

Observability and Monitoring Interview Answers

What is the difference between Monitoring and Observability?

Monitoring is about watching known failure modes and metrics that you've predetermined are important—you set up dashboards and alerts for specific things you expect might go wrong. It answers questions you've thought to ask ahead of time.

Observability, on the other hand, is about being able to understand and debug your system by asking arbitrary questions you didn't anticipate. It's the property of a system that allows you to understand its internal state based on its external outputs.

- With monitoring, you might track error rates and CPU usage.
- With observability, when something weird happens that you've never seen before, you can slice and dice the data—filtering by user ID, request path, or any other dimension—to understand what's happening.

Monitoring tells you something is wrong; observability helps you figure out why. In modern distributed systems where failure modes are unpredictable, observability is essential because you can't monitor for every possible issue. Good observability means your system emits rich, high-cardinality data that lets you explore and investigate rather than just watching predetermined metrics.

What are the three pillars of observability? How do they complement each other?

The three pillars are **logs, metrics, and traces**.

- **Logs** are discrete event records capturing what happened at a specific point in time—errors, state changes, or significant events. They provide detailed context but can be expensive to store and search at scale.
- **Metrics** are numerical measurements aggregated over time—request counts, latency percentiles, error rates. They're cheap to store and great for spotting trends and triggering alerts, but lack the granular detail of individual events.
- **Traces** show the path of a request through a distributed system, connecting related operations across multiple services with timing information. They're excellent for understanding request flows and identifying bottlenecks but represent only a sample of requests.

These pillars complement each other perfectly: metrics alert you to a problem, traces help you narrow down which service or component is involved, and logs provide the detailed context about

what went wrong.

For example, metrics show latency increased, traces identify that the database service is slow, and logs reveal the specific query that's timing out. Modern observability platforms connect these pillars—you can jump from a metric spike to relevant traces to associated logs, following the investigation naturally.

What is high-cardinality data and what challenge does it pose for observability systems?

High-cardinality data refers to dimensions or labels with many unique values—things like user IDs, request IDs, email addresses, or IP addresses.

Traditional metrics systems struggle with high cardinality because they store time series for every unique combination of label values. If you have 10 labels each with 100 possible values, you potentially have 100^{10} time series, which becomes computationally and economically infeasible.

This is a challenge because high-cardinality data is often exactly what you need for effective debugging—knowing which specific user experienced an issue or which particular endpoint is slow. Many older monitoring systems limit cardinality or charge heavily for it.

Modern observability platforms address this through different data models—using columnar storage, sampling intelligently, or treating events as discrete rather than continuous time series. The key insight is that you need high-cardinality data for investigation but not necessarily for real-time metrics.

Solutions include storing high-cardinality data in logs or traces rather than metrics, using exemplars in Prometheus to link metrics to specific trace examples, or using purpose-built observability databases that handle high cardinality efficiently.

What is the RED method for monitoring microservices?

RED stands for Rate, Errors, and Duration—the three key metrics for monitoring request-driven services.

- **Rate** is the number of requests per second your service is handling, which tells you about traffic patterns and load.
- **Errors** is the number or percentage of failed requests, showing service reliability.
- **Duration** measures how long requests take, typically expressed as latency percentiles like p50, p95, and p99.

These three metrics give you a complete picture of a service's health from the user's perspective. If

rate drops unexpectedly, maybe upstream services aren't sending traffic. If errors spike, something's broken. If duration increases, performance is degrading.

The beauty of RED is its simplicity and universality—it applies to any request-response service regardless of technology. It's often contrasted with USE method (Utilization, Saturation, Errors) which focuses on resources rather than requests.

For microservices where requests flow through multiple services, tracking RED metrics at each service boundary lets you quickly isolate where problems are occurring. I typically implement RED metrics as my baseline monitoring for every service, then add service-specific metrics on top.

Why is structured logging essential for a modern backend system?

Structured logging formats log entries as structured data—typically JSON—rather than unstructured text strings. This is essential because it makes logs machine-readable and queryable.

Instead of logging "User john@example.com logged in from IP 192.168.1.1", structured logging captures:

```
{  
  "event": "user_login",  
  "user_email": "john@example.com",  
  "source_ip": "192.168.1.1",  
  "timestamp": "2026-01-18T10:30:00Z"  
}
```

This structure enables powerful querying—I can easily find all logins from a specific IP, all actions by a specific user, or correlate events across services. Log aggregation tools like Elasticsearch, Splunk, or Loki can index structured fields efficiently, making searches fast even across terabytes of logs.

Structured logs integrate naturally with observability platforms where you want to jump from a trace to relevant logs filtered by trace ID. They're also easier to process in pipelines—parsing structured JSON is reliable whereas parsing custom text formats is brittle and requires regular expression maintenance.

Structured logging enables high-cardinality analysis since each field can be independently indexed and filtered. In distributed systems, consistency in log structure across services makes correlation and investigation dramatically easier.

What is a correlation ID and how is it used?

A **correlation ID**, also called a request ID or trace ID, is a unique identifier attached to a request that follows it through the entire system as it flows across multiple services.

When a request enters the system, we generate a unique ID and include it in all logs, metrics, and traces related to that request. As the request calls downstream services, we propagate the correlation ID through headers or message metadata. This lets us correlate all activity related to a single user request across all the services involved.

If a user reports an error, they can provide their correlation ID, and we can instantly find all logs across every service that touched that request. In logging, I include the correlation ID as a structured field in every log entry. In distributed tracing, the correlation ID becomes the trace ID that connects all spans.

This is invaluable for debugging distributed systems where a single user action might touch 10+ services—without correlation IDs, finding related logs is nearly impossible. I typically generate correlation IDs at the edge (API gateway or load balancer) and ensure all services extract and propagate them. For async operations like queue processing, the correlation ID moves with the message.

What information should you avoid putting in logs?

Several types of information should never appear in logs due to security, privacy, or compliance concerns.

Most critical is authentication credentials—passwords, API keys, access tokens, or session tokens. These give attackers direct access if logs are compromised.

Personally identifiable information (PII) like Social Security numbers, credit card numbers, health information, or even email addresses and phone numbers in jurisdictions with strict privacy laws should be excluded or redacted. Financial information including account numbers or transaction details requires special handling.

Encryption keys or cryptographic secrets of any kind should never be logged. Even internal system secrets like database connection strings or service-to-service authentication tokens shouldn't appear.

Beyond security, I avoid logging full request and response bodies which might contain any of the above plus create storage and performance issues. Instead, I log metadata about requests—size, duration, status code—and use correlation IDs to retrieve full details from short-term storage if needed.

When sensitive data must be logged for debugging, I use redaction or hashing, and ensure those logs have stricter access controls and shorter retention periods.

What are log-based metrics?

Log-based metrics are aggregated measurements derived from log data rather than being directly instrumented in code. Instead of having your application maintain counters and gauges, you emit structured logs and let your log aggregation system compute metrics from those logs.

For example, rather than instrumenting a counter for failed login attempts, you log every login attempt with success/failure status, then create a metric by counting logs where status equals "failed".

The advantage is flexibility—you can define new metrics from historical logs without deploying code changes. If you realize you need to track API calls from a specific user agent, you can create that metric retroactively from existing logs. This reduces instrumentation burden since you log events once and can derive multiple metrics.

However, log-based metrics have tradeoffs: they're more expensive than native metrics since you're storing detailed logs, there's typically higher latency in metric availability, and you're limited to metrics derivable from logged data.

I use log-based metrics for lower-volume events where flexibility matters—security events, business metrics, or things I'm exploring. For high-volume metrics like request rates, I prefer native instrumentation with efficient metric libraries for performance and cost reasons.

Compare the push vs. pull model for metrics collection.

In the **push model**, applications actively send their metrics to a central collector—the app periodically pushes data to the metrics backend.

In the **pull model**, a central collector scrapes metrics from applications by making HTTP requests to metrics endpoints that the apps expose.

Each has distinct advantages:

- **Push** is better for:
 - Short-lived jobs that might not exist long enough to be scraped
 - Applications behind NAT or firewalls where inbound connections aren't possible
 - Scenarios where you want metrics sent immediately on critical events
 - Simpler for developers—just call an API and you're done
- **Pull**, popularized by Prometheus, provides:
 - Better control to the monitoring system—the collector knows if targets are down because scrapes fail
 - Central control of the collection rate
 - Built-in service discovery integration
 - Prevention of metric flooding if an app goes haywire since it can only send as fast as it's scraped
 - The monitoring system maintains the authoritative list of what should be monitored

In practice, I often use hybrid approaches—Prometheus with pushgateway for batch jobs, or OpenTelemetry Collector that can receive pushed metrics and expose them for pulling.

The choice depends on architecture—pull works great for Kubernetes environments with service discovery, while push might be better for serverless or edge computing scenarios.

What are the four main metric types used by systems like Prometheus?

The four types are **Counter**, **Gauge**, **Histogram**, and **Summary**.

- **Counter:** A cumulative metric that only increases—like total requests served or total errors. You query counters with rate functions to see requests per second. Counters reset to zero when processes restart, which is expected and handled by the query language.
- **Gauge:** A metric that can go up or down—like current memory usage, active connections, or queue depth. Gauges represent a point-in-time snapshot.
- **Histogram:** Samples observations (like request durations) and counts them in configurable buckets, plus provides a sum of all values and total count. Histograms let you calculate percentiles and are aggregatable across instances.
- **Summary:** Similar but calculates quantiles on the client side—useful when you want exact percentiles for a single instance but they can't be aggregated across multiple instances.

In practice, I use:

- Counters for things that accumulate (requests, errors, bytes sent)
- Gauges for current state (memory usage, goroutines, queue size)
- Histograms for distributions (latency, request size)
- I rarely use Summaries because their non-aggregatable nature limits flexibility

The metric type determines what queries and operations are valid—you can't `rate()` a Gauge or average a Counter, so choosing correctly is important.

Why are histograms often preferred over summaries for measuring latency?

Histograms are preferred because they're aggregatable across multiple instances while summaries are not.

When you have multiple replicas of a service, you want to calculate overall p95 latency across all instances—histograms make this possible by storing bucket counts that can be summed, then calculating percentiles from the combined buckets. Summaries calculate quantiles on each instance separately and you can't meaningfully combine them.

Histograms also let you calculate different percentiles at query time—you instrument once with reasonable buckets and can later ask for p90, p95, p99 without code changes. Summaries bake the percentiles in at collection time. Histograms support Prometheus's `histogram_quantile` function which approximates percentiles from buckets.

The tradeoff is that histograms require choosing bucket boundaries upfront—poor choices can make percentile calculations inaccurate. Summaries give exact percentiles for a single instance and don't require bucket configuration.

In practice, for service latency I almost always use histograms with standard bucket configurations like exponential buckets from 0.001 to 10 seconds. The aggregability is crucial in distributed systems where you need system-wide latency views. I only use summaries for very specific cases where per-instance exact percentiles matter and aggregation isn't needed.

What are the core components of a distributed trace? (Trace, Span)

A **trace** represents the complete journey of a request through a distributed system. It's composed of **spans**, which are individual units of work.

Each span represents one operation—a database query, an HTTP call to another service, or a function execution. Spans have several key properties:

- A unique span ID
- A trace ID that connects all spans in the trace
- A parent span ID that establishes the hierarchy
- Start and end timestamps showing duration
- The operation name
- Tags/attributes providing additional context

The relationships between spans form a tree structure showing which operations triggered which others. For example, a trace for "user checkout" might have:

- A root span for the API request
- Child spans for calls to inventory service and payment service
- Grandchild spans for their database queries

Each span captures timing, so you can see where latency occurs. Spans also record errors and include baggage—key-value pairs propagated through the trace.

Modern tracing uses the W3C Trace Context standard for propagating trace and span IDs across service boundaries via HTTP headers. The trace gives you the full picture of a distributed transaction, while individual spans let you drill into specific operations to identify bottlenecks or failures.

What is OpenTelemetry and what problem does it solve?

OpenTelemetry (OTel) is a unified, vendor-neutral standard for collecting telemetry data—traces, metrics, and logs.

Before OTel, the observability landscape was fragmented—Jaeger for tracing, Prometheus for metrics, various logging standards, each with different instrumentation libraries and formats. If you wanted to switch from Jaeger to Zipkin or Datadog, you'd need to re-instrument your code.

OpenTelemetry solves this by providing a single set of APIs and SDKs for all telemetry types across all major languages. You instrument your code once with OTel, and can send data to any compatible backend by changing configuration, not code.

OTel includes automatic instrumentation for common frameworks and libraries, reducing manual instrumentation burden. The OpenTelemetry Collector is a vendor-agnostic proxy that receives, processes, and exports telemetry data, enabling sophisticated pipelines. OTel also standardizes semantic conventions—common attribute names and formats—improving interoperability.

This is huge for the industry because it prevents vendor lock-in, makes it easier to adopt observability, and allows best-of-breed backend choices. In practice, I use OTel SDKs for all new services, configure the collector to handle data processing and routing, and gain flexibility to change observability backends without touching application code.

What is trace context propagation?

Trace context propagation is the mechanism for passing trace and span IDs across service boundaries so that spans from different services can be correctly connected into a single distributed trace.

When service A calls service B, it needs to communicate the trace ID and A's span ID so that B knows it's part of the same trace and can create child spans with proper parent relationships. This is typically done through HTTP headers—the W3C Trace Context standard defines `traceparent` and `tracestate` headers that carry this information.

In message-based systems, the context is included in message metadata or headers. The propagation requires both injection (adding context to outgoing requests) and extraction (reading context from incoming requests). OpenTelemetry SDKs handle this automatically for common protocols.

Without proper propagation, you'd have disconnected spans instead of a complete trace, making it impossible to see the full request path. Propagation must work across heterogeneous systems—services written in different languages, using different frameworks, communicating via different protocols. This is why standards like W3C Trace Context are critical.

In practice, ensuring propagation works correctly requires verifying that all HTTP clients, message producers, and RPC frameworks properly inject context, and all servers and consumers extract it.

Compare head-based vs. tail-based sampling for traces.

Sampling is necessary because tracing every single request in high-volume systems is prohibitively expensive.

Head-based sampling makes the keep/drop decision at the start of a trace—when a request enters the system, you decide whether to trace it, typically based on a sampling rate like 1% or whether it's from a specific customer. This is simple and efficient but has a critical flaw: you don't know yet if the trace will be interesting. You might drop a trace that experienced errors or high latency.

Tail-based sampling defers the decision until after the trace completes. The entire trace is held temporarily, then you examine it—keeping all traces with errors, high latency, or other interesting characteristics, while sampling successful fast traces at a lower rate. This ensures you capture the traces you actually want to investigate.

The tradeoff is complexity and cost—you need infrastructure to buffer traces, make sampling decisions, and distribute those decisions across all services involved in the trace. Tail-based sampling typically happens in collectors rather than in applications.

In practice, I start with head-based sampling for simplicity, then move to tail-based sampling as volume grows and the value of retaining interesting traces justifies the complexity. Some platforms combine both—head-based for extreme scale, tail-based for refinement.

What is the difference between symptom-based alerting and cause-based alerting? Which is preferred?

Symptom-based alerts fire when users are impacted—high error rates, increased latency, or service unavailability. These alert on the symptoms users experience.

Cause-based alerts fire on suspected root causes—high CPU usage, disk full, database connection pool exhausted. The key difference is whether the alert reflects actual user impact.

Symptom-based alerting is strongly preferred because it's what actually matters. If CPU is at 95% but users aren't affected, waking someone up doesn't help anyone. Conversely, users might be impacted for reasons you didn't predict, which symptom-based alerts catch but cause-based alerts miss.

Symptom-based alerting aligns with SLO-based monitoring—alert when you're burning through your error budget, indicating degraded user experience.

That said, cause-based metrics are still valuable for investigation and dashboards—when a symptom alert fires, you consult cause-based metrics to understand why.

I implement symptom-based alerts on user-facing metrics like error rates above SLO thresholds or

latency p99 degradation. Cause-based metrics like resource utilization go on dashboards for context and might trigger lower-severity notifications for proactive remediation before users are impacted. The goal is actionable alerts that indicate real user problems rather than noisy alerts on internal system states.

Explain SLOs, SLIs, and SLAs.

These are related but distinct concepts from SRE practice.

- **SLI (Service Level Indicator):** A specific metric that measures service quality from the user's perspective—like request success rate, latency percentiles, or availability. It's the measurement itself.
- **SLO (Service Level Objective):** A target for an SLI—like "99.9% of requests succeed" or "95% of requests complete in under 200ms". It's an internal goal that defines acceptable service quality.
- **SLA (Service Level Agreement):** A business contract with consequences—if we fail to meet our SLA, we might owe refunds or credits. SLAs are typically less strict than SLOs to provide a buffer.

For example, your SLO might be 99.9% but your SLA is 99.5%. This gives you room for incidents without contractual breaches.

In practice, I define SLIs based on user-facing metrics that matter—using the four golden signals as a starting point. I set SLOs based on business needs and technical feasibility, measuring them over rolling windows like 30 days. I track SLO compliance and use error budgets to balance reliability and feature velocity. SLAs are business decisions that reference SLOs but include commercial terms.

Not every service needs an SLA, but every user-facing service should have SLOs.

What is an error budget?

An **error budget** is the amount of unreliability you can tolerate while still meeting your SLO. If your SLO is 99.9% availability, your error budget is 0.1%—meaning the service can be down 43 minutes per month.

This flips the conversation from "maximize uptime" to "how do we spend our error budget wisely?" Error budgets enable rational decision-making about reliability versus feature velocity.

- If you have error budget remaining, you can take risks—deploy more frequently, do risky database migrations, or experiment with new infrastructure.
- If you've exhausted your error budget, you slow down and focus on reliability—freezing risky deployments, addressing technical debt, and improving systems.

This prevents arguments between development and operations—both teams share ownership of the budget. Error budgets are typically tracked over rolling windows like 30 days, and you measure burn rate—how quickly you're consuming the budget. Fast burn rates trigger alerts even if you haven't exhausted the budget yet.

In practice, I calculate error budgets from SLO targets, track them on dashboards visible to the whole team, and use them as input to release decisions. When budget is low, I conduct blameless postmortems to understand what consumed it and how to improve. Error budgets make reliability a data-driven conversation.

What is alert fatigue and how do you combat it?

Alert fatigue occurs when teams receive so many alerts—especially false positives or low-priority notifications—that they start ignoring them or responding slowly even to critical issues. It's a serious problem that degrades system reliability.

Combat strategies include:

- **Ruthless selectivity:** Only alert on symptoms that require immediate human intervention, not on things that can wait or self-heal.
- **Proper alert routing:** Critical issues page on-call engineers, warnings go to Slack, informational items only appear on dashboards.
- **Dynamic thresholds:** Use anomaly detection rather than static thresholds to reduce false positives.
- **Alert consolidation:** If 5 instances are down, send one alert about the auto-scaling group, not 5 instance alerts.
- **Alert dependencies:** Suppress downstream alerts when root causes are known.
- **Actionable alerts:** Every alert should have a runbook explaining what it means and what to do.
- **Regular review:** Tune or disable alerts that fire frequently without requiring action.
- **Aggregation during incidents:** Prevent overwhelming teams during major incidents.
- **Track metrics:** Monitor time-to-acknowledge and false positive rate to measure effectiveness.

Treat alert tuning as ongoing work, not a one-time setup. The goal is high signal-to-noise ratio where every alert demands and deserves attention.

What is eBPF and what is its role in modern observability?

eBPF (extended Berkeley Packet Filter) is a technology that lets you run sandboxed programs in the Linux kernel without changing kernel code or loading kernel modules.

For observability, this is revolutionary because it enables deep inspection of system behavior with minimal overhead. eBPF programs can attach to kernel events—network packets, system calls, function entries/exits—and collect detailed telemetry. This enables observability use cases that were previously impossible or impractical.

You can:

- Trace every network connection
- Capture distributed traces without application instrumentation
- Profile CPU usage with minimal overhead
- Track file system operations
- Monitor security events

Tools like Pixie, Cilium, and Parca use eBPF for network observability, service mesh data planes, and continuous profiling.

eBPF-based observability doesn't require changing application code or even restarting services--you can observe running systems transparently. It's particularly powerful in Kubernetes where eBPF can provide network visibility between pods, trace requests across the cluster, and collect metrics at the kernel level. The safety guarantees of eBPF mean these programs can't crash the kernel.

In practice, I use eBPF-based tools for network debugging, security monitoring, and performance profiling, especially in environments where instrumentation is difficult or where I need deep system-level visibility beyond what application-level telemetry provides.