[Sumo Logic](https://help.sumologic.com/docs/get-started/) is an all-in-one cloud data analytics platform focused on Security, Operations, and Business Intelligence use cases.

[Sumo Logic](https://youtu.be/wlwo-RLKRIQ) provide:

* Monitoring, analyzing, troubleshooting, and visualizing data from your application and network environment.
* Elastic processing to collect, manage, and analyze the log data, regardless of type, volume, or location.
* Real-time insights into online operations and customer behavior.

The collection of logs in Sumo Logic is managed by the collectors and they help in designing the [best Sumo Logic](https://youtu.be/D4WO5DlqD6o) deployment. A [Collector](https://help.sumologic.com/docs/get-started/overview/#sumo-logic-components) is a small application that gathers log data from your servers and sends it to the Sumo Logic Cloud.

Sumo ingests individual metric data points from your metric sources. [Quantization](https://help.sumologic.com/docs/metrics/introduction/metric-quantization/) is the process of aggregating metric data points for time series over an interval.

Perhaps the most convenient way to start [capturing telemetry from Java](https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/java/) (or, generally speaking, JVM) is to incorporate [OpenTelemetry Instrumentation for Java](https://github.com/open-telemetry/opentelemetry-java-instrumentation" \t "_blank). It automatically detects when one of the [popular libraries](https://github.com/open-telemetry/opentelemetry-java-instrumentation#supported-libraries-frameworks-and-application-servers) is being used in the service and injects the instrumentation without writing any code. It’s also possible to mix automatic instrumentation with manual instrumentation if needed. This method works for all Java 8+ JVMs.

It is very simple to [configure **traceId** and **spanId** data injection](https://help.sumologic.com/docs/apm/traces/get-started-transaction-tracing/opentelemetry-instrumentation/java/traceid-spanid-injection-into-logs-configuration/) into user logs in Java applications. In general, it is enough to add instrumented versions of the logging packages into project dependencies. **Log4j2** and **logback** loggers are supported since OpenTelemetry-Java-Instrumentation version [0.10.1](https://github.com/open-telemetry/opentelemetry-java-instrumentation/tree/v0.10.1).

A [Logback appender](https://github.com/SumoLogic/sumologic-logback-appender#sumologic-logback-appender) that sends straight to Sumo Logic.

[**Sending parsed log data to Datadog**](https://www.datadoghq.com/blog/datadog-sumo-logic-log-management-integration/#sending-parsed-log-data-to-datadog)

[Datadog integrates with Sumo Logic](https://docs.datadoghq.com/integrations/sumo_logic/) in two ways: you can forward Sumo Logic log data to your Datadog event stream, and you can use Sumo Logic as a notification channel from Datadog alerts and events. In other words, each service can inform the other.

Once you [set up a webhook connection](https://help.sumologic.com/docs/alerts/webhook-connections/datadog/), you can send a Sumo Logic alert to your Datadog account.

[**Opentelemetry Datadog integration (TraceId/SpanId issues)**](https://github.com/quarkusio/quarkus/discussions/30562)

[**OpenTelemetry Collector**](https://opentelemetry.io/docs/collector/#introduction)

[**How to collect, customize, and standardize Java logs**](https://www.datadoghq.com/blog/java-logging-guide/)

A [Logger](https://www.datadoghq.com/blog/java-logging-guide/#loggers) is an instantiated object that logs messages for a defined scope of an application, application component, or service. enriching our log messages with [formats](https://www.datadoghq.com/blog/java-logging-guide/#layouts) and [additional metadata](https://www.datadoghq.com/blog/java-logging-guide/#enrich-your-java-logs). [Log levels](https://www.datadoghq.com/blog/java-logging-guide/#log-levels) let you filter what events a Logger will capture depending on their severity. log levels can vary depending on the framework ex. (Trace, Debug, Info, Warn, Error, Fatal). Some frameworks, such as Log4J and Log4J2, let you create custom log levels so you can add more granularity to your logging. Each level corresponds to a [numeric value](http://logging.apache.org/log4j/2.x/manual/customloglevels.html).

[Appenders](https://www.datadoghq.com/blog/java-logging-guide/#appenders)—or Handlers in some frameworks—define the endpoints for Loggers.

* [Writing logs to a console](https://www.datadoghq.com/blog/java-logging-guide/#log-to-console) - System.out or System.err
* [Writing logs to a file](https://www.datadoghq.com/blog/java-logging-guide/#log-to-a-file) - Logging to files also avoids network issues that might affect [streaming logs](https://www.datadoghq.com/blog/java-logging-guide/#stream-logs-to-an-endpoint) to a remote endpoint. using a log shipper or monitoring service, such as Datadog, to tail those files and send the messages to an endpoint. more about using FileAppenders with Logback in their [documentation](https://logback.qos.ch/manual/appenders.html#FileAppender).
* [Streaming logs to a remote endpoint](https://www.datadoghq.com/blog/java-logging-guide/#stream-logs-to-an-endpoint) - A SocketAppender, or SocketHandler, can send log messages to a network endpoint, specified by host and port.

Java logging frameworks—in particular Log4J2 and Logback—provide out-of-the-box support for a variety of other Appenders. You can view documentation for more information on endpoints for [Log4J2](https://logging.apache.org/log4j/2.0/manual/appenders.html), [Logback](https://logback.qos.ch/manual/appenders.html), and [JUL](https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html#a1.5).

Standardizing your log messages makes it easier to filter, sort, and search for specific logs. Most frameworks provide [Layouts](https://www.datadoghq.com/blog/java-logging-guide/#layouts), also called Formatters, which specify how an Appender should format log messages and what information is included. Pattern layouts can help make logs more readable, particularly if you’re logging to the console. [Logging to JSON](https://www.datadoghq.com/blog/java-logging-guide/#log-to-json) lets you add custom information to logs in the form of new key-value pairs. Java does not have a native JSON library, so logging backends require a separate dependency to format or encode messages in JSON. The [Logstash Logback JSON encoder](https://github.com/logstash/logstash-logback-encoder) and [Jackson](https://github.com/FasterXML/jackson) are two commonly used libraries.

A central monitoring service like Datadog can also let you correlate your logs with [request traces](https://docs.datadoghq.com/tracing/setup/java/) and infrastructure metrics—such as from your web servers, databases, etc.—giving you full insight into how your application is performing. See [our documentation](https://docs.datadoghq.com/tracing/advanced/connect_logs_and_traces/?tab=java&_gl=1*hau3g4*_gcl_au*OTIxMzA4MzM5LjE2Nzk2OTM5MzU.*_ga*MTAyOTU5OTM5NS4xNjc5NjkzOTM1*_ga_KN80RDFSQK*MTY4NTEzNTQ1MS4xMi4xLjE2ODUxMzk5NDcuNDUuMC4w*_fplc*MmtCM05rdVBwNUcybzhvcmlPNEpvTzA5eEVLaWNTUlVvTlVqaGpLNmRQRGNtYWozMzN5d3JCaVZuUVdqSTgyJTJGTTliYUZhc2E5N3dwbE1iZ1ZIb3BlaGRkcXpaVUtSZzZPa1dySW4yMSUyQlNSSXI1dTZFN2w3Q1NMZFV2OFlnQSUzRCUzRA..&_ga=2.61653161.402167876.1685132095-1029599395.1679693935) for more information on including trace IDs in your Java logs.

[**Sumo Logic vs. Datadog**](https://www.sumologic.com/compare/sumo-logic-vs-datadog/)

[**Distributed tracing**](https://www.datadoghq.com/knowledge-center/distributed-tracing/)