happiest minds
The Mindful IT Company
Born Digital . Born Agile

Observability in **Cloud-Native Applications**

Abstract

Given the distributed auto-scaling nature, frequent deployments and the many potential points of failures and latencies of modern cloud-based applications in ephemeral environments; these systems are increasingly more complex to monitor and debug in production, thus making observability a fundamental requirement to cut down time spent on triage. With Metrics, Logs, and Traces forming the so-called "Pillars of Observability[1]", this paper seeks to explore –

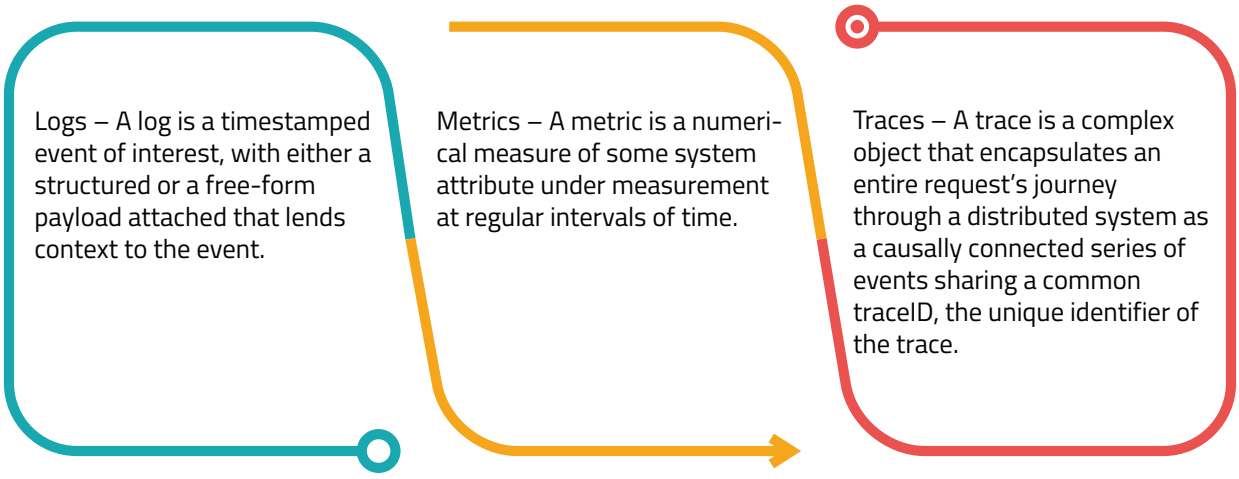
1. Each pillar in-depth – their respective capabilities, how they combine to provide a holistic view of a complex distributed system, and their respective limitations when used in isolation.
2. Open Standards and Tools.
3. Analytics on Observability Data – critical path analysis, anomaly detection, error analysis, etc.
4. Current and Emerging Trends.

Introduction

Traditional monitoring typically revolves around defining a few crucial metrics and having automated monitoring and alerts on any threshold violations on those metrics. These metrics serve as broad symptoms to indicate a larger problem in a specific part of the system. However, they provide limited help in nailing down the exact source or root cause, as they cannot be correlated with the appropriate log events. Tracing fills this very gap by helping replay and investigate what happened with an individual transaction or a group of transactions (e.g., a sudden degradation in a component) - thereby providing a built-in debuggability in production code execution, where a debugger cannot be attached.

Monitoring looks out for failures that are anticipated and defined upfront, whereas observability is a broader and a more open-minded approach to navigate and discover the unknown unknowns too. A holistic approach includes health-check endpoints that report the status of running applications to monitoring tools that consume them, application and system metrics that can help in alerting anomalies, timestamped logs rich in context to help investigate the root cause of an issue, and context-rich tracing that can help correlate and debug distributed call traces to pinpoint exact sources of latencies and errors.

Observability is hence comprised of 3 key pillars –



Traces are request-centric and tell the complete story of a request or transaction, whereas metrics are system-centric, providing a high-level view of whether the system works as expected without answering the WHY. Logs provide a timestamp, ordering and the finest-grained detail about events.

Logs

Logging refers to capturing timestamped events traditionally as free-form text, or as structured logs which are more machine-friendly and suitable for log indexing, search and analytics. Timestamps provide a global ordering of events to logs generated from disparate or distributed processes. Event data from all service instances are shipped to a single system for aggregated/centralized indexed storage, querying, analysis and processing.

Unlike metrics, log entries can contain fields with high cardinality, which makes it possible to analyze logs and slice and dice on various dimensions to identify potential causes of issues. Logs are rich in context, and once aggregated, they support search and debugging at a very fine level of granularity over the entire request or transaction at once. Since logs are exhaustive and generally not sampled, they represent the totality of requests and can be used for auditing, billing (from metering logs), analytics for detecting frauds and attacks etc. Logging is also very easy to adopt. On the downside, compared to metrics, they are resource-intensive in their entire lifecycle of generation (as they are usually blocking), transmission, indexing and storage, querying and processing, to retention. They also increase with traffic or load conditions. Unlike traces, logs do not capture causal relationships between events, and hence may lose out on critical clues while investigating issues. The traceID should however be put into the MDC and made part of the log conversion pattern, to ensure consistent logging of the trace identifier in logs, which would in turn help in correlating logs with traces.

ELK (Elasticsearch-Logstash-Kibana)[2] is a popular open source log management solution. Elasticsearch provides log storage with text indexing and search, Logstash or fluent provides log shipping (from hosts) and an optional transformation before writing to log stores like Elasticsearch. Kibana provides a visualization web interface for Elasticsearch data. For high-volume logging, a broker like Kafka is often used as a buffer before Logstash, to keep up with the spikes.

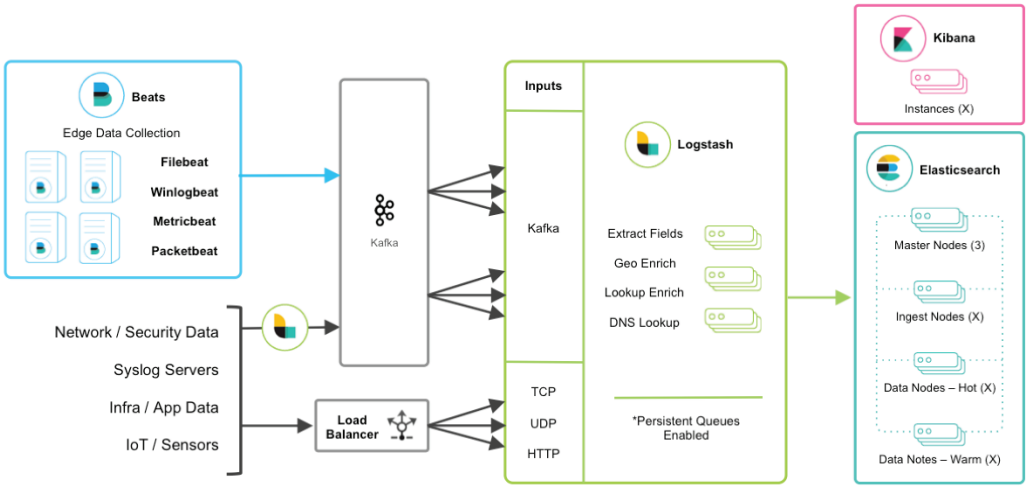


Figure 1: ELK Architecture (Source: http://elastic-stack.readthedocs.io)

Metrics

Metrics deal with aggregate-level measurements about a process (e.g., throughput, latency, data volumes), and they can be monitored over time to identify trends. Being numbers, they are very resource-efficient right from computation (even in distributed mode), to transmission, compression, storage, retrieval and retention. Since events are reduced to aggregates, the memory usage due to instrumentation remains constant w.r.t the number of events, being a function of the number of metrics being tracked and their dimensions. Once collected, they also lend themselves well to further statistical analysis, and for monitoring and alerting when critical thresholds are breached. On the downside, metrics lack context and though good overall "signals" of the system health, they lack the effectiveness that logs and traces can boast of in deeper investigation of issues.

Metrics support dimensions/labels, which lend them somewhat of a relatively static "context". Metrics and their dimensions may refer to instrumentation-side internals, e.g., an outbound RPC endpoint invocation that needs monitoring, HTTP classes of response codes (3XX, 4XX, 5XX) or, they may refer to the targets being scraped – e.g., the service, the cluster, the region, etc.

A metric is identified by the metric name and any (optional) set of labels. The observed values for the metric are reported at every time interval. The metric's observed value, the labels that provide additional meta-information/context to the observation and the timestamp of the observation are stored in the metrics system.

The number of unique values a label can assume is called its cardinality. e.g., for the metric api_hits_total, if one of the labels of interest is the HTTP method (GET, POST), the cardinality of this label is 2. Similarly, other dimensions of interest could be the region of execution – e.g., if the application is deployed in 3 regions, namely {ap-south-1, ca-central-1, and eu-west-1}, that makes the cardinality of this label as 3.

For any metric, a time-series gets created for every applicable combination of its labels (over the label values) – 6, in the above example. Hence cardinality explosion is something to watch out for and avoid, by limiting labels to those dimensions of interest which can assume a small bounded set of values only (typically < 10). This helps in keeping the performance of scraping, indexing, and queries modest. Since high cardinality can overwhelm the timeseries database, most metrics systems impose constraints on cardinality. This limitation again requires complementing metrics with event logs, which have no cardinality limits as such.

Metrics are of two main types –

Counters – represent an ever-increasing measurement. e.g. execution or invocation count, error count. A typical monitoring use-case is to measure the rate of change of the counter, to observe the trend and use it for alerting. Counters reset to 0 whenever the application restarts. System uptime itself can be a valuable counter.

Gauges – represent a point-in-time snapshot of a given measurement that can go up/down over time. e.g., connection pool usage, mem/cpu/disk usage, #active_threads, #active_executions. It could even be a business metric like #active_carts on an ecommerce site.

Client instrumentation libraries typically expose health and metrics endpoints by default and provide OS and runtime engine/process-level statistics (e.g., cpu metrics, jvm metrics, logging metrics, up time metrics, file descriptor metrics) out of the box. Additionally, any key libraries like RPC libraries can be wrapped, instrumented and reused across applications/microservices, for engineering consistency.

Services are typically measured by RED (Rate, Error, Duration) metrics –

Request Rate – #requests/sec the service is serving. Similarly, in case of lambda functions, these would be #invocations/sec.

Request Error Rate - #failed_requests/sec. For functions, this would be #errored_invocations/sec.

Request Duration – duration of the request in a time unit. In case of functions, this would be the duration it took to process an event (including initialization time for the first event).

Infrastructure and offline processing workloads are typically measured by USE (Utilization, Saturation, Errors) metrics –

Utilization – proportion or, %age of resources used (e.g., cpu utilization, memory utilization, connection pool utilization, thread-pool utilization). In case of functions, this could be the number of function instances running as a %age of the maximum concurrency allowed for the function.

Saturation – a measure of extra work that is queued up for servicing, e.g., queue size. In the case of functions, the #invocation requests that are throttled can serve as a measure of saturation.

Errors – count of errors.

A leading open-source tool in the metrics space is Prometheus[3], which supports multiple service discovery mechanisms to dynamically discover and scrape targets in orchestrated environments like Kubernetes. Applications can be instrumented using either Prometheus client libraries or other standard libraries, to expose metrics over HTTP (or, JMX). Exporters can be used to expose these metrics in a format that can be scraped by Prometheus. Prometheus periodically scrapes the configured targets to pull out the exposed metrics and persist them to a time-series database. Grafana provides a user-interface for query/visualization and for alerting on metrics. Prometheus also supports a push-mechanism for ephemeral targets, e.g., lambdas or batch jobs that run and die (not continuous applications). Push gateways can be used to push job metrics, e.g., last run duration, last run timestamp, last success timestamp (can be used for alerts) etc.

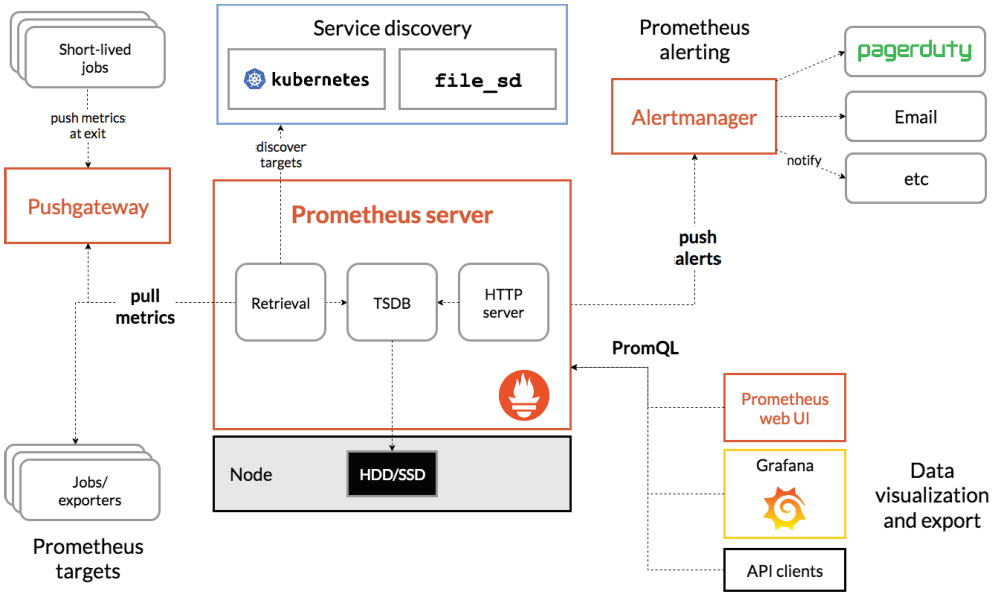


Figure 2: Prometheus Architecture (Source: Prometheus)

Traces

Tracing[4] provides a means to observe the behavior of an end-to-end request/transaction, as it propagates across different endpoints of a distributed system. Conceptually, a completed trace is a transaction tree like what a distributed call graph would stand for, with child nodes representing different dependencies/endpoints that participate in the entire request processing, with each call timed individually. Hence traces are visualized as Gantt charts, representing service dependencies, start time (representation simplified to 0-based instead of a real timestamp here) and durations.

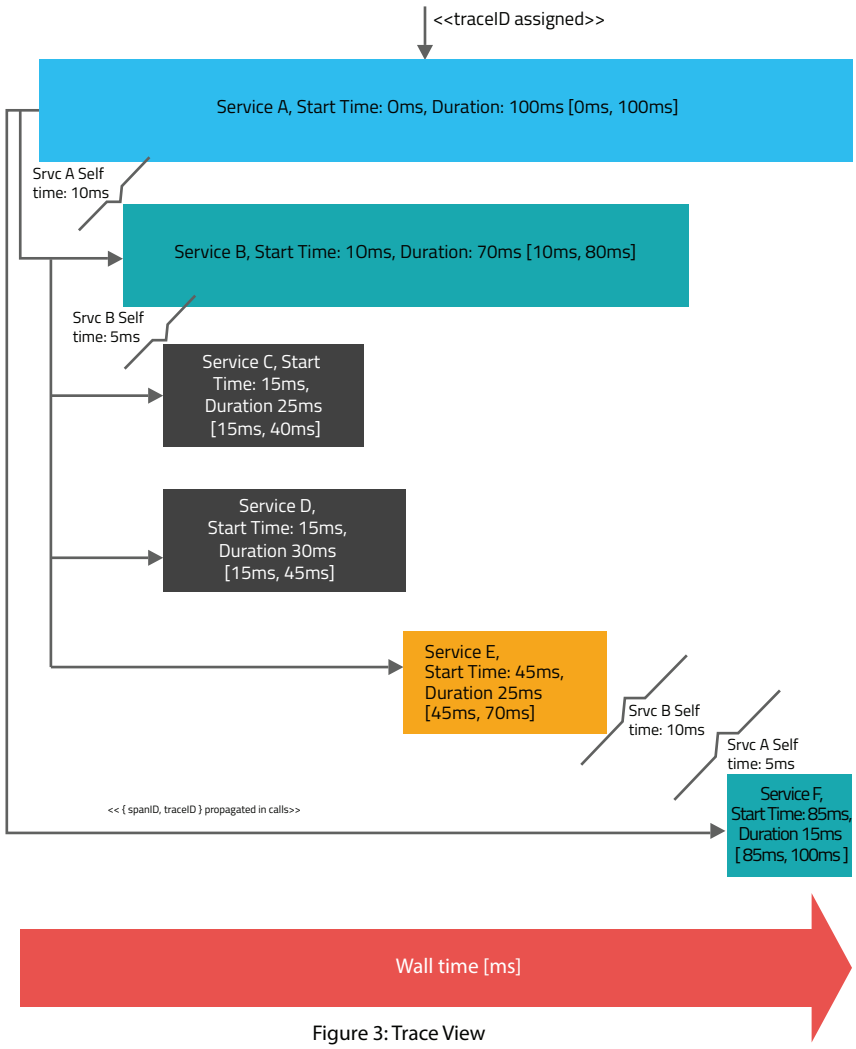


Figure 3: Trace View

Figure 4: A real trace visualization (Source: Zipkin)

Tracing points are usually the ingress and egress points in the services, where a request either enters or, exits a process boundary. Hence a request intercepting layer makes a good tracing point. For every outgoing RPC call, a new child span should be started and closed on call completion (for async calls, this is the on-complete callback point) and any exceptions encountered with the call should be logged to the errored span. In case of messaging, a new child span should be started by the message producer before publishing the message to the message bus and closed once the message is enqueued.

Services are instrumented with tracing libraries. At the system ingress boundary, like the API Gateway, a traceID is assigned to any incoming request as it's end-to-end correlation id, which gets propagated along to all downstream services the request traverses through, via http headers/message headers etc. Any invoked downstream service links it's spans with the respective parent span and trace, thus preserving the lineage. In-process propagation can be achieved via a Threadlocal variable used to embed the current span to make it is accessible anywhere in the execution path within the process.

Once a span returns, the corresponding span gets completed, and tracing agents running on the services capture and send the spans across to a tracing backend/server, where a trace gets reconstructed and persisted to a pluggable storage (commonly Elasticsearch or, Cassandra). Thereupon, the trace can be queried over APIs or, via the Tracing WebUI. The Tracing WebUI supports searching and visualizing traces by traceID or, tags; and supports complex trace comparisons to examine the behavior of an anomalous trace vis-a-viz a normal one.

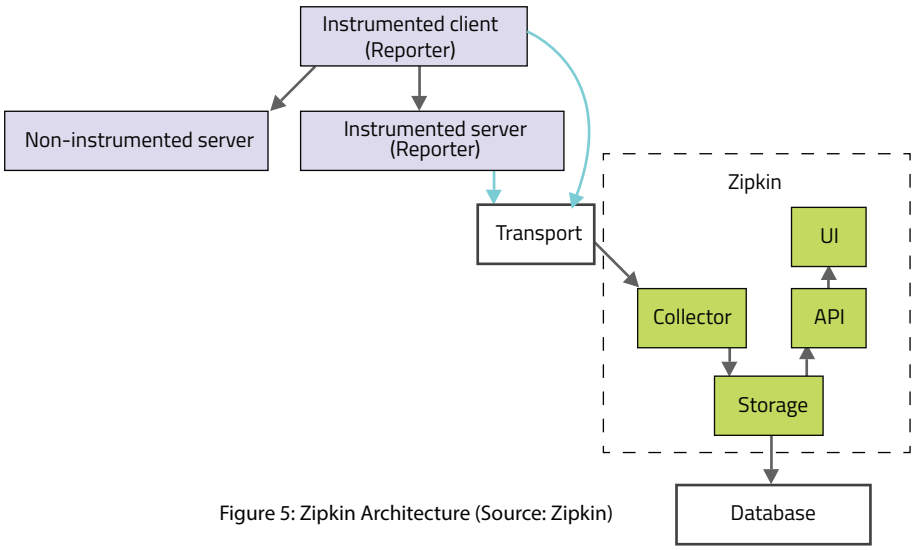


Figure 5: Zipkin Architecture (Source: Zipkin)

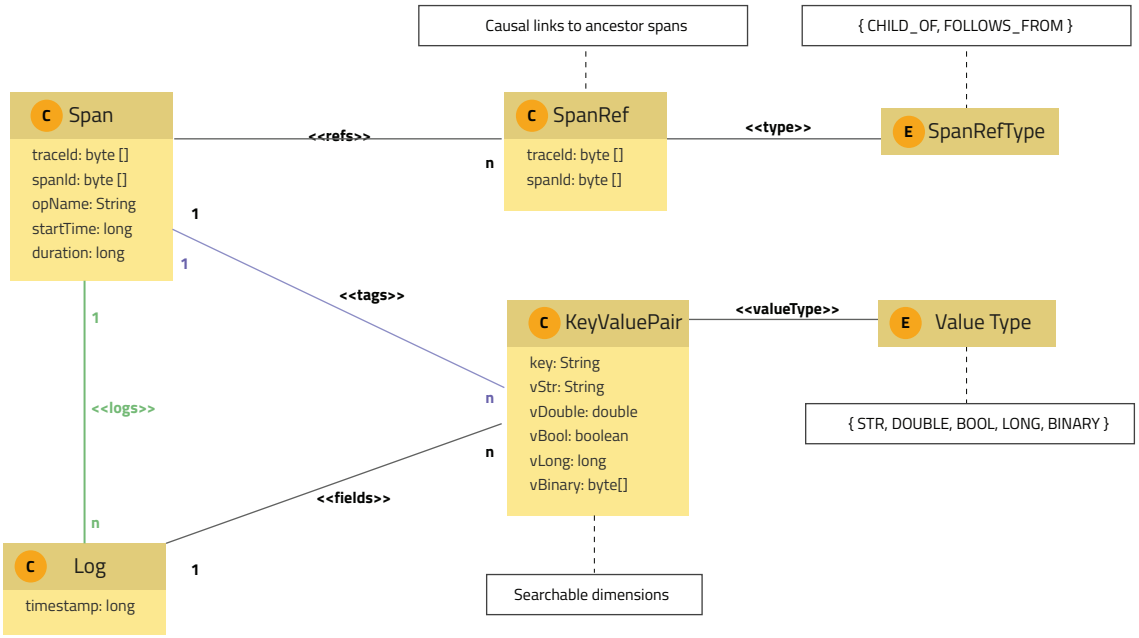


Figure 6: Trace Object Model

Trace: Represents the end-to-end execution path of a request flowing through a distributed system, as a DAG. A trace comprises of spans.

Span: Span is a named and timed operation in the request processing. Spans can, in turn, spawn nested child spans, even concurrent ones. Spans provide causal relationships within a trace. References model direct causal relationships between a child span and its parent, represented by edges in a DAG. A span may have zero (in case of root span) or many causal parents. Nested spans can either have a child-of (typically one only) or, a follows-from kind of a reference with the parent span, depending on whether the caller expects a response (e.g., RPC protocol) or not (e.g., a fire-and-forget messaging protocol). A span also supports k-v tags, and structured logs.

Tags: Spans can be tagged (e.g., userID, cluster info, outbound endpoint invocation details, sql-query references, response/error codes), to support multi-dimensional queries in analysis of traces. Tags provide a mechanism for context-rich tagging of spans.

Logs: Structured events, providing context around "what happened" with the span. These can be used to log important events about the span, e.g., an error or, any event that impacts the course of the request (e.g., a cache hit/miss, empty results from polling a remote system).

Tracing is an intrusive technique and harder to implement compared to logs and metrics, as it involves code instrumentation with tracing libraries like the open source Jaeger[5] and Zipkin[6] However, it beats blackbox techniques of monitoring by supporting rich contextual information and finer granularity within traces, which can be queried and analyzed for deep insights. Tracers are designed to be low-overhead, propagating only IDs in-band to the downstream services. Completed spans are reported to tracing servers out-of-band (async). Sampling (simple random or, adaptive sampling) of traces is another means of keeping the runtime overhead of tracing low in high-traffic systems. A downside of sampling, however, is that it limits visibility into infrequently occurring latency issues (e.g., p99 latency) or, rare failures.

Without tracing, it would be difficult to observe the runtime behavior, understand causality, and isolate points of failures or latencies in a live distributed environment made of complex synchronous and asynchronous interactions between microservices, serverless compute components, and external systems. Beyond just a request-centric view, traces, when analyzed in aggregate, can uncover valuable insights into which service or resource is a bottleneck, and help in making informed optimization decisions about which services impact the end-user experience the most, which services are on the critical path etc.

Analytics

As there are human limits to how much manual debugging and analysis can practically be done on traces, it is useful to build analytics on these rich volumes of data to discover patterns versus outliers, establish correlations and derive insights which would otherwise be buried under individual traces. Common analytics use-cases include service dependency graph, critical path analysis, error analysis, latency analysis, anomaly detection etc.

Features can be computed out of individual traces e.g., total latency of the trace, trace start time, no. of spans in the trace, no. of outbound network calls in the trace, entry point of the root span, the client type (web client, android app, partner applications etc.), origin of the request, server region etc., which can be fed to an ML model e.g., an anomaly detection model, to flag anomalies. These features can be computed in near real-time and can even be used to monitor trends. Tracing can also contribute to KPIs like the end-to-end transaction processing time in a distributed system.

Dependency graph reconstructs the downstream dependencies of each service, thereby representing the totality of the architecture in action. The dependency graph can help visualize how exactly requests flow through the entire service landscape. Error analysis can help with classes of errors and error stats per service. Latency analysis can help pinpoint latency issues between inter-service calls. Critical path analysis can help in optimization efforts, by identifying spans that are bottlenecks in the end-to-end request execution and hence should be addressed first to shorten the trace execution time.

For critical path analysis, a DAG of the trace would be the starting point. A gap[7] in the Open Tracing specification is how to model sibling relationships that denote more of a sequencing constraint (a preceding sibling must finish before the following sibling can start) than a causal relationship. In the absence of a formal support for this kind of reference type within the OpenTracing model, a custom workaround needs to be employed e.g., by introducing a special custom tag (say "next-of") on spans to hold a reference to any preceding sibling spans that must be sequenced right before it. When the DAG is constructed, apart from the regular child-of and follows-from edges, edges should be generated for this otherwise missing reference type as well. The diagrams that follow depict the resultant DAG corresponding to the trace in Figure 3, and its respective linearized version.

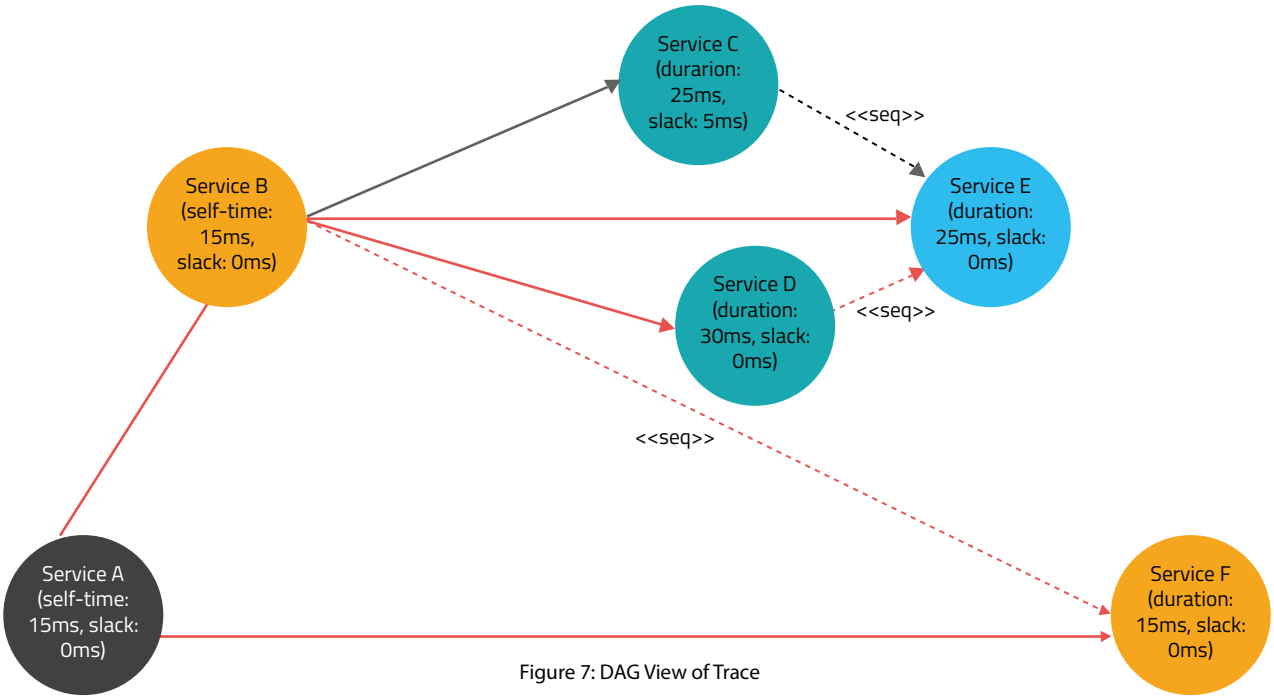


Figure 7: DAG View of Trace

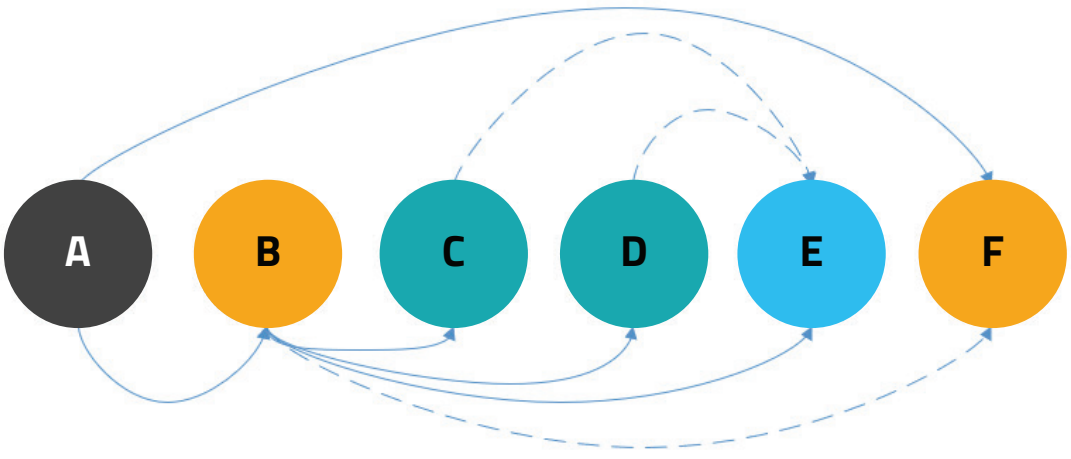


Figure 8: Topologically Ordered View of Trace

The critical path can be obtained by traversing the graph in a forward pass to compute the early start (ES) and early finish (EF) for each node, and a backward pass to compute the late finish (LF) and late start (LS) for each node.

01 ES: earliest a span can possibly start, considering predecessor spans	02 EF (= ES + span duration): earliest a span can possibly finish, considering predecessor spans
03 LF: latest a span can finish without delaying the trace finish	04 LS (= LF – span duration): latest a span can start without delaying the trace finish

All nodes with 0 slack (LF – EF) indicate tasks that cannot be delayed without impacting the overall trace execution time. This sequence forms the critical path, as highlighted in red.

The obvious critical path in a DAG is the longest path in time from the origin node to the terminal node. All spans that fall on this path have a slack of 0. This can be computed as follows –

Topologically sort the DAG, G = (V, E). This would yield the vertex order {A, B, C, D, E, F} with appropriate edges between them.	Recursively find the longest path from the origin node to the terminal node. For vεV, this is obtained by computing the max path cost (in time) from the origin node to v over all predecessor edges of v.	The longest path thus obtained is the critical path.

Trace-level critical paths can be further analyzed by grouping them on root span endpoints (application entry points) to arrive at critical paths globally. This can be computed at the batch processing layer using the MapReduce paradigm.

The tracing infrastructure can include Kafka as an intermediate buffer between the span collector and storage. Apart from absorbing traffic spikes, Kafka can enable a real-time pipeline for streaming window-based aggregations on the trace either based on heuristics or, a predetermined time interval to allow for all spans of a given trace to be “seen” before computing features on it. Historical analysis on traces can be run to incorporate any new feature introduced. A batch data store like Hadoop can enable an analytics pipeline on historical data. The output of the analysis could be written to various kinds of sinks like a TSDB, an event bus (for alerting), or the tracing store (Elasticsearch or Cassandra) itself, depending on the nature of the analysis. Apache Flink and Apache Spark Streaming/Batch are the commonly used open-source frameworks for Big Data Analytics, with the same program reused in real time and batch mode.

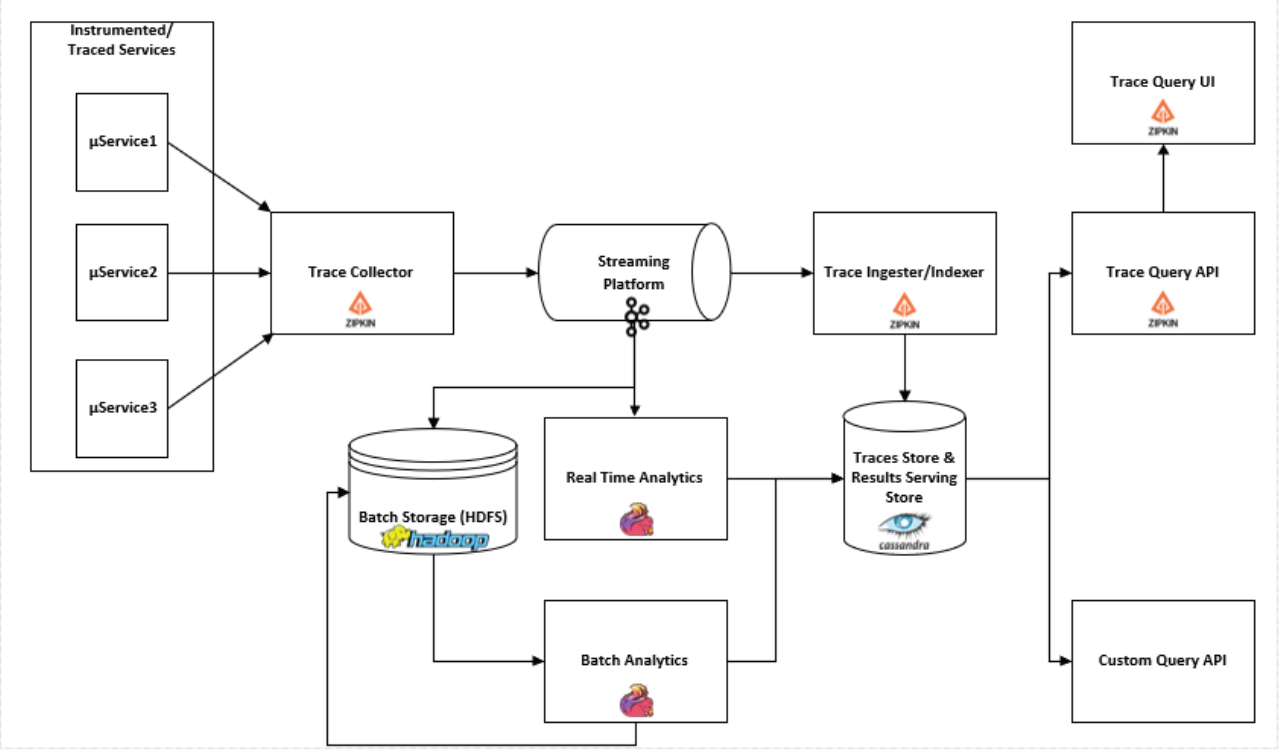


Figure 9: Trace Analytics Pipeline

Log analytics pipelines too commonly use Kafka as the event bus, to enable real-time complex queries and analytics on streaming log data. A lot of log analytics is also done in batch mode.

Future Directions

OpenTracing and OpenCensus are two competing open standards for observability, each with a strong community. Unlike OpenTracing, OpenCensus collects and exports metrics too. OpenTelemetry is an evolving standard that merges these two standards into a single instrumentation standard that aims to standardize how telemetry data gets collected and sent to backend telemetry platforms. The OpenTelemetry backend is pluggable and compatible with leading backends supported by either project today. OpenTelemetry offers backward compatibility and an easy migration path for both the projects while setting forth a unified set of specifications and libraries for cloud-native observability telemetry.

Conclusion

Response time is critical to any business, as latency or reliability issues in end-user experience can significantly impact brand reputation and adoption, thereby leading to a loss in revenue. While traditional monitoring may suffice for a service landscape that is relatively straightforward and contained; with highly distributed and complex architectures, tracing becomes a must. Wherever observability is carefully designed for in software, it offers a deep capability for exploration and root-cause analysis in a live system, thereby improving troubleshooting capabilities and accelerating issue identification and resolution, more often proactively. This built-in observability in turn leads to more robust software over time.

Author Bio

Bidisha Gangopadhyay *is a Principal Architect at Happiest Minds, where she is a member of the Central Architecture Group. She has over 16 years of industry experience spanning Enterprise Architecture, Technology Consulting, Presales & Solutioning, and Individual Contributor roles. Her primary areas of interest include Distributed Systems, Big Data and the JVM ecosystem at large.*

References

[1] https://www.oreilly.com/library/view/distributed-systems-observability/

[2] https://elastic-stack.readthedocs.io/en/latest/e2e_kafkapractices.html

[3] https://prometheus.io/docs/introduction/overview/

[4] https://opentracing.io/

[5] https://zipkin.io/

[6] https://www.jaegertracing.io/

[7] https://github.com/opentracing/specification/issues/142