CoreCapture: Anomaly-Aware System Trace Sampling for Resource-Constrained Mobile Devices

Yongqian Sun
Nankai University
Tianjin, China
sunyongqian@nankai.edu.cn

Xiaotian Wang
Nankai University
Tianjin, China
2120240811@mail.nankai.edu.cn

Yicheng Sui*
Nankai University
Tianjin, China
suiyicheng@nankai.edu.cn

Jintao Feng
Nankai University
Tianjin, China
2211371@mail.nankai.edu.cn

Yue Zhang
Nankai University
Tianjin, China
2211604@mail.nankai.edu.cn

Shenglin Zhang
Nankai University
Tianjin, China
zhangsl@nankai.edu.cn

Bowen Zheng
Huawei
Nanjing, China
zhengbowen1@huawei.com

Qiang Li
Huawei
Xian, China
liqiang78@huawei.com

Wei Liang
Huawei
Xian, China
sailighty.liang@huawei.com

Dan Pei
Tsinghua University
Beijing, China
peidan@tsinghua.edu.cn

Abstract

Diagnosing performance anomalies on production mobile devices requires System Traces, which are whole-system runtime records spanning kernel, services, and applications. Yet each trace dump consumes over 100 MB, and the daily upload budget shared across all diagnostic components limits its collection to a few dumps per day; anomalies occurring after the quota is exhausted are permanently lost. We present CoreCapture, an on-device system that compresses each dump by filtering out steady-state data while preserving anomaly-relevant diagnostic context. CoreCapture employs a dual-stream anomaly detector (DS-TraceNet) that separately models semantic and temporal features to identify anomalous traces, followed by a spatio-temporal retention mechanism (STA-ER) that expands each anomaly to a complete interaction-event window and prunes irrelevant threads. Evaluation on production traces shows that CoreCapture reduces per-dump volume by approximately 70% while preserving 95% event-level anomaly coverage. Deployed on production smartphones, it adds only 4–5% CPU and 13–15 MB memory overhead on device per alert-triggered dump. Our source code demo and two data cases are publicly available at https://github.com/linjiaxiaoyue/CoreCapture.

CCS Concepts: • **Software and its engineering → Operational analysis**; **Software maintenance tools**; *Software evolution.*

Keywords: System Trace, Anomaly-Aware Sampling, Mobile Devices, On-Device Anomaly Detection, Performance Diagnosis

ACM Reference Format:
Yongqian Sun, Xiaotian Wang, Yicheng Sui, Jintao Feng, Yue Zhang, Shenglin Zhang, Bowen Zheng, Qiang Li, Wei Liang, and Dan Pei. 2026. CoreCapture: Anomaly-Aware System Trace Sampling for Resource-Constrained Mobile Devices. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857128

1 Introduction

Performance anomalies on mobile devices, including UI jank, touch delays, and app freezes, degrade user experience [17–19, 42]. To diagnose these anomalies, engineers examine complete operational units such as a frame render, an input dispatch cycle, or a user interaction episode [24, 39]. We call each such unit an *event*. An event produces a *trace*: a sequence of *tracepoints*, where each tracepoint records one system operation (a function call, a scheduling event, or an IPC transaction) along with its thread context, CPU core,

*corresponding author



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857128

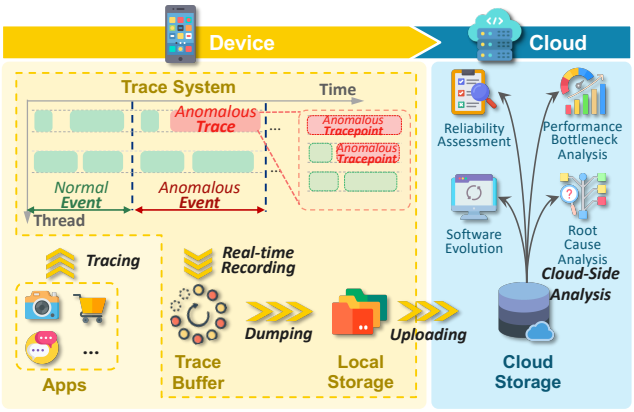


Figure 1. End-to-end Trace System workflow.

and timestamp. The Trace System, built on Linux Ftrace [20], instruments critical paths such as input dispatch, view rendering, binder transactions, and CPU scheduling, and continuously writes tracepoints into a fixed-size in-kernel buffer. Figure 1 shows the end-to-end pipeline: when an anomaly is detected, the buffer is dumped to local storage and uploaded to the cloud for offline analysis. While this pipeline is architecturally sound, its practical effectiveness is constrained by resource limitations on mobile devices.

The operational reality is that trace collection is heavily throttled. A single 15-second trace window produces roughly 144 MB of raw data, creating storage pressure and I/O load that measurably interfere with foreground workloads. To keep this impact manageable, production systems allocate a daily upload budget of roughly 400 MB, shared across all diagnostic components including performance monitors, crash reporters, and user-initiated diagnostics, not only trace sampling. A single dump of the buffered trace data is approximately 144 MB, already consuming over a third of this shared budget; even two or three dumps per day can exhaust the allocation. The common alternative [15, 16], dumping a fixed-length segment when an anomaly is suspected, has two weaknesses: the window may miss the actual anomalous interval, and the captured data is dominated by irrelevant steady-state activity. Both strategies share a fundamental problem: unlike server-side systems where issues can often be reproduced under controlled conditions, mobile performance anomalies arise from specific combinations of user interaction, background activity, thermal state, and resource contention that are difficult to recreate, making missed anomaly data permanently irrecoverable.

A natural question is therefore whether the system can retain less data overall but keep only the anomaly-relevant portion. Yet existing trace-based anomaly detection methods [21, 43, 51, 52] are not directly applicable here: they optimize for balanced precision and recall on server-side resources, whereas the on-device setting demands near-perfect recall under strict resource constraints, where any missed anomaly is permanently lost and retention decisions must be made in real time with minimal CPU and memory overhead. We therefore define a new task, **Anomaly-Aware Trace Sampling (AATS)**, with the objective of maximizing anomaly coverage while minimizing the data retention ratio, subject to the requirement that retained segments contain sufficient diagnostic context for root-cause analysis.

> ► **Finding: Anomaly Detection ≠ Anomaly-Aware Sampling**
>
> - **Output purpose.** Anomaly detection identifies anomalous *points*; AATS must retain diagnostic *fragments* with sufficient context for root-cause analysis.
> - **Optimization target.** Anomaly detection balances precision and recall; AATS requires near-perfect recall because any missed anomaly is permanently lost.
> - **Execution environment.** Anomaly detection runs on servers with ample resources; AATS must run on-device under strict CPU and memory constraints.

Achieving these objectives under on-device resource constraints requires solving two technical problems.

Challenge 1: Heterogeneous anomaly signals. Production system traces contain three characteristic anomaly patterns: duration anomalies, interval anomalies, and cross-type anomalies (§ 2.3). The cross-type pattern is revealing: different types of system operations have different normal duration ranges, meaning anomaly detection must be type-aware. A sufficiently large model [49] could theoretically learn both semantic and temporal features jointly, but AATS requires continuous on-device execution under strict CPU and memory constraints. In a lightweight model [47], the high-dimensional semantic features risk dominating gradient updates, suppressing the low-dimensional temporal signals and degrading detection of duration and interval anomalies. The challenge is therefore to achieve type-aware detection within the resource budget of a mobile device.

Challenge 2: Extracting diagnostic context around detected anomalies. AATS must not only detect anomalies but also retain the surrounding context needed for root-cause analysis. However, there is no established rule for how much context suffices: the causal chain behind a frame-rendering jank may span hundreds of milliseconds or multiple threads, depending on the specific anomaly. The system must therefore crop a diagnostic fragment around each detected anomaly by making trade-offs along two dimensions (§ 2.4): temporally, extending far enough to capture the causal chain without wasting storage on irrelevant activity; at the thread level, including enough threads to preserve cross-thread dependencies without diluting the fragment with noise.

We present **CoreCapture**, an on-device system that addresses both challenges through DS-TraceNet (dual-stream anomaly detection) and STA-ER (spatio-temporal anomaly event retention).

We make the following contributions:

1. To the best of our knowledge, we are the first to formalize Anomaly-Aware Trace Sampling (AATS) as a new systems task with explicit constraints and an optimization objective, addressing a critical data capture challenge in mobile systems, and present CoreCapture as its concrete realization on production mobile devices.
2. We propose Dual-Stream System Trace Network (DS-TraceNet), a detection model that separates semantic and temporal modalities into parallel specialized branches, allowing each branch to independently optimize for its respective anomaly signal and resolving the modality tension identified in Challenge 1.
3. We design Spatio-Temporal Anomaly Event Retention (STA-ER), a cropping mechanism that extracts diagnostic fragments centered on detected anomalies by defining temporal boundaries via event windows and thread-level boundaries via critical-thread filtering.
4. We evaluate CoreCapture on large-scale real-world production traces, achieving approximately 70% data reduction while maintaining 95% event-level anomaly coverage with throughput exceeding 11,000 traces per second.

2 Exploratory Study

To empirically ground the two challenges identified in § 1, we examine production system traces from real user-reported fault tickets. We first establish the operational context (§ 2.1) and data abstractions (§ 2.2), then present observations from six months of production traces that motivate CoreCapture's design (§ 2.3 and § 2.4).

2.1 System Trace Collection Workflow

Figure 1 in § 1 shows the end-to-end pipeline. We now describe the device-side data flow in more detail. Modern mobile OSes provide a performance instrumentation subsystem built on the Linux Ftrace framework [20]. The recording stage has three steps:

Recording. Tracepoints are inserted into key code paths: input dispatch, view rendering, binder and IPC transactions, CPU frequency changes, and memory allocation. The instrumented code paths write timestamped tracepoint records into a fixed-size ring buffer in the kernel. Application developers can also add custom tracepoints through user-space instrumentation APIs.

Dumping. When a dump command is issued, whether by a diagnostic component or a user-initiated bug report, the buffer contents are flushed to persistent storage. A single dump spanning approximately 15 seconds of continuous execution produces roughly 144 MB of data, and the daily upload budget of approximately 400 MB is shared across all diagnostic components, leaving limited room for additional collections.

Table 1. Tracepoint format: one record per system operation.

Field	Example	Description
Thread Name	`render_service`	The executing thread
TID	`1458`	Thread ID
PID	`1458`	Process ID
CPU	`001`	CPU core number
Timestamp	`187065.212543`	Seconds since boot
Tracepoint Name	`H:CleanRenderNodes 58856`	The traced operation name

Uploading. The dumped file is then uploaded to the cloud for offline analysis.

2.2 System Trace and Event Abstraction

A single dump produces a flat stream of *tracepoints*. Each tracepoint records one system operation along with its thread context, CPU core, timestamp, and tracepoint name (Table 1). The format follows Ftrace [20] and is shared across Android (Atrace [32]), HarmonyOS (HiTrace [30]), and other mobile OSes. Duration tracepoints additionally record the elapsed time between the entry and exit of a code region.

Trace events. When a user reports UI jank, an engineer does not examine individual tracepoints; the engineer looks for a complete frame-rendering cycle (from the VSync signal to frame commit) and asks whether that cycle was anomalous. We call such end-to-end system actions *events*. An event corresponds to one complete user interaction or system operation, and in either case, it constitutes a complete, self-contained unit of diagnostic information.

2.3 Anomaly Patterns in Production Traces

The data structure described above raises a natural question: can we detect anomalous patterns in trace data purely from their numerical properties?

We collected system traces from fault tickets of a Top-Tier Mobile Vendor over six months. The corpus spans both system apps and popular third-party apps, with 300+ maintenance tickets and 100+ anomaly categories. Yet when anomalies occur after the daily trace quota is exhausted, the anomaly data is permanently lost. For example, a user reporting intermittent UI jank may find all three daily trace collections already used by other diagnostics, leaving engineers without data.

Manual inspection by experienced diagnostic engineers, supplemented by automated root-cause analysis on the cloud platform, reveals three recurring anomaly patterns in anomaly-bearing events (Figure 2):

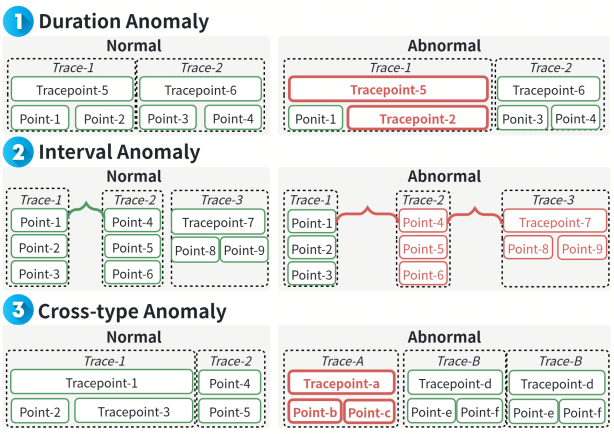


Figure 2. Three characteristic anomaly patterns in production system traces.

2.3.1 Duration Anomaly. A specific tracepoint, such as `DrawFrame`, shows an execution duration far above its normal range. We treat a tracepoint p_i as duration-anomalous if its duration exceeds the $\mu + 3\sigma$ threshold of the empirical distribution for that tracepoint. The threshold is type-dependent: a tracepoint that normally finishes in 1 ms is anomalous at 5 ms, while another tracepoint with a normal duration of 30 ms may still be within the normal range at the same absolute deviation.

2.3.2 Interval Anomaly. Successive invocations of the same tracepoint sometimes show abnormal inter-arrival times. For periodic operations such as VSync event reception at 5.5 ms intervals, a gap of 50 ms or more usually signals a scheduling stall or resource contention. As with duration anomalies, the acceptable interval range is type-dependent.

2.3.3 Cross-type Anomaly. Normal duration distributions vary across tracepoints. In a cross-type anomaly, the same duration is normal for one tracepoint type yet anomalous for another. Figure 2(3) illustrates this: `Trace-1` has a normal execution time of 50 ms, so a 50 ms observation is routine; but `Trace-A` has a normal duration well below 15 ms, and it would be anomalous in that type. A detector must therefore incorporate the semantic identity of each tracepoint rather than relying on absolute numerical thresholds. One that ignores call semantics will either miss subtle anomalies or produce excessive false positives. The three patterns show that neither temporal features nor semantic context alone suffices; a practical detector must reason about both dimensions jointly.

► Finding 1: Joint Temporal-Semantic Anomaly Reasoning

Neither temporal features nor semantic context alone suffices for anomaly detection: the same absolute duration is normal for one tracepoint but anomalous for another. These observations validate the premise of Challenge 1.

2.4 Diagnostic Context Beyond Anomalies

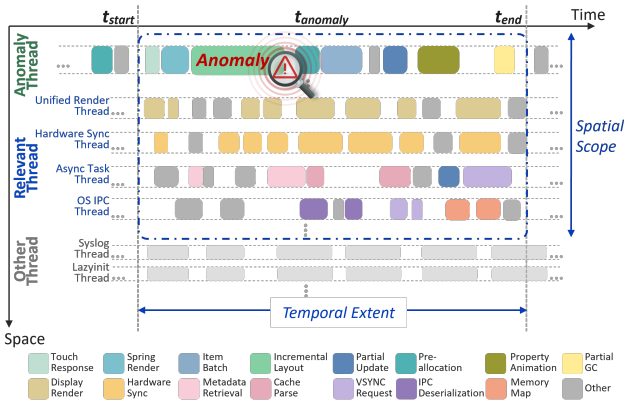


Figure 3. Anomaly-centric spatio-temporal data cropping and retention.

To understand whether detected anomalies alone suffice for diagnosis, we examined how diagnostic engineers actually use trace data in practice. Across the fault tickets in our dataset, engineers consistently reported that isolating the anomalous sequence was only the first step: reconstructing the root cause required examining the surrounding events that preceded and followed the anomaly. For example, as shown in Figure 3, the anomaly thread's lifecycle begins with a touch response, followed by a spring animation and swiper item batching. It then launches an incremental item layout rendering task whose duration is markedly prolonged, indicating a severe performance anomaly, before dispatching a component partial update. Afterwards, an idle-period lazy preallocation runs, and a property animation callback executes with slightly elevated duration.

At the same time, expanding retention to capture full context reintroduces data volume problems. A typical event window contains trace records from hundreds of concurrently scheduled threads, most of which are unrelated to the anomaly under diagnosis. Including all of them would negate the storage savings that selective sampling is supposed to achieve. The system must therefore extend retention temporally to event boundaries that capture causal context, while pruning irrelevant threads to maintain data efficiency.

► Finding 2: Context Is Essential for Diagnosis

Diagnostic engineers report that isolated anomaly sequences are insufficient for root-cause analysis; the surrounding causal context must be retained. However, expanding retention to event-level windows reintroduces thread-level redundancy, requiring thread-level pruning. This confirms Challenge 2.

3 Methodology

We propose **CoreCapture** for Anomaly-Aware Trace Sampling (AATS). Figure 4 shows the overall architecture. § 3.1 and § 3.2 introduce the preprocessing and feature pipeline. § 3.3 presents DS-TraceNet to address heterogeneous anomaly signals (**Challenge 1**), and § 3.4 presents STA-ER to retain diagnostic context around detected anomalies (**Challenge 2**).

3.1 Data Preprocessing

3.1.1 Parameter Filtering. Raw trace data are highly unstructured records containing timestamps, process contexts, and other execution metadata. Among these fields, the tracepoint name is the most informative attribute, as it directly captures system operations and execution behavior. Tracepoint names often include dynamic parameter information. For example, the first tracepoint shown in Figure 5 records a VSync reception event, where parameters such as `name` and `vsyncId` specify the target entity and signal identifier, respectively.

To remove such noise, we employ Drain [14] for template extraction and extend its tokenization strategy by treating additional punctuation symbols as delimiters. Numeric tokens and alphanumeric identifiers are replaced with the wildcard symbol <*>, producing templates that retain semantic information while removing dynamic values (Figure 5).

3.1.2 Sequence Construction. Each trace contains a hierarchy of tracepoints with explicit call-and-return relationships: one root tracepoint as the entry point, and multiple sub-tracepoints triggered in order (e.g., Figure 6 shows `H:RenderFrame` and its sub-tracepoints for a rendering cycle).

We linearize this hierarchy into a single sequence following depth-first order, which retains the execution order of the nested tracepoints, avoiding the cost of tree modeling in the on-device inference path.

3.1.3 Time Extraction. As shown in Figure 6, we extract two temporal features per trace:
Duration Time: the total execution time of the trace, computed from its root tracepoint.
Gap Time: the time gap between the current trace's root tracepoint and the end of the preceding trace in the same thread.

3.2 Feature Engineering

3.2.1 Semantic Embedding. Tracepoint names encode code semantics and execution logic, making their vector representations important for automated anomaly analysis. For example, the tracepoint `H:FlushDirtyNodeUpdate` carries informative functional semantics that may correlate with system anomalies.

Real production environments face two major challenges. First, even inside the OS provider, where the tracepoint names of a single build can in principle be enumerated from source, the vocabulary is not closed: system updates, tracepoint expansion, and user-space instrumentation keep adding names and continuously shift the vocabulary distribution and produce out-of-vocabulary (OOV) tokens. Second, mobile devices have limited storage and computing resources with infrequent updates, making frequent model retraining and redeployment infeasible [22, 54]. Traditional one-hot encoding performs poorly here: its dimension rises linearly with vocabulary size and brings heavy storage burden for expanding vocabularies [23, 33]. Besides, it fails to represent OOV tokens properly, as such tokens are converted into zero vectors and damage sequence semantics [2, 57].

To address these challenges, we adopt FastText [3] to vectorize tracepoint-name token sequences. Unlike Word2Vec [26], which treats complete words as the basic modeling unit, FastText represents each token as a collection of character-level n-grams [1, 35]. By leveraging subword information, FastText can generate meaningful embeddings for unseen OOV tokens through previously observed character fragments, thereby effectively alleviating the OOV problem [5].

We further optimize FastText for lightweight mobile deployment. First, we train FastText on a task-specific corpus built from all tracepoint-name token sequences in our dataset, aligning the embedding space with the target domain and reducing model redundancy. We then adjust the hash bucket count to a value slightly larger than the actual number of n-grams (down from the default 2 million) to save storage while keeping all valid n-grams. We set the n-gram length from 3 to 6 to balance semantic quality and completeness. The resulting model handles OOV tokens while remaining lightweight, meeting on-device deployment constraints.

3.2.2 Time Embedding. Raw time-series observations typically exhibit a right-skewed long-tail distribution, where most values are concentrated in the low-value range while a small number of anomalies extend to tens or hundreds of milliseconds. Directly using these raw values can cause optimization difficulties, reduce training stability, and weaken feature representation due to distribution skewness and extreme outliers. To mitigate this issue, we apply a logarithmic transformation to the time feature, which regularizes the distribution, compresses extreme values, and reduces the

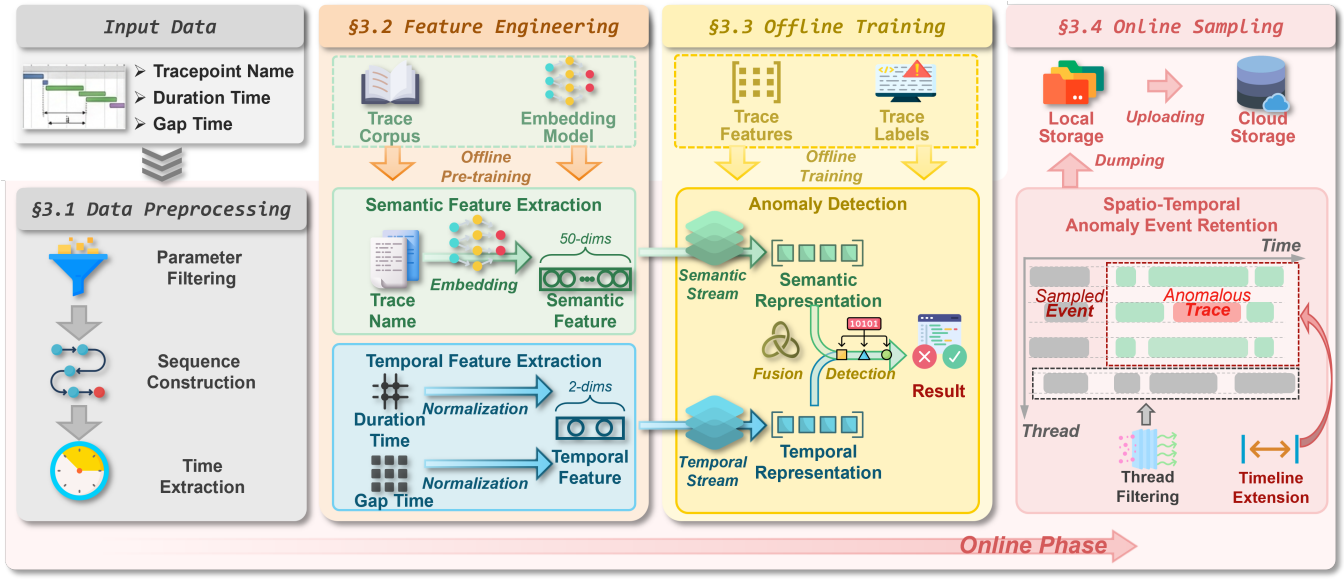


Figure 4. The overall architecture of CoreCapture.

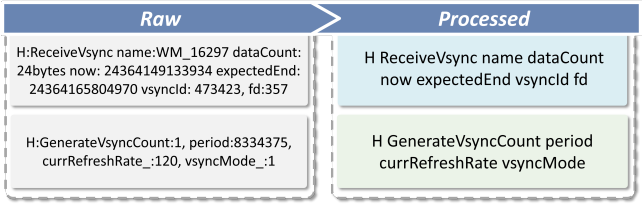


Figure 5. Parameter filtering of raw tracepoint names.

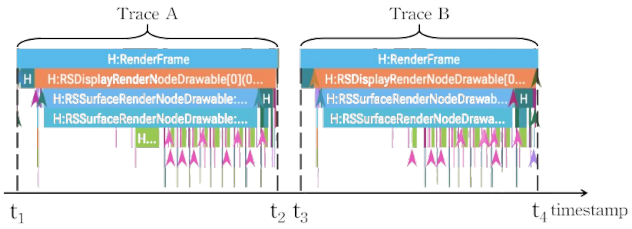


Figure 6. Two example traces for time extraction.

impact of scale differences, thereby facilitating more stable and effective learning of temporal patterns.

3.3 Offline Training

A trace encapsulates two heterogeneous modalities: discrete semantic information and continuous temporal states. Structural or logical anomalies manifest as semantic drift, while performance regressions are reflected in abnormal temporal latencies [11, 55]. Jointly feeding high-dimensional semantic features and low-dimensional temporal features into a monolithic network inevitably causes optimization conflicts: semantic features tend to dominate gradient updates and overshadow temporal signals, while high-variance temporal anomalies can mask subtle semantic drifts [31, 40]. This modality imbalance severely limits the model's capacity to capture underlying anomalous system states [10].

To bridge the gap and overcome the monolithic architectural constraints, we adopt the multi-branch paradigm from computer vision, whose core philosophy lies in the structural decoupling of disparate modalities, such as separating spatial appearance from temporal motion dynamics, to capture features independently via parallel specialized branches [9, 56]. Inspired by this design, we propose a lightweight Dual-Stream System Trace Network (DS-TraceNet) consisting of two parallel streams: the Semantic Stream and the Temporal Stream (as illustrated in Figure 7).

3.3.1 Semantic Stream. The Semantic Stream is responsible for extracting and encoding the high-dimensional structural and behavioral features embedded in traces. For each trace, we obtain an ordered sequence of tracepoint name tokens: $S = \{s_1, s_2, \ldots, s_n\}$, where each s_i denotes the i-th tracepoint name token in the linearized sequence. As described in § 3.2, we map each token s_i into a dense continuous vector space via the lightweight, pre-trained FastText model:

$$v_i^s = \text{FastText}(s_i) \in \mathbb{R}^{d_s} \quad (1)$$

where d_s is the embedding dimension. The token-level embeddings are then aggregated to yield a single trace-level semantic representation $\mathbf{h}^s \in \mathbb{R}^{d_s}$ that encodes the overall functional semantics and invocation structure of the entire trace. Then, the $\mathbf{h}^s$ is fed into the Semantic Stream branch, a compact feed-forward network with batch normalization and dropout regularization, to produce a refined semantic representation $\mathbf{f}^s$. This progressive design forces the branch

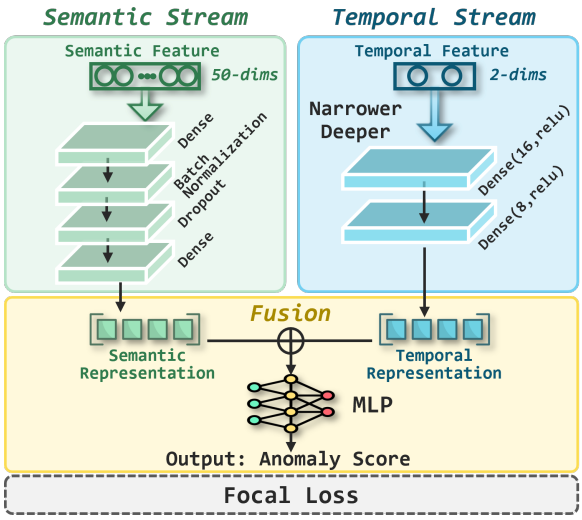


Figure 7. Architecture of DS-TraceNet.

to distill the most discriminative semantic patterns from the raw embedding space, enabling the model to capture structural deviations in traces that are characteristic of semantic drift anomalies.

3.3.2 Temporal Stream. The Temporal Stream is dedicated to processing the continuous temporal state features of each trace. As described in § 3.1.3, two temporal features are extracted per trace: the Duration Time δ and the Gap Time δ_g. Both features undergo a log transformation to form a compact 2-dimensional temporal feature vector:

$$\mathbf{h}^t = \left[\ln(1+\delta),\ \ln(1+\delta_g)\right] \in \mathbb{R}^{d_t} \tag{2}$$

where $d_t = 2$. Despite its low dimensionality, $\mathbf{h}^t$ carries highly sensitive anomaly signals, as performance regressions and scheduling anomalies are directly manifested as abnormal deviations in duration and gap values.

The Temporal Stream branch processes $\mathbf{h}^t$ through a narrower yet deeper architecture than the Semantic Stream's. This design concentrates the representational capacity on learning fine-grained temporal sensitivity to produce a refined temporal representation $\mathbf{f}^t$, enabling the model to detect subtle latency anomalies that would otherwise be suppressed under a joint training regime with high-dimensional semantic features.

3.3.3 Cross-Modality Fusion and Anomaly Scoring. To evaluate the system execution state, the high-level representations from both streams are fused into a unified cross-modal feature. This late-fusion strategy preserves the independently learned semantic representation $\mathbf{f}^s$ and temporal representation $\mathbf{f}^t$ while preserving the intra-modal discriminative structure and enabling the fusion layer to model inter-modal correlations. The fused feature $\mathbf{F}$ is computed as:

$$\mathbf{F} = \sigma\left(W_f\left[\mathbf{f}^s \parallel \mathbf{f}^t\right] + b_f\right) \tag{3}$$

where $\parallel$ denotes feature concatenation, W_f and b_f are learnable parameters of the fusion layer, and σ denotes the ReLU activation function. This design is motivated by the complementary nature of the two modalities: structural or logical anomalies manifest as semantic drift captured by $\mathbf{f}^s$, while performance regressions manifest as temporal anomalies captured by $\mathbf{f}^t$. The fusion MLP learns operation-conditioned temporal patterns from their joint representations, so lexically similar names alone do not determine the anomaly score. Late fusion ensures that neither modality suppresses the other prior to integration, which is critical given the dimensional asymmetry between the two streams.

The fused representation $\mathbf{F}$ is then fed into a lightweight multi-layer perceptron (MLP) to produce the final anomaly probability:

$$y_{\text{pred}} = \text{MLP}(\mathbf{F}) \in [0,1] \tag{4}$$

A trace is classified as anomalous when y_{pred} exceeds a predefined decision threshold τ, i.e., the predicted label $\hat{y} = \mathbf{1}[y_{\text{pred}} > \tau]$.

3.3.4 Model Optimization and End-to-End Training. Loss Function. A key challenge in real-world trace anomaly detection is the severe class imbalance, as anomalous traces are rare in production environments [21]. Standard cross-entropy loss treats all samples equally, causing the model to be dominated by the majority class and underrepresent anomalies. To address this, we adopt Focal Loss as the training objective:

$$\mathcal{L}_{\text{focal}} = -\alpha\,(1-p_t)^{\gamma}\log(p_t) \tag{5}$$

where p_t denotes the predicted probability of the ground-truth class:

$$p_t = \begin{cases} y_{\text{pred}} & \text{if } y_{\text{true}} = 1 \\ 1 - y_{\text{pred}} & \text{otherwise} \end{cases} \tag{6}$$

α is a class-balancing factor and γ is a focusing parameter that down-weights easy samples, directing training toward hard misclassified examples. We set $\alpha = 0.75$ and $\gamma = 2.0$: the higher α assigns greater weight to the minority anomalous class, while γ suppresses the dominant gradient contribution from easy normal samples without over-penalizing confident correct predictions.

To further mitigate class imbalance at the data level, we apply sample weighting during training, assigning higher weights to anomalous samples so that their gradient contributions are amplified relative to the dominant normal class. The entire DS-TraceNet is trained end-to-end using the Adam optimizer, which provides stable adaptive optimization across heterogeneous components.

3.4 Online Sampling

With the lightweight model trained offline, the online sampling pipeline operates on the device to capture critical data while pruning steady-state redundancy. The pipeline proceeds in three stages: real-time anomaly detection, spatio-temporal event retention, and selective sampling.

3.4.1 Model Inference and Real-Time Anomaly Detection. The trained model is deployed on-device in a lightweight inference runtime and integrated with the trace data workflow. As trace data streams in, the model processes each trace and produces an anomaly score $y_{\text{pred}} \in [0, 1]$. Traces with $y_{\text{pred}} > \tau$ are flagged as anomalous and serve as anchors for the subsequent event retention stage.

3.4.2 STA-ER: Spatio-Temporal Anomaly Event Retention. To achieve aggressive compression of the sampling data while avoiding over-pruning, we need to strike a balance between large-scale steady-state data elimination and the preservation of necessary spatio-temporal context for anomaly analysis. We propose a Spatio-Temporal Anomaly Event Retention (STA-ER) strategy. STA-ER eliminates redundancy along two orthogonal dimensions: the temporal dimension, which determines the time range to retain around a detected anomaly, and the thread dimension, which determines which threads within that range are worth keeping. Together, these two filtering passes maximize data reduction while preserving analytical completeness.

Temporal Dimension: Anomaly-Centric Event Extension. We cannot simply retain isolated anomalous traces, as they are insufficient for downstream anomaly analysis: a single trace lacks the causal context required to reconstruct a complete anomaly scenario [8]. The triggering conditions precede the anomaly, and the propagation effects follow it. Therefore, retaining data at the granularity of isolated traces would cause semantic fragmentation of critical call chains, making end-to-end anomaly chain reconstruction impossible [25, 38]. We must instead elevate the retention granularity to a semantically meaningful unit that captures the full context surrounding each detected anomaly.

We design an anomaly-centric extension mechanism that bidirectionally expands the retention range around each detected anomalous trace to preserve both preceding and subsequent context. Through systematic observation of production system traces, we find that user interactions with the device screen provide natural and semantically coherent segmentation boundaries. Each interaction, a tap, a swipe, or a gesture, generates a series of touch tracepoints scattered across the timeline, and these records cluster into discrete episodes with well-defined onsets and offsets. We therefore define an event E_k as the contiguous time interval bounded by the first and the next touch tracepoint of a single user interaction. For anomalous traces s_t with anomaly scores that exceed a predefined event threshold, the event E_k containing such traces is marked as an *anomalous event* and denoted as a core event E_k^*. This core event E_k^* is retained in full, up to a 10 s cap that prevents a long interaction interval from producing an unboundedly large fragment. This ensures that the full interaction episode surrounding each anomaly is captured without truncation.

Thread Dimension: Relevance-Driven Thread Sampling. A retained event window still contains trace records from hundreds of concurrently scheduled threads, the vast majority of which are irrelevant to performance anomaly analysis. To eliminate this intra-window redundancy, we apply a knowledge-base-guided thread filter to screen the core thread set $\mathcal{T}_{core}$ within the event.

Core Thread Identification. During offline analysis of historical anomaly cases, performance engineers classify all threads by their functional roles on the operating system, and systematically record the indispensable thread groups for diagnosing different performance problems, including underlying kernel threads, multimedia threads, UI rendering threads, network threads, native system service threads, and third-party application threads. This domain knowledge is compiled into a *Core Thread Knowledge Base* ($\mathcal{K}_{thread}$). At online sampling time, threads present in a retained event window are matched against $\mathcal{K}_{thread}$, and only those belonging to the identified $\mathcal{T}_{core}$ are preserved. Threads outside this set are stripped from the retained window, achieving a further reduction in data volume without sacrificing analytical completeness.

3.4.3 Selective Persistence and Cloud Transmission. Only anomalous event windows retained by the Spatio-Temporal Anomaly Event Retention (STA-ER) strategy, after undergoing thread-level pruning, are persisted as core data $\mathcal{D}_{core}$, which is defined by the following decision rule:

$$\mathcal{D}_{core} = \begin{cases} \text{Persist} & \exists s_t \in E_k^* \wedge \text{Thread}(s_t) \in \mathcal{T}_{core} \\ \text{Prune} & \text{otherwise} \end{cases} \tag{7}$$

Data segments representing system steady-state behavior, which fall outside the identified critical ranges, are excluded from dumping. These core data $\mathcal{D}_{core}$ are then uploaded to the cloud for in-depth anomaly analysis and root cause diagnosis. This selective data persistence mechanism alleviates the excessive I/O bandwidth consumption and storage pressure typically associated with full trace collection.

4 Evaluation

To show the effectiveness of CoreCapture, we evaluate it by answering the following questions:

- **RQ1:** How does CoreCapture compare against strong baselines across different scenarios?
- **RQ2:** What is the contribution of the DS-TraceNet to the overall detection performance of CoreCapture?

- **RQ3:** How sensitive is CoreCapture to key hyperparameters?

4.1 Experimental Setup

4.1.1 Datasets. We construct a practical dataset from a real production environment, covering three representative anomaly-prone applications under two typical runtime scenarios (Fling and Animation). The dataset contains 11, 8, and 5 maintenance tickets for *App*_1, *App*_2, and *App*_3, respectively, in the Fling scenario, and 8 tickets for each application in the Animation scenario. In total, the evaluation covers two user-perceived anomaly scenarios across three mainstream applications, forming six application–scenario combinations with 48 independent maintenance tickets, 43 anomaly categories, 870 events, and 661,480 traces. The two scenarios were selected because our production data indicates that users perceive their degradation most directly, and the tickets are evenly divided between them (24 per scenario). These maintenance tickets are used to generate raw system trace files. We divide the dataset into training and test splits and conduct statistics at two granularities: traces and events. Due to variations in execution time and system states across traces, the number of traces and events varies across samples. Table 2 shows the distribution of normal and anomalous traces, which exhibits severe class imbalance typical of production environments, where anomalous instances are rare. Table 3 further quantifies the total number of normal and abnormal events in the training and test sets.

4.1.2 Implementation Details. During experiments, we generally use default hyperparameters from the corresponding anomaly detection algorithm papers. If the training or detection performance on a particular dataset is suboptimal, hyperparameters are adjusted on the training split to optimize the results. All experiments are performed on a server configured with one Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz, 192 GB of RAM, and two NVIDIA A30 GPUs, each with 24 GB of GPU memory.

Table 2. Statistics of normal and anomalous trace-level samples in training and test sets across different scenarios.

	Fling		Animation	
	Normal	Abnormal	Normal	Abnormal
Train	109088	478	133795	117
Test	242952	188	174756	106

4.1.3 Baseline.

We select three trace-based anomaly detection methods as our baselines.

TraceAnomaly [21] encodes call paths and response times into service trace vectors, then applies a deep Bayesian network with posterior flow to model normal trace patterns and score anomalies.

PUTraceAD [51] models each trace as a span-level causal graph with semantic, temporal, and status-code features, learning representations via a graph attention network and training with the nnPU estimator using only a small set of labeled anomalies.

GTrace [43] clusters traces by shared substructures and trains a group-wise VAE that decouples structural and latency information through a structure-to-latency prediction objective, with caching and merged-graph batching for inference efficiency.

Table 3. Statistics of normal and anomalous event-level interaction episodes in training and test sets across different scenarios.

	Fling		Animation	
	Normal	Abnormal	Normal	Abnormal
Train	213	33	166	21
Test	208	23	178	28

4.2 Evaluation Metrics

To comprehensively validate the effectiveness of our method, we evaluate its performance at two granularities: the *trace level* and the *event level*.

At the *trace level*, we compare the core anomaly detection capability of our model against baseline methods. At this granularity, we use standard classification metrics, including $Precision_s$, $Recall_s$, and F1-score$_s$.

At the *event level*, we assess the effectiveness of the sampling mechanism using event-level recall ($Recall_e$), defined as the proportion of anomalous data retained among all true anomalous interaction episodes. Furthermore, we evaluate the efficiency of data reduction using the Average Compression Ratio (*ACR*). For a given set of M original trace files, let $|T_k|$ denote the size of the k-th complete trace file, and let $|C_k|$ denote the size of its persistently retained data. *ACR* is formulated as:

$$ACR = \frac{1}{M} \sum_{k=1}^{M} \frac{|C_k|}{|T_k|} \tag{8}$$

A smaller *ACR* reflects higher data reduction efficiency, demonstrating the capability to suppress steady-state noise while preserving high-fidelity diagnostic contexts.

4.3 Overall Performance (RQ1)

4.3.1 Anomaly Detection Performance.

To evaluate the effectiveness of CoreCapture, we compare it against three baselines at both trace and event levels across the Fling and Animation scenarios.

Table 4. Anomaly detection performance at trace level.

Method	Fling			Animation		
	P	R	F1	P	R	F1
CoreCapture	0.792	0.809	0.800	0.662	0.868	0.751
TraceAnomaly	0.223	0.245	0.234	0.069	0.108	0.084
PUTraceAD	0.699	0.580	0.634	1.000	0.047	0.090
GTrace	0.078	0.158	0.104	0.091	0.135	0.108

Table 5. Event-level Recall and Average Compression Ratio.

Method	Fling		Animation	
	Recall	ACR	Recall	ACR
CoreCapture	0.947	0.277	0.964	0.358
TraceAnomaly	0.696	0.300	0.714	0.375
PUTraceAD	0.682	0.248	0.071	0.029
GTrace	0.739	0.253	0.429	0.227

Trace-Level Analysis. As shown in Table 4, CoreCapture consistently achieves the best Recall and F1 across all scenarios. In Fling, it attains the highest F1 of **0.800**, outperforming the strongest baseline, PUTraceAD (0.634), by a large margin. In Animation, CoreCapture maintains a Recall of **0.868** and F1 of **0.751**, whereas all baselines deteriorate substantially, with F1 below 0.11. In particular, PUTraceAD achieves perfect Precision but an extremely low Recall (0.047), indicating an overly conservative detection strategy. TraceAnomaly and GTrace exhibit extremely low precision and recall across both scenarios. They cannot effectively extract information to capture intricate, cross-type anomaly patterns. Overall, CoreCapture avoids the precision–recall imbalance observed in the baselines and preserves substantially more anomalous traces without inflating false alarms. **Event-Level Analysis.** Table 5 demonstrates the superior anomaly observability of our method. It captures **94.7%** and **96.4%** of actual performance anomalies in Fling and Animation, respectively. This ensures that even if certain localized execution traces are marginally missed at the micro-level, the overarching user-interaction episode containing the anomaly cascade is still retained. Crucially, this comprehensive anomaly coverage is achieved alongside a highly efficient data reduction footprint. CoreCapture achieves an ACR of **0.277** in Fling and **0.358** in Animation. In Animation, PUTraceAD and GTrace report misleadingly low ACR values of 0.029 and 0.227, respectively, while their $Recall_e$ values drop catastrophically to 0.071 and 0.429. This shows that they miss many anomalous events, which is unacceptable in production environments. CoreCapture successfully strikes an optimal, production-ready balance between data reduction and contextually high-fidelity diagnosis.

4.3.2 Execution Efficiency.

Table 6. Comparison of detection speed and memory peak.

Method	Detection Speed (Traces/s)		Memory (GB)
	Fling	Animation	
CoreCapture	13763.50	11657.47	0.18
TraceAnomaly	70.77	68.59	45.11
PUTraceAD	490.58	434.98	15.39
GTrace	7210.42	3803.36	2.32

To evaluate the deployability of methods in resource-constrained edge environments, we compare execution efficiency and runtime overhead. Table 6 shows detection speed and peak memory usage under identical settings, where CoreCapture achieves the highest throughput with the lowest memory consumption in both scenarios.

Detection Speed Analysis. CoreCapture exhibits a massive performance advantage in throughput compared to the baseline methods. Specifically, in Fling, we process **13,763.50 traces/s**, outperforming GTrace by 1.91×, PUTraceAD by 28.1×, and TraceAnomaly by 194.5×. In Animation, we sustain **11,657.47 traces/s**, exceeding GTrace by 3.07×, PUTraceAD by 26.8×, and TraceAnomaly by 170.0×. Even compared to GTrace, which accelerates inference through caching and merged-graph batching, CoreCapture consistently achieves better performance in both scenarios.

Memory Footprint. CoreCapture maintains an exceptionally low peak memory usage of only **0.18 GB** in the server-side benchmark (Table 6), which is 12.9× lower than GTrace, 85.5× lower than PUTraceAD, and 250.6× lower than TraceAnomaly. In percentage terms, this corresponds to 92.2%–99.6% less memory than baselines. The incremental on-device memory overhead is 13–15 MB (§ 5.2). In contrast, the baselines incur substantial memory overhead due to their memory-intensive representations, such as graph construction and high-dimensional event modeling. By keeping both compute latency and memory overhead within minimal bounds, CoreCapture proves its architectural capability to operate as an online, low-impact agent on resource-constrained edge devices.

4.4 Ablation Study (RQ2)

DS-TraceNet's design rests on two choices: keeping the two modalities, and processing them in separate streams. We ablate each in turn.

The first removes one modality at a time, isolating what each stream contributes (Table 7).

In both scenarios, dropping the semantic stream cuts trace-level F1 by about 55–61%, while dropping the temporal stream cuts it by about 10–12%; event-level recall drops in both cases, and CoreCapture keeps the best or comparable ACR.

Table 7. Ablation of semantic and temporal modalities.

Scenario	Variant	Trace level			$Recall_e$	ACR
		P	R	F1		
Fling	Semantic-only	0.449	0.303	0.362	0.737	0.288
	Temporal-only	0.665	0.793	0.723	0.714	0.307
	CoreCapture	**0.792**	**0.809**	**0.800**	**0.947**	**0.277**
Animation	Semantic-only	0.295	0.292	0.294	0.829	0.360
	Temporal-only	0.593	0.755	0.664	0.764	0.430
	CoreCapture	**0.662**	**0.868**	**0.751**	**0.964**	**0.358**

Table 8. Ablation study: impact of DS-TraceNet on detection and sampling performance.

Scenario	Level	Metric	CoreCapture	w/o DS-TraceNet
Fling	Event	$Recall_e$	**0.947**	0.842
		ACR	**0.277**	0.281
	Trace	$Recall_s$	**0.809**	0.628
		F1	**0.800**	0.669
Animation	Event	$Recall_e$	**0.964**	0.536
		ACR	0.358	**0.125**
	Trace	$Recall_s$	**0.868**	0.500
		F1	**0.751**	0.642

The second replaces the dual streams with a single joint-optimized stream to isolate the architectural benefit. Table 8 reports the performance variations across the Fling and Animation scenarios at both the trace and event levels, contrasting the full dual-stream DS-TraceNet against this single-stream variant, which retains both modalities.

Trace-Level Analysis. As shown in Table 8, the single-stream variant clearly degrades detection accuracy. Recall is heavily affected, dropping from 0.809 to 0.628 in Fling and from 0.868 to 0.500 in Animation. Concurrently, the F1 falls by 0.131 and 0.109 across the two scenarios, respectively. Removing the decoupled architecture forces joint optimization of semantic embeddings and low-dimensional temporal signals, where the former dominates gradient propagation and suppresses the latter, hindering effective detection of heterogeneous anomaly patterns. **Event-Level Analysis.** As shown in Table 8, the single-stream variant in Fling degrades $Recall_e$ from 0.947 to 0.842 while increasing the ACR from 0.277 to 0.281, indicating a simultaneous reduction in anomaly coverage and pruning efficiency. Conversely, in Animation, the ablated configuration yields a lower ACR but at the cost of a catastrophic $Recall_e$ drop from 0.964 to 0.536. This extreme reduction comes from model detection collapse: the single-stream network misclassifies massive anomalies as steady-states, causing aggressive over-pruning that discards over 42.8% of critical diagnostic data. This trade-off shows that DS-TraceNet is necessary for balancing data reduction with anomaly observability.

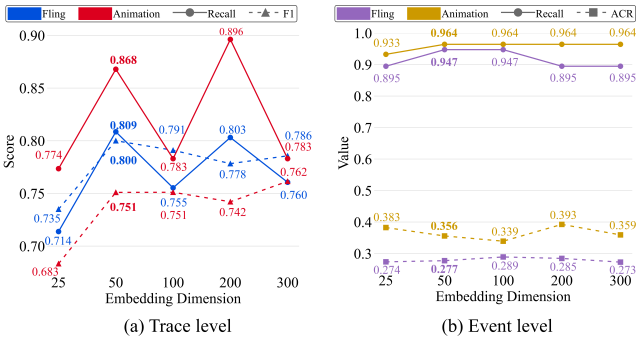


Figure 8. Impact of embedding dimension.

4.5 Hyperparameters Sensitivity (RQ3)

4.5.1 Embedding Dimension. Since 50, 100, and 300 are commonly adopted embedding dimensions in prior studies [1, 45], we evaluate dimensions of 25, 50, 100, 200, and 300 across both operational granularities. As shown in Figure 8, at the trace level, both Recall and F1 achieve their best results at 50 dimensions (0.809 and 0.800, respectively) in Fling, whereas Animation exhibits only marginal gains with larger dimensions. Considering the limited improvement and the additional computational overhead introduced by high-dimensional embeddings, we select 50 dimensions. At the event level, dimensions below 50 lead to noticeably lower Recall, indicating insufficient feature representation. Increasing the embedding size beyond 50 yields only marginal improvements in metrics, while incurring additional memory and computational overhead. In particular, 50 dimensions already achieve near-optimal Recall and maintain competitive ACR. Therefore, we select 50 as the embedding dimension to balance diagnostic performance and model efficiency.

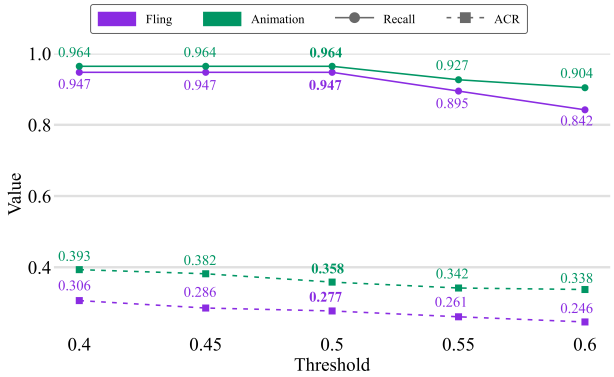


Figure 9. Comparison of Recall and ACR across different event thresholds.

4.5.2 Event Threshold. We evaluate the sensitivity of CoreCapture to the event threshold: an event is classified as anomalous only when its anomaly score exceeds the predefined threshold. Figure 9 shows a clear trade-off between Recall and ACR. As the threshold increases, ACR consistently decreases, indicating improved compression efficiency, whereas Recall remains stable at lower thresholds and begins to decline noticeably beyond 0.5. Therefore, thresholds below 0.5 preserve detection capability but yield limited compression, while larger thresholds further reduce ACR at the cost of missed anomalies. Considering the dual objectives of high Recall and low ACR, we select 0.5 as the final event threshold.

5 Deployment

CoreCapture has been deployed for over two months on internal devices with the beta system managed by a major mobile OS provider, where it samples trace data in the background. Validation has covered multiple mainstream devices running relatively recent systems, including lower-end devices.

5.1 Workflow of Deployment

Figure 10 shows the three-stage deployment workflow: *Preparation*, *Integration*, and *Sampling*.

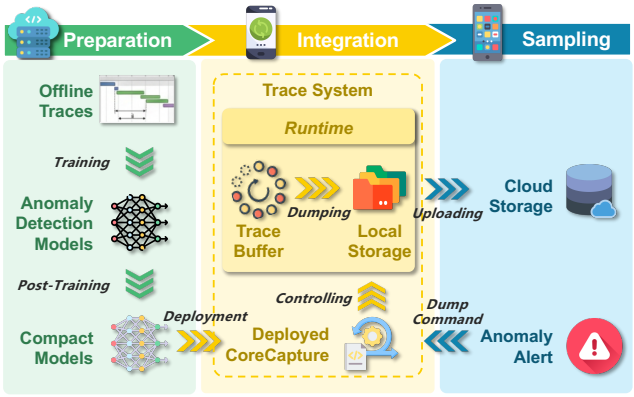


Figure 10. Deployment workflow of CoreCapture.

Preparation. Historical offline traces are used to train the anomaly detection models described in § 3.3. Post-training optimization, including quantization, converts the trained DS-TraceNet and task-specific FastText artifacts into compact models suitable for mobile deployment. The training procedure stays entirely offline; only the lightweight inference artifacts are placed on the device. Training DS-TraceNet completes in under 10 minutes on an NVIDIA A30 GPU; retraining is needed only when the tracepoint vocabulary evolves.

Integration. CoreCapture is deployed directly inside the Trace System as a module, rather than as a separate daemon. By residing in the system partition, it maintains stable system privileges across reboots and controls the dumping path from the runtime Trace Buffer to Local Storage, augmenting the existing trace infrastructure without replacing the underlying recording mechanism. CoreCapture adds no new tracing source or permission and exposes no application-visible channel, preserving the security boundary.

Sampling. At runtime, the monitoring stack triggers dump commands as described in § 2.1. CoreCapture processes the dumped data through its online inference and STA-ER pipeline (§ 3.4), retaining only anomaly-relevant event windows for subsequent upload to the cloud.

5.2 Details of Deployment

Deploying an anomaly detection model on a production mobile device imposes resource constraints that differ sharply from the server-side benchmarking environment used in § 4. We measured CoreCapture's runtime overhead on actual production devices using the OS's built-in system profiling tools, across multiple dump cycles. On a typical trace dump of approximately 144 MB, CoreCapture's inference adds a CPU overhead of 4–5% and a memory overhead of 13–15 MB during each alert-triggered dump (Figure 10). Each invocation consumes approximately 0.11 mAh, and at about 10 alert-triggered dumps per device per day in performance-anomaly scenarios, the daily total is approximately 1.1 mAh. To avoid affecting foreground apps, the inference engine runs on the device's efficiency cores and leverages NPU acceleration where available. The baseline methods evaluated in § 4 (TraceAnomaly, PUTraceAD, and GTrace) cannot be deployed on-device at all: their memory footprints reach 2.3–45 GB (Table 6) and their inference latencies are incompatible with real-time processing on a mobile device.

5.3 Effectiveness of Deployment

CoreCapture is the only method among those evaluated that satisfies the resource constraints of production mobile deployment, and it benefits both device-side sampling and cloud-side analysis.

5.3.1 Device-Side Benefits. The existing trace infrastructure enforces a daily upload budget of approximately 400 MB, shared across all diagnostic components. Given the 400 MB daily budget and an average dump size of 144 MB, each application is effectively limited to at most three trace dumps per day. Under these constraints, anomalies occurring after the quota is exhausted are never recorded.

With CoreCapture's data reduction (approximately 70% on average, as shown in § 4), the volume of retained data per dump is reduced by 2.8–3.6×, allowing the system to sample more traces within the same daily budget.

5.3.2 Cloud-Side Benefits. Under the previous full-dump approach, each trace dump contained substantial steady-state redundancy; engineers typically spent more than 10 minutes per anomalous trace to locate the anomalous segments, based on internal diagnostic workflow statistics. With CoreCapture, the uploaded data has already been filtered to retain only anomaly-relevant events with their diagnostic context, reducing the per-trace analysis time to less than 2 minutes.

5.4 Case Studies

We use a representative production anomaly to qualitatively compare three data sampling strategies (Figure 11).

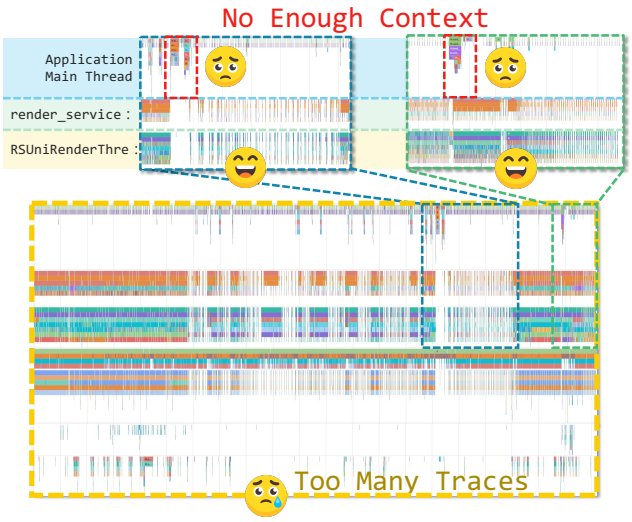


Figure 11. Case study: comparing three data retention strategies on a representative production anomaly.

Before CoreCapture, the only available strategy was a full dump of the entire trace buffer. As shown in the yellow region at the bottom of Figure 11, this preserves the complete raw trace at 149.7 MB, including long steady-state intervals and many unrelated threads, and engineers must manually sift through this data to locate the anomaly-relevant region.

An alternative is to use only the anomaly detection results from DS-TraceNet and sample the flagged anomalous traces without the surrounding context. As shown in the red regions at the top, the isolated snippet lacks the execution preceding and following the anomaly on related threads.

CoreCapture addresses this by completing the AATS task at the event level. DS-TraceNet identifies the anomalous trace as the sampling anchor; STA-ER then samples the surrounding context by expanding temporally to the complete interaction event and filtering to the core threads needed for analysis (§ 3.4). The result is 15.1 MB, a compression ratio of 0.1 relative to the full dump, shown as the two retained windows at the top of Figure 11.

As Figure 3 illustrates, if only the anomalous `Incremental Layout` anchor were retained, the preceding `Spring Render` and the subsequent abnormal `Hardware Sync`, `Cache Parse`, and `Property Animation` would be absent from the retained context; STA-ER preserves causal context, so that downstream diagnosis can draw on both the preceding execution and the downstream effects. The retained context feeds the cloud-side analyses of Figure 1: root cause analysis, performance bottleneck analysis, reliability assessment, and software evolution.

6 Related Work

6.1 On-device System Data Analytics

For system call data, E-Spion [27], HADES-IoT [4] and Zeus-IoT [34] adopt various whitelists and intercept system calls to implement security verification on resource-constrained terminals, while MIDAS [46] and Celdrán et al. [6] identify anomalies by analyzing system call sequences and extracting behavioral features. Although Draco [37], Goyal et al. [12] and R-CAID [13] achieve strong detection performance, their high computational and memory overhead as well as dependence on dedicated hardware hinder their usage on mobile and other end devices. In the field of system log analytics, LightLog [41] reduces computational load via lightweight temporal convolutional networks; DistilLog [29] combines semantic compression, knowledge distillation and lightweight recurrent networks; MetaLog [48] supports cross-system detection with meta-learning, yet its excessive resource consumption also prevents on-device deployment. Limited by computation, memory and power, these methods fail to support real-time on-device inference.

6.2 Trace-based Anomaly Detection

Existing trace-based anomaly detection methods can be broadly categorized into three groups. VAE-based methods, such as TraceAnomaly [21], CRISP [53], TraceVAE [44], and GTrace [43], learn latent distributions or reconstruction patterns of normal traces for anomaly detection. GNN-based methods, including BSDG [36], PUTraceAD [51], TraceCRL [50], and TraceGra [7], model traces as invocation graphs to capture service dependencies and interactions. LSTM-based methods, such as Multimodal LSTM [28], treat traces as temporal sequences and use recurrent models. While these approaches have achieved strong performance in offline trace analysis for microservice traces, they are difficult to deploy on production devices due to strict resource constraints and high computational and memory overhead. Moreover, they often either focus on limited temporal signals or weakly couple semantic and temporal information, making it challenging to capture complex cross-type anomaly patterns.

7 Conclusion

Mobile performance anomalies are difficult to reproduce, making every missed trace dump a permanent diagnostic loss. We formalized this setting as Anomaly-Aware Trace Sampling (AATS) and presented CoreCapture, an on-device system that transforms trace collection from a quota-limited, best-effort snapshot mechanism into an anomaly-aware sampling pipeline. CoreCapture addresses the two fundamental challenges of this task through DS-TraceNet, a dual-stream detection model that resolves the tension between semantic and temporal anomaly signals under lightweight constraints, and STA-ER, a spatio-temporal retention mechanism that retains diagnostic context by expanding anomalies to complete interaction-event windows while pruning irrelevant threads. Large-scale evaluation on production traces demonstrates that CoreCapture reduces per-dump trace volume by approximately 70% while preserving approximately 95% of anomaly-bearing events, achieves detection throughput exceeding 11,000 traces per second with only 0.18 GB peak memory in the server-side benchmark, and is the only method among those evaluated that satisfies the resource constraints of production mobile deployment. These results confirm that aggressive data reduction and high anomaly coverage are not inherently in tension. CoreCapture enables the system to collect more dumps within the same daily budget, directly increasing the number of anomalies that can be captured and diagnosed.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), the Fundamental Research Funds for the Central Universities (XXX-63253249), and the Tianjin Key Research and Development Program (Grant No. 25YFYFFG00690).

References

[1] Deepak Suresh Asudani, Naresh Kumar Nagwani, and Pradeep Singh. 2023. Impact of word embedding models on text analytics in deep learning environment: a review. *Artificial intelligence review* 56, 9 (2023), 10345–10425.

[2] Tristan Bilot, Baoxiang Jiang, Zefeng Li, Nour El Madhoun, Khaldoun Al Agha, Anis Zouaoui, and Thomas Pasquier. 2025. Sometimes Simpler is Better: A Comprehensive Analysis of {State-of-the-Art}{Provenance-Based} Intrusion Detection Systems. In *34th USENIX Security Symposium (USENIX Security 25)*. 7193–7212.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the association for computational linguistics* 5 (2017), 135–146.

[4] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. 2019. HADES-IoT: A practical host-based anomaly detection system for IoT devices. In *Proceedings of the 2019 ACM Asia conference on computer and communications security*. 479–484.

[5] Elie Bursztein, Marina Zhang, Owen Vallis, Xinyu Jia, and Alexey Kurakin. 2023. Retvec: Resilient and efficient text vectorizer. *Advances in Neural Information Processing Systems* 36 (2023), 60902–60921.

[6] Alberto Huertas Celdrán, Pedro Miguel Sánchez Sánchez, Chao Feng, Gérôme Bovet, Gregorio Martínez Pérez, and Burkhard Stiller. 2022. Privacy-preserving and syscall-based intrusion detection system for IoT spectrum sensors affected by data falsification attacks. *IEEE Internet of Things Journal* 10, 10 (2022), 8408–8415.

[7] Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. 2023. TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications* 204 (2023), 109–117.

[8] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.

[9] Yutong Chen, Ronglai Zuo, Fangyun Wei, Yu Wu, Shujie Liu, and Brian Mak. 2022. Two-stream network for sign language recognition and translation. *Advances in neural information processing systems* 35 (2022), 17043–17056.

[10] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, et al. 2020. Towards intelligent incident management: why we need it and how we make it. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1487–1497.

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 3–18.

[12] Akul Goyal, Xueyuan Han, Gang Wang, and Adam Bates. 2023. Sometimes, you aren't what you do: Mimicry attacks against provenance graph host intrusion detection systems. In *30th Network and Distributed System Security Symposium*.

[13] Akul Goyal, Gang Wang, and Adam Bates. 2024. R-caid: Embedding root cause analysis within provenance-based intrusion detection. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3515–3532.

[14] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[15] Haiyu Huang, Cheng Chen, Kunyi Chen, Pengfei Chen, Guangba Yu, Zilong He, Yilun Wang, Huxing Zhang, and Qi Zhou. 2025. Mint: Cost-Efficient Tracing with All Requests Collection via Commonality and Variability Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 683–697.

[16] Haiyu Huang, Xiaoyu Zhang, Pengfei Chen, Zilong He, Zhiming Chen, Guangba Yu, Hongyang Chen, and Chen Sun. 2024. Trastrainer: Adaptive sampling for distributed traces with system runtime state. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 473–493.

[17] Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. 2024. More apps, faster hot-launch on mobile devices via fore/background-aware GC-swap co-design. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 654–670.

[18] Changlong Li, Yu Liang, Rachata Ausavarungnirun, Zongwei Zhu, Liang Shi, and Chuan Jason Xue. 2023. Ice: Collaborating memory and process management for user experience on resource-limited mobile devices. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 79–93.

[19] Wentong Li, Li-Pin Chang, Yu Mao, and Liang Shi. 2025. {PMR}: Fast Application Response via Parallel Memory Reclaim on Mobile

Devices. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1569–1584.

[20] Linux Kernel Documentation. [n. d.]. ftrace - Function Tracer. https://docs.kernel.org/trace/ftrace.html. Accessed: 2026-06-10.

[21] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 48–58.

[22] Sicong Liu, Hao Luo, XiaoChen Li, Yao Li, Bin Guo, Zhiwen Yu, YuZhan Wang, Ke Ma, YaSan Ding, and Yuan Yao. 2024. AdaKnife: Flexible DNN offloading for inference acceleration on heterogeneous mobile devices. *IEEE Transactions on Mobile Computing* 24, 2 (2024), 736–748.

[23] Yuan-Yuan Liu, Chia-Yang Lin, Yin-Cheng Wang, Wei-Chih Hsu, Chien-Hui Lin, and Kuo-Hao Chang. 2025. An empirical study on enhancing wafer quality: Integrating big data and AI in virtual metrology for thin-film processing. *Advanced Engineering Informatics* 68 (2025), 103747.

[24] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance Debugging with {In-Production},{Just-In-Time} Method Tracing on Android. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 787–803.

[25] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. 2020. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1176–1189.

[26] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[27] Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. 2019. E-spion: A system-level intrusion detection system for iot devices. In *Proceedings of the 2019 ACM Asia conference on computer and communications security*. 493–500.

[28] Sasho Nedelkoski, Jorge Cardoso, and Odej Kao. 2019. Anomaly detection from system tracing data using multimodal deep learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 179–186.

[29] Huy-Trung Nguyen, Lam-Vien Nguyen, Van-Hoang Le, Hongyu Zhang, and Manh-Trung Le. 2024. Efficient log-based anomaly detection with knowledge distillation. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 578–589.

[30] OpenHarmony Documentation. [n. d.]. Introduction to HiTraceMeter. https://gitee.com/openharmony/docs/blob/HEAD/en/application-dev/dfx/hitracemeter-intro.md. Accessed: 2026-06-11.

[31] Xiaokang Peng, Yake Wei, Andong Deng, Dong Wang, and Di Hu. 2022. Balanced multimodal learning via on-the-fly gradient modulation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8238–8247.

[32] Perfetto Documentation. [n. d.]. ATrace: Android system and app trace events. https://perfetto.dev/docs/data-sources/atrace. Accessed: 2026-06-11.

[33] Zifeng Qi, Haiguang Chen, Mingxing Liu, and JianJiang Wang. 2022. Bie—Modernism with Cultural Calculations in Multiple Dimensions. In *International Conference on Human-Computer Interaction*. Springer, 120–136.

[34] Alireza Roshandel and Manuel Egele. 2025. Zeus-IoT: Comprehensive Code Signing to Prevent IoT Device Weaponization. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*. IEEE, 845–857.

[35] Anil Sharma and Suresh Kumar. 2023. Ontology-based semantic retrieval of documents using Word2vec model. *Data & knowledge engineering* 144 (2023), 102110.

[36] Kuanzhi Shi, Jing Li, Yuecan Liu, Yuzhu Chang, and Xuyang Li. 2022. BSDG: Anomaly Detection of Microservice Trace Based on Dual Graph Convolutional Neural Network. In *International Conference on Service-Oriented Computing*. Springer, 171–185.

[37] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and operating system support for system call security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 42–57.

[38] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. 2021. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 419–429.

[39] Jiawei Wang, Nian Liu, Arnau Casadevall-Saiz, Yutao Liu, Diogo Behrens, Ming Fu, Ning Jia, Hermann Härtig, and Haibo Chen. 2025. Enabling Efficient Mobile Tracing with BTrace. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–338.

[40] Weiyao Wang, Du Tran, and Matt Feiszli. 2020. What makes training multi-modal classification networks hard?. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12695–12705.

[41] Zumin Wang, Jiyu Tian, Hui Fang, Liming Chen, and Jing Qin. 2022. LightLog: A lightweight temporal convolutional network for log anomaly detection on the edge. *Computer Networks* 203 (2022), 108616.

[42] Yuanpei Wu, Dong Du, Chao Xu, Yubin Xia, Ming Fu, Binyu Zang, and Haibo Chen. 2025. D-vsync: Decoupled rendering and displaying for smartphone graphics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 326–341.

[43] Zhe Xie, Changhua Pei, Wanxue Li, Huai Jiang, Liangfei Su, Jianhui Li, Gaogang Xie, and Dan Pei. 2023. From point-wise to group-wise: A fast and accurate microservice trace anomaly detection approach. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1739–1749.

[44] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Huai Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. 2023. Unsupervised anomaly detection on microservice traces through graph vae. In *Proceedings of the ACM Web Conference 2023*. 2874–2884.

[45] Jingyun Xu, Jiayuan Xie, Yi Cai, Zehang Lin, Ho-Fung Leung, Qing Li, and Tat-Seng Chua. 2023. Context-aware dynamic word embeddings for aspect term extraction. *IEEE Transactions on Affective Computing* 15, 1 (2023), 144–156.

[46] Yiwen Xu, Zijing Yin, Yiwei Hou, Jianzhong Liu, and Yu Jiang. 2022. MIDAS: safeguarding iot devices against malware via real-time behavior auditing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4373–4384.

[47] Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, and Pinjia He. 2024. Deep learning or classical machine learning? an empirical study on log-based anomaly detection. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.

[48] Chenyangguang Zhang, Tong Jia, Guopeng Shen, Pinyan Zhu, and Ying Li. 2024. Metalog: Generalizable cross-system anomaly detection from logs with meta-learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[49] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *Proceedings of the 44th international conference on software*

engineering. 623–634.

[50] Chenxi Zhang, Xin Peng, Tong Zhou, Chaofeng Sha, Zhenghui Yan, Yiru Chen, and Hong Yang. 2022. TraceCRL: contrastive representation learning for microservice trace analysis. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 1221–1232.

[51] Ke Zhang, Chenxi Zhang, Xin Peng, and Chaofeng Sha. 2022. Putracead: Trace anomaly detection with partial labels based on gnn and pu learning. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 239–250.

[52] Shenglin Zhang, Zhongjie Pan, Heng Liu, Pengxiang Jin, Yongqian Sun, Qianyu Ouyang, Jiaju Wang, Xueying Jia, Yuzhi Zhang, Hui Yang, et al. 2023. Efficient and robust trace anomaly detection for large-scale microservice systems. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 69–79.

[53] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 655–672.

[54] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen. 2022. A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities. *Proc. IEEE* 110, 3 (2022), 334–354.

[55] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[56] Xuelin Zhu, Jiuxin Cao, Jiawei Ge, Weijia Liu, and Bo Liu. 2022. Two-stream transformer for multi-label image classification. In *Proceedings of the 30th ACM international conference on multimedia*. 3598–3607.

[57] Yao Zhu, LI Zhenyuan, Yangyang Wei, and Shouling Ji. 2025. The Case for Learned Provenance-based System Behavior Baseline. In *Forty-second International Conference on Machine Learning*.