2026 Design, Automation & Test in Europe Conference (DATE 2026)

Topology-Aware Circuit Breaking on Critical Paths in Microservice Systems

Lin Wang, Xin Li, Yanling Bu, Tianhao Zhang, Meiyan Teng, and Yanchao Zhao
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

***Abstract*—In microservice architectures, the complex web of inter-service dependencies makes systems vulnerable to cascading failures, where a single slow microservice can degrade overall application performance. Conventional circuit-breaking mechanisms often lack the precision to handle these issues effectively, as they treat services uniformly or depend on static, local thresholds. This can lead to either overly aggressive or delayed responses, resulting in inefficient system stabilization. This paper introduces the Topology-Aware Circuit Breaker (TACB), a traffic management mechanism that addresses this challenge by focusing on the services that matter most. The core idea of TACB is to dynamically identify the request's critical path, the longest execution path in the service call graph, which dictates the end-to-end latency. By concentrating its analysis and circuit-breaking actions exclusively on the services along this path, TACB intelligently ignores non-critical services and adapts to the real-time state of the distributed system. TACB continuously monitors service stability on the critical path and applies targeted circuit breaking to any service exhibiting performance degradation. This ensures that protective measures are applied precisely where and when they are needed. We implemented TACB on an Istio-based service mesh and evaluated it using the DeathStarBench benchmark suite. Experimental results demonstrate that our approach achieves significant improvements in system resilience, reducing end-to-end latency by over 50% and improving overall throughput compared to default and random strategies.**

***Index Terms*—microservices, traffic management, cascade failure, circuit breaker**

I. INTRODUCTION

Driven by containerization technologies such as Docker [1] and Kubernetes [2], microservice architecture (MSA) has become the dominant paradigm for building large-scale distributed systems [3]. By decomposing monolithic applications into fine-grained and loosely coupled services, MSA substantially improves scalability, flexibility, and fault isolation. This architectural style has been widely adopted in production environments at companies such as Amazon [4] and Alibaba [5], where backends often comprise thousands of microservices processing millions of requests per second. Nevertheless, this architectural paradigm introduces considerable operational complexities. Dense service dependencies create long and dynamic invocation chains, where the slowdown or failure of even a single component may rapidly propagate upstream and trigger cascading failures that compromise overall system reliability [6].

In a microservice environment, the performance degradation or unresponsiveness of a downstream service blocks its upstream callers, leading to the consumption of critical resources such as threads and connections [7]. As requests accumulate, these upstream services exhaust resources and eventually fail, amplifying the fault across the system. This cascading effect represents one of the most disruptive failure modes in microservice-based systems. Addressing such failures is inherently difficult due to two fundamental challenges. The first is the unreliability of services running in shared and dynamic environments, where resource contention, network fluctuations, or software defects can easily cause performance degradation. The second is the uncertainty and dynamism of service dependencies. Modern microservice deployments involve thousands of services with complex and constantly evolving invocation patterns shaped by feature iterations, A/B testing, and workload shifts. For instance, Uber's backend system comprises approximately 4,000 microservices that communicate internally via RPCs, creating 40,000 unique RPC endpoints [8]. The sheer scale and dynamism of inter-service dependencies present great challenges for effective traffic management and fault isolation.

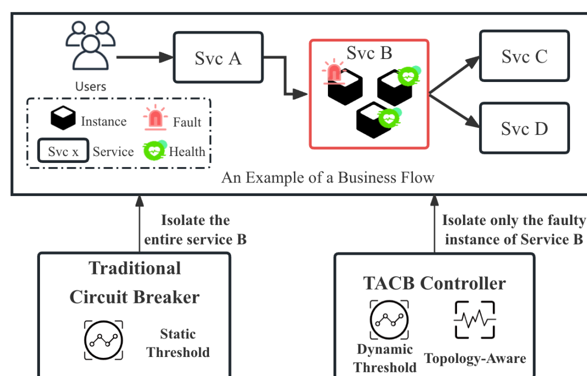

Fig. 1. Comparison example of conventional circuit breaker and TACB.

The circuit breaker pattern has emerged as a widely adopted resilience strategy [9], [10]. By interrupting requests to persistently failing services, circuit breakers prevent upstream resources from being depleted. However, conventional circuit breakers exhibit critical shortcomings that limit their efficacy in modern systems. As shown in Fig. 1, they typically rely on fixed thresholds such as error rates or timeouts, which cannot adapt to runtime variability. Their perspective is local, focusing on individual services in isolation and ignoring the broader execution context. Additionally, they tend to treat all services uniformly, without considering their varying contributions to overall performance. As a result, conventional circuit breakers often lead to premature or delayed intervention in circuit breaker operation, failing to provide effective protection in complex distributed environments.

978-3-9826741-1-7/DATE26/© 2026 EDAA

To overcome these limitations, this paper introduces the Topology-Aware Circuit Breaker (TACB), a lightweight mechanism designed to enhance resilience in microservice systems. TACB operates on the core principle of dynamically identifying the critical path of each request and applying circuit-breaking actions exclusively to unstable services along this path. TACB integrates distributed tracing for topology discovery, employs latency-based heuristics for stability assessment, and leverages service mesh infrastructure for fine-grained, instance-level control. This targeted approach avoids unnecessary interventions on non-critical services while ensuring timely protection at performance-sensitive points.

The contributions of this paper are summarized as follows:

- **Problem Analysis and Motivation:** Through an analysis of complex microservice interaction patterns, we reveal the inadequacy of conventional, localized fault-tolerance strategies in preventing cascading failures. This work establishes the critical need for topology-aware resilience mechanisms.
- **Management Mechanism:** We propose the **T**opology-**A**ware **C**ircuit **B**reaker (TACB) mechanism, which dynamically adapts to service state changes and focuses on key services for circuit breaking to maintain system quality.
- **Evaluation:** We implement TACB on the Istio service mesh and conduct extensive experiments using a comprehensive benchmark. Results show TACB reduces end-to-end latency by 50%-80%.

We review the related works in Section II. Section III gives the problem statement of traffic management. Our proposed solution framework is detailed in Section IV. Following this, Section V conducts experiments and evaluates the efficiency of the controller. Finally, we offer our conclusion in Section VI.

II. Related Works

In recent years, traffic management in microservices has garnered significant research attention. While early work focused on foundational patterns, the field has evolved towards more dynamic, intelligent, and context-aware solutions.

A. Critical Path Analysis for Performance Optimization

The critical path has been effectively applied to analyze performance and optimize scheduling [8], [11]–[13]. Zhang et al. [8] introduced CRISP, a tool that performs critical path analysis on large-scale RPC traces to identify performance bottlenecks. Similarly, other research has leveraged critical path information for topology-aware scheduling. For instance, Li et al. [12] proposed MOTAS. This framework maps microservice topology to underlying cluster topology to improve network efficiency, while Qiu et al. [13] used critical path analysis to guide resource allocation and reduce CPU usage.

These studies effectively demonstrate the value of understanding application topology and critical paths for performance optimization. However, they primarily focus on offline analysis or resource scheduling rather than real-time resilience. Thus, while critical path analysis provides an excellent diagnostic tool to identify performance-sensitive components, it lacks an inherent mechanism for live traffic intervention and resilience.

B. Advancements in Circuit Breaker Mechanisms

The circuit breaker pattern is a cornerstone of microservice resilience [9], [14]. Initial implementations, popularized by systems like Hystrix [15], relied on static thresholds (e.g., error rate or volume). While effective, these static approaches struggle to adapt to the highly dynamic nature of cloud environments, where traffic patterns and resource availability can fluctuate dramatically [16], [17].

To address these limitations, recent research has explored more adaptive and predictive mechanisms. One major trend is the application of Artificial Intelligence (AI) and Machine Learning (ML) to create intelligent, self-healing systems. For example, some studies propose using Deep Reinforcement Learning (DRL) to automatically learn optimal fault recovery policies from system metrics [18]–[20]. Other works leverage Large Language Models (LLMs) to build comprehensive incident management frameworks that can perform anomaly detection and root cause analysis from multimodal data (metrics, logs, and traces) [21]. Although powerful, these AI-driven methods often introduce significant complexity, incur high computational costs, and operate as black boxes with limited interpretability, making their decisions difficult to trust in mission-critical systems.

In contrast, TACB offers a different approach to intelligent circuit breaking. Conceptually, our approach aligns with the Theory of Constraints (TOC), which postulates that the throughput of any system is determined by its bottleneck. By focusing exclusively on the critical path, TACB applies targeted circuit breaking with fine-grained accuracy.

III. System Model and Problem Statement

We model a microservice application as a set of interdependent services, each possibly deployed with multiple instances for scalability and redundancy. Rather than modeling detailed resource usage (such as CPU, memory, and network bandwidth), we focus on capturing performance degradation through response latency, which can be observed via distributed tracing.

For each end-to-end request, the runtime execution is represented as a vertex-weighted directed acyclic graph (DAG) $G = (V, E)$, where V is the set of vertices, representing the microservices. E is the set of directed edges, representing the RPC calls between services. Each service $v \in V$ has an associated execution time, defined by a weight function $w : V \to \mathbb{R}^+$, where $w(v)$ is the latency of service v. Formally, let $\mathcal{P}_{s\to t}(G)$ denote the set of all directed paths from the entry node s to the exit node t in the service-call graph G. The critical path CP is defined as the path with the maximum cumulative latency:

$$CP = \arg\max_{p \in \mathcal{P}_{s\to t}(G)} \sum_{v \in p} w(v), \tag{1}$$

where $w(v)$ represents processing latency of service node v. This path uniquely determines the end-to-end latency and directly represents the performance bottleneck of the request. By focusing on the critical path, we aim to optimize the application's performance by addressing bottlenecks that have the most significant impact on latency.

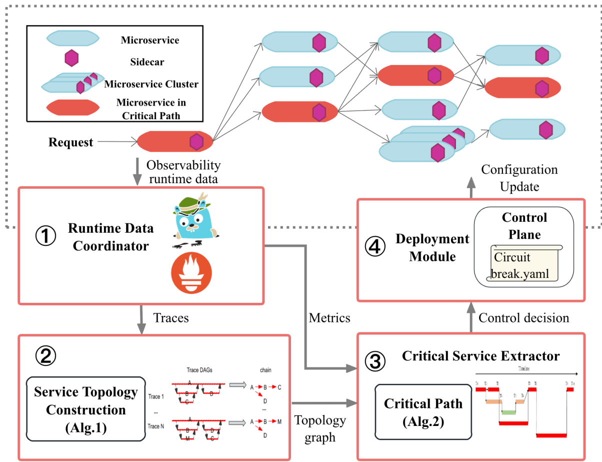

Fig. 2. The logic structure of TACB circuit breaker.

Conventional circuit breakers often make decisions based solely on local service metrics, ignoring the service's role in the request's topology. A service may appear unhealthy in isolation but lie on a non-critical branch, while a bottleneck on the critical path may escape timely protection. Furthermore, static thresholds limit adaptability under dynamic cloud-native workloads. Therefore, the key challenge is to design a circuit-breaking mechanism that is topology-aware. The proposed mechanism must dynamically identify services along the critical path and apply fine-grained, adaptive protection only where it will significantly impact end-to-end performance. By using the DAG to guide these decisions, we ensure that circuit-breaking actions are applied efficiently and only to those services that directly contribute to request latency, optimizing system resilience without unnecessary interventions.

IV. MECHANISM

In this section, we describe the overall architecture and implementation of the TACB mechanism. The key insight of TACB is that system resilience can be significantly improved by focusing protective measures on services on the request's critical path. This is achieved through a multi-stage process involving real-time topology detection, critical path analysis, and fine-grained control decisions, as illustrated in Fig. 2.

A. Runtime Data Coordinator

The efficacy of TACB's topology-aware approach relies on a hybrid data model that combines two sources: distributed traces from Jaeger [22] and aggregated latency metrics from Prometheus [23]. This monitoring system scrapes and stores time-series data. Traces are essential for reconstructing the runtime service topology and identifying per-request critical paths, while latency metrics are used to assess the real-time stability of individual instances. To mitigate the prohibitive overhead of full tracing, we employ a 10% head-based sampling strategy. This rate is chosen as it provides a sound balance between data fidelity and performance impact. Prior studies [12] have shown that a 10% sampling frequency can capture a representative view of system behavior for analysis while imposing a negligible cost on the application, with observed impacts on throughput and latency of less than 1%. For stability assessment, TACB primarily uses the 50th (P_{50}) and 99th (P_{99}) percentile latencies as key performance indicators (KPIs). The divergence between the median (P_{50}) and tail (P_{99}) latency is a strong indicator of performance degradation and forms the quantitative basis for our service stability heuristic.

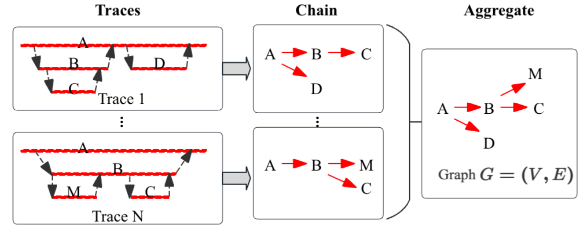

Fig. 3. An illustration of building the service dependency graph.

B. Topology Graph Construction

The topological foundation of our mechanism is the service dependency graph. In modern microservice systems, where dependencies are complex and constantly evolving, static definitions of service topology are often inadequate. Therefore, we dynamically construct this graph by processing distributed tracing data. As illustrated in Fig. 3, this process involves aggregating numerous individual request traces. The figure depicts how distinct request paths, such as A→B→C in one trace and A→B→M in another, are merged to form a comprehensive directed graph $G(V, E)$. This resulting graph provides an accurate, macro-level model of the system's runtime topology.

Our efficient approach for this construction is formally detailed in Alg. 1. The algorithm operates in linear time, with a time complexity of $O(S_{total})$, where S_{total} represents the total number of spans across all processed traces. This efficiency ensures minimal overhead during runtime analysis. This efficiency is achieved by using a hash map ($spanMap$) for quick lookups, which avoids an inefficient quadratic search that would involve comparing every span against every other span. The algorithm iterates through the collected traces to identify unique microservices, which become the graph's vertices (V), and then establishes directed edges (E) between them based on the parent-child relationships found in the spans. Each edge represents a direct RPC call. The resulting graph is a foundational prerequisite for our topology-aware mechanism.

C. Critical Path Analysis

This stage is the core of the TACB decision-making process. Given that the service topology and the resulting critical path can change from one request to another, this process must be continuous. Using the service dependency graph and real-time trace data, we identify the critical path for each incoming traced request. A critical path represents the sequence of causally dependent operations that dictates the total end-to-end latency

Algorithm 1: Build Service Dependency Graph from Traces

Input: A collection of traces T
Output: A directed graph $G = (V, E)$

```
1 Initialize G with empty vertex set V and edge set E;
2 for each trace tr ∈ T do
3     Construct spanMap (span.id → span.serviceName) from tr;
4     Add all unique service names in spanMap to V;
5     for each span s ∈ tr with parentId do
6         Add edge (spanMap[parentId], spanMap[s.id]) to E;
7     end
8 end
9 return G;
```

Algorithm 2: Identify Critical Path from Trace

Input: A single trace S (spans with parent-child links)
Output: Ordered list of services on the critical path CP

```
1  if S is empty then
2      return empty list;
3  end
4  spanMap ← (constructed as in Algorithm 1);
5  lastSpan ← arg max_{s∈S}(s.startTime + s.duration);
6  if lastSpan is null then
7      return empty list;
8  end
9  Initialize CP ← empty list;
10 currentSpan ← lastSpan;
11 while currentSpan ≠ null do
12     Prepend currentSpan.serviceName to CP;
13     if currentSpan has a parentId then
14         currentSpan ← spanMap[currentSpan.parentId];
15     end
16     else
17         currentSpan ← null;
18     end
19 end
20 return CP;
```

of a request. By focusing our circuit-breaking efforts on this path, we can most effectively improve performance and prevent cascading failures.

Our method for identifying the critical path is detailed in Alg. 2. The intuition is straightforward: the final operation to complete in a distributed trace must be the terminus of the critical path. We identify the critical path endpoint as the span with the maximum completion time (start_time + duration), under the assumption that the longest path must end at this point. This is valid because any path ending at an earlier point cannot be longer than one ending at the latest point. By starting at this final span and backtracking through its unique chain of parent spans, we can reconstruct the exact sequence of services that determined the overall request latency. This backtracking approach is both efficient and deterministic.

The algorithm's time complexity is $O(k)$, where k is the number of spans in a single trace. This linear complexity arises because the primary operations, finding the span with the maximum completion time and subsequently backtracking through its antecedents, each require traversing the spans at most once. The low computational cost ensures that critical path analysis can be performed in real-time. It enables the TACB controller to make timely, fine-grained control decisions based on the most current state of the system, thereby providing adaptive and precise resilience.

D. Stability Heuristic and Control Logic

This section defines the metric used to assess service health and outlines the control logic for making control decisions.

1) Service Stability Heuristic: A service's stability is evaluated using key latency indicators aggregated by Prometheus over a recent time window. A service instance is determined to be unstable if it meets either of the following criteria, which respectively capture absolute performance degradation and significant performance jitter. A service instance is flagged as unstable when its performance degrades, characterized by its P_{99} latency exceeding a service-level threshold (θ_{abs}) or its latency jitter (P_{99}/P_{50}) surpassing a configured ratio (θ_{ratio}). This dual-condition heuristic is formally expressed as:

$$isUnstable = (P_{99} > \theta_{abs}) \lor (\frac{P_{99}}{P_{50}} > \theta_{ratio}). \qquad (2)$$

A data-driven approach is employed to set these parameters. This process involves first conducting a series of baseline runs of the target microservice application under its typical operational load without any fault injection. During this profiling phase, a representative dataset of P_{99} latency and P_{99}/P_{50} ratio metrics is collected for the services that commonly appear on critical paths. Subsequently, the statistical distribution of these baseline metrics is analyzed. The thresholds are then set based on a high percentile of the observed values. For example, θ_{abs} for a service can be set to its 95th percentile P_{99} latency observed during the baseline period. This data-driven methodology ensures that the thresholds are tailored to the specific performance profile of each service.

2) Control Loop: The TACB controller operates in a periodic loop to continuously assess the system state. In each cycle, it first processes sampled traces from the preceding time window to determine the current set of services along critical paths. For each of these services, the controller retrieves the latest P_{50} and P_{99} latency metrics from Prometheus and evaluates their stability using the `isUnstable`. Whenever an instance is classified as unstable, the controller immediately formulates a control decision, typically ejecting the instance from the load-balancing pool for a short duration. The resulting decisions are then forwarded to the delivery module for enforcement.

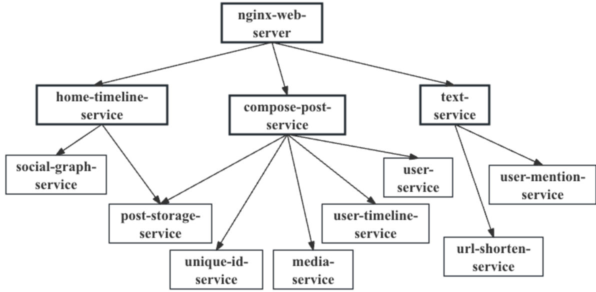

Fig. 4. Service invocation topology of a key workflow in the Social Network.

E. Control Decision Enforcement

Once a control decision is formulated by the controller, it must be propagated and enforced within the service mesh's data plane. This phase translates the logical policy into a concrete configuration update. The TACB mechanism leverages the Istio [24] service mesh, which provides a dedicated infrastructure layer for service-to-service communication, to implement its decisions. The controller modifies the `DestinationRule` custom resource corresponding to the unstable service. Specifically, it updates the `outlierDetection` field to configure an ejection policy based on the decision's parameters.

This updated configuration is persisted in the ETCD [25] database. The Istio control plane detects this change and propagates the new policy in the form of xDS updates to the Envoy proxies of all upstream services. Upon receiving the update, the Envoy proxies will temporarily remove the unstable instance from their load-balancing pool, effectively enforcing the circuit-breaking action at the data plane level without any changes to the application code. A key advantage of TACB's approach, implemented via Istio's outlier detection, is its ability to operate at the instance level rather than the service level.

V. Performance Evaluation

Extensive and comprehensive experiments are conducted to evaluate the performance of TACB in different scenarios.

A. Evaluation Setup

Experimental Settings: All experiments are performed on the OpenStack platform. 4 physical machines are included in the platform, each using Intel(R) Xeon(R) series processors, containing 8 to 16 CPU cores, and equipped with 16 GB to 32 GB physical memory.

Benchmark: To ensure our evaluation is both realistic and reproducible, all experiments are conducted using the Social Network application from the open-source benchmark *DeathStarBench* [26]. This benchmark is composed of 36 microservices in various languages, featuring deep and complex call graphs designed to emulate large-scale systems like Twitter or Facebook. Its inherent complexity is crucial for our evaluation: it ensures that the performance bottleneck is both non-trivial and highly dynamic, shifting with runtime conditions, as illustrated for a key workflow in Fig. 4. Furthermore, the application's mixed workload of CPU, network, and I/O-bound operations creates the realistic and unpredictable

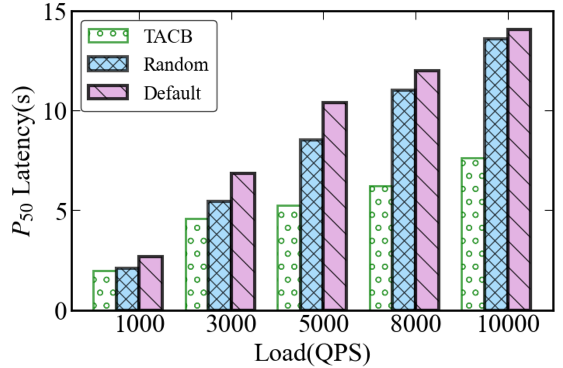

Fig. 5. Comparison of median (P_{50}) latency with varying request loads.

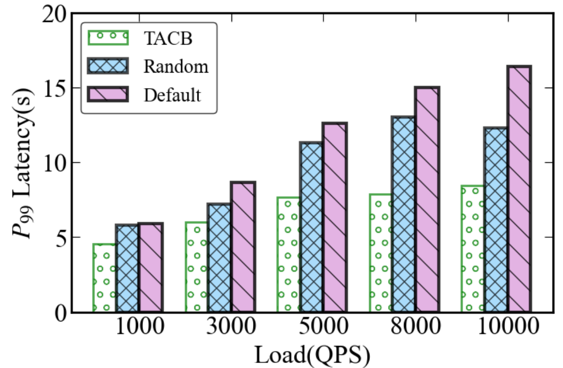

Fig. 6. Comparison of tail (P_{99}) latency with varying request loads.

latency scenarios that TACB is designed to mitigate. All the microservice components are deployed as Docker containers in the experimental environment.

Competing Approaches: In the experiments, we explore three strategies in this section: (1) *Default:* Circuit breaker in default mode in Istio. (2) *Random:* Randomly selects a subset of services on the critical path to apply circuit breaking, regardless of their stability. (3) *TACB:* Our proposed controller.

Load Testing Tool: We use *wrk2* [27] to generate client-side load and evaluate our methodologies under various deployment scenarios. To simulate realistic conditions, tests were run with request rates varying from 1,000 to 10,000 queries per second (QPS). By using this precise load generation, we aimed to create a challenging environment, ensuring that our results reflect practical performance metrics and scalability under diverse conditions.

B. Request Return Latency

As shown in Fig. 5 and Fig. 6, TACB consistently outperforms both the Default and Random strategies across all tested request volumes, in terms of both median (P_{50}) and tail (P_{99}) latency. TACB achieves measurable latency improvements even under the lowest tested load of 1,000 QPS. At this load, the system experiences substantial traffic but not the widespread overload failures that the Default is primarily designed to address. This indicates that the observed benefit stems from TACB's design rather than external factors. Unlike traditional mechanisms that react mainly to severe overload, TACB detects

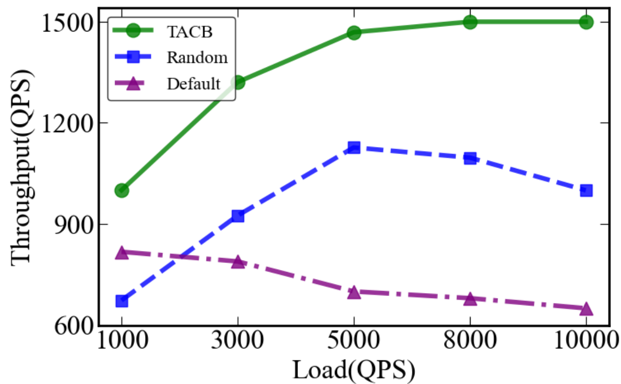

Fig. 7. Throughput comparison.

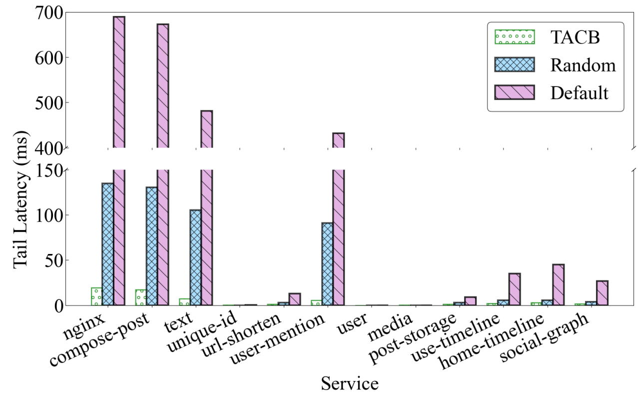

Fig. 8. Per-service tail latency comparison.

and responds to performance degradations on the critical path, including transient bottlenecks caused by cold starts, garbage collection pauses, or resource contention. By selectively isolating unstable instances identified through the stability metric, TACB prevents individual slow nodes from delaying requests, thereby improving overall latency for the entire workload.

These results directly demonstrate TACB's effectiveness. The benefits become progressively more significant as the request load increases, where resource contention exacerbates these bottleneck effects. This confirms TACB's effectiveness and robustness across a wide range of operating conditions.

C. Throughput

The throughput comparison in Fig. 7 reveals the performance differences between the strategies under increasing load. At 1000 QPS, the Random policy's throughput is slightly below the Default. As the request volume escalates, TACB demonstrates robust performance, maintaining a stable throughput. Conversely, the default policy, lacking an adaptive mechanism, begins to exhibit errors that lead to request blocking and a consequent decline in the number of completed requests per second. Similarly, the Random policy is effective only at lower request densities. As the load intensifies, its arbitrary nature proves suboptimal, resulting in performance degradation rather than improvement.

D. End-to-End Latency

The results in Fig. 8 demonstrate that TACB sharply reduces tail latency, particularly for upstream services. Under the Default strategy, entry-point services such as nginx and compose-post service exhibit severe degradation, with tail latencies inflating to more than 600 ms (milliseconds). This inflation does not originate from their own processing but from accumulated waiting time on slow downstream dependencies. Once a deep service becomes unstable, the resulting delay cascades upward, forcing upstream services to block and amplifying response times across the chain.

TACB addresses this problem through its topology-aware design. By dynamically identifying the critical path and isolating unstable service instances, TACB eliminates the bottleneck at its root. As soon as the problematic node is bypassed, upstream services revert to their intrinsic processing cost, which is only tens of milliseconds (below 20 ms), instead of carrying hundreds of milliseconds of accumulated delay. This suppression of cascading delays directly explains the substantial quantitative gains. TACB consistently reduces end-to-end P99 latency by 50%–80% compared with Default and Random baselines, while simultaneously improving throughput through fine-grained, targeted protection.

TABLE I
RESOURCE OVERHEAD OF TACB NORMALIZED PER 1,000 REQUESTS

Resource (per 1,000 reqs)	Default	TACB	Overhead
CPU (mCPU·s)	28.4	29.1	+0.7 (+2.5%)
Memory (MiB)	7.8	8.9	+1.1 (+14%)

E. Overhead Analysis

We evaluated the resource overhead of TACB under peak load conditions (10,000 QPS). The overhead on each sidecar is minimal as TACB reuses Istio's telemetry pipeline, leading to a small increase in CPU and memory consumption. Table I summarizes these costs. The resource overhead introduced by TACB is minimal (less than 2.5% CPU and 1.1 MiB memory per 1,000 requests), making it a cost-effective solution given the significant latency improvements (50–80%).

VI. Conclusion

This paper presents TACB, a topology-aware circuit breaker that provides a new perspective on microservice resilience by shifting from isolated, local metrics to a global perspective centered on the dynamic critical path. Our experimental evaluation demonstrates that this topology-aware approach delivers substantial performance improvements, reducing end-to-end latency up to 80% while maintaining or improving system throughput across diverse load conditions. In the future, we intend to investigate extending the topology-aware approach to other resilience patterns, such as adaptive rate limiting and load shedding on the critical path.

Acknowledgment

This work is supported in part by the Science and Technology Major Project of Jiangsu Province of China under Grant BG2024043.

References

[1] "Docker - accelerate how you build, share, and run applications," [Online]. Available: https://www.docker.com/, Accessed on 2025.

[2] "Kubernetes: Production-grade container orchestration," [Online]. Available: https://kubernetes.io/, Accessed on 2024.

[3] J. Thönes, "Microservices," *IEEE software*, vol. 32, no. 1, pp. 116–116, 2015.

[4] "The amazon web service," [Online].Available:https://aws.amazon.com/cn/, Accessed on 2025.

[5] "The alibaba cloud office website," [Online]. Available: https://www.alibabacloud.com/, Accessed on 2024.

[6] K. Asanović, S. Pudar, C. Celio, Y. Lee, R. Iyer, U. Jevtić, and J. Zhu, "Silk: A low-latency, high-bandwidth rack-scale interconnect," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 401–415.

[7] Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–9.

[8] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "Crisp: Critical path analysis of large-scale microservice architectures," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 655–672.

[9] G. Aquino, R. Queiroz, G. Merrett, and B. Al-Hashimi, "The circuit breaker pattern targeted to future iot applications," in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings 17*. Springer, 2019, pp. 390–396.

[10] K. Falahah, W. Surendro, and S. Danar, "Circuit breaker in microservices: State of the art and future prospects," *IOP Conference Series: Materials Science and Engineering*, vol. 1077, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:234077380

[11] Y. Shi, S. Xu, S. Kai, X. Lin, K. Xue, M. Yuan, and C. Qian, "Timing-driven global placement by efficient critical path extraction," in *2025 Design, Automation Test in Europe Conference (DATE)*, 2025, pp. 1–7.

[12] X. Li, J. Zhou, X. Wei, and D. Li, "Topology-aware scheduling framework for microservice applications in cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1635–1649, 2023.

[13] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "Firm: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 805–825.

[14] L. Zheng, Z. Chen, J. He, and H. Chen, "Mulan: Multi-modal causal structure learning and root cause analysis for microservice systems," in *Proceedings of the ACM Web Conference 2024*, ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 4107–4116.

[15] "Netflix/hystrix," [Online]. Available: https://github.com/Netflix/Hystrix/, Accessed on 2025.

[16] Z. Wu, Y. Deng, Y. Zhou, L. Cui, and X. Qin, "Hashcache: Accelerating serverless computing by skipping duplicated function execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 12, pp. 3192–3206, 2023.

[17] H. Huang, C. Chen, and K. Chen, "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*, ser. ASPLOS '25. NY, USA: Association for Computing Machinery, 2025, p. 683–697.

[18] Z. Li, H. Sun, Z. Xiong, Q. Huang, Z. Hu, D. Li, S. Ruan, H. Hong, J. Gui, J. He, Z. Xu, and Y. Fang, "Noah: Reinforcement-learning-based rate limiter for microservices in large-scale e-commerce services," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 9, pp. 5403–5417, Sep. 2023.

[19] H. Guo, J. Yang, J. Liu, L. Yang, L. Chai, J. Bai, J. Peng, X. Hu, C. Chen, D. Zhang, X. Shi, T. Zheng, L. Zheng, B. Zhang, K. Xu, and Z. Li, "Owl: A large language model for it operations," 2024. [Online]. Available: https://arxiv.org/abs/2309.09298

[20] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 553–565. [Online]. Available: https://doi.org/10.1145/3611643.3616249

[21] H. Guo, J. Yang, J. Liu, J. Bai, B. Wang, and Z. Li, "Logformer: a pre-train and tuning pipeline for log anomaly detection," in *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'24/IAAI'24/EAAI'24. AAAI Press, 2024.

[22] "Jaeger overview," [EB/OL], https://www.jaegertracing.io/ Accessed on 2024.

[23] "Prometheus - monitoring system time series database," [Online]. Available: https://prometheus.io/, Accessed on 2024.

[24] "Istio - service mesh. simplified," [Online]. Available: https://istio.io/ Accessed on 2025.

[25] "Etcd - a distributed, reliable key-value store for the most critical data of a distributed system." [Online]. Available: https://etcd.io/, Accessed on 2024.

[26] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, and N. Katarki, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18.

[27] "Wrk2," [Online]. Available: https://github.com/giltene/wrk2/, Accessed on 2025.