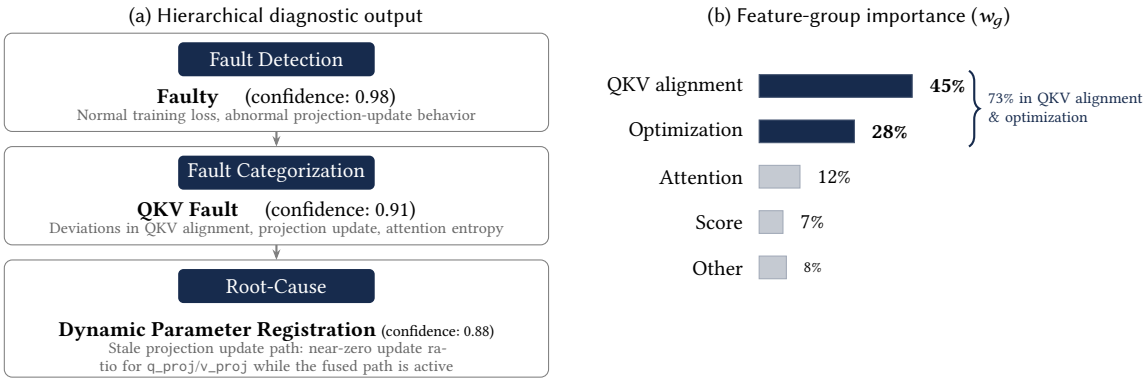


Fig. 14. DEFault++ diagnosis and feature-group importance for the stale QKV fusion fault in Listing 2.1

We use `diffusers#11903` as a real-world case study for DEFault++. The fault redirects the forward pass to a fused QKV projection while LoRA updates the original projection modules. In this case, training completes without an explicit failure, but the updates do not affect the projection path used during execution. Figure 14 shows the DEFault++ output for this case. DEFault++ detects the run as faulty, assigns it to the QKV category, and diagnoses the root cause as dynamic parameter registration. The explanation assigns most of the normalized feature importance to QKV Alignment and Optimization, which together account for 73% of the score. These groups point to a mismatch between the projection modules updated by LoRA and the fused QKV projection used in the forward pass. The GitHub bug fix addresses the same mismatch by ensuring that LoRA updates the projection module executed by the model. This case also shows that DEFault++ generalizes within the dynamic parameter registration root cause rather than memorizing a single operator. The training operator `QFG` freezes QKV gradients while the parameters stay in the forward path, whereas `diffusers#11903` is the opposite case, where gradients flow but the parameters leave the forward path. DEFault++

recognizes the second case through the same near-zero QKV update-activity signature, which suggests the root-cause label spans sub-mechanisms within a category rather than fitting one operator.

10 Developer Study

We evaluate DEFault++ through a developer study using real-world transformer debugging scenarios derived from reproduced GitHub issues. Participants review four scenarios and select the most appropriate repair action for each one. For two scenarios, participants receive only the debugging artifacts available from the reproduced issue (e.g., logs, model outputs). For the other two scenarios, participants receive the same artifacts together with the diagnosis generated by DEFault++. We use the study to measure how DEFault++ affects repair correctness, confidence, and self-reported completion time, and how participants perceive the usefulness of its diagnoses.

Developer Study Design. We use a within-subjects, counterbalanced design with two conditions [89]. In the baseline condition, participants receive the scenario description, observed behavior, logs, and metrics. In the DEFault++ assisted condition, they receive the same materials plus the DEFault++ diagnosis, including fault status, fault category, predicted root cause, and supporting feature group. Each participant completes all four scenarios, with two scenarios assigned to each condition. We counterbalance condition assignment across two survey forms, following prior scenario-based developer studies of DL debugging techniques [52, 74].

The four scenarios cover Masking, Positional, KV Cache, and Variant faults (Table 16). For S1, DEFault++ provides a correct diagnosis at all three levels. For S2, S3, and S4, DEFault++ correctly detects the fault and predicts the correct fault category, but predicts an incorrect root cause (Table 17). We include these imperfect-diagnosis scenarios to evaluate whether category-level diagnosis and supporting feature groups can still guide repair selection when the root-cause prediction is incorrect.

The primary outcome is repair correctness, scored against the ground-truth fix from the source issue. We also collect self-reported confidence on a 5-point scale and self-reported time as a perceived-effort measure. After completing the scenarios, participants rate the clarity, usefulness, and practical value of the DEFault++ diagnosis on 5-point Likert scales and answer preference questions comparing the two conditions.

Table 16. Developer-study scenarios and correct repair actions

Scenario	Source Issue	Category	Repair Action
S1: Cached decoding visibility	pytorch #103082	Masking	Fix the causal mask for cached decoding so that the new token attends to the correct cached key positions.
S2: Position-dependent scoring drift	transformers #19045	Positional	Recompute relative-position distances from the true token positions during cached decoding.
S3: Incremental state update mismatch	nsa-pytorch #20	KV Cache	Write the current token's key-value vectors into the cache before computing attention and prediction scores.
S4: Local-attention layer assignment	transformers #35896	Variant	Fix the layer-selection condition so that local/sliding-window attention is applied to the intended layers.

Table 17. DEFault++ diagnosis shown to participants in each developer-study scenario

Scenario	Level 1	Level 2	Level 3 (predicted)	Level 3 (ground truth)
S1 (Masking)	Faulty ✓	Masking ✓	Dynamic mask ✓	Dynamic mask
S2 (Positional)	Faulty ✓	Positional ✓	Indexing ✗	Relative position
S3 (KV Cache)	Faulty ✓	KV Cache ✓	Cache invalidation ✗	Update synchronization
S4 (Variant)	Faulty ✓	Variant ✓	Dynamic dispatch ✗	Variant configuration

Developer Study Participants. The Dalhousie University Research Ethics Board approved the study.[2] Participants are eligible if they have experience with transformer architectures in training, fine-tuning, inference, debugging, or evaluation [52, 74]. We recruit 21 participants through graduate-student mailing lists, academic networks, and direct outreach (Table 18a). Participation is anonymous, voluntary, and uncompensated, and runs through Microsoft Forms.

Table 18. Developer study: participant demographics and repair outcomes (N = 21 participants)

(a) Participant demographics

Characteristic	Value	N
ML/DL experience	<1 year	1
	1–2 years	6
	3–5 years	10
	>5 years	4
Prior transformer debugging experience	Yes	15
	No	6
Current Role (multi-select)	Research assistant	10
	Graduate student	11
	Industry practitioner	9

(b) Repair accuracy and confidence by scenario and condition

Scenario	Condition	N	Correct	Accuracy	Confidence
S1 (Masking)	Baseline	12	8/12	66.7%	3.75
S1 (Masking)	Assisted	9	9/9	100.0%	4.11
S2 (Positional)	Baseline	9	4/9	44.4%	2.78
S2 (Positional)	Assisted	12	10/12	83.3%	3.75
S3 (KV Cache)	Baseline	12	7/12	58.3%	3.50
S3 (KV Cache)	Assisted	9	8/9	88.9%	4.11
S4 (Variant)	Baseline	9	5/9	55.6%	3.33
S4 (Variant)	Assisted	12	8/12	66.7%	4.08
Total	Baseline	42	24/42	57.1%	3.38
Total	Assisted	42	35/42	83.3%	4.00

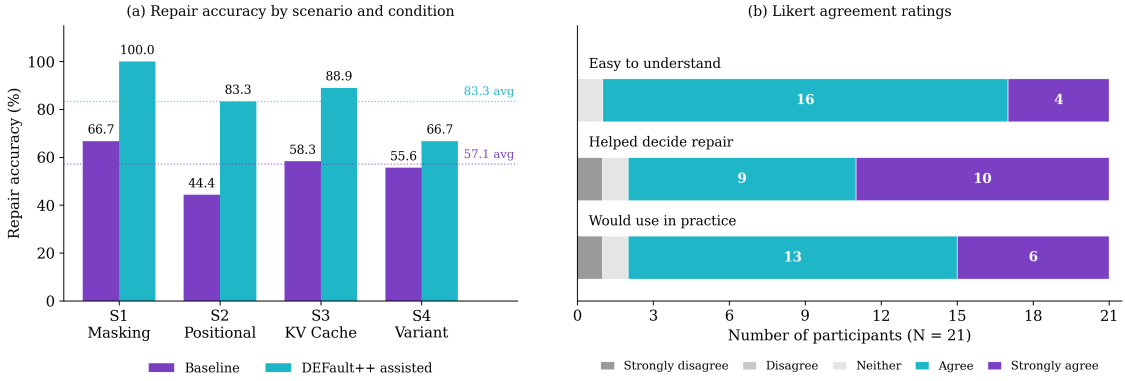


Fig. 15. Developer study results: (a) Repair accuracy by scenario and condition, (b) Likert agreement ratings

Developer Study Results. We report descriptive statistics, following prior vignette-based debugging studies such as UMLAUT [74] and KUnit [52]. Each participant sees only two scenarios per condition, so we treat the difference between conditions as an effect-size estimate rather than as a hypothesis test.

Participants select the correct repair more often with DEFault++ assistance than in the baseline condition. Repair accuracy increases from 57.1% in the baseline condition to 83.3% in the assisted condition (Table 18b and Fig. 15). Confidence also increases from 3.38 to 4.00. The largest gains appear in S2 and S3, where DEFault++ predicts the correct fault category but an incorrect root cause. This suggests that category-level diagnosis and supporting feature groups can still help developers choose the correct repair direction when the root-cause prediction is imperfect.

[2]https://www.dal.ca/research-and-innovation/support-for-researchers/responsible-conduct-research/human-ethics.html

Participants also rate the DEFault++ output positively. Most participants agree or strongly agree that the diagnosis is easy to understand, helps them decide on a repair, and would be useful in practice (Fig. 15).

11 Threats to Validity

Threats to Internal Validity. In our evaluation, information leakage could occur if runs from the same model–task pair were split across training and test folds. We address this risk with nested grouped cross-validation, which keeps each model–task pair within a single outer fold and limits preprocessing and hyperparameter selection to the inner training data. The joint training objective could also affect the evaluation because DEFault++ shares an encoder across diagnostic levels. As a result, the loss from one level may influence the representation used by another level. We mitigate the risk by restricting the separation loss to faulty samples within their ground-truth category and by evaluating its contribution through ablation.

Threats to Construct Validity. Mutation-generated faults may not fully represent faults that developers encounter in real-world transformer models [28, 33]. We reduce this risk by deriving each mutation operator from documented root causes in attention and DNN fault taxonomies [27, 29, 33], which are based on real-world examples. Class imbalance may also affect the evaluation results. We mitigate this risk by using macro-averaged F1 and inverse-frequency class weighting [6]. The mutation-killing test might fail to identify weak or noisy mutants, even when the intended change is successfully applied through fault injection. This may create a conservative fault set, but it improves label reliability because each retained faulty run differs statistically from its matched clean run. Extreme mutants may also inflate diagnostic performance when their effects are easy to detect [28]. We reduce this risk by varying fault severity and target layer, and by reporting mutation scores separately for each category. A final construct concern is the causal reading of the outputs. The training labels are causally grounded by construction, since each mutant differs from its clean counterpart through one targeted change under matched seeds. The Fault Propagation Graph and the diagnostic model, however, capture statistical associations rather than causal mechanisms, because the graph follows the computation graph and the model predicts mutation-generated labels. The RQ_4 group ablation and the real-world trace-to-code checks show that the reported feature groups are consistent with the predicted faults, but they do not establish a causal explanation, so we treat FPG edges as plausible propagation paths rather than as verified causal links.

Threats to External Validity. Generalization beyond the evaluated model–task pairs remains an external validity concern. We reduce this risk by using grouped outer folds, so each method is tested on pairs not seen during training or hyperparameter selection. Comparisons with prior techniques may also be affected by differences in prediction targets, since those techniques do not support DEFault++'s multi-level diagnoses (Table 2). We therefore restrict direct baseline comparison to fault detection. In the developer study, differences between participants could affect repair choices. We reduce this risk with a counterbalanced within-subjects design, where each participant completes scenarios with and without DEFault++ support.

12 Limitations and Future Work

DEFault++ currently diagnoses one injected fault per training run. This setting isolates each fault for categorization and root-cause diagnosis, but it does not cover programs with several co-occurring faults. Extending DEFault-bench to multi-fault configurations and studying how diagnostic behavior changes when several faults appear in the same run is a natural next step.

The Fault Propagation Graph (FPG) currently acts as a structural prior. Its edges mark possible propagation paths between transformer components, but they do not encode dependency strength or propagation mechanism. Calibrating

these edges with fault-injection measurements and annotating them with dependency types would let DEFault++ distinguish stronger, weaker, and mechanism-specific propagation paths, and would let the model learn edge weights rather than treat every edge alike.

Our evaluation covers four encoder-only and three decoder-only models with 66M–125M parameters and 6–12 layers. It does not cover encoder-decoder models, vision models, multimodal models, mixture-of-experts architectures, or larger transformers. Since each mutation operator targets a component type, DEFault-bench extends to another model in the same architecture family by mapping its modules to those component types, although DEFault++'s trained classifier weights are model-specific and need retraining.

Our explanation evaluation measures faithfulness to DEFault++'s own decision process. DEFault-bench provides root-cause labels but not ground-truth feature-group explanations, so future work can strengthen the evaluation by adding independently validated feature-group annotations or by designing controlled faults with known feature-level effects.

Finally, our developer study is modest (N = 21) and vignette-based rather than live, so we treat self-reported time as estimated effort rather than as a direct measure of task completion time. A larger live study that varies the type and severity of diagnostic errors would better characterize how practitioners use DEFault++ during debugging.

13 Conclusion

We present DEFault++, a transformer-specific hierarchical technique for fault detection, fault categorization, and root-cause diagnosis. DEFault++ extends our prior hierarchical DNN fault diagnosis to transformer architectures and reports which feature groups support each diagnosis. It organizes component-level measurements with a Fault Propagation Graph, a structural prior derived from the dependency paths in transformer computation. To train and evaluate it, we construct DEFault-bench, a benchmark of 5,556 labeled transformer runs generated with transformer-specific mutation testing. On DEFault-bench, DEFault++ reaches F1 of 0.826–0.909 for detection, 0.850–0.868 for categorization, and 0.851–0.867 for hierarchical root-cause diagnosis. In a developer study with 21 participants, repair accuracy rises from 57.1% without DEFault++ to 83.3% with it. The ablation and the developer study together point to one design lesson, that component-level measurements organized along the transformer's propagation structure separate faults that model-level training signals leave indistinguishable. We release DEFault-bench and the DEFault++ implementation in our replication package [32].

References

[1] 2024. FlashAttention kernel is not utilized because scaled dot product attention received a non-null attn mask. https://github.com/Dao-AILab/flash-attention/issues/1321.

[2] Muhammad Adnan, Akhil Arunkumar, Gaurav Jain, Prashant J. Nair, Ilya Soloveychik, and Purushotham Kamath. 2024. Keyformer: KV Cache Reduction through Key Tokens Selection for Efficient Generative Inference. In *Proceedings of the 7th MLSys Conference (MLSys)*.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer Normalization. *arXiv preprint arXiv:1607.06450* (2016). doi:10.48550/arXiv.1607.06450

[4] Houssem Ben Braiek and Foutse Khomh. 2023. Testing feedforward neural networks training programs. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–61. doi:10.1145/3529318

[5] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Samuel Pittman, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. *arXiv preprint arXiv:2104.00006* (2021). doi:10.48550/arXiv.2104.00006

[6] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. 2018. A systematic study of the class imbalance problem in convolutional neural networks. *Neural networks* 106 (2018), 249–259. doi:10.1016/j.neunet.2018.07.011

[7] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. Deepfd: Automated fault diagnosis and localization for deep learning programs. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 573–585.

doi:10.1145/3510003.3510099

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021). doi:10.48550/arXiv.2107.03374

[9] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D Manning. 2019. What Does BERT Look at? An Analysis of BERT's Attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP*. 276–286. doi:10.18653/v1/W19-4828

[10] Huangliang Dai, Shixun Wu, Jiajun Huang, Zizhe Jian, Yue Zhu, Haiyang Hu, and Zizhong Chen. 2025. FT-Transformer: Resilient and Reliable Transformer with End-to-End Fault Tolerant Attention. arXiv:2504.02211 [cs.DC] doi:10.48550/arXiv.2504.02211 arXiv:2504.02211v2.

[11] Alana de Santana Correia and Esther Luna Colombini. 2022. Attention, please! A survey of neural attention models in deep learning. *Artificial Intelligence Review* 55, 8 (2022), 6037–6124. doi:10.1007/s10462-022-10148-x

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies*. 4171–4186. doi:10.18653/v1/N19-1423

[13] Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. 2021. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In *International Conference on Machine Learning*. 2793–2803.

[14] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. Deepfault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering*. Springer International Publishing, Cham, 171–189. doi:10.1007/978-3-030-16722-6_10

[15] Pál Erdős and Alfréd Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6 (1959), 290–297.

[16] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* 542, 7639 (2017), 115–118. doi:10.1038/nature21056

[17] Kawin Ethayarajh. 2019. How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 55–65. doi:10.18653/v1/D19-1006

[18] Gemini Team. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023). doi:10.48550/arXiv.2312.11805

[19] Aaron Gokaslan and Vanya Cohen. 2019. OpenWebText Corpus. http://Skylion007.github.io/OpenWebTextCorpus.

[20] Phillip Good. 2005. *Permutation, Parametric, and Bootstrap Tests of Hypotheses* (3rd ed.). Springer. doi:10.1007/b138696

[21] Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, Ismet Burak Kadron, Corina Pasareanu, and Sarfraz Khurshid. 2019. Symbolic execution for importance analysis and adversarial generation in neural networks. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 313–322. doi:10.1109/ISSRE.2019.00039

[22] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On Calibration of Modern Neural Networks. In *International Conference on Machine Learning*. PMLR, 1321–1330.

[23] Fengxiang He, Bohan Wang, and Dacheng Tao. 2020. Piecewise linear activations substantially shape the loss surfaces of neural networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[24] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

[25] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. Deepmutation++: A mutation testing framework for deep learning systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1158–1161. doi:10.1109/ASE.2019.00126

[26] Hugging Face. 2024. Transformers Library. https://github.com/huggingface/transformers. Accessed: 2025-04-21.

[27] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 1110–1121. doi:10.1145/3377811.3380395

[28] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. Deepcrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 67–78. doi:10.1145/3460319.3464825

[29] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 510–520. doi:10.1145/3338906.3338955

[30] Niful Islam, Ragib Shahriar Ayon, Deepak George Thomas, Shibbir Ahmed, and Mohammad Wardat. 2026. When Agents Fail: A Comprehensive Study of Bugs in LLM Agents with Automated Labeling. *arXiv preprint arXiv:2601.15232* (2026). doi:10.48550/arXiv.2601.15232

[31] Foad Jafarinejad, Krishna Narasimhan, and Mira Mezini. 2021. NerdBug: automated bug detection in neural networks. In *Proceedings of the 1st ACM International Workshop on AI and Software Testing/Analysis*. 13–16. doi:10.1145/3464968.3468409

[32] Sigma Jahan. 2026. DEFault++ Replication Package. https://github.com/SigmaJahan/DEFaultplusplus-Transformer-Debugging. Available at: https://github.com/SigmaJahan/DEFaultplusplus-Transformer-Debugging.

[33] Sigma Jahan, Saurabh Rajput, Tushar Sharma, and Mohammad Masudur Rahman. 2026. Why Attention Fails: A Taxonomy of Faults in Attention-Based Neural Networks. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*.

[34] Sigma Jahan, Saurabh Singh Rajput, Tushar Sharma, and Mohammad Masudur Rahman. 2025. Taxonomy of Faults in Attention-Based Neural Networks. arXiv:2508.04925 [cs.SE] doi:10.48550/arXiv.2508.04925

[35] Sigma Jahan, Mehil B Shah, Parvez Mahbub, and Mohammad Masudur Rahman. 2025. Improved Detection and Diagnosis of Faults in Deep Neural Networks Using Hierarchical and Explainable Classification . In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 2944–2956. doi:10.1109/ICSE55347.2025.00224

[36] Erik Jones, Robin Jain, and Percy Liang. 2020. Automatically Finding Bugs in a Classifier: A Systematic Approach. *arXiv preprint arXiv:2003.02907* (2020). arXiv:2003.02907 doi:10.48550/arXiv.2003.02907

[37] Hans Kamp and Barbara Partee. 1995. Prototype theory and compositionality. *Cognition* 57, 2 (1995), 129–191. doi:10.1016/0010-0277(94)00659-9

[38] Prannay Khosla, Yonglong Tian, Huiwen Wang, Ce Liu, Phillip Isola, Dilip Krishnan, Yao Tian, Tatsunori B. Hashimoto, Alan Yuille, Quoc V. Le, Cordelia Schmid, and Trevor Darrell. 2020. Supervised Contrastive Learning. In *Advances in Neural Information Processing Systems*.

[39] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.

[40] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[41] Pang Wei Koh, Thao Nguyen, Yew Siang Tang, Stephen Mussmann, Emma Pierson, Been Kim, and Percy Liang. 2020. Concept Bottleneck Models. In *International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5338–5348.

[42] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. 2019. Similarity of Neural Network Representations Revisited. In *International Conference on Machine Learning*. PMLR, 3519–3529.

[43] Kyla H. Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2025. ChatDBG: Augmenting Debugging with Large Language Models. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. doi:10.1145/3729355

[44] Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 3538–3545. doi:10.1609/aaai.v32i1.11604

[45] Yuhang Liang, Xinyi Li, Jie Ren, Ang Li, Bo Fang, and Jieyang Chen. 2025. ATTNChecker: Highly-Optimized Fault Tolerant Attention for Large Language Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. doi:10.1145/3710848.3710870

[46] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Joshi Mandar, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692* (2019). doi:10.48550/arXiv.1907.11692

[47] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[48] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems*, Vol. 30.

[49] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. Deepmutation: Mutation testing of deep learning systems. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111. doi:10.1109/ISSRE.2018.00021

[50] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 175–186. doi:10.1145/3236024.3236082

[51] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. 2020. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 25–31. doi:10.1109/DSN-W50199.2020.00014

[52] Ruchira Manke, Mohammad Wardat, Foutse Khomh, and Hridesh Rajan. 2025. Mock Deep Testing: Toward Separate Development of Data and Models for Deep Learning. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2970–2982. doi:10.1109/ICSE55347.2025.00220

[53] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics* 19, 2 (1993), 313–330.

[54] Sergei Maslov and Kim Sneppen. 2002. Specificity and Stability in Topology of Protein Networks. *Science* 296, 5569 (2002), 910–913. doi:10.1126/science.1065103

[55] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. *arXiv preprint arXiv:1609.07843* (2016). doi:10.48550/arXiv.1609.07843

[56] Mohammad Mehdi Morovati, Amin Nikanjam, and Foutse Khomh. 2024. Fault Localization in Deep Learning-based Software: A System-level Approach. *arXiv preprint arXiv:2411.08172* (2024). arXiv:2411.08172 [cs.SE] doi:10.48550/arXiv.2411.08172

[57] Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. 2021. On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines. In *International Conference on Learning Representations*. https://openreview.net/forum?id=nzpLWnVAyah

[58] Ramaravind K. Mothilal, Amit Sharma, and Chenhao Tan. 2020. DiCE: Diverse Counterfactual Explanations for Machine Learning Classifiers. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency (FAT*)*. ACM, 607–617. doi:10.1145/3351095.3372850

[59] Gireen Naidu, Tranos Zuva, and Elias Mmbongeni Sibanda. 2023. A review of evaluation metrics in machine learning algorithms. In *Computer science on-line conference*. Springer, 15–25. doi:10.1007/978-3-031-35314-7_2

[60] Amin Nikanjam, Houssem Ben Braiek, Mohammad Mehdi Morovati, and Foutse Khomh. 2021. Automatic fault detection for deep learning programs using graph transformations. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021), 1–27. doi:10.1145/3470006

[61] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. 2021. A review on the attention mechanism of deep learning. *Neurocomputing* 452 (2021), 48–62.

[62] OpenAI. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[63] Denis Paperno, German Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Roberto Fernández. 2016. The LAMBADA Dataset: Word Prediction Requiring Broad Discourse Context. In *Proceedings of ACL 2016*. doi:10.18653/v1/P16-1144

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. 8024–8035.

[65] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038. doi:10.1109/ICSE.2019.00107

[66] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. SoapFL: A Standard Operating Procedure for LLM-Based Method-Level Fault Localization. *IEEE Transactions on Software Engineering* 51, 4 (2025), 1173–1187. doi:10.1109/TSE.2025.3543187

[67] Ketai Qiu, Niccolò Puccinelli, Matteo Ciniselli, and Luca Di Grazia. 2025. From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025). doi:10.1145/3709353

[68] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. OpenAI blog, 9 pages.

[69] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. A Multi-Agent Approach to Fault Localization via Graph-Based Retrieval and Reflexion. *arXiv preprint arXiv:2409.13642* (2024). arXiv:2409.13642 [cs.SE] doi:10.48550/arXiv.2409.13642

[70] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32. doi:10.1609/aaai.v32i1.11491

[71] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, 4902–4912. doi:10.18653/v1/2020.acl-main.442

[72] Lucas Roquet, Fernando Fernandes dos Santos, Paolo Rech, Marcello Traiola, Olivier Sentieys, and Angeliki Kritikakou. 2024. Cross-layer reliability evaluation and efficient hardening of large vision transformers models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6. doi:10.1145/3649329.3655688

[73] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. *arXiv preprint arXiv:1910.01108* (2019). doi:10.48550/arXiv.1910.01108

[74] Eldon Schoop, Forrest Huang, and Björn Hartmann. 2021. UMLAUT: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–16. doi:10.1145/3411764.3445538

[75] Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical Networks for Few-shot Learning. In *Advances in Neural Information Processing Systems*, Vol. 30.

[76] Adam Stein, Arthur Wayne, Aaditya Naik, Mayur Naik, and Eric Wong. 2025. Where's the Bug? Attention Probing for Scalable Fault Localization. *arXiv preprint arXiv:2502.13966* (2025). doi:10.48550/arXiv.2502.13966

[77] Florian Tambon, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, and Giuliano Antoniol. 2023. Mutation Testing of Deep Reinforcement Learning Based on Real Faults. https://arxiv.org/abs/2301.05651 arXiv:2301.05651 [cs.LG].

[78] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, Gothenburg, Sweden, 303–314. doi:10.1145/3180155.3180220

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[80] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*. 5797–5808. doi:10.18653/v1/P19-1580

[81] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of ICLR 2019*. doi:10.18653/v1/W18-5446

[82] Xin Wang, Yi Qin, Yi Wang, Sheng Xiang, and Haizhou Chen. 2019. ReLTanh: An activation function with vanishing gradient resistance for SAE-based DNNs and its application to rotating machinery fault diagnosis. *Neurocomputing* 363 (2019), 88–98.

[83] Zhijie Wang, Yuheng Huang, Da Song, Lei Ma, and Tianyi Zhang. 2023. DeepSeer: Interactive RNN Explanation and Debugging via State Abstraction. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. ACM. doi:10.1145/3544548.3580852

[84] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 561–572. doi:10.1145/3510003.3510071

[85] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2023. An effective data-driven approach for localizing deep learning faults. *arXiv preprint arXiv:2307.08947* (2023). doi:10.48550/arXiv.2307.08947

[86] Mohammad Wardat, Wei Le, and Hridesh Rajan. 2021. Deeplocalize: Fault localization for deep neural networks. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 251–262. doi:10.1109/ICSE43902.2021.00034

[87] Ni Wayan Surya Wardhani, Masithoh Yessi Rochayani, Atiek Iriany, Agus Dwi Sulistyono, and Prayudi Lestantyo. 2019. Cross-validation metrics for evaluating classification performance on imbalanced data. In *Proceedings of the International Conference on Computer, Control, Informatics and Its Applications (IC3INA)*. IEEE, 14–18. doi:10.1109/IC3INA48034.2019.8949568

[88] Shihao Weng, Yang Feng, Jincheng Li, Yining Yin, Xiaofei Xie, and Jia Liu. 2026. AtPatch: Debugging Transformers via Hot-Fixing Over-Attention. *arXiv preprint arXiv:2601.21695* (2026). doi:10.48550/arXiv.2601.21695

[89] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. doi:10.1007/978-3-642-29044-2

[90] Tom Nuno Wolf, Fabian Bongratz, Anne-Marie Rickmann, Sebastian Pölsterl, and Christian Wachinger. 2024. Keep the faith: Faithful explanations in convolutional neural networks for case-based reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 5921–5929. doi:10.1609/aaai.v38i6.28406

[91] Dangwei Wu, Beijun Shen, Yuting Chen, He Jiang, and Lei Qiao. 2021. Tensfa: detecting and repairing tensor shape faults in deep learning systems. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 11–21. doi:10.1109/ISSRE52982.2021.00014

[92] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. 2020. On Layer Normalization in the Transformer Architecture. In *International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR. http://proceedings.mlr.press/v119/xiong20b.html

[93] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 627–638. doi:10.1145/3468264.3468612

[94] Shuangfei Zhai, Tatiana Likhomanenko, Etai Littwin, Dan Busbridge, Jason Ramapuram, Yizhe Zhang, Jiatao Gu, and Joshua M. Susskind. 2023. Stabilizing Transformer Training by Preventing Attention Entropy Collapse. In *International Conference on Machine Learning*. 40770–40803.

[95] Hao Zhang and W. K. Chan. 2019. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 376–387. doi:10.1109/ASE.2019.00043

[96] Haoyi Zhang, Jing Li, Xu Li, Jie Bing, Mingnan Chen, Dongxiang Yu, Jianxin Tang, and Beichen Zhang. 2024. Layer-Condensed KV Cache for Efficient Inference of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Bangkok, Thailand, 11192–11205. doi:10.18653/v1/2024.acl-long.602

[97] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. Autotrainer: An automatic dnn training problem detection and repair system. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 359–371. doi:10.1109/ICSE43902.2021.00043

[98] Yuhao Zhang, Luyao Ren, Liqian Chen, Yingfei Xiong, Shing-Chi Cheung, and Tao Xie. 2020. Detecting numerical bugs in neural network architectures. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 826–837. doi:10.1145/3368089.3409720