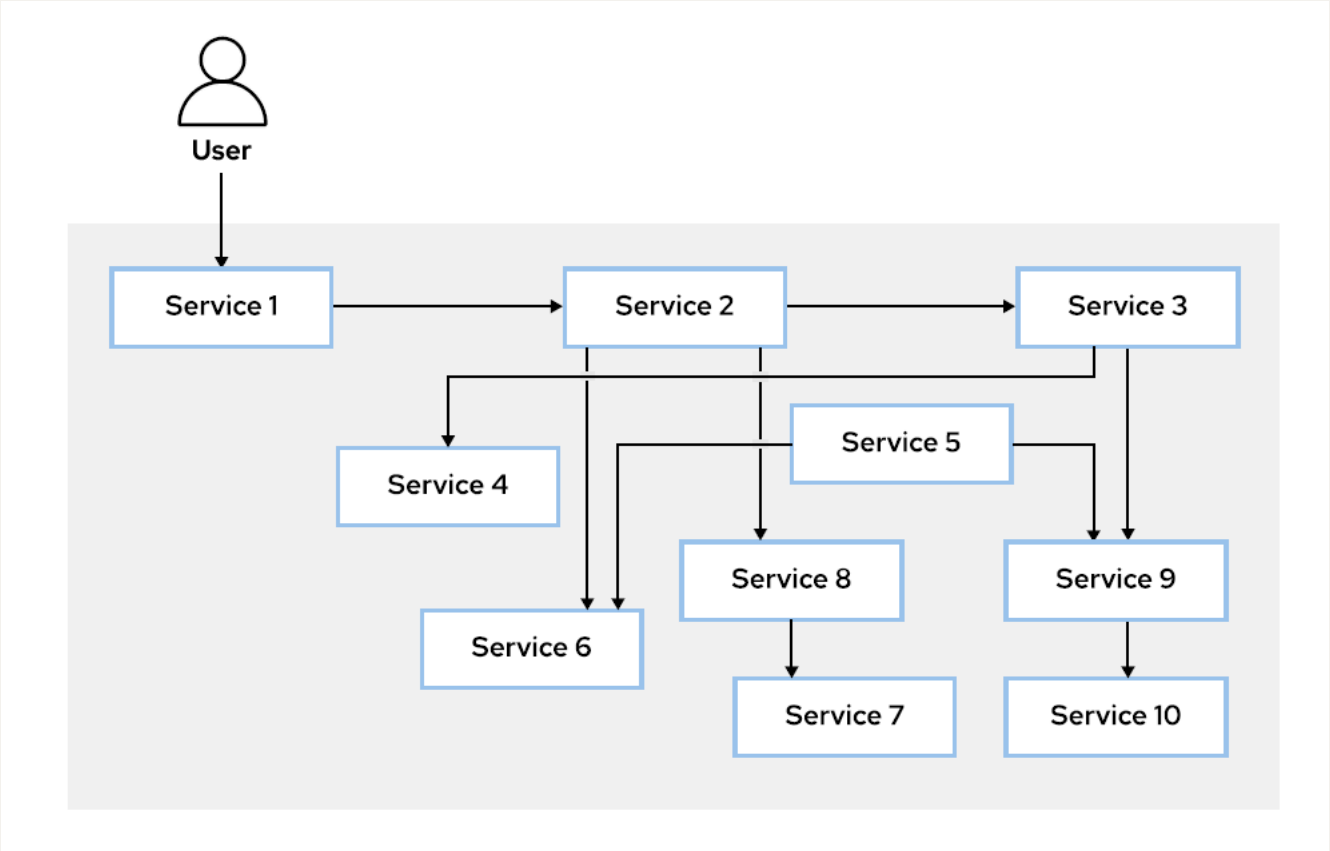
Quarkus Tracing

Quarkus usa

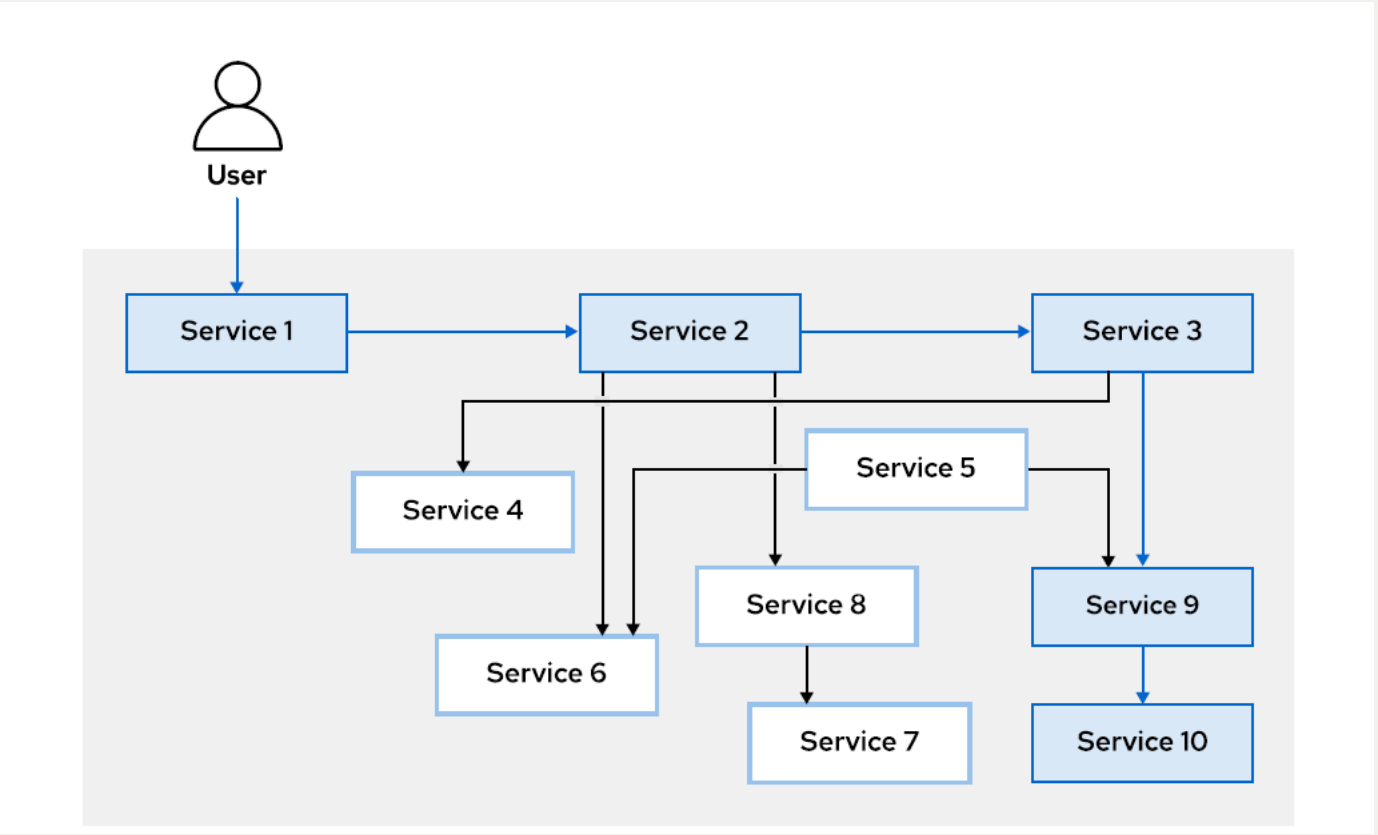
- OpenTelemetry



Sin tracing

Quarkus usa

- OpenTelemetry



Con tracing

Traces y Span en Tracing Distribuido

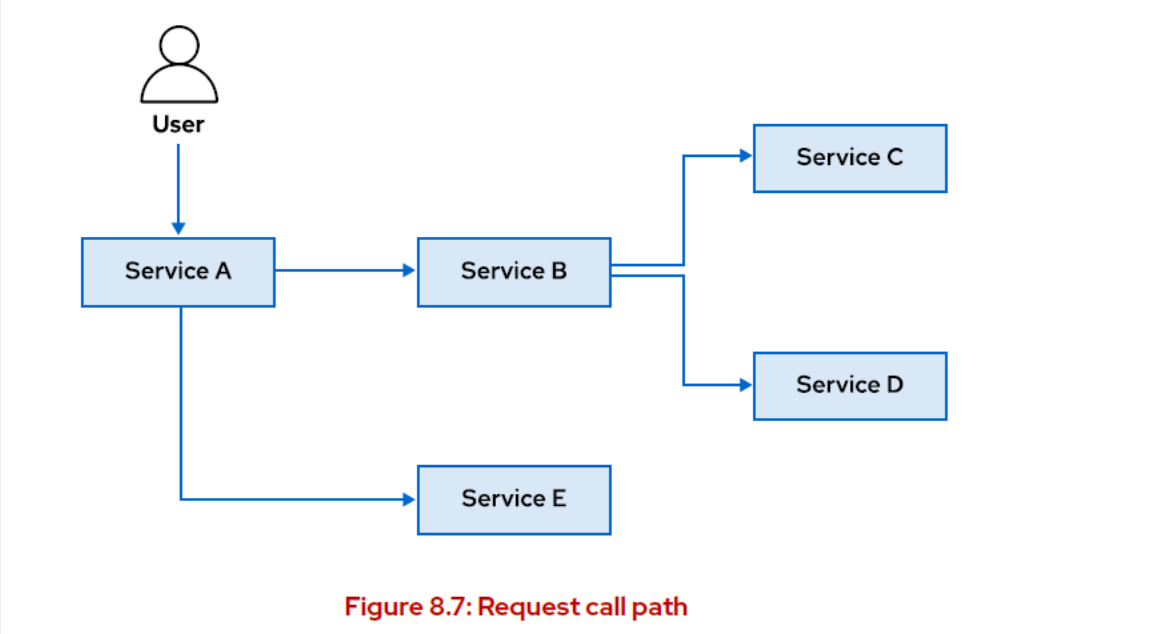


Figure 8.7: Request call path

Span

A Span represents a logical unit of work, which has a unique name, a start time, and the duration of execution. To model the service call flow across a set of services, Jaeger nests spans by execution order.

Trace

A Trace is an execution path of a request through a set of services, and is comprised of one or more spans.

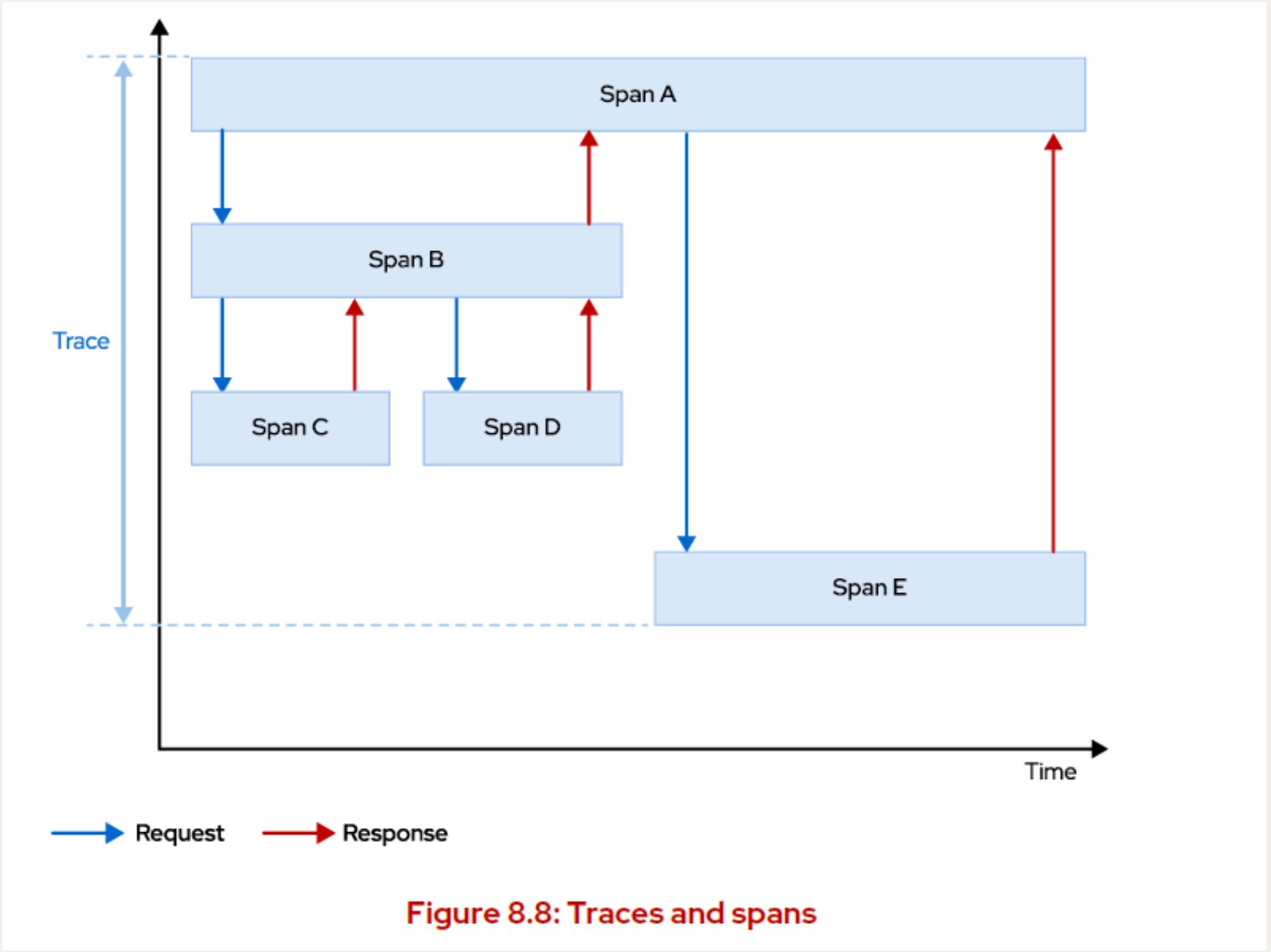


Figure 8.8: Traces and spans

Introducing Jaeger

Jaeger is a distributed tracing platform. Jaeger allows developers to configure their services to enable gathering runtime statistics about their performance.

Jaeger is made up of several components, which work together to collect, store, and display tracing data.

Jaeger Client

The OpenTracing API defines standard APIs for instrumentation and distributed tracing of microservices-based applications. Jaeger clients are language specific implementations of the OpenTracing API. They are used to configure applications for distributed tracing.

Jaeger Agent

The Jaeger agent is a network daemon, which listens for span data sent over User Datagram Protocol (UDP), which it batches and sends to the collector. The agent is meant to be placed on the same host as the application being traced.

Jaeger Collector

The Collector receives runtime statistics from the agents and places them in an internal queue for processing. This allows the collector to return a response immediately to the agent.

Storage

Collectors require a persistent storage back end. Jaeger has a pluggable mechanism for storage.

Query

Query is a service that retrieves runtime statistics from storage.

Jaeger Console

Jaeger provides a web-based console that lets you visualize your distributed tracing data. By using the console, you can trace the path of requests as they flow through the services in your application.

1)

```xml
</dependency>
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-opentelemetry</artifactId>
</dependency>
```

2)

```
# Enable Tracing (OpenTelemetry)
quarkus.otel.service.name=adder
quarkus.otel.traces.sampler=traceidratio
quarkus.otel.traces.sampler.arg=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=%X{spanId}, parentId=%X{parentId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n
quarkus.otel.exporter.otlp.traces.endpoint=http://localhost:4317
quarkus.otel.exporter.otlp.traces.protocol=grpc
```

Propiedad	Descripción	Ejemplo / Nota
`quarkus.otel.service.name=adder`	Define el **nombre del servicio** que aparecerá en las herramientas de observabilidad (como Jaeger, Tempo, Dynatrace).	Aquí el servicio se llama `adder`. Esto ayuda a identificar de qué microservicio provienen los spans y traces.
`quarkus.otel.traces.sampler=traceidratio`	Define el **tipo de muestreo (sampling)** para traces. `traceidratio` significa que se usará un ratio (porcentaje) basado en el trace ID para decidir si se toma o no el trace.	Otros valores comunes: `always_on` (siempre toma el trace), `always_off` (nunca), `parentbased_traceidratio`.
`quarkus.otel.traces.sampler.arg=1`	Es el **argumento del sampler**, en este caso el ratio. Un valor de `1` significa que **se capturan el 100% de los traces**. Si fuera `0.5`, se capturaría el 50% de las solicitudes.	Útil para reducir carga en ambientes de producción (por ejemplo, usar 0.1 = 10%).
`quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=%X{spanId}, parentId=%X{parentId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%n`	Configura el **formato de los logs en consola**. Inserta los campos de OpenTelemetry (traceId, spanId, etc.) en cada línea de log para correlacionar logs con traces.	- `traceId` = ID global de la solicitud. - `spanId` = ID de la operación actual. - `parentId` = Span padre (para jerarquía). - `sampled` = indica si el trace fue muestreado.
`quarkus.otel.exporter.otlp.traces.endpoint=http://localhost:4317`	Define la **URL del endpoint OTLP (OpenTelemetry Protocol)** al que Quarkus enviará los datos de tracing.	Normalmente es el agente o colector OTLP (por ejemplo, Jaeger, Tempo, Dynatrace).
`quarkus.otel.exporter.otlp.traces.protocol=grpc`	Define el **protocolo de transporte** para enviar los traces. Puede ser `grpc` (binario, más rápido) o `http/protobuf`.	`grpc` es preferido porque consume menos recursos y es más eficiente para traces de alto volumen.

NOTAS

- Por defecto, OpenTelemetry le agrega tracing a todos los REST endpoints donde hagas @GET y @POST.
- Puedes usar opcionalmente @Traced en tu capa de servicio o capa de datos para hacer Troubleshooting y encontrar los cuellos de botella.

Jaeger ScreenShots

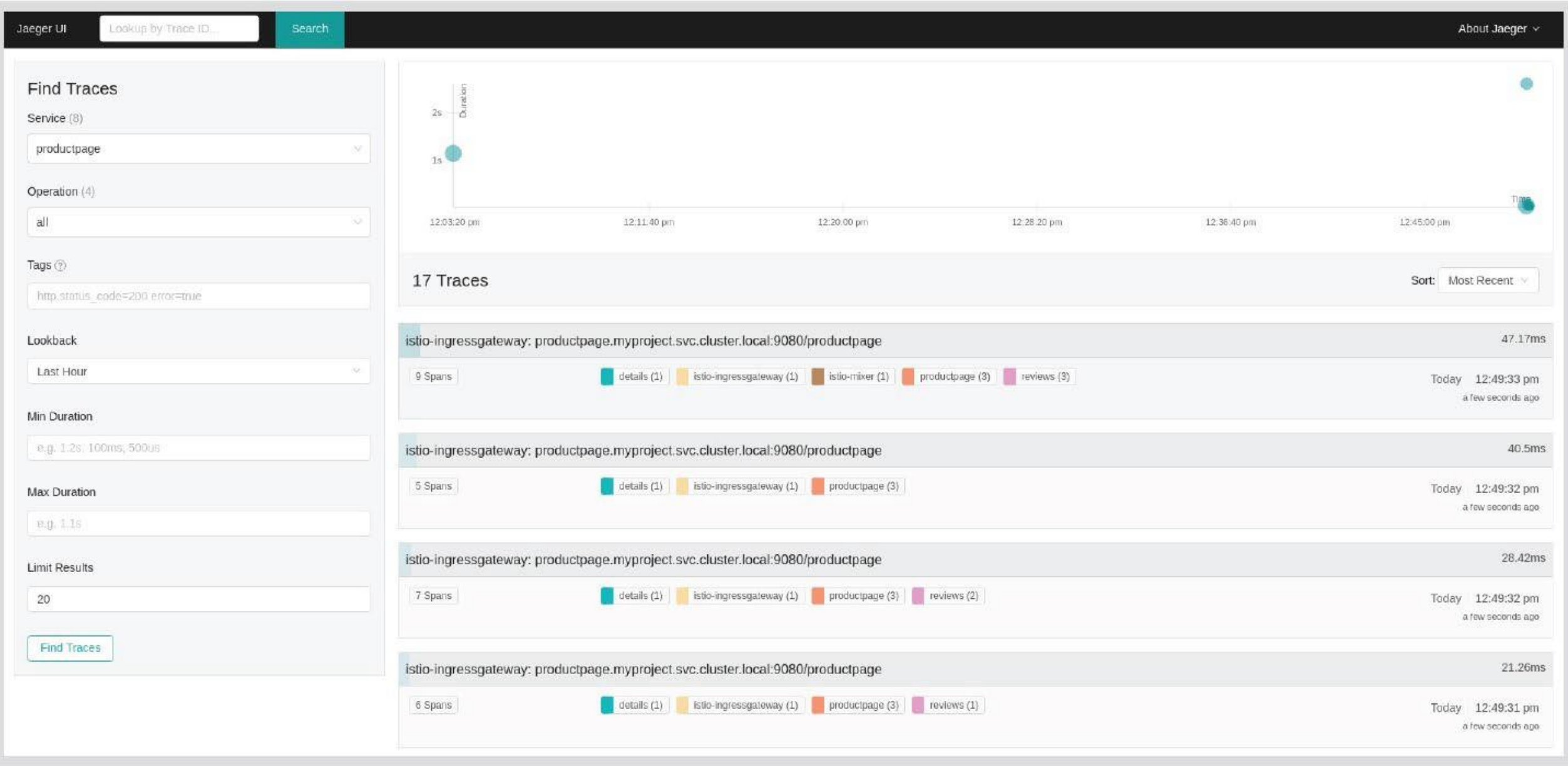


Figure 8.9: List of traces

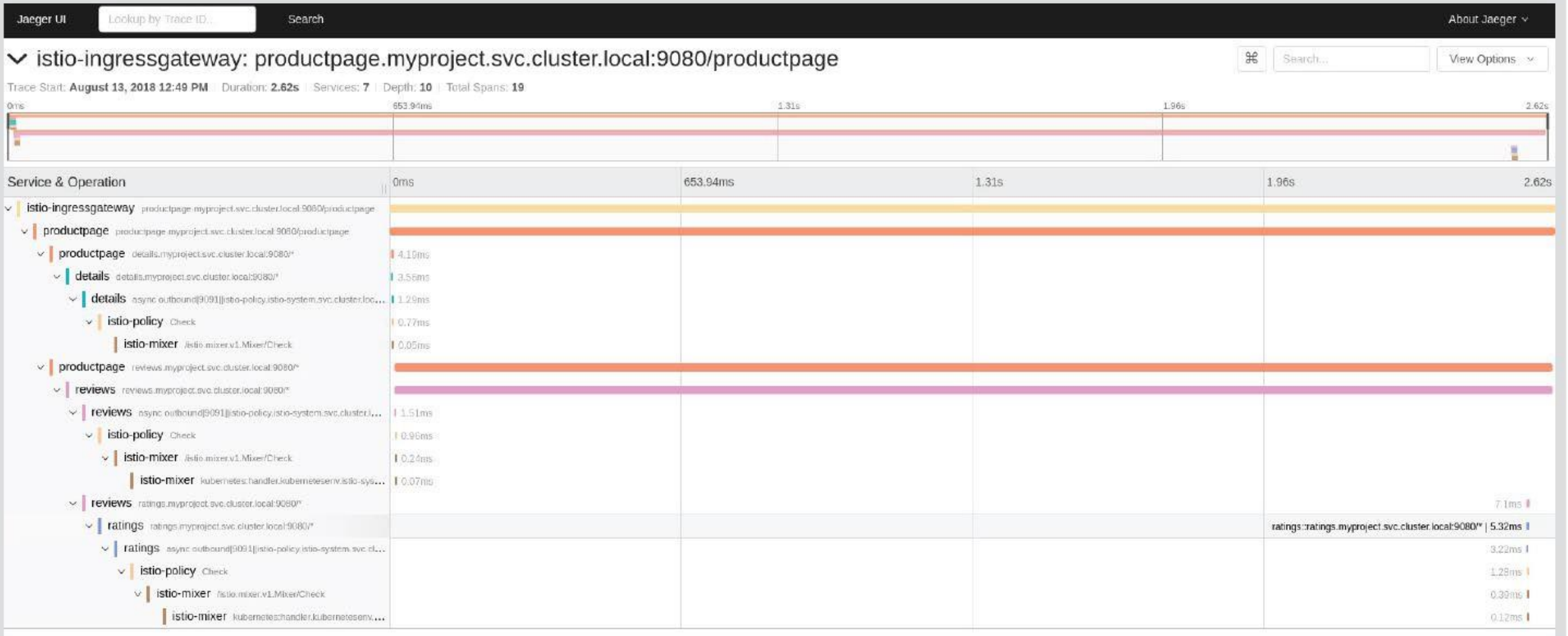


Figure 8.10: Trace details

Recursos

- https://www.jaegertracing.io/

  https://opentelemetry.io/