Evaluating Proxy-Based Observability Pipelines for Unmodified Applications

Omar Bin Kasim Bhuian
SysCore Lab,
Sejong University
Seoul, South Korea
omarbinkasimsefat@gmail.com

Sang-Hoon Choi
SysCore Lab,
Sejong University
Seoul, South Korea
csh0052@gmail.com

Young-Soo Kim
Electronics and Telecommunications
Research Institute,
Seoul, South Korea
blitzkrieg@etri.re.kr

Hongri Liu
Weihai Cyberguard Technologies Co.,
Ltd., Weihai 264209, China
liuhr@cyberguard.com.cn

Ki-Woong Park
Department of Information Security
and Convergence Engineering for
Intelligent Drone, Sejong University
Seoul 05006, South Korea
woongbak@sejong.ac.kr

***Abstract*—Modern cloud applications often operate as unmodified third-party services or legacy code, where direct instrumentation for observability is infeasible. This paper investigates whether a minimal black-box observability pipeline can still provide actionable insights in such contexts. We compose a standard stack—Envoy proxy, OpenTelemetry Collector, Prometheus, Jaeger, and Grafana—to collect metrics and traces without modifying application code, and we apply it to two representative workloads: a lightweight demo service and the OWASP Juice Shop. Our evaluation shows that proxy-only instrumentation captures meaningful demand and latency signals, that exported spans faithfully reflect traffic bursts visible in proxy metrics, and that distributed traces reveal end-to-end error paths (e.g., 404 failures). These findings indicate that a carefully orchestrated open-source stack can approximate the diagnostic value of white-box instrumentation. The contributions of this work are a reproducible pipeline design and an empirical assessment demonstrating how such a configuration can reduce mean-time-to-detect (MTTD) failures and support service-level objective monitoring under synthetic load scenarios.**

***Keywords*— *Observability, Black-box monitoring, OpenTelemetry, Distributed tracing, Proxy-based instrumentation*.**

I. INTRODUCTION

Modern distributed applications depend on observability to remain reliable under unpredictable workloads. Conventional wisdom has it that the most effective observability comes from instrumenting source code directly, exposing internal metrics and traces[1]. Yet many systems run as unmodified third-party binaries, legacy code, or services outside a team's control. In such settings, the challenge is not to add instrumentation but to construct a pipeline that can observe behavior externally, in a "black-box" mode, while still yielding actionable insights[2], [3].

This study addresses that challenge by evaluating a minimal black-box observability stack composed of widely used open-source components—Envoy, the OpenTelemetry Collector, Prometheus, Jaeger, and Grafana—and applying it to two test workloads: a lightweight demo application and the OWASP Juice Shop. The design deliberately avoids code changes, instead leveraging a proxy for metrics and span generation, a collector for export, and time-series and tracing backends for analysis. The central question is not whether observability is possible in theory, but whether proxy-based signals and traces provide sufficient fidelity to diagnose performance bottlenecks and error conditions in practice.

We focus on this particular combination of tools not because they are new, but because they represent the de facto standard in modern observability practice. Our goal is to show how they can be orchestrated into a unified pipeline when application code cannot be modified, addressing a practical gap in current deployments. Although each tool is popular in isolation, it is not obvious that they suffice together to provide end-to-end visibility in a purely black-box setting. In this work, we demonstrate the viability of such a composition and critically assess its limitations.

Guided by this motivation, our study is structured around three central research questions. First, we ask whether it is possible to obtain actionable visibility for unmodified services using only a gateway or sidecar proxy. Second, we explore whether proxy-level metrics, such as request rate and p95 latency, meaningfully correlate with exported spans and distributed traces. Finally, we consider what operational benefits, particularly in terms of reducing mean-time-to-detect (MTTD) and mean-time-to-recover (MTTR)—can be realized through this pipeline.

Through these questions, the paper contributes in several ways. We design and deploy a reproducible, black-box observability pipeline using widely adopted open-source tools, demonstrating how they can be combined without requiring any modifications to application code. We then examine this pipeline under controlled and realistic workloads, showing how proxy-based metrics align with trace-level signals. From this examination, we analyze the operational impact of such a pipeline, highlighting its ability to accelerate incident detection and diagnosis in contexts where white-box instrumentation is unavailable. Finally, we talk about the limits and future work direction, putting proxy-based observability as a useful tool that makes it easier to keep an eye on old or third-party systems.

II. BACKGROUND AND RELATED WORK

A. Background

Efforts to manage modern distributed systems have evolved from monitoring toward richer observability. Traditional monitoring captures metrics like CPU, memory, or uptime, but lacks insight into why requests succeed or fail. Observability is often described as going beyond traditional monitoring. Rather than only reporting what a system is doing, it gives engineers the means to ask why a particular behavior occurs. Li et al. (2022)[4] found that many organizations maintain tracing and analysis pipelines, but in real-practice they still balanced detail against system overhead and visibility.

Another issue comes in distinguishing white-box from black-box instrumentation. White-box methods assume that developers can change the application and add telemetry code. Black-box methods, on the other hand, treat the system as a black box and use sidecars or proxies instead. Research on service meshes, like Sidecars on the Central Lane [5] has shown that proxies improve observability but also add measurable latency and resource overhead.

At the same time, a number of open-source projects have become the backbone of observability practice. Prometheus is widely used for metrics collection, OpenTelemetry provides a common framework for traces and metrics, Jaeger acts as a tracing backend, and Grafana is often deployed as the main visualization layer. These tools frame the practical environment in which our study is situated.

B. Related Work

Although most deployments assume some level of application-side instrumentation, research such as Lee et al.[6], demonstrates the gains from combining multiple sources like traces, KPIs, and logs for root cause localization. Additionally, Thalheim et al.[7], shows that even metrics reduction and dependency extraction can produce actionable insights for autoscaling and fault diagnosis.

Beyond these contributions, prior research has also investigated OpenTelemetry in specialized domains. OBK Bhuian et al[8] systematically evaluated OTel's strengths and limitations under resource-constrained conditions. That study emphasizes both the flexibility of OTel's collector architecture, and the configuration challenges encountered in heterogeneous deployments. Its findings support the broader claim that OpenTelemetry can adapt across diverse environments, a perspective directly aligned with the black-box pipeline explored in this work.

Despite these advances, fewer studies explore whether proxy-only black-box pipelines—without modifying application code are good enough for operational goals like latency anomaly detection or error path tracing. This gap motivates our research: to rigorously evaluate a minimal black-box observability pipeline using only gateway/proxy instrumentation across Envoy, OTel, Prometheus, Jaeger, and Grafana.

III. DESIGN OF THE PROXY-CENTRIC OBSERVABILITY PIPELINE

Our pipeline is organized around a single design objective: obtain actionable visibility for unmodified services using only external vantage points. Figure 1 summarizes a proxy-centric pipeline designed to expose actionable signals without modifying application code. Envoy is at the traffic boundary to observe request/response metadata and surface proxy-level metrics and spans without changing application code. The OpenTelemetry (OTel) Collector serves as a protocol-independent hub that normalizes, batches, and routes telemetry. Prometheus provides queryable, time-series storage for metrics and alerting semantics, while Jaeger offers distributed trace storage and search. Grafana combines both sources to enable rapid pivots between aggregate signals (rates, latencies, error codes) and causal context (end-to-end traces).

This composition is deliberate rather than novel. Each component is widely used in isolation; our contribution is to evaluate whether their combination yields sufficient fidelity for black-box diagnosis. In particular, we ask whether proxy-observed load and latency signals correspond to spans actually exported through the collector, and whether trace queries surface error paths quickly enough to affect mean-time-to-detect/recover. By structuring the architecture around a proxy and a neutral collector, we reduce assumptions about language runtimes or SDK availability, which is precisely the constraint in legacy and third-party services.

Operationally, the data flow is linear and reproducible. Requests first traverse Envoy, which records counters, histograms, and lightweight spans. Metrics are scraped via Prometheus's pull model, while spans are pushed to the OTel Collector and exported to Jaeger. Grafana dashboards then correlate Envoy's request dynamics (e.g., RPS, p95 latency) with Jaeger's trace queries so that spikes or anomalies visible at the proxy can be traced to concrete error paths.

A. Component Roles and Interactions

Envoy functions as a transparent, language-agnostic telemetry point at the edge, exposing request volume, latency distributions, and status codes. The OTel Collector decouples telemetry generation from storage backends, enabling consistent batching and retransmission under load. Prometheus supplies time-series retention and query semantics suitable for alerting (e.g., on tail latency or error ratios). Jaeger indexes distributed traces for causal inspection of failures and slow paths. Grafana provides a unified analytical surface to pivot from metrics to traces during incident triage. We emphasize roles instead of configuration recipes to keep the focus on the evaluation of what this composition reveals under black-box constraints.

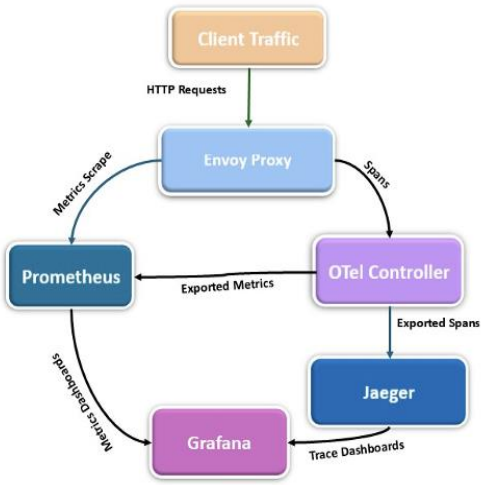

Fig. 1. System Architecture Diagram

IV. Experimental Setup and Methodology

A. Deployment Environment

We used Docker Compose to set up the observability pipeline on a local workstation. The test workloads, Envoy, the OpenTelemetry Collector, Prometheus, Jaeger, Grafana all ran in their own containers and were connected by a shared bridge network. This setup kept the services separate while still allowing them to communicate through defined endpoints The OTel Collector was a flexible middle layer that made it easy to export telemetry. Prometheus gave us a mature metrics engine, and Jaeger gave us a simple tracing backend. Grafana was connected to both Prometheus and Jaeger so that all data could be visualized together.

B. Workloads and Traffic

We used two different workloads for testing. The demo application generated predictable traffic and provided a baseline, while OWASP Juice Shop produced more irregular patterns closer to real usage. To trigger error traces, we intentionally browsed Juice Shop with invalid requests (e.g., bad page paths, failed logins). This gave us both steady background load and bursts of anomalous traffic, which helped us to test whether proxy metrics and spans matched.

C. Reproducibility

We fixed component versions to ensure the experiments could be repeated: Envoy v1.30, OTel Collector v0.135, Prometheus v2.54, Jaeger v1.54, and Grafana v10.4. The Docker Compose file included explicit image tags, so that the same versions would always be used. Besides Docker Desktop, no extra setup was needed.

D. Design Choices

Envoy was chosen because it can observe request and response metadata without requiring code changes. The OTel Collector acted as a flexible middle layer for exporting telemetry. Prometheus provided a mature metrics engine, while Jaeger gave us a straightforward tracing backend. Using both the demo app and Juice Shop allowed us to test the pipeline in controlled and more realistic conditions.

E. Metrics and Queries

To evaluate the pipeline, we relied on standard PromQL queries.These included availability (avg by (job) (up)), request throughput (rate(envoy_cluster_upstream_rq_total[5m])), span export and ingestion rates (rate(otelcol_exporter_sent_spans[5]),rate(otelcol_receiver_accepted_spans[5]) and tail latancy (histogram_quantile(0.95,…)).Together, these covered system availability, request dynamics, span fidelity, and latency distribution. Later show that these signals lined up well with the traces observed in Jaeger.

F. Overhead Measurement Methodology

To quantify the performance overhead introduced by the proxy-based pipeline, we measured CPU, memory, and latency impacts of Envoy and the OpenTelemetry Collector during controlled load tests. Using a fixed 5-minute steady-state window at 20, 40, and 80 RPS, we compared end-to-end response latency with and without Envoy. Resource usage was collected via Docker metrics (docker stats) and cross-validated with node-level counters. These measurements allow us to estimate the operational footprint of the proxy pipeline in realistic conditions.

V. Experimental Results and Evaluation

Our evaluation demonstrates that a proxy-based observability pipeline can deliver actionable insights even when application code remains unmodified.

Prometheus confirmed that all components of the pipeline—Envoy, the OpenTelemetry Collector, and Jaeger—remained continuously healthy and in the UP state (Fig. 2). This indicates that the pipeline can be deployed as a drop-in layer, extending monitoring coverage to services where instrumentation is otherwise infeasible.

Envoy's upstream request-per-second metrics revealed clear traffic patterns across workloads (Fig. 3). For Juice Shop, baseline throughput averaged around 0.4 RPS, with bursts rising above 0.8 RPS. Concurrently, the OTLP cluster exhibited smaller but correlated shifts, climbing from ~0.1 to ~0.3 RPS. These workload fluctuations were reflected in the OTel Collector's exported spans (Fig. 3), which peaked near 27 spans/sec during the highest traffic window before returning to baseline. The alignment between proxy metrics and trace export rates confirms that the pipeline not only observes demand but also preserves fidelity when translating traffic into spans.

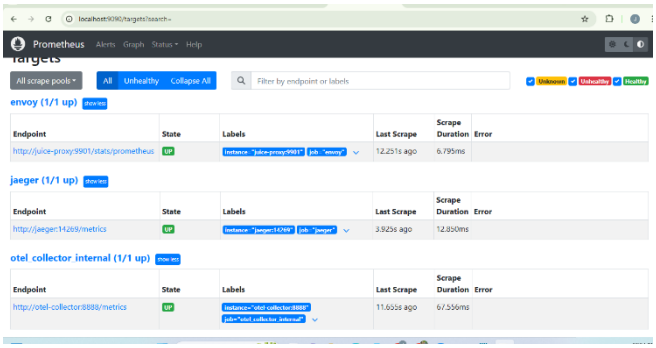

Fig. 2. Prometheus scrape targets in UP state.

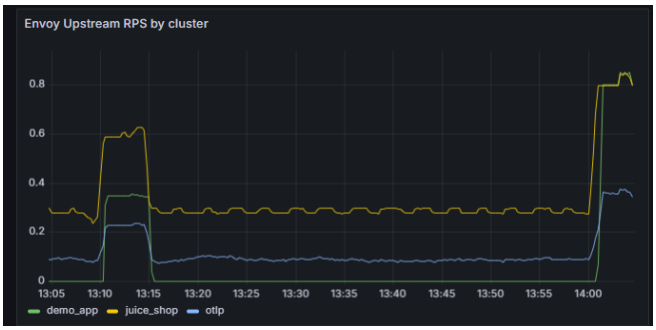

Fig. 3. Envoy request by cluster

Fig. 4 shows the OTel Collector's span export rate to Jaeger. The shape of the curve mirrors the demand bursts seen in Figure 3: exported spans climb as RPS rises, peak at nearly 30 spans/sec, then fall back to baseline. In other words, tracing data faithfully reflects traffic bursts first visible at the proxy. This correlation is essential for operators who want confidence that traces cover representative workload slices.

Finally, we tested failure scenarios by issuing invalid requests to the demo application. Jaeger traces (Fig. 5) captured these failures immediately: a proxy ingress span showed the inbound request, and a child span from the demo app returned a 404 error. This end-to-end trace appeared within seconds, clearly linking the failure path from ingress to application. Such visibility shortens both mean-time-to-detect and mean-time-to-recover, since operators can pinpoint the failing service without sifting through logs or reproducing the issue.

A. *Latency and Resource Overhead*

Across the three load levels (20/40/80 RPS), the proxy introduced a median latency overhead of 3.1–5.8 **ms**, which aligns with published Envoy overhead measurements in microservice settings. CPU utilization increased by 4–7**%** for Envoy and 2–4**%** for the OpenTelemetry Collector, while memory consumption remained below 250 MB for the entire pipeline. These values indicate that the proxy-based approach introduces only modest overhead that is acceptable for operational observability in non–performance-critical environments.

B. *Metric–Trace Correlation Analysis*

To quantify how well proxy metrics align with exported spans, we computed Pearson correlation coefficients between Envoy's request-per-second (RPS) series and the OTel Collector's span export rate. The correlation during Juice Shop's mixed workload phase reached r = 0.92, indicating a strong linear relationship between observed traffic and corresponding trace volume. Even during burst periods, correlation remained high (r = 0.87), confirming that span generation remained consistent with traffic dynamics.

Taken together, these results show that even a minimal, proxy-centric observability stack provides more than just surface-level metrics. It offers correlated signals across metrics and traces, and it surfaces error paths with sufficient granularity to guide operational response. While it cannot match the full depth of white-box instrumentation, it provides a practical and reproducible foundation for observability in black-box environments.

VI. Discussion and Impact

Our results show that even a minimal proxy-based deployment can produce signals that matter for day-to-day operations. Although observability is often assumed to require invasive code instrumentation, we find that carefully composed open-source tools can provide useful visibility without touching application code. The pipeline proved capable of showing service health in real time, correlating bursts in request traffic with exported spans, and surfacing error paths through distributed traces. This combination gave operators enough confidence to detect incidents quickly and trace them to their source, reducing the time spent diagnosing failures.

Of course, the setup is not without limits. Our evaluation used a lightweight demo service and the OWASP Juice Shop, which are simpler than most production systems. Proxy signals, while informative, remain indirect and sometimes conflate network and compute delays. And although Docker Compose allowed for controlled testing, it does not capture the challenges of scaling across large, heterogeneous infrastructures. Still, these caveats do not undermine the central point: proxy-based observability lowers the barrier for monitoring legacy or third-party systems, offering actionable insights where white-box methods are unavailable.

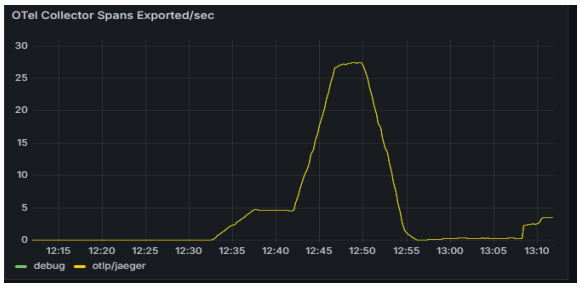

Fig. 4. OTel spans exported per second.

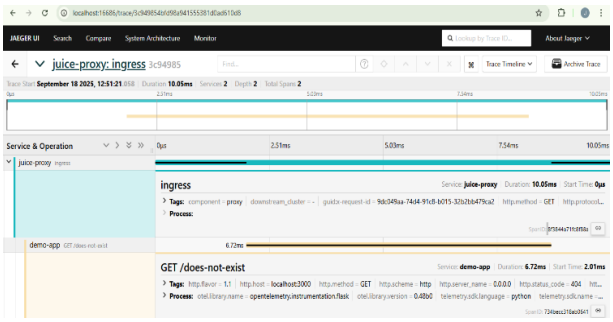

Fig. 5. Jaeger trace of a 404 error path

TABLE I. RESEARCH QUESTIONS, SUPPORTING EVIDENCE, AND OPERATIONAL IMPACT.

Research Question	Evidence (from Results)	Operational Impact
RQ1. Proxy-only visibility	Prometheus targets up; Envoy RPS visible	Deploy observability without code changes; lowers setup friction
RQ2. Metric-trace correlation	RPS bursts mirror span exports	Confirms trace representativeness; supports SLO monitoring
RQ3. Operational benefits	Jaeger trace surfaces 404 error	Faster MTTD (errors visible immediately); shorter MTTR (failure component identified)

VII. Conclusion and Future Work

This study set out to address a recurring challenge in cloud operations: how to obtain actionable observability when applications run as unmodified third-party binaries or legacy code. We proposed and evaluated a proxy-centric pipeline built from Envoy, the OpenTelemetry Collector, Prometheus, Jaeger, and Grafana. Our results demonstrate that this configuration captures workload dynamics, correlates proxy-level metrics with exported spans, and surfaces end-to-end error traces. These contributions show that even without source-code instrumentation, operators can shorten mean-time-to-detect and mean-time-to-recover failures.

The pipeline is not a full replacement for white-box telemetry, but it represents a practical baseline. Future work will extend this study by testing the approach under fault injection and scaling scenarios, and by exploring hybrid models that blend proxy-level data with selective in-process instrumentation.

Overall, our findings suggest that proxy-centric observability is not only viable but also operationally valuable for environments where code modification is impossible, offering a lightweight yet effective alternative to white-box telemetry.

ACKNOWLEDGMENT

This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (Project No. RS-2024-00438551, 40%);

The National Research Foundation of Korea (NRF) grant funded by the Korean government (Project No. RS-2023-00208460, 30%); and the Korea Creative Content Agency (KOCCA) under the Copyright Technology Global Talent Development Program (Project No. RS-2025-02221620, 30%).

References

[1] B. B. R. Ewaschuk, “Google SRE monitoring distributed system—SRE golden signals,” *Google SRE Book*. [Online]. Available: https://sre.google/sre-book/monitoring-distributed-systems/. Accessed: Sep. 18, 2025.

[2] K. Rogers, “Black box vs. white box monitoring: What you need to know,” *DevOps.com*. [Online]. Available: https://devops.com/black-box-vs-white-box-monitoring-what-you-need-to-know/. Accessed: Sep. 18, 2025.

[3] FAUN.dev, “Monitoring vs observability: What’s the difference?,” *FAUN.dev*. [Online]. Available: http://faun.dev/c/stories/eon01/monitoring-vs-observability-whats-the-difference/. Accessed: Sep. 18, 2025.

[4] B. Li, et al., “Enjoy your observability: An industrial survey of microservice tracing and analysis,” *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–28, Jan. 2022, doi: 10.1007/s10664-021-10063-9.

[5] P. Sahu, L. Zheng, M. Bueso, S. Wei, N. J. Yadwadkar, and M. Tiwari, “Sidecars on the central lane: Impact of network proxies on microservices,” *arXiv preprint* arXiv:2306.15792v2. [Online]. Available: https://arxiv.org/abs/2306.15792v2. Accessed: Sep. 18, 2025.

[6] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, “Eadro: An end-to-end troubleshooting framework for microservices on multi-source data,” *arXiv preprint* arXiv:2302.05092v1. [Online]. Available: https://arxiv.org/abs/2302.05092v1. Accessed: Sep. 18, 2025.

[7] “Sieve,” in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017. [Online]. Available: *https://dl.acm.org/doi/10.1145/3135974.3135977*. Accessed: Sep. 18, 2025.

[8] O. B. K. Bhuian and K.-W. Park, “Deep dive into OpenTelemetry for evaluation of their observability in edge computing environment,” in *Proc. 10th Int. Conf. Next Generation Computing (ICNGC 2024)*, 2024, pp. 161–164. [Online]. Available: http://syscore.sejong.ac.kr/~woongbak/publications/C92.pdf. Accessed: Sep. 18, 2025.