HiLL PUBLISHING

Engineering Advances, 2026, 6(3), 233-237
https://www.hillpublisher.com/journals/ea/
ISSN Online: 2768-7961

A Conceptual Framework for OpenTelemetry-based Distributed Tracing and Burst Traffic Bottleneck Localization in Complex Microservice Architecture

Chenghao Shi

Georgia Institute of Technology, Atlanta, GA 30332, USA.

How to cite this paper: Chenghao Shi. (2026) A Conceptual Framework for OpenTelemetry-based Distributed Tracing and Burst Traffic Bottleneck Localization in Complex Microservice Architecture. *Engineering Advances*, 6(3), 233-237.
DOI: 10.26855/ea.2026.09.017

Received: July 29, 2026
Accepted: August 28, 2026
Published: September 25, 2026

*Corresponding author: Chenghao Shi, Georgia Institute of Technology, Atlanta, GA 30332, USA.

© 2026 by the author(s).
This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives (CC BY-NC-ND) license, which permits non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited and is not modified or adapted.
https://creativecommons.org/licenses/by-nc-nd/4.0/

Abstract

Complex microservice architectures can experience queue accumulation, resource contention, and cross-service latency propagation when request volume rises sharply within a short interval, while CPU thresholds or service response times alone provide limited evidence for locating the earliest point of degradation. This paper proposes a conceptual framework that uses OpenTelemetry trace and span records to reconstruct request dependencies across services, align span duration, critical-path contribution, and CPU, memory, and network conditions on a common timeline, and derive service-level bottleneck scores from latency deviation, critical-path position, and resource-state variation. The framework is intended to separate primary bottlenecks from downstream nodes affected by propagated delay. A planned evaluation design based on Google Cloud Online Boutique and the public RE2-OB dataset is outlined, including 90 labeled failure cases covering CPU, memory, disk, socket, network delay, and packet loss; Metric Threshold, Latency Ranking, and Trace-Critical Localization are proposed as comparison methods, with Top-1, Top-3, precision, recall, F1-score, P95/P99 latency, and telemetry overhead as planned metrics.

Keywords

Conceptual framework; OpenTelemetry; distributed tracing; microservice architecture; burst traffic; bottleneck localization

Introduction

A single request in a microservice application may pass through frontend, checkout, payment, inventory, recommendation, and other independently deployed services, so local resource pressure can accumulate along the call chain and appear as end-to-end latency; monitoring runtime resources, service behavior, and end-to-end execution has consequently become a continuing engineering concern in complex microservice systems [1]. Distributed tracing preserves Trace IDs, Span IDs, parent-child relations, and execution timing, allowing individual requests to be reconstructed across service boundaries, although dynamic dependencies, large span volumes, and latency propagation complicate automated diagnosis [2]. Burst traffic makes this distinction harder because downstream services may slow while waiting for an overloaded upstream component, causing latency ranking alone to confuse propagated delay with the original bottleneck; reliable localization consequently requires call order, critical-path position, and resource state to be examined together.

This paper proposes a conceptual framework for OpenTelemetry-based distributed tracing and burst traffic

bottleneck localization in complex microservice architectures. The paper focuses on framework design and a planned evaluation scheme; it does not report experimental results or claim verified effectiveness. The performance, overhead, and applicability of the framework remain to be validated in future work.

1. Bottleneck Characteristics Under Burst Traffic

1.1 End-to-End Latency Evolution Under Burst Traffic

When request volume rises sharply within a short interval, performance degradation in a microservice call chain usually begins with local queue accumulation and resource contention; once the request rate approaches the processing capacity of a service, thread occupancy, CPU scheduling delay, and network transmission time lengthen the duration of related spans, while parent calls spend more time waiting for downstream responses, causing P95 and P99 latency to deviate earlier than the average response time. Because microservices are connected by sequential dependencies, saturation at an upstream node propagates additional waiting time along parent-child spans and causes several downstream services to exhibit latency growth within the same observation window, even when their local resource states remain close to normal. Burst-related degradation should therefore be examined against request arrival order, span timing, and resource variation so that the transition from a local slowdown to chain-wide latency propagation can be identified accurately [3].

1.2 Bottleneck Identification Along Service Call Chains

The service with the longest observed latency does not necessarily represent the earliest point of congestion, because a primary bottleneck commonly exhibits a concurrent deviation in span duration, residence time on the critical call segment, and CPU, memory, or network state, whereas a propagation-affected downstream service may show longer waiting time while its local resource consumption remains within the normal range. Trace IDs group all spans belonging to the same request, Span IDs and Parent Span IDs preserve invocation direction, and start timestamps together with span durations establish the temporal order of abnormal behavior; arranging these fields according to service dependencies makes it possible to distinguish primary bottlenecks, propagation-affected services, and normal nodes without relying solely on service-latency ranking. Existing trace-based anomaly analysis in microservice environments also indicates that span-level temporal information provides sufficiently fine-grained evidence for locating abnormal execution within distributed service chains [4].

2. Proposed OpenTelemetry-Driven Bottleneck Localization Method

2.1 Trace Reconstruction and Critical-Path Extraction

OpenTelemetry spans belonging to the same request are grouped by Trace ID, while Span ID and Parent Span ID are used to recover caller–callee dependencies and form a request-level directed acyclic graph $G = (V, E)$; each node stores service name, operation name, start timestamp, end timestamp, duration, and status, and every edge preserves the invocation order between two spans. For traces containing parallel child calls, total latency is not obtained by summing all span durations, because overlapping execution intervals would count the same elapsed time more than once; the analysis instead separates span self time from child waiting time, searches the root-to-terminal path with the largest accumulated blocking interval, and defines its latency as L_{cp}, while the contribution of service s is expressed as $C_s = L^{cp}s/Ltrace$, so services with an early temporal deviation and a persistent share of critical-path latency receive higher diagnostic priority [5]. The output of this stage contains the reconstructed call graph, the critical path, service-level latency contributions, and the temporal order of abnormal spans, which supplies causal context for the resource alignment performed in the next stage.

2.2 Span–Resource Temporal Alignment

Each critical span is mapped to runtime measurements collected from the same service instance within its execution interval $[t_{start}, t_{end}]$, including CPU utilization, memory usage, network delay, request queue state, and error status; measurements with different scales are normalized against a stable baseline, where $Z_x = (x - \mu_x)/\sigma_x$ describes the deviation of metric x, and span delay is expressed as $D_s = (L_s - L_s^{base})/L_s^{base}$. A service becomes a stronger bottleneck candidate when the rise in span duration and the deviation of one or more local resource indicators occur in the same temporal window, whereas a downstream span that becomes longer after an upstream delay while its local resource state remains near baseline is classified as propagation-affected; trace and profiling

information have previously been combined to distinguish abnormal service execution with finer runtime context [6]. This alignment keeps latency evidence tied to the resource state that existed when the request was processed, reducing errors caused by inspecting aggregated CPU or memory values outside the actual span interval.

2.3 Bottleneck Scoring and Localization

The reconstructed critical-path evidence and time-aligned resource state are combined into a service score $B_s = \alpha D_s + \beta C_s + \gamma R_s$, where D_s denotes normalized span-latency deviation, C_s represents critical-path contribution, and R_s summarizes normalized resource deviation; the three terms are scaled to comparable ranges before scoring, while the coefficients are fixed from validation cases or an explicitly stated equal-weight setting so that the ranking procedure remains reproducible. As shown in Fig. 1, telemetry moves from OpenTelemetry instrumentation to OTLP export and the Collector, after which Trace ID aggregation, DAG reconstruction, critical-path extraction, span–resource alignment, and bottleneck scoring are executed in sequence; a service is ranked as the primary bottleneck when a high score coincides with the earliest abnormal timestamp and a local resource deviation, while nodes showing delayed spans without corresponding local pressure are retained as propagation-affected services, and the runtime cost of OpenTelemetry instrumentation and telemetry collection is considered when the final configuration is evaluated [7]. The resulting output contains a ranked bottleneck list, propagation labels, and healthy-node classifications, providing a consistent basis for evaluating localization accuracy and telemetry overhead under burst traffic conditions.

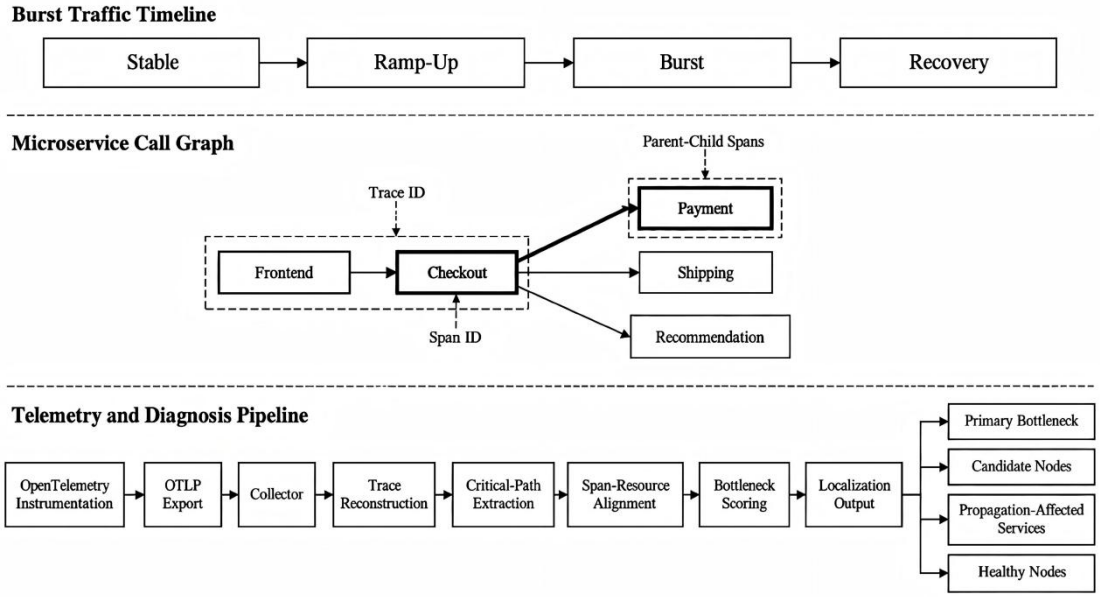

Figure 1. OpenTelemetry-based burst traffic bottleneck localization framework.

3. Proposed Evaluation Design Under Burst Traffic

3.1 Proposed Evaluation Environment and Dataset Configuration

Google Cloud Online Boutique is used as the system under test, with its microservices running in containers and OpenTelemetry instrumentation recording the spans produced by service calls; OTLP transports the telemetry to the Collector, where Trace ID, Span ID, Parent Span ID, timestamps, and duration fields are retained for later reconstruction. Locust generates the workload, beginning from the official configuration of 10 users with a spawn rate of 1 user per second, and the observation period is divided into stable, ramp-up, and burst phases; workload levels of 150, 300, 600, and 1,200 concurrent users are adopted as stress references from containerized OpenTelemetry overhead experiments conducted on AWS and Azure, with repeated runs maintained at each level to limit the influence of short-lived request fluctuations [7].

Each run records request rate, throughput, error rate, P95/P99 latency, and Collector-side CPU and memory usage, while trace records are matched to the service instance active within the same request interval. No unsupported fixed duration is assigned to the burst phase; its start, peak, and recovery boundaries are taken directly from

workload-generator timestamps, and the stable phase supplies the baseline distribution for span duration and resource measurements. This configuration places workload variation, trace timing, and runtime resource state on the same temporal axis, allowing each localization result to be traced back to a specific request and service instance.

3.2 Proposed Baselines, Evaluation Metrics, and Reproducibility Settings

The localization test uses labeled RE2-OB failure cases under three methods: Metric Threshold ranks services according to normalized CPU, memory, and network deviations; Latency Ranking orders services according to the increase in span duration relative to the stable baseline; Trace-Critical Localization reconstructs parent-child spans and combines critical-path contribution with resource deviation from the same execution window. The root-cause service encoded for each case is treated as ground truth, Top-1 is counted when the first-ranked service matches this label, Top-3 is counted when the root cause appears among the first three candidates, and precision, recall, and F1-score are calculated over the same complete case set, following the service-level failure-localization setting used in distributed-trace diagnosis [8].

As shown in Table 1, RE2-OB contains 90 failure cases covering CPU, memory, disk, socket, network delay, and packet loss; each fault type is applied to five root-cause services with three repetitions for every service-fault pair, yielding 15 cases per fault type. The dataset retains metrics, logs, and traces together, with 69-77 metrics available for checking resource behavior within each failure window; all three localization methods read the same cases and use the same normalization rules so that differences in sample selection do not alter the comparison.

Table 1. Fault composition of the RE2-OB dataset

Fault Condition	Fault Class	Target Services	Repetitions per Service	Cases
CPU	Resource	5	3	15
Memory	Resource	5	3	15
Disk	Resource	5	3	15
Socket	Resource	5	3	15
Network Delay	Network	5	3	15
Packet Loss	Network	5	3	15
Total	6 fault types	—	—	90

3.3 Planned Localization Accuracy and Telemetry Overhead Analysis

If implemented, the three methods would produce a ranked service list for every labeled case; the planned analysis would group the results by the six fault categories and then aggregate for Top-1, Top-3, precision, recall, and F1-score. Error inspection concentrates on two situations: the longest span belongs to a propagation-affected service, or a service shows clear local resource deviation without holding the longest span in the call chain; in either case, the abnormal timestamp and critical-path position are rechecked to determine whether Metric Threshold or Latency Ranking has assigned a cascading delay to the root cause. Public Online Boutique experiments on cascading microservice failures provide corresponding evidence for interpreting dependency-driven failure propagation [9].

Telemetry overhead is planned to be evaluated by comparing tracing-enabled and tracing-disabled runs under identical workload levels, using P95/P99 latency, throughput, Collector CPU, Collector memory, and trace volume as recorded quantities while preserving the same request script and repetition count. External OpenTelemetry measurements reported throughput reductions of 0.40%-8.40% on AWS and 0.30%-5.50% on Azure, with larger response-time and latency penalties appearing under heavier load; these figures are retained only as external configuration references and are not presented as localization results from the present experiment. No final experimental statistics are reported in this paper; the evaluation design is proposed for future work, with localization accuracy and telemetry cost examined under the same workload conditions.

4. Limitations

This paper proposes a conceptual framework and does not report experimental results. The framework has not been implemented or tested in a real microservice environment. No data were collected, and no comparison with existing methods was performed. Therefore, the effectiveness, performance overhead, and scalability of the proposed

framework remain unverified. Future work should implement the framework, conduct experiments under bursty traffic, and compare it with existing tracing and bottleneck localization methods.

5. Conclusion

This paper proposes a conceptual framework for OpenTelemetry-based distributed tracing and burst traffic bottleneck localization in complex microservice architectures. The framework includes trace reconstruction, critical-path extraction, span–resource temporal alignment, bottleneck scoring, and propagation labeling. It is designed to align request execution order, critical-path latency, and service runtime states on a shared timeline. A planned evaluation design based on Online Boutique and RE2-OB is also outlined. Future work may implement the framework, conduct burst-traffic experiments, and compare it with existing methods.

References

[1] Waseem M, Liang P, Shahin M, et al. Design, monitoring, and testing of microservices systems: the practitioners' perspective. J Syst Softw. 2021;182:111061.

[2] Bento A, Correia J, Filipe R, et al. Automated analysis of distributed tracing: challenges and research directions. J Grid Comput. 2021;19:9.

[3] Usman M, Ferlin S, Brunstrom A, et al. A survey on observability of distributed edge & container-based microservices. IEEE Access. 2022;10:86904-86919.

[4] Kohyarnejadfard I, Aloise D, Azhari S V, et al. Anomaly detection in microservice environments using distributed tracing data analysis and NLP. J Cloud Comput. 2022;11:25.

[5] Khanahmadi M, Shameli-Sendi A, Jabbarifar M, et al. Detection of microservice-based software anomalies based on OpenTracing in cloud. Softw Pract Exp. 2023;53(8):1681-1699.

[6] Panahandeh M, Hamou-Lhadj A, Hamdaqa M, et al. ServiceAnomaly: an anomaly detection approach in microservices using distributed traces and profiling metrics. J Syst Softw. 2024;209:111917.

[7] Hammad Y, Al-Said Ahmad A, Andras P. An empirical study on the performance overhead of code instrumentation in containerised microservices. J Syst Softw. 2025;230:112573.

[8] Purfallah Mazraemolla Z, Rasoolzadegan A. An effective failure detection method for microservice-based systems using distributed tracing data. Eng Appl Artif Intell. 2024;133:108558.

[9] Soldani J, Forti S, Roveroni L, et al. Explaining microservices' cascading failures from their logs. Softw Pract Exp. 2025;55(5):809-828.