Monitoring

TEKsystems Global Services

WORKFORCE DEVELOPMENT

Observability in Modern Systems

- Observability is the ability to understand what is happening inside a system by looking at the data it produces. In modern distributed systems, this data must explain behavior without requiring manual debugging or guesswork.
    - Observability is about system introspection
    - Explains behavior without direct access
    - Critical for distributed systems
    - Relies on telemetry from the system

Monitoring vs Observability

- Monitoring tells you whether a system is healthy based on predefined checks and thresholds. Observability tells you what the system is doing internally without needing to predict every failure mode.
  - Monitoring is reactive
  - Observability is exploratory
  - Monitoring answers known questions
  - Observability enables unknown questions

Why Logs and Metrics Are Not Enough

- Logs and metrics describe system behavior, but they lack causality and context. They cannot explain how a single request flows across multiple services.
  - Metrics show trends
  - Logs show events
  - Neither shows request paths
  - Neither explains dependencies

The Distributed Systems Problem

- Modern applications are built from many services that communicate over the network. Failures and latency now emerge from interactions between systems, not single components.
  - Requests span services
  - Failures propagate
  - Latency compounds
  - Root cause becomes unclear

What Is OpenTelemetry

- OpenTelemetry is an open standard for generating and exporting telemetry data from applications. It provides a unified way to collect traces, metrics, and logs across systems.
  - Vendor-neutral standard
  - Open-source CNCF project
  - Cross-language support
  - Exports to any backend

The Three Signals

- OpenTelemetry defines three core observability signals that describe system behavior. Each signal answers a different class of operational questions.
  - Traces show request flows
  - Metrics show numerical behavior
  - Logs show events
  - Signals complement each other

What Is a Trace

- A trace represents the full lifecycle of a single request as it moves through a system. It shows how work is broken down across services and components.
  - One trace per request
  - Spans form a tree
  - Parent-child relationships
  - Latency visible end-to-end

What Is a Span

- A span represents a single operation within a trace. It contains timing information and metadata about the work performed.
  - Start and end time
  - Belongs to one trace
  - Can have children
  - Records errors

Why Tracing Exists

- Tracing exists to explain behavior that cannot be inferred from metrics or logs alone. It provides causality across service boundaries.
  - Explains slow requests
  - Explains failures
  - Explains dependencies
  - Enables root cause analysis

OTEL Is Vendor Neutral

- OpenTelemetry focuses on generating telemetry, not storing or visualizing it. This allows the same instrumentation to work with any observability backend.
  - No vendor lock-in
  - Works with many APM platforms
  - Same code across environments
  - Backends can change freely

Why Standards Matter

- Without standards, every vendor requires custom instrumentation and agents. Standards allow tools to interoperate across organizations and platforms.
  - Avoids proprietary agents
  - Enables tool portability
  - Reduces engineering overhead
  - Improves ecosystem growth

OTEL in Cloud Native Systems

- Cloud-native systems are dynamic, ephemeral, and distributed by design. Observability must adapt automatically as services scale and move.
  - Services scale horizontally
  - Instances are short-lived
  - Infrastructure is abstracted
  - Tracing must be automatic

Telemetry as a Control Plane

- Telemetry acts as a feedback system between applications and operators. It enables real-time decision making based on system behavior.
  - Supports incident response
  - Enables capacity planning
  - Drives automation
  - Feeds reliability engineering

OTEL vs APM Agents

- Traditional Application Performance Monitoring (APM) agents are tightly coupled to specific vendors. OTEL decouples instrumentation from analysis.
  - APM agents are proprietary
  - OTEL is open standard
  - OTEL supports multiple backends
  - Instrumentation is portable

What OTEL Does Not Do

- OpenTelemetry does not store, visualize, or alert on data. It only produces and exports telemetry.
  - No built-in dashboards
  - No built-in alerting
  - No data storage
  - Focuses only on collection

Why Tracing Became Critical

- As systems moved from monoliths to microservices, request paths became invisible. Tracing restores visibility across service boundaries.
  - Monoliths had single stack traces
  - Microservices break call stacks
  - Network hides execution flow
  - Tracing reconstructs execution paths

Tracing vs Logging

- Tracing shows structured request paths, while logging shows unstructured events. They answer different operational questions.
  - Tracing is request-centric
  - Logging is event-centric
  - Tracing shows causality
  - Logging shows symptoms

Tracing vs Metrics

- Metrics aggregate system behavior, while tracing shows individual experiences. Both are required for full observability.
  - Metrics show trends
  - Tracing shows individual journeys
  - Metrics detect anomalies
  - Tracing explains anomalies

OTEL Core Components

- OpenTelemetry is built from several core components that work together to generate and move telemetry. Each component plays a specific role in the observability pipeline.
  - Instrumentation libraries
  - SDKs
  - Collectors
  - Exporters
  - Backends

Instrumentation Libraries

- Instrumentation libraries hook into application frameworks and libraries to generate telemetry. They create spans, metrics, and logs automatically.
  - Framework specific
  - Language specific
  - Often auto-injected
  - Minimal code changes

OTEL SDK

- The SDK is responsible for creating, processing, and exporting telemetry data. It lives inside the application process.
  - Creates tracers and meters
  - Manages span lifecycle
  - Handles sampling
  - Sends data to exporters

OTEL Collector

- The OTEL Collector is a standalone service that receives, processes, and forwards telemetry. It acts as a telemetry gateway.
  - Runs out of process
  - Receives OTLP
  - Can transform data
  - Forwards to backends

Why Use a Collector

- Collectors centralize telemetry processing and reduce coupling between applications and backends. They enable routing and enrichment of data.
  - Decouples apps from vendors
  - Enables sampling control
  - Supports multiple backends
  - Simplifies network topology

OTEL Exporters

- Exporters define how telemetry leaves the system. They translate internal data into backend-specific protocols.
  - OTLP exporter
  - Jaeger exporter
  - Zipkin exporter
  - Vendor specific exporters

OTLP Protocol

- OTLP is the standard protocol used by OpenTelemetry to transmit telemetry. It is designed to be efficient and vendor neutral.
  - HTTP or gRPC
  - Binary encoding
  - Standard across languages
  - Default OTEL protocol

Sampling

- Sampling controls how many traces are collected and stored. It balances visibility with performance and cost.
  - Always on sampling
  - Probabilistic sampling
  - Tail-based sampling
  - Parent-based sampling

Head vs Tail Sampling

- Head sampling happens when a trace starts, while tail sampling happens after the trace completes. Tail sampling allows smarter decisions.
  - Head sampling is fast
  - Tail sampling sees full trace
  - Tail sampling is more expensive
  - Tail sampling enables filtering

Resource Attributes

- Resource attributes describe the entity producing telemetry. They provide identity and context across systems.
  - Service name
  - Environment
  - Region
  - Version

Context Propagation

- Context propagation allows trace data to flow across service boundaries. It ensures that distributed systems share the same trace.
  - Trace context travels with requests
  - Implemented via headers
  - Works across protocols
  - Required for distributed tracing

Trace Context Headers

- Trace context is usually transmitted through standardized HTTP headers. These headers allow downstream services to join the same trace.
  - traceparent header
  - tracestate header
  - W3C standard
  - Supported across vendors

What Is Baggage

- Baggage is key-value metadata that travels with a trace. It provides business and operational context.
  - Propagates across services
  - Used for custom metadata
  - Not used for metrics
  - Should be kept small

Correlation

- Correlation links traces, metrics, and logs together. It enables operators to move between signals seamlessly.
  - Trace ID in logs
  - Span ID in logs
  - Metrics tagged with trace data
  - Enables cross-signal navigation

Service Graphs

- Service graphs visualize communication between services. They are built automatically from trace data.
  - Shows dependencies
  - Shows request volume
  - Shows error rates
  - Shows latency edges

Trace to Logs

- Trace to logs allows operators to jump from a trace directly into relevant logs. This reduces time to root cause.
  - Uses trace ID
  - Works with Loki
  - Filters logs automatically
  - Improves debugging speed

Trace to Metrics

- Trace to metrics allows operators to correlate a single trace with aggregate system behavior. It connects micro and macro views.
  - Uses span attributes
  - Generates RED metrics
  - Supports SLOs
  - Supports capacity analysis

RED Method

- The RED method is a common way to monitor services using request data. It focuses on user experience.
  - Rate
  - Errors
  - Duration
  - Derived from traces

USE Method

- The USE method focuses on resource utilization. It complements RED by describing infrastructure health.
  - Utilization
  - Saturation (CPU queue, swap space)
  - Errors
  - Infrastructure focused

OTEL in Incident Response

- OTEL provides the fastest path from symptom to root cause during incidents. It replaces guesswork with evidence.
  - Find failing service
  - Follow request path
  - Inspect errors
  - Confirm fix with live traces

OTEL and the LGTM Stack

- OpenTelemetry acts as the data plane for the LGTM stack. It feeds traces into Tempo, logs into Loki, and metrics into Prometheus or Mimir.
  - OTEL generates telemetry
  - Tempo stores traces
  - Loki stores logs
  - Prometheus or Mimir stores metrics

Tempo in the LGTM Stack

- Tempo is a distributed tracing backend designed for scale. It stores traces and integrates directly with Grafana.
  - Receives OTLP
  - Stores trace data
  - No indexing by default
  - Optimized for cost

Why Tempo Does Not Index

- Tempo avoids heavy indexing to reduce cost and complexity. It relies on Grafana for querying and navigation.
  - Lower storage cost
  - Faster ingestion
  - Relies on metadata
  - Optimized for large volumes

Grafana as the Control Plane

- Grafana acts as the single interface for all observability data. It connects traces, logs, and metrics.
  - Unified dashboards
  - Explore view
  - Trace navigation
  - Signal correlation

Trace Exploration in Grafana

- Grafana allows interactive exploration of traces. Engineers can drill into spans and follow dependencies.
  - Search by service
  - Search by duration
  - View span trees
  - Inspect errors

Trace to Logs in LGTM

- LGTM enables jumping from a trace directly into Loki logs. This connects request paths with raw events.
  - Trace ID used as filter
  - Automatic log queries
  - Reduces investigation time
  - Improves debugging flow

Trace to Metrics in LGTM

- LGTM connects trace attributes with metrics. This enables SLOs and service level monitoring.
  - RED metrics from spans
  - Latency percentiles
  - Error rates
  - Throughput

Service Graphs in LGTM

- Service graphs visualize dependencies using trace data. They help understand system topology.
  - Auto-generated
  - Shows call relationships
  - Shows traffic volume
  - Highlights bottlenecks

OTEL Data Flow

- OTEL defines how telemetry flows from applications to backends. This pipeline must be reliable and observable.
  - App generates telemetry
  - SDK processes data
  - Collector receives data
  - Backends store data

Why Platform Teams Care

- Platform teams use OTEL to provide observability as a service. It becomes shared infrastructure.
  - Standardized telemetry
  - Reduced onboarding time
  - Consistent tooling
  - Improved reliability

OTEL in FastAPI

- FastAPI can be instrumented with OpenTelemetry to automatically generate traces for every request. This requires no changes to application logic.
  - Auto-instrumentation supported
  - Works with ASGI middleware
  - Creates spans per endpoint
  - Captures request metadata

Auto-Instrumentation

- Auto-instrumentation injects telemetry into libraries at runtime. It eliminates the need to manually create spans for basic operations.
  - Uses monkey patching
  - Configured at startup
  - Minimal code changes
  - Covers common frameworks

opentelemetry-instrument

- The opentelemetry-instrument command wraps the application process. It bootstraps the OTEL SDK before the app starts.
  - CLI wrapper
  - Initializes providers
  - Loads instrumentation packages
  - Requires correct dependencies

OTLP Endpoints

- OTLP endpoints define where telemetry is sent. They must be reachable from the application environment.
  - HTTP or gRPC
  - Usually port 4317 or 4318
  - Collector or backend
  - Critical for connectivity

Custom Spans

- Custom spans allow developers to trace business logic. They provide visibility into internal operations.
  - Manually defined
  - Used for critical code paths
  - Add custom attributes
  - Expose performance bottlenecks

Error Tracing

- Unhandled exceptions automatically mark spans as errors. This allows traces to represent failures.
  - Exception recorded in span
  - Status set to error
  - Searchable in Tempo
  - Supports incident analysis

Outbound Calls

- Outbound HTTP calls can also be traced if instrumentation exists. This enables full request graphs.
  - HTTP client instrumentation
  - Database instrumentation
  - Message queue instrumentation
  - Shows dependency latency

Sampling in Production

- Sampling strategies must balance cost and visibility. Production systems rarely sample 100 percent.
  - Lower cost
  - Reduced storage
  - Focus on errors
  - Tail-based filtering

Performance Overhead

- OTEL introduces minimal overhead when configured correctly. Overhead increases with sampling and export volume.
  - CPU overhead
  - Network overhead
  - Storage overhead
  - Generally acceptable in prod

Common Production Patterns

- OTEL is typically deployed with a collector and centralized configuration. This simplifies management and scaling.
  - Sidecar collectors
  - DaemonSet collectors
  - Central sampling rules
  - Multi-backend routing

MCQ 1

- What is the main purpose of OpenTelemetry?
- A. Store logs and metrics
- B. Provide dashboards and alerts
- C. Generate and export telemetry
- D. Replace Prometheus

MCQ 1 – Answer

- What is the main purpose of OpenTelemetry?
- A. Store logs and metrics
- B. Provide dashboards and alerts
- **C. Generate and export telemetry**
- D. Replace Prometheus

MCQ 2

- Which component is responsible for receiving and forwarding telemetry out of process?
- A. SDK
- B. Instrumentation library
- C. Collector
- D. Exporter

MCQ 2 – Answer

- Which component is responsible for receiving and forwarding telemetry out of process?
- A. SDK
- B. Instrumentation library
- **C. Collector**
- D. Exporter

MCQ 3

- What does context propagation enable?
- A. Log aggregation
- B. Metric sampling
- C. Distributed tracing across services
- D. Alert routing

MCQ 3 – Answer

- What does context propagation enable?
- A. Log aggregation
- B. Metric sampling
- **C. Distributed tracing across services**
- D. Alert routing

MCQ 4

- Why is tail-based sampling powerful?
- A. It samples faster
- B. It reduces CPU usage
- C. It sees the full trace before deciding
- D. It eliminates exporters

MCQ 4 – Answer

- Why is tail-based sampling powerful?
- A. It samples faster
- B. It reduces CPU usage
- **C. It sees the full trace before deciding**
- D. It eliminates exporters

MCQ 5

- Which LGTM component stores distributed traces?
- A. Loki
- B. Prometheus
- C. Grafana
- D. Tempo

MCQ 5 – Answer

- Which LGTM component stores distributed traces?
- A. Loki
- B. Prometheus
- C. Grafana
- **D. Tempo**