

Cloud Observability and Distributed Tracing in HTTP Webservers and Temporal Workflows

Vedanth Sriram

Dept. of Information Science and Engineering
RV College of Engineering
Bangalore, India
vedanthsriram.is22@rvce.edu.in

Dr. Merin Meleet

Dept. of Information Science and Engineering
RV College of Engineering
Bangalore, India
merinmeleet@rvce.edu.in

Abstract—Cloud-native setups now lean heavily on tools like Temporal to manage extended business processes. Though tracing works well for instant back-and-forth calls, it tends to fall apart when jumping between async stages - leaving gaps that make troubleshooting harder. Here comes a tracing method built to handle replays, tailored for Temporal environments using Elastic APM alongside standard W3C Trace Context. It adds specially crafted interceptors into Go and Python SDKs, stitching traces together seamlessly - from web entry points through task routing, step-by-step actions, all the way into databases or remote services. With this setup, each piece talks to the next without losing context, even after pauses or retries. Facing tough issues unique to Temporal, the system tackles how workflows replay, must run predictably, yet pass tracking data between different languages. Built for real use, it runs where Go and Python services connect via Temporal Cloud. Tests showed almost no added delay during typical tasks, even when rebuilding full traces across staggered operations. Choices made during development reveal compromises, ways to monitor code that replays, along with hurdles in gathering vast amounts of live data from persistent workflows. Engineers aiming to link tracing in mixed-language apps gain a solid starting point - no need to adopt unproven or overly tangled OpenTelemetry setups.

Index Terms—Elastic APM, Microservices, Distributed Tracing, Observability, Temporal, W3C Trace Context

I. INTRODUCTION

Tiny services now shape most big apps in the cloud. Their strength lies in staying separate, adapting fast, failing quietly. Splitting large programs into small pieces lets teams adjust parts without touching everything else. Yet tracking what happens across these fragments gets messy quickly. One click by a person might race through many tiny services, jump between background tasks, tap several data stores, ping outside systems before answering back. Problems pop up anywhere along that path. Finding exactly where things went wrong feels like hunting fog in a storm.

Seeing how services talk together gets easier with tools like Dapper or newer ones built on OpenTelemetry. A single ID starts at the front door of the system, then travels through HTTP or RPC calls via set header formats. But when things run out of order, standard tracking methods fall apart. Workflows in systems such as Temporals keep going without waiting for the original trigger to stay alive. Later on - sometimes minutes, sometimes much longer - a web server's task might run somewhere far removed. A separate worker

process picks it up, possibly inside another physical cluster altogether. Different programming language? That can happen too. Because of this gap in time and place, standard tracing tools often fail to link parent and child steps. The connection slips away at every handoff point.

Bits breaking apart cause big hiccups in daily operations. One problem? Service maps inside monitoring systems stay half-drawn, hiding how pieces truly connect. When background jobs run sluggish database calls, those signals float loose - untied to any user move that started them. Without clear links between broken workflows and incoming API hits, finding root causes drags on far longer than needed. Tracing one action across systems means hunting through scattered log files, task timelines, and separate metric views just to see what happened.

A new tracing system works inside Temporal-based async environments while knowing when replays happen. Built on W3C Trace Context, it moves trace IDs through Temporal payloads so parents and children stay linked correctly. Because of how it reads execution state, duplicates vanish even when workflows restart. Instead of guessing where data flows, it follows actual paths - whether in Go or Python SDKs - with tight alignment across languages. Hidden under normal operations, database steps join the bigger trace picture without breaking stride. What makes it different? It acts predictably across services yet respects the locked-down conditions Temporal requires. Not every tool can walk that line without cracking open the engine first.

II. SYSTEM ARCHITECTURE

A. Runtime Environment and Infrastructure

Out there, the setup runs on a network of linked tasks guided by workflows, tied together with web APIs. Instead of Kubernetes, it leans on Nomad to handle where things run - each piece tucked neatly in Docker boxes. These bits talk to one another using Consul, built right into the structure for finding what's online. Behind the scenes, details like monitoring keys slip quietly into containers when they start, pulled from Vault so nothing stays exposed. Work happens step by step, triggered without waiting, powered by Temporal behind PostgreSQL stores holding data tight.

Built to handle many languages on purpose. Web requests come in through Go, running on Gin for speed. Workflow tasks needing power also run in Go. On the other hand, Python handles data analysis jobs with Flask under the hood. Parsing tricky formats happens in Python too. Integration scripts lean toward Python when needed. Communication between systems uses Temporal Cloud with two-way TLS turned on. That keeps workflows tracking correctly across services. For testing things locally, developers use Temporalite instead. It mimics the main system but runs light and stays in memory.

Straight into Elastic APM goes the data, no middle layers involved. Direct links between agents and server replace any extra collector steps. Out of nowhere, Go apps appear connected using Elastic's own Go agent, plus tailored tools for Gin, also SQL tracking. Meanwhile, Python pieces hook in through their matching Elastic APM module for Python. The flow stays lean, built without detours. Skipping middle layers cuts down system clutter along with running costs, yet demands close attention to passing context properly due to missing step-by-step data shaping tools. What runs behind the scenes, including setup specifics, gets briefly laid out in Table I.

TABLE I
TOOL COMPONENTS AND ENVIRONMENT DETAILS

Component	Technology and Implementation
Orchestration Discovery and Secrets	Nomad Job templates and Docker containers Consul for discovery, HashiCorp Vault for APM credentials
Temporal Server	Temporal Cloud with mTLS for production, Temporalite for local development
Go Services	Gin framework and go.elastic.co/apm/v2
Python Services	Flask framework and elastic-apm library
APM Backend	Elastic APM Server using direct agent to server export

B. Request Lifecycle

A single journey through a monitored task moves step by step across different systems. It kicks off when someone outside sends an HTTP call to a gateway built in Go or Python. Right after arrival, tools inside the gateway catch that message to check for a trace ID. Without one present, the system creates a fresh UUID v4 code on its own. Then comes the start of a main monitoring record - recording details like what came in, which HTTP verb was used, where it came from, and exactly when it arrived.

Starting from the root transaction, business logic runs inside the request handler until it triggers a Temporal workflow. Right at that point, the Temporal client SDK takes over to send the signal. Wrapped around this call, custom interceptors on the client step in quietly. They pull out the current APM transaction details along with the app's request ID. Into a transfer-friendly structure those values get packed. Before anything reaches the Temporal Server, that data slips straight into the workflow headers alongside the rest of the message. Later on, a free worker picks up the job by checking the assigned queue. A moment before running the actual workflow

code, special tools step in through custom entry points. Hidden inside message headers, the encoded tracking data gets pulled out carefully. That decoded signal tells the nearby monitoring tool to open a fresh activity link pointing backward. Storing each phase of progress happens first within the time-aware server's memory core. One at a time, tasks wait quietly until summoned for processing steps ahead. Instead of skipping forward blindly, the system honors prior traces as guiding threads.

While the workflow runs, it sets up different tasks. Because of outbound interceptors in the workflow, tracing details move into each task call. Once a task begins, inbound interceptors tied to it start smaller tracking units that last as long as the task does. During this time, anything that follows - like calling outside services or querying databases - is recorded under those tracking units. This builds a full map of the entire process from start to finish.

III. REPLAY-AWARE TRACE PROPAGATION

A. Cross-Boundary Metadata Injection

Built around the W3C Trace Context, the tracing setup uses a common way to pass tracking IDs between services. Because it follows this norm, Go and Python parts work together without hiccups. Compatibility stays strong even if shifting later toward OpenTelemetry becomes necessary. The exact headers sent via Temporal's payload appear in Table II.

TABLE II
TRACE CONTEXT PROPAGATION IN TEMPORAL

Header Key	Format	Purpose
traceparent	W3C standard string	Main trace context identifying the trace and parent span
x-pure-requestid	custom UUID string	Company level identifier for log correlation

A version number kicks off the W3C traceparent string, followed by a unique trace ID separated with dashes. Next comes a span ID - this one points to the direct parent in the chain. After that sits a flags piece showing sampling status. Instead of building its own format, the Temporal client pulls data from the current Elastic APM context during outgoing workflow launches. Then it lines up those pieces into the expected dashed structure. The full sequence lands as a standardized text snippet.

Picking how to turn data into bytes matters a lot here. Instead of packing headers inside JSON or using format tied to one programming language, they go straight into raw bytes. When working in Go, you switch the string directly into an array of bytes, then slot it into Temporal's storage structure. Over in Python, things shift by turning the text into UTF-8 byte form. These bare-metal bytes make certain any language can talk to another - no hiccups. So when a Go-based server tucks away tracing details, a Python process pulls them back apart just fine. Misreads during unpacking simply do not happen.

B. Workflow Hierarchy and Interceptor Logic

Just setting up an ongoing trace isn't enough - getting the structure right matters just as much. A developer could pull the incoming trace info and launch the workflow with it, yet still reuse that same context for each activity. That approach flattens everything, making every step seem like it came straight from the initial HTTP call. Instead of seeing which actions belong to which process, they all sit at the same level. When viewing traces in Kibana, figuring out what ran inside which workflow becomes guesswork.

Fixing the hierarchy starts with the worker's inbound interceptor making one key change. Once the workflow transaction begins - using the original traceparent - something different happens next. The stored traceparent inside the execution context gets replaced on purpose. Rather than keep the incoming parent span ID, a fresh traceparent is built instead. This new version uses the span ID just created for the workflow itself. When the outbound interceptor later sends context to following activities, it carries this updated identity. That shift means every activity clearly links under the workflow as its real parent. As a result, the APM view shows a clean cascade of steps - each aligned right beneath the main flow.

C. Deterministic Replay-Awareness

Every time a worker vanishes mid-task, another one steps in without delay. From scratch, it walks back through every recorded step of the workflow. The system leans on event sourcing to stay dependable under pressure. Each past action gets rerun as if repeating a path already traveled. What changes? Outside impacts get silenced by the SDK during playback. Code executes once more, yet nothing reaches beyond the boundaries set before.

Most APM tools do not detect when workflows are being replayed. When replays happen without intervention, each one triggers another full batch of transactions sent to the monitoring backend. These repeated entries clutter logs, distort performance numbers, and turn troubleshooting into guesswork. Built-in awareness of replay states lets this system avoid creating redundant records altogether.

Right there in the Go code, the workflow interceptor looks at the built-in replay flag from the context. When it sees replay mode active, it goes ahead anyway and pulls trace headers out of the metadata. That step matters - without it, fresh activities after catch-up would lose connection to the initial trace. Yet here's the twist: no APM workflow transaction gets made at all. During replay, the APM agent stays quiet on purpose, like it's holding its breath.

Python hits a snag here. Inside this setup, workflows run in a tight sandbox meant to block unpredictable actions - pulling system time, making random values, reaching out over networks. That isolation keeps things predictable. But the Elastic APM tool leans hard on networking to send its reports, also using thread-specific storage to track state - moves that the sandbox flat-out denies. Working around it? The Python layer peeks at whether the workflow is currently replaying. When it sees real-time execution, not playback, it taps into

a special lenient mode offered by Temporal's toolkit, briefly loosening those sandbox rules. Right where things run free, that's when the APM deal kicks in - trace details get adjusted on the fly. When those steps finish, tight controls snap back fast, keeping the system predictable even though data sneaks out just in time.

IV. DOWNSTREAM INSTRUMENTATION

Most delays happen right where work gets done. When code runs, that is when things break or slow down. Seeing into these moments means placing trackers directly in those tasks. Without clear sight there, the full picture stays hidden. Where business rules execute becomes the key spot to watch.

Inside Go programs, tasks often talk to outside private systems using HTTP or special remote call methods. Wrapped around the usual HTTP transport layer, outgoing web requests get tracked automatically. Using request-making functions that include context matters a lot here. Without attaching the right task context, each HTTP call creates a disconnected monitoring segment - floating free, missing from the full journey map. When hitting databases, a modified PostgreSQL driver handles tracking behind the scenes. A single SQL query inside any activity brings up a child span showing how long it took plus the cleaned-up version of the command. Instead of sending every detail by default, the system lets admins filter out noisy database traces using environment settings - so only meaningful operations get recorded. This way, heavy but unimportant traffic doesn't clog pipelines, yet key flows still leave clear tracks behind.

Python's flexibility shines when tools like Elastic APM step in during startup. Right away, the agent begins watching how key packages - urllib3, requests, psycpg2 - are used. Each time a script reaches out over the network or talks to a database, it gets noticed. Context tells the system which operation is running, so each action links correctly into place. Less setup means fewer repetitive tasks for analysts working models. Full visibility stays intact, even as services multiply across systems.

V. EVALUATION AND PERFORMANCE

A. Latency Measurements

Testing how well the tracking system works involved using a typical Temporal workflow that manages changes to virtual machines. This check happened inside isolated containers so only the monitoring tool's effect could be seen clearly. With the full monitoring setup turned on, ten test requests were sent through. Afterward, another ten went through but with monitoring switched off using config settings. What each round took in time is shown in Table III.

Surprisingly, turning off APM led to marginally longer response times compared to when it was active. This small gap wasn't caused by the monitoring tools - stats show it matches normal fluctuations in the Docker network and how tasks are scheduled. What stands out? The extra steps like intercepting requests, formatting headers, and starting transactions leave no trace in overall performance.

TABLE III
END TO END PERFORMANCE EVALUATION FOR TEN REQUESTS

Metric	APM Enabled	APM Disabled
Average	3.771 milliseconds	5.097 milliseconds
Minimum	2.524 milliseconds	2.619 milliseconds
Maximum	6.355 milliseconds	8.789 milliseconds
Median	3.388 milliseconds	4.922 milliseconds
95th Percentile	4.799 milliseconds	6.633 milliseconds

B. Per-Component Overhead

Looking closer at how well it runs, Figure IV shows the added delay in microseconds each part contributes, drawn from performance measurements.

TABLE IV
PER COMPONENT INSTRUMENTATION OVERHEAD ESTIMATES

Component	Unit Overhead	Relative Impact
HTTP Middleware	10 microseconds per request	Less than 0.1 percent of API call
Outbound HTTP	3 microseconds per call	Less than 0.01 percent of network call
Database Span	20 microseconds per query	Approximately 1 percent of fast query
Temporal Transaction	30 microseconds per workflow	Less than 0.001 percent of execution

Even when handling slow tasks - like talking to outside systems that take many seconds - the tiny cost of starting a transaction, just 30 microseconds, barely matters at all. Instead of slowing things down, the system slips work like packing data, grouping messages, and sending them over the network into a separate background process run by the Elastic APM agent. That offloading means your main code runs free, untouched by those behind-the-scenes chores.

C. Observability Improvements

What stood out most was how clearly things became visible after the change. Before setting up the special interceptors, tracking data broke apart right where Temporal took over. When someone looked into a broken user request using Kibana, they only saw an HTTP POST marked as success - then nothing. From that point on, worker actions, task crashes, and issues in the database showed up scattered, unrelated, each logged in its own place. Finding what really went wrong meant hunting through separate logs by hand, linking pieces without help.

After setup, Kibana's service maps showed clear trace paths linking Python and Go services. Rooted in the original API call, the APM interface built a full step-by-step view. Opening each segment exposed the workflow run as a sub-node, followed by deeper layers marking individual actions - each stamped with duration and context. With everything linked,

engineers could match slow responses to failing background jobs at a glance. Fixing live issues got faster; hunting through logs by hand became obsolete.

VI. LIMITATIONS

Though the system links distributed tracing through Temporal lines well, some flaws still stand out. Not every part scales smoothly under heavy load, especially since nothing gets left out when collecting data. Each incoming request, each running workflow, each active task - each one triggers a complete trace record. Full visibility shows up clearly in early tests with light traffic, yet that clarity fades when volume spikes. Handling millions of workflows per day fills storage fast, overwhelming the Elastic APM Server's ability to take in more. The flood of traces hits hard, slowing everything down just when speed matters most.

Heavy workloads can push systems beyond normal limits. Worker memory might struggle when queues grow deep, yet this wasn't tested at full capacity. Trace data adds up fast, even if each bit seems tiny. Processing many actions at once increases pressure on the CPU. How much it slows things down remains unclear without real-world testing. The total effect across thousands of tasks needs closer observation.

One more thing. Right now, the system does not handle certain advanced Temporal functions well - especially continue-as-new. If a workflow runs too long and builds up heavy history, Temporal can restart it cleanly with a blank slate through that feature. Since this process actually spawns a completely new instance behind the scenes, tracing gets disconnected. The link only stays alive if someone pulls out the existing traceparent value and pushes it into the next run's headers. But at present, that transfer step has not been built in.

Thirty days of waiting for someone to click approve? That breaks how APM usually tracks things. While the system waits, it keeps a record alive - too long, really. Data gets cleared out by then, rotated away before the trace finishes. Visuals glitch or vanish because the pieces are gone. Instead of one stretched-out timeline, better to snap separate events together later. Continuous tracking just does not fit when delays stretch across weeks.

VII. CONCLUSION

This study introduced a resilient tracing system aware of replays, built for distributed setups where multiple languages meet delayed Temporal workflows through Elastic APM. Instead of ignoring gaps, it fills visibility holes between services that speak different coding tongues while handling async handoffs. Rather than fail at boundary lines, it moves W3C headers across deferred operations with precision few achieve. Not held back by limits, it works within tight Python SDK sandboxes without breaking rules or rhythm. Where structures misalign, it fixes broken chains in workflow parent-child relationships automatically. Without doubling effort, it spots repeated runs and blocks redundant data before clutter begins.

This setup keeps tracking strong from start to finish without slowing down key operations. Instead of separate views, it combines API gateway actions with background tasks into one clear picture. That clarity makes problem solving faster when systems get complicated. Looking ahead, efforts will shift to smarter sampling methods that cut down data storage needs. Long jobs that hang around too long in storage are also getting fixes. On top of that, plans include smoother steps to match OpenTelemetry standards directly.

REFERENCES

- [1] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 3–18.
- [2] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google Technical Report, 2010.
- [3] J. Kaldor et al., "Canopy: An end-to-end performance tracing and analysis system," in Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 34–50.
- [4] B. Li et al., "Enjoy your observability: an industrial survey of microservice tracing and analysis," Empirical Software Engineering, vol. 27, no. 1, p. 25, 2021.
- [5] X. Zhou et al., "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 243–260, 2019.
- [6] OpenTelemetry Authors, "OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability," 2024. Available: <https://opentelemetry.io>
- [7] R. R. Sambasivan et al., "So, you want to trace your distributed system? Key design insights from years of practical experience," Carnegie Mellon University, Technical Report CMU-PDL-14-102, 2014.
- [8] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," ACM Transactions on Computer Systems, vol. 35, no. 4, pp. 1–28, 2018.
- [9] S. Ooka et al., "Constructing a service process model based on distributed tracing for conformance checking of microservices," in 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, 2022, pp. 1045–1055.
- [10] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in Network Operations and Management Symposium, 2020, pp. 1–9.
- [11] T. Chen et al., "Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications," in 2021 IEEE 10th International Conference on Cloud Networking, 2021, pp. 40–47.
- [12] P. Las-Casas et al., "Sifter: Scalable Sampling for Distributed Tracing, without Feature Extraction," in Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 312–325.
- [13] A. Nedelkoski et al., "Anomaly detection and classification using distributed tracing and deep learning," in 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2019, pp. 241–250.
- [14] G. Yu et al., "MicroRank: End-to-end latency issue localization with extended rule-based graph propagation in microservices," in Proceedings of the Web Conference 2021, 2021, pp. 2287–2298.
- [15] D. Al-Azzoni et al., "Informed and Assessable Observability Design Decisions in Cloud-native Microservice Applications," arXiv preprint arXiv:2403.00633, 2024.