2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-9302-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISSRE66568.2025.00041

Integrating GraphSAGE and Mamba for Self-Supervised Spatio-Temporal Fault Detection in Microservice Systems

Shenglin Zhang[†‖], Yingke Li[†], Jianjin Tang[†], Chenyu Zhao[†], Wenwei Gu[†], Yongqian Sun[†¶*]
Dan Pei[§]
[†]Nankai University, zhangsl@nankai.edu.cn, {2120220705, 1120250461, 1120250460}@mail.nankai.edu.cn,
gwwdaxue@gmail.com, sunyongqian@nankai.edu.cn
[‖]Haihe Laboratory of Information Technology Application Innovation
[¶]Tianjin Key Laboratory of Software Experience and Human Computer Interaction
[§]Tsinghua University, peidan@tsinghua.edu.cn

***Abstract*—Monitoring and fault detection in microservice systems is crucial for ensuring service stability. However, most existing methods either rely heavily on labeled data or fail to model complex spatial-temporal dependencies across services. To address these limitations, we propose ChronoSage, a spatio-temporal fault detection framework that integrates GraphSAGE and Mamba for unified graph-stream-based modeling. GraphSAGE captures the evolving topological structures by aggregating neighborhood features, while Mamba efficiently models long-range temporal dependencies through a selective state-space mechanism. We adopt a self-supervised training strategy to reduce label dependence and enhance generalization. Experiments on two real-world datasets demonstrate that ChronoSage achieves superior accuracy and efficiency compared to state-of-art baselines, such as ART and Eadro. The results validate ChronoSage's ability to support system-level fault detection in dynamic microservice environments, achieving an F1-score of 0.872 on D1 and 0.972 on D2, surpassing all compared methods.**

***Index Terms*—Microservice Systems; Fault Detection; Multi-modal Monitoring Data; Spatio-Temporal Model**

I. Introduction

With the development of modern software architecture, microservice systems have become the mainstream choice by significantly improving the elasticity, scalability and development efficiency of the system [1]. Microservice systems contain a large number of instances and complex dependencies. Once a fault occurs, it may degrade service performance, affect the user experience, and even cause substantial losses. A typical example occurred in June 2023, when Amazon Web Services (AWS) experienced a significant outage in its US-EAST-1 region. The incident, lasting over two hours, was attributed to a fault in the capacity management subsystem of AWS Lambda, leading to elevated error rates and latencies across more than 100 services, including API Gateway and the AWS Management Console. This disruption impacted numerous organizations, such as news agencies, airlines, and government services, highlighting the cascading effects of faults within microservice architectures [2]. Therefore, timely and accurate detection of system faults has become one of the key tasks for maintaining service quality.

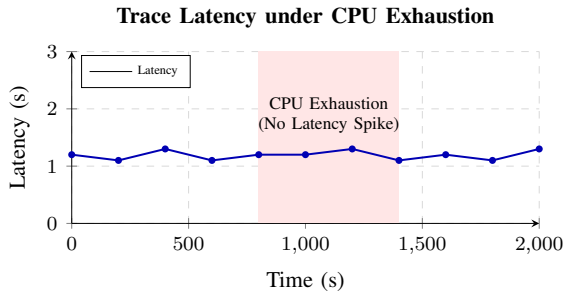
We make the following distinctions between anomalies and faults in microservice systems. Anomalies refer to unexpected patterns or behaviors in system data that deviate from the norm [3]. Faults are actual defects or malfunctions within the system that can cause service degradation or outages [4]. Although anomalies do not invariably precipitate faults, faults are consistently accompanied by a series of anomalous events. We adopt "anomaly" when discussing the detection of irregular patterns in single-modal data and "fault" when referring to the identification and diagnosis of system-level issues.

Many efforts [5]–[7] have been devoted to anomaly detection, dealing with only one type of monitor data. This limits their accuracy and generalization in real-world scenarios. Metric data can reflect resource usage, but lack contextual information. Logs provide execution traces, but do not capture service dependencies. Traces reveal invocation paths but miss the contextual and performance details of the anomalies. For example, as shown in Figure 1, under the fault injection of CPU exhaustion, the trace-based latency remains stable and does not exhibit obvious anomalous behaviors. In contrast, the metrics (specifically CPU usage) show abrupt spikes and sustained high usage, clearly reflecting the anomaly. Single-modal monitoring data typically cannot provide a comprehensive and accurate characterization of system behavior. Consequently, relying solely on the anomalous behavior of a single type of monitoring data is likely to lead to false alarms in fault detection.

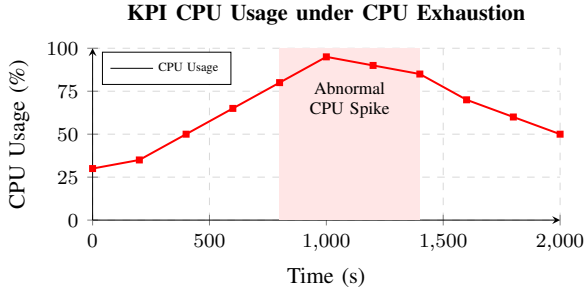
We aim to leverage multi-modal monitoring data to address the problem of fault detection in microservice systems. Numerous approaches have been proposed, which can be categorized into supervised and unsupervised methods. Supervised approaches rely on labeled data to learn mappings between features and anomalies. Early approaches often used SVM or random forests on statistical features of metrics, while more recent methods like TimesNet [5] utilize deep models with

* Yongqian Sun is the corresponding author.

979-8-3503-9302-6/25/$31.00 ©2025 IEEE
DOI 10.1109/ISSRE66568.2025.00041



(a) Trace data shows no anomaly despite resource fault.



(b) KPI captures resource anomaly clearly.

Fig. 1: **Case Study:** Trace fails to reflect CPU-related fault, while KPI reveals it.

multi-scale convolutions to extract periodic patterns from time series. However, these methods often ignore the rich contextual information embedded in service call chains. Unsupervised methods operate without labeled data. BARO [16] combines multivariate Bayesian online change point detection with a nonparametric statistical test, RobustScorer, enabling label-free analysis of multivariate time-series metrics. Nevertheless, it ignores logs and traces, may struggle with data exhibiting strong parametric distributions, and lacks modeling of service interaction patterns due to the absence of graph structures. Hades [12] integrates metrics and logs and uses isolation forests to identify outliers based on deviations from normal data distributions. However, it cannot capture inter-service dependencies, limiting its ability to detect global anomalies.

GNNs have gained attention for their natural ability to model service invocation relationships. Eadro [10] uses Graph Attention Networks (GATs) and multimodal data fusion for system-level classification but depends on labeled data and assumes a static graph structure. MSTGAD [15] uses GATs to model spatial dependencies among microservices and as a self-supervised learning method, it leverages a small number of labeled traces. However, the spatial graph structure remains static, based on predefined service dependencies, and does not adapt to dynamic or evolving system behaviors. This reliance on a fixed topology may limit the model's generalization capability in real-world environments where service interactions are often incomplete, noisy, or subject to change. TraceVAE [6] employs a variational autoencoder with graph structures to jointly model traces and temporal features, enhancing fault scoring. ART [14] unifies metrics, logs, and traces into time series, constructs dynamic graphs per minute, and combines Transformer, GRU [21], and GraphSAGE [29] to capture spatio-temporal features. However, ART still struggles with long sequence degradation and misses fine-grained fault patterns. Nevertheless, GNN-based methods still face two key challenges:

- **Limited ability to model long-term temporal dependencies.** Monitoring data in microservices exhibit strong spatio-temporal coupling, but most existing methods focus on snapshots of the system and fail to capture temporal evolution
- **Heavy reliance on labeled data.** Many supervised methods require manually labeled anomalies, which are rare and costly in real-world systems, hindering their practical deployment.

To address the challenges of system-level fault detection in microservices, we propose ChronoSage (Spatio-temporal Fault Detection model with Mamba and Dynamic Graphs), a novel framework that integrates both spatial and temporal modeling capabilities.

- **Spatio-temporal integration for precise fault detection.** ChronoSage leverages the spatial representation power of GraphSAGE and the temporal modeling efficiency of Mamba. Specifically, it performs embedding computation on dynamic graphs using GraphSAGE to incorporate evolving service interaction patterns into node features. These graph embeddings are then processed by the Mamba model, which captures temporal dependencies through its state space mechanism, enabling accurate analysis of long-range time-series data.
- **Label-efficient learning to reduce annotation cost:** C h ron o S a ge adopts a self-supervised learning strategy to automatically extract meaningful patterns from system data features without requiring manual label. By incorporating the temporal evolution of service interactions and the spatial dependencies between microservices, ChronoSage significantly improves fault detection accuracy compared to traditional unsupervised methods.

Experimental results on two real-world microservice datasets demonstrate the effectiveness of ChronoSage in fault detection. ChronoSage achieves superior performance, reaching an F1-score of 0.872 on dataset D1 and 0.972 on dataset D2. These results validate ChronoSage's ability to support highly accurate system-level fault detection in dynamic microservice environments.

II. Background

A. Microservice Systems and Multimodal Monitoring Data

Microservice architectures decompose complex applications into a collection of small, independently deployable services, each dedicated to a specific business function [10]. The modular design of microservice architectures empowers development teams to work independently and in parallel, thereby accelerating the overall development lifecycle. Communication

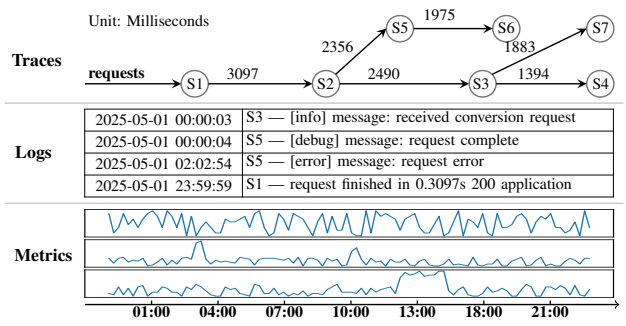


Fig. 2: Multimodel Monitoring Data

between services is achieved through lightweight and efficient protocols such as HTTP/REST or gRPC. Microservice architecture not only enhances development productivity but also significantly improves system scalability, flexibility, and maintainability.

ChronoSage use the concept of a system instance to describe the runtime entities within a microservice system. A system instance may refer to a microservice instance or a host instance. Together, these instances form the underlying infrastructure that supports high availability, scalability, and fault tolerance in the system.

In microservice systems, such monitoring data typically includes three primary modalities: metrics, logs, and traces [17], as shown in Figure 2.

- **Metrics.** Metrics represent quantitative measurements of system state over time, often collected at regular intervals by monitoring agents such as Prometheus. Represented as time series data, metrics provide real time insights into service status. For instance, system-level metrics such as CPU usage and memory utilization reflect resource consumption, while service-level metrics such as request throughput and error rate serve as indicators of microservice health. A spike in HTTP 5xx error rates, for example, may suggest code-level bugs or faults in dependent services.
  Metrics anomaly detection in time series data is fundamental for identifying system faults, involving detecting sudden spikes or drops, trend shifts, and level changes in key performance indicators (KPIs) [30].
- **Logs.** Logs capture system states and behaviors through fields such as timestamps, node identifiers, and messages that combine predefined structures with values generated during execution. These predefined structures, established by developers, ensure consistency in event semantics across the system. In contrast, the values generated during execution reflect specific runtime information, including execution duration, status codes, and output formats. For instance, in Figure 2, the log entry ”S1—request finished in 0.3097s 200 application” illustrates this composition. Here, ”S1—request finished in” represents the standardized structure indicating the completion of a request, while ”0.3097s,” ”200,” and ”application” are runtime-specific values denoting processing time, HTTP status code, and response content type, respectively. In this context, ”S1” refers to a particular microservice instance within the system. This structured logging approach facilitates consistent formatting and efficient analysis of system events.
  Traditional log anomaly detection methods often rely on keyword matching (e.g., ”ERROR”, ”fail”), which can lead to false positives due to network jitter or user login faults, etc. Recent approaches adopt structured pipelines, such as log parsing, feature extraction, and anomaly detection, to more accurately identify abnormal behaviors [31].
- **Traces.** Traces capture the end-to-end execution path of a request as it traverses through multiple microservices. A single user request may trigger a cascade of service invocations. Figure 2 presents a concrete trace example, where each node (*S1–S7*) denotes a microservice instance involved in processing the request. The numeric labels (e.g., 3097, 2490) represent the latency in milliseconds associated with each inter-service call. The overall execution path starts from *S1*, proceeds through *S2* and *S3*, and then branches into multiple downstream services (*S4*, *S5*, *S6*, *S7*). The trace structure demonstrates both sequential and parallel service invocations.
  In microservice architectures, trace-based fault detection focuses on identifying anomalies in service call sequences, such as sudden increases in response times or abnormal call paths [32].

B. Mamba

Time series forecasting aims to predict future values based on historical temporal dependencies and patterns. It plays a critical role in domains such as industrial monitoring, energy management, financial analysis, and cloud operations. Traditional deep learning approaches, such as RNNs [33], LSTMs [34], 1D-CNNs [35], and Transformers [18], have achieved notable success in time series forecasting field. However, they face significant challenges in modeling long-range dependencies and scaling to long sequences. Unlike RNNs and LSTMs, which suffer from vanishing gradients due to their recursive nature [19], Transformers incur quadratic complexity with respect to sequence length, limiting their efficiency when handling very long time series [20].

To address these limitations, ChronoSage adopt Mamba [36], a recently proposed Selective State Space Model (SSM), as our backbone for time series prediction. Mamba combines high computational efficiency with strong long-range modeling capability, making it well-suited for large-scale time-series tasks.

Mamba achieves linear time complexity ($O(N)$) while effectively modeling long-term dependencies. Its core innovations include:

- Dynamic state transitions: Mamba replaces fixed state matrices with learnable, input-conditioned transitions, enabling adaptive modeling across time steps.

- Hardware-aware parallelism: Through a convolutional representation, Mamba supports parallel computation while maintaining low memory usage even for long sequences.
- Selective information retention: Mamba avoids the step-by-step hidden state updates of RNNs, using a more flexible mechanism to preserve important information over time.

Formally, given input x_t and hidden state h_{t-1}, the Mamba update is defined as:

$$h_t = A(x_t)h_{t-1} + B(x_t)x_t + C(x_t) \tag{1}$$

where $A(\cdot)$,$B(\cdot)$,$C(\cdot)$ are learnable functions representing state transitions, input weights, and bias terms, respectively. The output is computed via $\hat{y}_t = D(h_t)$, where $D(\cdot)$ maps the hidden state to the prediction space.

Compared with alternative long-sequence models such as GRU or Transformer, Mamba offers the following advantages specific to our context:

- Efficiency for long sequences: Mamba scales linearly with sequence length, allowing it to model 1-minute resolution data over extended time windows without incurring memory bottlenecks.
- Robustness to input noise and length variation: Mamba does not rely on positional encodings, making it naturally suited for handling variable-length and uneven monitoring sequences.
- Better inductive bias for system dynamics: The state-space formulation aligns well with evolving system behaviors in microservices, enabling the model to track performance shifts and fault precursors over time.

These properties make Mamba particularly well-suited as the temporal modeling module in ChronoSage, allowing the framework to capture system evolution patterns efficiently and detect subtle deviations that precede faults in real-world deployments.

C. SSL

Self-Supervised Learning (SSL) is a machine learning paradigm that reduces reliance on manually labeled data by learning from data itself. It involves designing pre-training tasks that create pseudo labels using data's inherent structures or properties, enabling models to learn general feature representations from unlabeled data [22]. These tasks generate pseudo labels automatically, compelling the model to extract meaningful features from the data to address the problem. Common SSL tasks include reconstruction tasks [23], prediction tasks (e.g., language modeling [24], image inpainting [25]), generative adversarial networks [26], and contrastive learning [27]. Compared to traditional supervised learning, SSL significantly lowers annotation costs. By employing diverse pre-training tasks, it extracts more generalizable features from massive unlabeled data. These features possess strong transferability, allowing rapid adaptation to various downstream tasks and providing robust initial models for them.

D. GraphSAGE

GraphSAGE is a graph neural network framework designed to efficiently generate node embeddings, particularly in large-scale and dynamic graph settings [28]. Unlike traditional methods that rely on the full graph structure, GraphSAGE learns node representations by sampling and aggregating information from a node's local neighborhood, significantly reducing computational complexity and enabling scalable training. Such characteristics makes it especially well-suited for real-time applications and evolving environments such as microservice systems [29].

Compared to other popular graph neural networks, GraphSAGE offers greater flexibility and scalability. Graph Convolutional Networks (GCNs) [37] rely on normalized Laplacian matrices to perform weighted aggregation over all neighbors in a static graph. While effective for small-scale and static graphs, GCNs operate under a transductive learning setting and require access to the entire graph structure, making them inefficient and less adaptable to large or dynamic graphs, particularly when new nodes are introduced. GATs [11], on the other hand, incorporate an attention mechanism during neighborhood aggregation, assigning different weights to different neighbors, allowing GATs to adapt to heterogeneous and highly dynamic graph structures. However, the attention mechanism introduces significant computational overhead, leading to higher training costs on large-scale graphs.

In contrast, GraphSAGE supports inductive learning, enabling generalization to previously unseen nodes without retraining the entire model. Its sampling-based design balances expressiveness with efficiency, making it particularly advantageous for fault detection and root cause analysis in large-scale, evolving systems.

E. Problem Definition

Fault detection is crucial for preventing system faults and ensuring stable operation, aiming to discover abnormal system status.Anomalies often indicate potential system faults. In this paper, we focus on system-level fault detection based on multimodal monitoring data collected at equal-space timestamps. The problem is formally defined as follows.

As shown in Figure 2, metrics are multivariate time series that monitor hardware, system, and various services. Logs record system runtime behavior as semi-structured text. A trace is made up of spans, each corresponding to an invocation. In addition, traces come with duration, status code, and other annotations. ChronoSage transform them into time series as detailed in section III-A, for every instance i at time t, the feature vector is extracted.

The goal of fault detection is to assess whether a given feature vector $F^{(i)}$ corresponds to an abnormal state at timestamp t. For each time step t, our model computes an *fault score* $s_t \in [0, 1]$, indicating the likelihood that the input $F^{(i)}$ is anomalous. A dynamic threshold θ is used for decision-making: if $s_t > \theta$, system at the t time point is classified as an fault. The ground truth label vector $\mathbf{y} \in \mathbb{R}^n$ consists

of binary values, where 0 denotes a normal instance and 1 indicates an fault.

III. Approach

In this section, we introduce ChronoSage, a multimodal system-level fault detection framework with GraphSAGE and Mamba. The framework aims to address the challenges mentioned above and is composed of four main components: (1) multimodal feature extraction that preprocesses multimodal data to form a comprehensive feature set for each service instance(2)graph stream construction that builds the dynamic graphs from the raw system data, (3) a multimodal spatio-temporal representation learning model that captures structural and temporal features, and (4) an fault scoring and detection module that uses prediction error and a dual-threshold strategy to determine abnormal nodes. The overall framework is illustrated in Figure 3.

To construct the input for ChronoSage, we extract and integrate multiple sources of monitoring data. Metric data (e.g., CPU usage, memory, I/O throughput) are resampled to 1-minute granularity using nearest-neighbor interpolation. Logs are parsed using the Drain algorithm to identify template patterns, and the occurrence frequencies of high-variance templates are computed per instance per minute. Trace data is converted into statistical features, including the average latency, request count, and proportions of different HTTP status code classes (2xx, 4xx, 5xx). Each node in the system-level graph represents a microservice instance and is associated with a feature vector combining metric, log, and trace statistics. A graph is constructed at every time step based on the service call relationships observed in that minute, resulting in a dynamic graph stream. A sequence of $w = 60$ such graphs is used as the model input, forming the basis for capturing both spatial and temporal dependencies.

A. Multimodal Feature Extraction

1) Multimodal Serialization: ChronoSage utilize three types of system monitoring data: trace, log, and metric. All three modalities are organized into a time series at the granularity of one minute, where each timestamp corresponds to a graph snapshot. Each node in the graph represents the aggregation of service instances deployed on the same machine.

For logs, ChronoSage apply Drain [38] to parse unstructured logs into structured event counts, representing the frequency of each log key per service instance per minute. For metrics, ChronoSage use linear interpolation to fill in missing values and align the data with the trace and log modalities. For trace data, ChronoSage extract statistical features such as mean, max, min, and standard deviation of span latency, response status code, and span count, and aggregate them by the callee service instance at each time interval.

2) Feature Extraction: After serialization, each service instance i is represented as a feature sequence $H^{(i)} = [h_1^{(i)}, h_2^{(i)}, ..., h_T^{(i)}]$. For each modality, ChronoSage apply sliding window Z-score normalization as follows:

$$\hat{H}^{(i)}_{\text{modal}}(t) = \frac{H^{(i)}_{\text{modal}}(t) - \mu_\omega(t)}{\sigma_\omega(t)} \tag{2}$$

where $\mu_\omega(t)$ and $\sigma_\omega(t)$ represent the mean and standard deviation computed over a sliding window of length ω ending at time t.

Finally, ChronoSage concatenate the normalized features from all three modalities for each node:

$$F^{(i)} = \left(F^{(i)}_{\text{trace}} \middle\| F^{(i)}_{\log} \middle\| F^{(i)}_{\text{metric}}\right) \tag{3}$$

This multimodal representation allows us to integrate heterogeneous monitoring signals into a unified feature space for downstream modeling.

B. Graph Stream Construction

Given the multimodal time-series features of each node, ChronoSage construct a graph stream $G = \{G_{t-w+1}, ..., G_t\}$ where w is the window size. Each graph snapshot $G_t = (V, E_t, F_t)$ at time t consists of a fixed node set V (all service instances), a dynamic edge set E_t, and node features F_t from the previous step.

The edge set E_t consists of two types of edges: (1) invocation edges extracted from trace data that reflect runtime interactions between service instances at time t, and (2) deployment topology edges, which represent fixed infrastructure-level dependencies and remain unchanged across time. Combining these two types edges ensures that the constructed graphs capture both dynamic behavior and static system structure.

The graph stream models the evolving system behavior, enabling the detection of anomalies in both spatial and temporal dimensions.

C. Spatio-Temporal Modeling

To model the evolving graph stream, ChronoSage propose a spatial-temporal representation learning module composed of GraphSAGE and Mamba. GraphSAGE captures the spatial structure in each snapshot by aggregating neighborhood features, while Mamba learns temporal dependencies across the dynamic graph sequence.

1) Spatial Modeling with GraphSAGE: ChronoSage use GraphSAGE [29] to learn structural representations from each snapshot G_t. For each node $v \in V$, the embedding is updated by aggregating its neighbors' features. ChronoSage adopt the mean aggregator defined as:

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}^{(k)}\left(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}\right)\right) \tag{4}$$

Here, $h_v^{(k)}$ is the k-th layer representation of node v, $\mathcal{N}(v)$ denotes the neighbors of v, and $W^{(k)}$ is a trainable weight matrix. ChronoSage use two GraphSAGE layers and apply layer normalization to stabilize training. The output is a spatially encoded feature vector for each node at each time step.

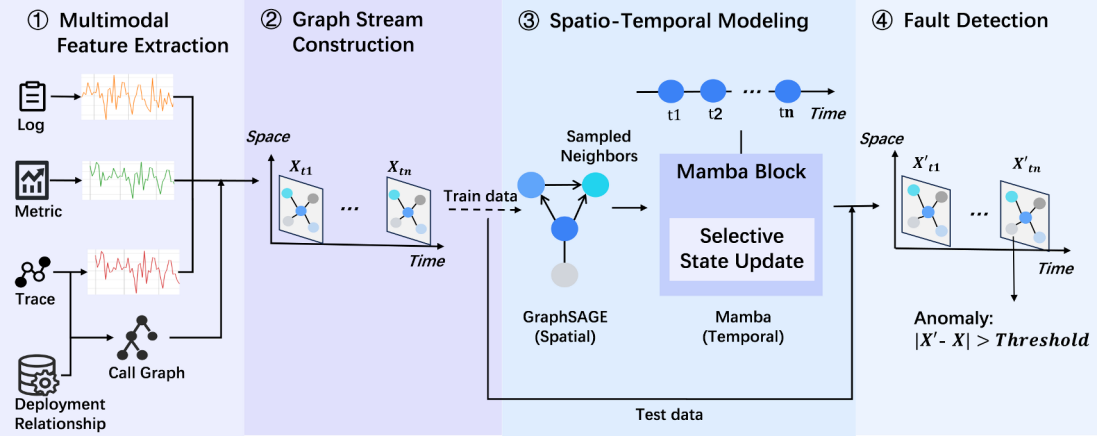


Fig. 3: Overview of the ChronoSage framework.

2) Temporal Modeling with Mamba: To effectively capture long-range temporal dependencies across the evolving sequence of graph snapshots, ChronoSage adopt **Mamba** [36], a selective state space model (SSM) designed for efficient long-sequence modeling with linear time and space complexity. Specifically, for each node, ChronoSage organize its spatial embeddings over the past w time steps into a sequential input $\{u_1, u_2, \ldots, u_w\}$, which is then fed into the Mamba module.

Mamba models the input sequence using a parameterized state space model, formalized as:

$$\frac{d}{dt}x(t) = Ax(t) + Bu(t), \quad y(t) = Cx(t) + Du(t), \tag{5}$$

where $x(t)$ denotes the latent state, $u(t)$ is the input, and $y(t)$ is the output. After discretization, the update becomes:

$$x_{k+1} = \tilde{A}x_k + \tilde{B}u_k, \quad y_k = \tilde{C}x_k + \tilde{D}u_k. \tag{6}$$

In practice, Mamba efficiently computes the output using a convolution-like operation:

$$y_k = \sum_{i=0}^{L-1} K[i] \cdot u_{k-i} + u_k \odot \Delta_k, \tag{7}$$

where $K[i]$ is a state-dependent kernel and $\Delta_k = \text{MLP}(u_k)$ is a dynamic input gate learned from the input itself. This formulation allows Mamba to selectively emphasize informative patterns while preserving temporal alignment.

ChronoSage organize the spatial embeddings of each node over the past w time steps into a sequence and feed it into Mamba, which outputs a prediction for the next time step:

$$\hat{F}_{t+1}^{(v)} = \text{Mamba}(F_{t-w+1}^{(v)}, \ldots, F_t^{(v)}) \tag{8}$$

The training objective is to minimize the prediction error between the predicted and actual feature vectors using mean squared error (MSE) loss:

$$\mathcal{L} = \frac{1}{|V|} \sum_{v \in V} \left\| \hat{F}_{t+1}^{(v)} - F_{t+1}^{(v)} \right\|_2^2 \tag{9}$$

ChronoSage use early stopping and gradient clipping to ensure stable convergence of the model.

D. Fault Detection

In the online phase, the trained model is used to make predictions for each node's feature vector at the next time step, and calculate the prediction error:

$$\Delta_t^{(v)} = \left\| \hat{F}_{t+1}^{(v)} - F_{t+1}^{(v)} \right\|_2 \tag{10}$$

ChronoSage then apply a two-stage thresholding strategy to identify anomalous nodes.

First, ChronoSage use the Interquartile Range (IQR) method to filter out extreme outliers:

$$(Q1 - k \cdot IQR,\ Q3 + k \cdot IQR) \tag{11}$$

where $IQR = Q3 - Q1$, and k is a hyperparameter.

Second, ChronoSage apply the Peaks Over Threshold (POT) method based on Extreme Value Theory (EVT) [39] to fit the tail of the error distribution and set an adaptive threshold. A node is labeled as anomalous if its prediction error exceeds the threshold.

The two-stage strategy allows us to filter out trivial fluctuations and focus on significant anomalies while maintaining robustness and adaptability to different systems.

IV. Evaluation

In this section, we conduct a comprehensive evaluation of ChronoSage using two real-world datasets collected from large-scale microservice systems. The goal is to validate the model's effectiveness, robustness, and efficiency in system-level fault detection tasks. Our experimental study aims to answer the following research questions (RQs):

- **RQ1:** How does ChronoSage perform in terms of fault detection accuracy compared to state-of-the-art methods?
- **RQ2:** How efficient is ChronoSage in both offline training and online detection, especially under real-time constraints?

- **RQ3:** What is the contribution of each core component (GraphSAGE and Mamba) to the overall performance of ChronoSage?
- **RQ4:** How sensitive is ChronoSage to different hyperparameter settings?

A. Experimental Setup

1) Dataset: We conduct experiments on two real-world datasets to evaluate the effectiveness of the proposed ChronoSage framework. Since ChronoSage adopts a SSL strategy and performs time-series prediction, we partition each dataset chronologically—using the earlier portion for training and the latter portion for testing. Detailed statistics of the datasets are summarized in Table I .

TABLE I: Detailed Statistics of Experimental Datasets

Dataset	#Inst.	#Norm.	#Fault	Type	Count
D1	46	3,979	155	Traces	44,858,388
				Logs	66,648,685
				Metrics	20,917,746
D2	18	12,805	281	Traces	214,337,882
				Logs	21,356,870
				Metrics	12,871,809

- **D1**: This dataset is collected from a simulated e-commerce system built on a microservice architecture. The deployment environment fully replicates a real-world cloud infrastructure used in production scenarios. The system consists of 46 running instances, including 40 microservice components and 6 virtual machine (VM) nodes. Each microservice instance is instrumented with container-level monitoring probes, while VMs collect system-level performance metrics in parallel. All faults in the dataset originate from real incidents in production environments and are reproduced via a controlled fault replay mechanism. With the support of a professional operations team, multiple rounds of fault injection experiments were conducted in May 2022, and each fault was annotated with root cause labels based on standardized diagnostic procedures.
  For D1, which includes complete trace logs, we directly construct dynamic interaction graphs by combining deployment topology with fine-grained service call relationships extracted from trace data. These graphs reflect both the static structure and dynamic interactions among services, enabling more accurate spatio-temporal modeling.
- **D2**: This dataset is sourced from the digital management system of a leading commercial bank. The system architecture consists of 18 core component instances, including microservice clusters, web server clusters, application servers, database clusters, and a Docker-based resource pool. Fault events were extracted from system logs recorded between January and June 2021. Two senior site reliability engineers independently performed root cause analysis and annotation using a double-blind labeling protocol, followed by cross-validation to ensure consistency and accuracy of the labels.
  Unlike D1, D2 lacks conventional microservice-level trace data. Its trace logs are more coarse-grained and infrastructure-oriented, providing only partial call paths among components (e.g., CMDB ID, span ID, parent ID, trace ID, and timestamp) without a fully structured service mesh. To address this limitation, we adopt a hybrid graph construction strategy: within each time window, we infer service interactions by identifying valid parent-child links from the trace metadata and augment the resulting structure with the deployment topology to compensate for missing direct call information. This enables us to generate approximate dynamic graphs even under partial observability.While the resulting graphs in D2 are less granular than those in D1, this self-constructed approach allows us to retain meaningful structural patterns necessary for spatial representation. More importantly, this strategy ensures the model can generalize across heterogeneous systems with varying data completeness, supporting consistent and fair evaluation across both datasets.

2) Experimental Environment: The proposed method is implemented using the PyTorch framework, and all experiments are conducted on a Linux server equipped with the following hardware configuration: dual Intel® Xeon® Gold 5218 processors (16 cores, 32 threads, base frequency 2.30 GHz), two NVIDIA® Tesla® V100S GPUs (each with 32 GB memory), and 192 GB of DDR4 RAM (with approximately 187 GiB usable). The Python version used is 3.8.20, and key dependencies include PyTorch 2.1.0, scikit-learn 1.3.2, and DGL 2.1.0.

a) Evaluation Metrics: In fault detection, the prediction outcomes can be categorized into four types: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). Specifically, TP refers to samples that are truly anomalous and correctly detected as anomalies; FP refers to normal samples that are incorrectly predicted as anomalies; TN denotes normal samples that are correctly identified as normal; and FN refers to anomalous samples that are missed by the model. Based on these outcomes, we compute two key evaluation metrics: **Precision** and **Recall**. Precision measures the proportion of correctly predicted anomalies among all samples predicted as anomalous, and a higher precision indicates a lower false positive rate. Recall measures the proportion of actual anomalies that are correctly identified, and a higher recall indicates a lower false negative rate. To balance the trade-off between precision and recall, we also use the **F1-score**, which is the harmonic mean of the two and serves as a comprehensive metric for evaluating fault detection

performance. The formulas are defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{12}$$

$$Recall = \frac{TP}{TP + FN} \tag{13}$$

$$F1\text{-}score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{14}$$

B. RQ1: Overall Performance of ChronoSage

To comprehensively evaluate the performance of ChronoSage on fault detection tasks, we compare it with seven state-of-the-art methods across different modalities. All baseline methods are implemented using their officially reported best hyperparameter settings to ensure fairness and result reliability. We conduct comparative experiments to evaluate the accuracy of our proposed method against baseline approaches on both D1 and D2 datasets. The accuracy results are summarized in Table II.

TABLE II: Accuracy Comparison Of Different Anomaly Detection Methods On Public Datasets

method	D1			D2		
	Precision	Recall	F1	Precision	Recall	F1
ChronoSage	**0.904**	0.842	**0.872**	**0.945**	**1.000**	**0.972**
ART	0.700	0.778	0.737	0.487	0.982	0.651
MSTGAD	0.562	0.336	0.421	0.416	0.684	0.517
Eadro	0.425	**0.946**	0.586	0.767	0.935	0.842
BARO	0.481	0.299	0.369	0.547	0.427	0.480
Hades	0.553	0.657	0.601	0.506	0.719	0.594
TimesNet	0.074	0.170	0.103	0.814	0.737	0.763
TraceVAE	0.706	0.534	0.609	0.378	1.000	0.549

The following analysis highlights the performance differences among the compared methods:

For the metric-based method TimesNet,it is more appropriate for tasks involving single-variable, periodic time series with minimal structural complexity, but its effectiveness diminishes in environments with intricate microservice architectures. TimesNet transforms time series data into 2D tensors to capture periodic patterns, but fails to consider the graph-structured dependencies among microservices, which are critical in modern distributed systems. This limitation explains its significantly better performance on D2 compared to D1, where D1 features richer interactions among microservices. Moreover, since TimesNet relies solely on metrics, it lacks contextual signals from other modalities, making it difficult to determine system-level anomalies—it can only assess whether a specific instance is abnormal at a particular time. BARO shows inferior performance in complex scenarios due to its limited data modality and modeling capacity. Its reliance on metrics alone fails to capture the full system context, while the lack of structural modeling overlooks dependencies between services. Moreover, its nonparametric nature, though flexible, may lead to suboptimal detection on data with clear distributional assumptions.

For the trace-based method TraceVAE, it uses a graph variational autoencoder to separately encode structural and temporal features of service traces. However, it is limited by its single-modality input and relies solely on reconstruction error, which can be sensitive to the sparsity and high dimensionality of trace data.

Among the multimodal methods, we compare against Eadro MSTGAD and ART. Eadro shows significantly better performance on D2 than single-modality baselines, underscoring the value of leveraging heterogeneous data sources. It uses Dilated Causal Convolution (DCC) to capture temporal dependencies and Graph Attention Networks (GAT) to model service call graphs. However, these two components are designed as independent modules, lacking deeper integration between temporal and structural modeling.

MSTGAD constructs a static spatial graph based on predefined microservice dependencies, where each node represents a service and edges capture fixed inter-service relationships. Temporal dependencies are modeled using dynamic temporal graphs across time windows. While trace data provides node-level features, it is not utilized for dynamic graph construction—every trace shares the same static graph topology. This limits the model's adaptability to evolving or incomplete system structures. Moreover, relying on GRU for temporal modeling may hinder performance in long-range dependency scenarios, and the fixed spatial graph restricts its generalization to systems with unknown or frequently changing topologies.

ART also incorporates metrics, logs, and traces, and adopts a similar architectural design as ours—using graph neural networks followed by temporal sequence modeling. It performs better on D1 due to the denser microservice interaction patterns, but its system-level fault score is computed by aggregating instance-level scores, which may obscure fine-grained local anomalies. Furthermore, ART struggles with long sequence prediction, and its performance tends to degrade as the sequence length increases due to limited capability in modeling long-range dependencies.

C. RQ2: Contribution of GraphSAGE and Mamba

To assess the contributions of the two core components—GraphSAGE and Mamba—we construct four ablated variants of our model under consistent preprocessing, training, and evaluation settings. The variants are defined as follows:

1) **C1**:Replaces GraphSAGE with GAT for spatial encoding.
2) **C2**: Replaces GraphSAGE with GCN.
3) **C3**: Replaces Mamba with Transformer for temporal modeling.
4) **C4**: Replaces Mamba with GRU.

The results, shown in Table III, demonstrate that the full ChronoSage model consistently outperforms all ablated versions on both D1 and D2 datasets. We analyze the performance gaps as follows:

C1 (GAT instead of GraphSAGE): While GAT uses attention mechanisms to learn weighted neighbor contributions, it suffers from higher computational complexity, especially on large graphs. Its reliance on first-order neighbors limits its ability to capture broader structural patterns, and the attention

TABLE III: Variants Performance Comparison

Dataset	Variants	Precision	Recall	F1
D1	**ChronoSage**	**0.904**	**0.842**	**0.872**
	C1	0.826	0.554	0.667
	C2	0.837	0.482	0.611
	C3	0.850	0.743	0.793
	C4	0.802	0.791	0.797
D2	**ChronoSage**	**0.945**	**1.000**	**0.972**
	C1	0.867	1.000	0.927
	C2	0.857	0.920	0.887
	C3	0.886	1.000	0.939
	C4	0.881	0.984	0.929

TABLE IV: Average offline training time (in seconds) and detection time (in 10^{-1} seconds) per anomaly detection run for each method.

Method	D1		D2	
	Offline	Detection	Offline	Detection
ChronoSage	436.783	0.414	368.732	0.698
ART	460.262	1.391	1085.767	0.454
MSTGAD	657.020	10.296	73.82	7.007
Eadro	510.570	0.887	795.416	0.337
Hades	1214.528	0.182	2073.041	0.086
TimesNet	6458.400	1.278	2538.678	0.105
TraceVAE	257777.735	13.493	199.491	0.083

weights can be sensitive to noise. In contrast, GraphSAGE aggregates multi-hop neighbors via sampling, providing more robust and scalable embeddings, particularly suited to complex, dynamic service graphs.

C2 (GCN instead of GraphSAGE): GCN is a classic transductive model that requires access to the full graph during training, making it less suitable for evolving graph structures. It also depends on the graph Laplacian, which increases memory and computation costs for large-scale systems. Moreover, GCN is prone to over-smoothing in deeper layers, causing node representations to become indistinguishable. GraphSAGE avoids these issues with its inductive design and hierarchical neighborhood aggregation.

C3 (Transformer instead of Mamba): Transformer excels at long-range modeling but has quadratic time complexity $O(n^2)$, which becomes prohibitive for long sequences. It also requires additional components like positional encoding to capture temporal order. In contrast, Mamba is a state-space-based model optimized for efficient long-sequence processing with linear complexity. Its architecture enables better scalability, robustness to short input lengths, and minimal preprocessing.

C4 (GRU instead of Mamba): GRU is effective in many sequence modeling tasks but struggles with long-term dependencies due to vanishing or exploding gradients. It also performs sequential updates, limiting parallelism. Mamba overcomes these limitations by using continuous-time state space modeling, which enhances long-range dependency capture and allows faster, parallel computation. Furthermore, Mamba's noise filtering during state updates improves robustness compared to GRU.

D. RQ3: Efficiency of ChronoSage

We record and compare the runtime performance of all methods, as shown in Table IV. For detection time, we simulate an online detection scenario by calculating the time required for each method to process one-minute intervals, in order to analyze their computational complexity. The results demonstrate that our method is capable of detecting anomalies within a very short time window.

In addition, BARO is a purely mathematical approach and does not follow the same modeling paradigm as the other deep learning-based methods. Therefore, it is not directly comparable in terms of performance metrics. The time values reported in the table represent the average inference time spent on each individual time series within the dataset. Specifically, BARO's inference times respectively on D1 and D2 are 483.99 and 1977.569 (in units of 10^{-1} seconds). In comparison, ChronoSage requires less time for both offline training and online detection than other multimodal baselines such as ART and Eadro, and also significantly outperforms single-modality methods. The exceptionally short runtime of TraceVAE on the D2 dataset is attributed to the fact that it only relies on trace data, and the trace structure in D2 is relatively simple.

This study investigates the impact of four key hyperparameters on model performance. Figure4 illustrates how the F1-score varies under different configurations.

State Dimension d_{state} : This parameter controls the size of the model's internal representation space. A small state size (e.g., 8) may limit the model's capacity to capture complex patterns, leading to underfitting. Conversely, larger values (e.g., 32 or 64) may increase model expressiveness but also raise the risk of overfitting and training instability. On our datasets, a moderate setting of 16 strikes a good balance between capacity and generalization.

Convolution Dimension d_{conv}: This parameter affects the model's ability to extract local temporal features. While smaller values (e.g., 3) may not capture sufficient context, overly large ones (e.g., 5 or 6) can lead to overfitting due to unnecessary complexity. Empirically, setting d_{conv}=4 achieves optimal performance in our experiments.

Sliding Window Size: This defines the length of the input sequence used for prediction. Short windows (e.g., 20 or 40) may not contain enough historical context, while excessively long windows (e.g., 100 or 120) can introduce noise. Notably, performance remains strong even at window size 120, demonstrating Mamba's robustness in modeling long-range dependencies.

Learning Rate: High learning rates (e.g., 0.05 or 0.07) can cause unstable training or convergence to suboptimal solutions. Lower learning rates (e.g., 0.001 or 0.005) enable more stable updates and better convergence, although very small values may slow training or lead to poor local minima. Overall, 0.001–0.005 offers a good trade-off between convergence speed and final model quality.

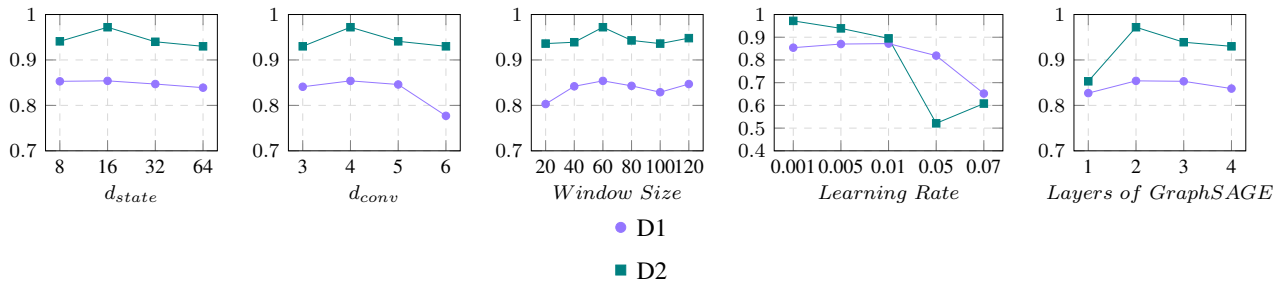


Fig. 4: Parameter Sensitivity Performance Comparison (F1-score)

E. Case Study

Our experimental results demonstrate that ChronoSage excels at detecting a variety of subtle and low-signal faults that are often missed by traditional fault detection methods. Among the successfully detected cases, several types of anomalies stand out:

- k8s container write I/O load
- node-level memory consumption
- k8s container CPU load

These fault types are particularly challenging due to their limited visibility in individual monitoring modalities and their tendency to manifest gradually over time.

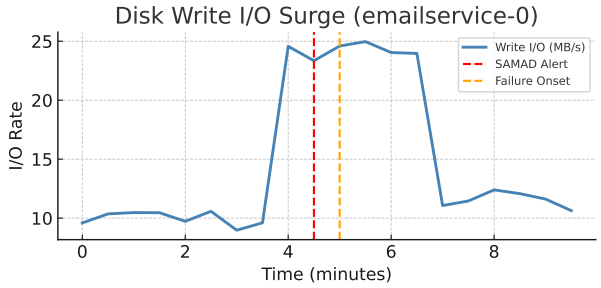


Fig. 5: Disk write I/O anomaly detection in `emailservice-0`. ChronoSage successfully identifies the I/O surge nearly two minutes prior to observable system degradation, despite no visible anomalies in CPU or memory metrics.

For example (Figure 5), *emailservice-0* pod had a disk write surge from unexpected batch logs. Traditional methods missed it (no CPU/memory spikes), but ChronoSage detected it via multi-modal modeling, alerting 2 minutes early (effective for low-signal anomalies). Another example (Figure 6): Memory usage gradually increased across co-located services on *node-4*, but no single service showed extreme values—making it undetectable via threshold-based methods. ChronoSage, however, leveraged GraphSAGE-captured cross-service spatial correlations in dynamic graphs to reveal abnormal memory patterns, highlighting its strength in identifying distributed, correlated anomalies through structural dependency modeling.

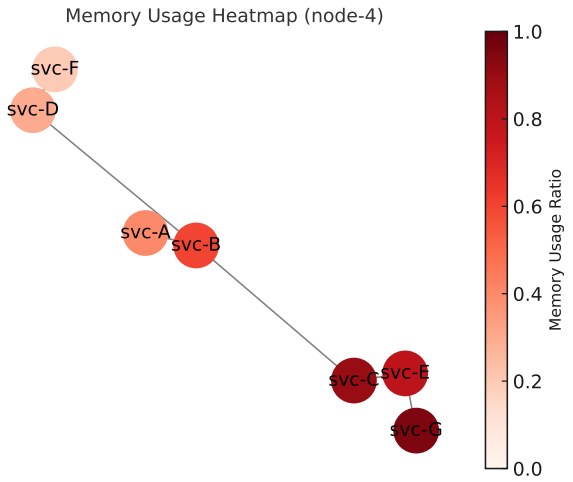


Fig. 6: Memory usage heatmap of services on `node-4`. ChronoSage detects a distributed anomaly by modeling spatial dependencies across services, identifying collective memory pressure even when individual service metrics remain within normal bounds.

F. Threat to Validity

Limited Experiment Scenario Coverage: The current experimental validation primarily relies on the AIOps Challenge datasets (D1 and D2), which exhibit gaps in scale and complexity compared to real-world production environments. Production microservice systems typically involve larger scales, more intricate interactions, and dynamic topologies (e.g., frequent service scaling or architecture evolution). The method's generalization capability in such realistic scenarios remains under-validated, potentially limiting its ability to address complex fault patterns in large-scale distributed systems.

High Dependency on Multimodal Data: The approach heavily depends on joint modeling of three modalities: metrics, logs, and traces. However, in practical scenarios, certain modalities may be missing or of low quality (e.g., incomplete logging configurations, noisy metric data). In cases where data acquisition is constrained, the method's robustness and applicability could degrade, particularly when critical modalities

TABLE V: Comparison of Key Characteristics Among Fault Detection Methods

Method / Reference	Multimodal	Supervised	Spatio-Temporal Modeling	Dynamic Graph Support
TimesNet [5]	No	Yes	No	No
Hades [12]	Yes	Yes	No	No
TraceVAE [6]	No	No	Yes	No
Eadro [10]	Yes	Yes	Yes	No
MSTGAD [15]	Yes	Yes	Yes	No
BARO [16]	No	No	No	No
ART [14]	Yes	Yes	Yes	Yes
ChronoSage (Ours)	Yes	No	Yes	Yes

are unavailable or unreliable.

V. Related Work

Anomaly detection methods have evolved from traditional statistical models to data-driven approaches, with increasing emphasis on deep learning and multimodal fusion. A number of recent works have explored different modalities and model structures for fault detection in microservice systems. As shown in Table V, our proposed method ChronoSage is fully self-supervised and supports dynamic graph construction based on real-time service interactions. Moreover, it explicitly models both spatial dependencies (via GraphSAGE) and temporal evolution (via Mamba), achieving fine-grained instance-level fault detection. We present a detailed analysis and summary of the related methods below.

TimesNet [5] captures multi-scale periodic patterns from metrics (via 2D tensor conversion and multi-kernel convolutions) but lacks log or trace context . Hades [12] fuses metrics and logs with cross-modal attention yet ignores trace dependencies and inter-service interactions . TraceVAE [6] encodes trace structure and temporal features via graph variational autoencoder but is constrained by single-modality input and sensitivity to noisy/sparse data .

In contrast, Eadro [10] and ART [14] use multi-modal data: Eadro processes modalities separately (fuses via GAT for static graphs), ART builds dynamic graphs (uses Transformer/GRU/GraphSAGE). Both are supervised (high annotation cost) and struggle with long-range temporal modeling.

In summary, existing methods either lack full multimodal integration (e.g., TimesNet, Hades, TraceVAE) or suffer from label dependence and weak long-sequence modeling (e.g., Eadro, ART). Our work addresses these gaps by proposing a self-supervised, scalable framework that unifies spatial and temporal learning across metrics, logs, and traces.

VI. Discussion

The proposed framework, ChronoSage, presents a novel solution for system-level anomaly detection in microservice environments by integrating graph neural networks with selective state space modeling. By explicitly modeling both spatial and temporal dependencies, ChronoSage achieves high-fidelity anomaly scoring even in highly dynamic and complex cloud-native systems. Compared with prior methods such as ART and Eadro, ChronoSage reduces reliance on labeled data through a self-supervised training strategy, thus offering better applicability in real-world settings where manual annotation is often costly or infeasible.

One of ChronoSage's notable advantages is its ability to dynamically construct service interaction graphs based on real-time call chain data. In contrast to approaches relying on static topologies, ChronoSage adapts to evolving system structures, improving the robustness of spatial representation. Additionally, the use of Mamba for temporal modeling enables efficient processing of long sequences while mitigating the inefficiencies and memory constraints commonly associated with RNNs and Transformers.

Despite these strengths, ChronoSage still encounters challenges in environments with incomplete or noisy telemetry. For example, missing trace information or inconsistent logging formats may degrade the accuracy of graph construction and feature extraction. Future work may incorporate uncertainty estimation or explore self-supervised learning strategies to mitigate the impact of partial supervision.

From a deployment perspective, ChronoSage features a modular architecture that allows seamless integration into existing AIOps pipelines. Its lightweight design ensures low inference latency, making it viable for real-time fault detection. However, practical deployment may still require fine-tuning of hyperparameters, such as window lengths and normalization settings, to accommodate system-specific behaviors and data characteristics.

Finally, while this work emphasizes technical performance, future research should also explore operational aspects, including interpretability and human-in-the-loop validation. Improving the explainability of model outputs is essential for fostering operator trust and achieving broader adoption in production-scale AIOps solutions.

VII. Conclusion

In this paper, we proposed ChronoSage, a spatio-temporal fault detection framework that integrates GraphSAGE and Mamba for modeling dynamic graphs and long-range temporal dependencies. By leveraging GraphSAGE's inductive neighborhood aggregation and Mamba's efficient state-space modeling, ChronoSage enables end-to-end optimization over multimodal monitoring data. Extensive experiments on two real-world microservice datasets demonstrate that ChronoSage outperforms state-of-the-art baselines in both detection accuracy and computational efficiency. Ablation studies confirm the individual contributions of GraphSAGE and Mamba, and hyperparameter analysis further validates the robustness of our

approach. Overall, ChronoSage offers a scalable and effective solution for real-time system-level fault detection in complex microservice environments.

VIII. Acknowledgement

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), and the Fundamental Research Funds for the Central Universities (XXX-63253249).

References

[1] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube." 583-590.

[2] Amazon Web Services, "Summary of the AWS Lambda Service Event in the Northern Virginia (US-EAST-1) Region," June 13, 2023. [Online]. Available:https://aws.amazon.com/message/12721/

[3] Zh. Zhong, Q. Fan, J. Zhang, M. Ma, Sh. Zhang, Y. Sun, Q. Lin, Y. Zhang, D. Pei. "A Survey of Time Series Anomaly Detection Methods in the AIOps Domain." 10.48550/arXiv.2308.00393. 2023.

[4] N. Trapani, L. Longo."Fault Detection and Diagnosis Methods for Sensors Systems: a Scientific Literature Review." IFAC-PapersOnLine. 56. 1253-1263. 10.1016/j.ifacol.2023.10.1749. 2023.

[5] H. Wu, T. Hu, Y. Liu, H. Zhou, J. Wang, M. Long,"TimesNet: Temporal 2D-Variation Modeling for General Time Series Analysis," arXiv preprint arXiv:2210.02186,2022.

[6] Z. Xie, H. Xu, W. Chen, W. Li, H. Jiang, L. Su et al."Unsupervised anomaly detection on microservice traces through graph vae." In Proceedings of the ACM Web Conference 2023, pp. 2874-2884. 2023.

[7] Y. Yuan, W. Shi, B. Liang, and B. Qin. "An approach to cloud execution failure diagnosis based on exception logs in openstack." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 124-131. IEEE, 2019.

[8] A. Russell-Gilbert, A. Sommers, A. Thompson, L. Cummins, S. Mittal, S. Rahimi, M. Seale, J. Jaboure, T. Arnold, and J. Church. "AAD-LLM: Adaptive Anomaly Detection Using Large Language Models." In 2024 IEEE International Conference on Big Data (BigData), pp. 4194-4203. IEEE, 2024.

[9] W. Guan, J. Cao, S. Qian, J. Gao, and C. Ouyang. "LogLLM: Log-based Anomaly Detection Using Large Language Models." arXiv preprint arXiv:2411.08561 (2024).

[10] C. Lee, T. Yang, Zh. Chen, Y. Su, and Michael R. Lyu. "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data." In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1750-1762. IEEE, 2023.

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. "Graph attention networks." arXiv preprint arXiv:1710.10903 (2017).

[12] C. Lee, T. Yang, Z. Chen, Y. Su, Y. Yang, and Michael R. Lyu. "Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention." In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1724-1736. IEEE, 2023.

[13] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. "The graph neural network model." IEEE transactions on neural networks 20, no. 1 (2008): 61-80.

[14] Y. Sun, B. Shi, M. Mao, M. Ma, S. Xia, S. Zhang, and D. Pei. "ART: A Unified Unsupervised Framework for Incident Management in Microservice Systems." In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 1183-1194. 2024.

[15] J. Huang, Y. Yang, H. Yu, J. Li, and X. Zheng. "Twin graph-based anomaly detection via attentive multi-modal learning for microservice system." In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 66-78. IEEE, 2023.

[16] L. Pham, H. Ha, and H. Zhang. "Baro: Robust root cause analysis for microservices via multivariate bayesian online change point detection." Proceedings of the ACM on Software Engineering 1, no. FSE (2024): 2214-2237.

[17] A. Bakhtin, J. Nyyssölä, Y. Wang, N. Ahmad, K. Ping, M. Esposito, M. Mäntylä, and D. Taibi, "LO2: Microservice API anomaly dataset of logs and metrics," arXiv preprint arXiv:2504.12067, 2025.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[19] P. Le and W. Zuidema. "Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive LSTMs." arXiv preprint arXiv:1603.00423 (2016).

[20] Z. Zhang, Y. Wang, S. Tan, B. Xia, and Y. Luo. "Enhancing Transformer-based models for long sequence time series forecasting via structured matrix." *Neurocomputing*, vol. 625, pp. 129429, 2025. doi:https://doi.org/10.1016/j.neucom.2025.129429

[21] K. Cho, B. V. Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." arXiv preprint arXiv:1406.1078 (2014).

[22] L. Jing and Y. Tian. "Self-supervised visual feature learning with deep neural networks: A survey."IEEE Transactions on Pattern Analysis and Machine Intelligence, 43(11), 4037–4058, 2020.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors." InNeurocomputing: Foundations of Research, pp. 696–699. MIT Press, 1988.

[24] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. "Improving language understanding by generative pre-training."OpenAI preprint, 2018.

[25] D. Pathak, P. Krahenbuhl, J. Donahue, T. Darrell, and A. A. Efros. "Context encoders: Feature learning by inpainting."arXiv preprint arXiv:1604.07379, 2016. https://doi.org/10.48550/arXiv.1604.07379

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative adversarial networks."Communications of the ACM, 63(11), 139–144, 2020. https://doi.org/10.1145/3422622

[27] R. Hadsell, S. Chopra, and Y. LeCun. "Dimensionality reduction by learning an invariant mapping." In2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), Vol. 2, pp. 1735–1742. IEEE, 2006. https://doi.org/10.1109/CVPR.2006.100

[28] W. L. Hamilton, R. Ying, and J. Leskovec. "Representation learning on graphs: Methods and applications." arXiv preprint arXiv:1709.05584, 2017.

[29] W. Hamilton, Zh. Ying, and J. Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems 30 (2017).

[30] Z. Zhong, Q. Fan, J. Zhang, M. Ma, S. Zhang, Y. Sun, Q. Lin, Y. Zhang, and D. Pei. "A survey of time series anomaly detection methods in the AIOps domain." arXiv preprint arXiv:2308.00393, 2023.

[31] M. Landauer, S. Onder, F. Skopik, and M. Wurzenberger. "Deep learning for anomaly detection in log data: A survey."Machine Learning with Applications, 12, 100470, 2023.

[32] J. Soldani and A. Brogi. "Anomaly detection and failure root cause analysis in (micro)service-based cloud applications: A survey."ACM Computing Surveys (CSUR), 55(3), 1–39, 2022.

[33] J. L. Elman"Finding structure in time." Cognitive science 14, no. 2 (1990): 179-211.

[34] A. Graves, and A. Graves. "Long short-term memory." Supervised sequence labelling with recurrent neural networks (2012): 37-45.

[35] G. Giannakakis, E. Trivizakis, M. Tsiknakis, and K. Marias. "A novel multi-kernel 1D convolutional neural network for stress recognition from ECG." In 2019 8th International Conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW), pp. 1-4. IEEE, 2019.

[36] A. Gu, and T. Dao. "Mamba: Linear-time sequence modeling with selective state spaces." arXiv preprint arXiv:2312.00752 (2023).

[37] T. N. Kipf,, and M. Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).

[38] P. He,J. Zhu, Z. Zheng, and M. R. Lyu. "Drain: An online log parsing approach with fixed depth tree." In 2017 IEEE international conference on web services (ICWS), pp. 33-40. IEEE, 2017.

[39] A. Siffer, P. A. Fouque, A. Termier, and C. Largouet. "Anomaly detection in streams with extreme value theory." In Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining, pp. 1067-1075. 2017.

[40] K. W. Church. "Word2Vec." Natural Language Engineering 23, no. 1 (2017): 155–62. https://doi.org/10.1017/S135132491600032X.