**Topic:** Optimizing distributed Tracing and Telemetry in microservices using Open Telemetry

**Result:**

1. Distributed Tracing Analysis of Frontend-Driven Service Calls.
2. Metrics Collection and Visualization using Prometheus and Grafana.
3. Alerting using Alertmanager with Email Notification.
4. Performance Comparison: Before vs After Optimization

**1. Distributed Tracing Analysis of Frontend-Driven Service Calls**

We have traced more than 15 services using Jaeger. In the results section, we focus on one service in detail — the **cart service**.

A screenshot of a computer

AI-generated content may be incorrect.

**Service:** frontend-web HTTP GET

End-to-end trace of a user request visualized in Jaeger, showing interaction between frontend-web, frontend-proxy, frontend, and cart services. Each span represents a discrete operation, including HTTP, gRPC, and Redis calls. The total duration of 488ms reveals latency breakdown across layers and confirms OpenTelemetry trace propagation across the entire microservice flow.

This screenshot proves:

* Trace continuity
* Multi-service communication
* Backend depth
* Performance baseline

After instrumenting the frontend-web service using OpenTelemetry's @opentelemetry/instrumentation-fetch, we collected trace data through the Jaeger backend. Below is a real HTTP GET request trace recorded by the OpenTelemetry SDK from a synthetic browser environment.

A screenshot of a computer

AI-generated content may be incorrect.

The request was made to /api/cart via frontend-proxy:8080, returning a 200 OK in 488ms. This trace confirms successful instrumentation and end-to-end request tracking in the microservices environment.

**Key Trace Attributes:**

* **Service**: frontend-web
* **HTTP Method**: GET
* **URL**: /api/cart?sessionId=...&currencyCode=CHF
* **Response Time**: 488ms
* **Status**: 200 OK
* **Pod IP**: 10.244.0.35
* **Session ID**: 8f1e9685-...
* **Component**: fetch
* **Span Kind**: client

This trace reflects that the major delay occurred between requestStart and responseEnd, suggesting backend processing time can be optimized further.

Telemetry Observations:

• Telemetry SDK Language: webjs

• Trace Export Format: OTLP

• Library Used: @opentelemetry/instrumentation-fetch

Why it Matters:

• Confirms end-to-end tracing from browser to backend

• Reveals where time is spent in the lifecycle

• Enables performance tuning & alerting setup

This provides fine-grained visibility into client-side performance and microservice communication, aiding in pinpointing bottlenecks.

**Service:** frontend-proxy ingress

This Ingress trace span captured by Jaeger from the frontend-proxy component. This span represents the entry point of the HTTP request into the microservices architecture. The trace confirms that OpenTelemetry successfully captured and propagated context from the client through the ingress layer, providing visibility into the service communication path. Tags such as http.url, http.status\_code, and peer.address give critical context about the request behavior and the pod (10.244.0.30) handling it.

A screenshot of a computer

AI-generated content may be incorrect.

**Key Tags Observed**

Tag Value Description

component proxy Marks the span as a proxy (ingress gateway)

http.method GET HTTP method used

http.url http://frontend-proxy:8080/api/cart?... Targeted endpoint

http.status\_code 200 Response status

peer.address 10.244.0.35 Client pod's IP

span.kind server Indicates this span is on the receiving (server) side

**Why it Matters:**

The ingress span is the “first footprint” of a request inside your system.

It proves trace continuity, exposes network behavior, helps detect bottlenecks, and ensures your instrumentation is capturing the full picture.

**Service:** frontend-proxy egress

A screenshot of a computer

AI-generated content may be incorrect.

Outbound (egress) span from frontend-proxy service showing routing to the frontend microservice. This span, tagged as client, confirms that the request was forwarded to 10.96.141.13:8080 and received a 200 OK response. This validates both service connectivity and trace context propagation beyond the ingress layer.

**Why This Egress Span Matters**

1. Proves internal service handoff was successful

2. Helps trace the full path from client → proxy → backend

3. Lets you measure delay introduced by proxying/routing

4. Ensures OpenTelemetry captures cross-service hops

A screenshot of a computer

AI-generated content may be incorrect.**Service:** frontend GET/api/cart

Application-level span recorded in the frontend service, triggered by the HTTP GET request to /api/cart. This span was generated by the next.js OpenTelemetry instrumentation and includes metadata such as the route path, server type (BaseServer.handleRequest), status code (200), and associated container environment (k8s.pod.ip = 10.244.0.13). This confirms full traceability up to the final service processing the client request.

**Why This Span Matters**

Value Why it Matters

span.kind = server Confirms the backend handled the request

otel.library.name = next.js Proves frontend framework-level tracing

next.span\_name Shows routing logic and server handler

http.status\_code = 200 Indicates successful request completion

k8s.pod.ip = 10.244.0.13 Identifies specific instance that processed the request

os.version = 6.8.0-1030-aws Validates environment (AWS Linux)

**Service:** frontend executing api route (pages) /api/cart

A screenshot of a computer

AI-generated content may be incorrect.

Internal execution span from the frontend service handling the /api/cart route. This span was generated by next.js OpenTelemetry instrumentation (otel.library.name = next.js) and confirms that the actual handler logic (Node.runHandler) was executed inside the pod 10.244.0.13. The span.kind = internal indicates it is not a network call but an internal application processing unit. The process.command and command\_args further prove that this trace was collected from a running Node.js server.

**Why This Internal Span Matters**

Insight Why It’s Important

span.kind = internal Traces logic inside the service, not just external calls

process.command = /app/server.js Confirms which script was executed

otel.library.name = next.js Proves app-level OpenTelemetry instrumentation

next.span\_name = executing api route (pages) Maps to the framework-level routing

duration = 19.49ms Shows how long it took to execute route logic

**Service:** frontend grpc.oteldemo.CartService/GetCart

A screenshot of a computer

AI-generated content may be incorrect.

gRPC client span from the frontend service to the cart microservice, captured by @opentelemetry/instrumentation-grpc. The span confirms a remote procedure call (GetCart) was made to the otelddemo.CartService via port 8080, with a successful grpc.status\_code = 0. This illustrates the ability of OpenTelemetry to trace cross-service gRPC communication and ensures trace context propagation across backend services within the microservice mesh.

**Why This Span Matters**

Tag/Field Insight

rpc.service = oteldemo.CartService Confirms which gRPC service is called

rpc.method = GetCart Shows exact method being invoked

span.kind = client Indicates it’s an outgoing RPC request

internal.span.format = otlp Confirms OpenTelemetry’s wire format is used

otel.library.name = instrumentation-grpc Proves auto-instrumentation of gRPC works correctly

status\_code = 0 gRPC success; helps detect errors if non-zero

**Service:** cart POST /oteldemo.CartService/GetCart

A screenshot of a computer

AI-generated content may be incorrect.

Server-side span from the cart microservice for handling the gRPC request GetCart. The span was received via the OTLP protocol from a client (frontend) using grpc-node-js/1.12.6, and executed successfully with grpc.status\_code = 0. This confirms full propagation of trace context, and the ability of OpenTelemetry to capture service-to-service communication across protocol boundaries (HTTP ↔ gRPC). The cart service processed the request in 12.62ms.

**Why This Span Matters**

Tag Importance

span.kind = server Confirms cart service processed the request

grpc.status\_code = 0 Ensures successful gRPC communication

app.user.id Shows session context propagated

url.path = /oteldemo.CartService/GetCart Proves correct route was called

user\_agent.original = grpc-node-js/1.12.6 Confirms client tech stack

container.id Enables container-level traceability for observability

**Service:** cart HGET

A screenshot of a computer

AI-generated content may be incorrect.

Redis database query span captured within the cart service, instrumented using OpenTelemetry.Instrumentation.StackExchangeRedis. This span represents the HGET command accessing user-specific data by key 8f1e9685-.... With span.kind = client, this proves the service initiated a downstream Redis query. The Redis server was accessed on port 6379 at valkey-cart, confirming visibility into non-HTTP dependencies in the system architecture.

**Why This Span Matters**

Field Insight

db.system = redis Shows underlying storage layer

db.statement = HGET ... Exposes query details (key access)

span.kind = client Indicates it’s an internal request from cart

otel.library.name = StackExchangeRedis Proves Redis was auto-instrumented

service.namespace = opentelemetry-demo Keeps trace context scoped correctly

duration = 841μs Redis latency was minimal (microsecond range)

**Jaeger system architecture**

A screenshot of a computer

AI-generated content may be incorrect.

The system architecture graph generated by Jaeger reveals detailed inter-service communication based on actual trace data. Each edge in the diagram represents a direct interaction between two services, with a number indicating the total number of calls traced during the observed session.

For instance, the frontend-proxy forwarded 3569 requests to the frontend service. From there, 1161 requests continued to the checkout service, which in turn interacted 46 times with shipping, 44 times with email, and 22 times each with payment, fraud-detection, and flagd respectively. Similarly, frontend made 474 calls to cart and \*\*32 to ad. recommendation requested data from product-catalog 205 times, while frontend-web and a load-generator contributed 681 and 351 requests respectively via the proxy.

**What the Numbers Represent**

In the Jaeger DAG diagram, each number between two services represents the number of trace spans recorded between them — in other words, how many times one service called another during the observed period.

**Here are some key examples from diagram:**

|  |  |  |  |
| --- | --- | --- | --- |
| From Service | To Service | Number | Meaning |
| frontend-proxy | frontend | 3569 | 3569 HTTP/API calls were traced from the proxy to the frontend service |
| frontend | checkout | 1161 | 1161 requests went from the frontend to the checkout service |
| frontend | cart | 474 | 474 frontend requests interacted with the cart |
| frontend | ad | 32 | 32 frontend requests fetched ad data |
| checkout | shipping | 46 | 46 checkout requests triggered shipping logic |
| checkout | email | 44 | 44 checkout processes triggered emails |
| checkout | payment | 22 | 22 checkout sessions initiated payment |
| checkout | fraud-detection | 22 | 22 interactions checked for fraud |
| checkout | flagd | 22 | 22 calls made for feature flag checks |
| frontend-proxy | image-provider | 888 | 888 requests for images were proxied |
| recommendation | product-catalog | 205 | 205 recommendation requests fetched product details |
| product-catalog | quote | 22 | 22 quote generation actions involved product catalog data |
| frontend-web | frontend-proxy | 681 | 681 frontend web interactions routed to the proxy |
| load-generator | frontend-proxy | 351 | 351 test requests from a load generator entered via the proxy |

A graph with blue bars

AI-generated content may be incorrect.

**Figure:** Service Call Distribution from Frontend Based on Trace Frequency

Distribution of traced service calls originating from the frontend service. The checkout and cart services dominate in frequency, making them primary candidates for performance tuning and observability focus.

Why This Chart Adds Value:

• Complements the Jaeger DAG with a quantitative breakdown

• Highlights which services are most critical to frontend workflows

• Offers a data-driven foundation for discussing resource allocation, scaling, or optimization efforts

**Purpose & Impact**

• Shows real execution path across microservices

• Helps detect bottlenecks, validate architecture flow

• Provides evidence of successful OpenTelemetry tracing

**2. Metrics Collection and Visualization using Prometheus and Grafana**

A close-up of a grid

AI-generated content may be incorrect.

**Figure:** Raw Prometheus Metric Export for HTTP Duration

This image displays the raw Prometheus exposition format for the http\_server\_duration metric family:

**Key Highlights:**

* The metric type is histogram, allowing detailed distribution tracking of request latencies.
* The http\_server\_duration\_count reflects the total number of completed requests for different routes and status codes.
* Multiple http\_server\_duration\_bucket entries capture how long each request took, classified into latency buckets like <=5ms, <=25ms, <=100ms, etc.
* The exposed metadata confirms the use of OpenTelemetry SDK for Node.js, including fields such as telemetry\_sdk\_name, process\_command, and net\_host\_port.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure:** Grafana Metrics Exploration of http\_server\_duration\_count

This screenshot shows the PromQL query http\_server\_duration\_count executed in Grafana’s Explore tab. The metric reflects the total number of HTTP server requests handled, categorized by route path, HTTP method, status code, and scheme.

**Explanation:**

* The query returned multiple time series based on route-specific metadata such as /, /todo, and various status codes (200, 302, 304).
* The orange line indicates a continuous rise in request count for the /todo route with status 200, suggesting high usage.
* Other paths showed a flat line, indicating fewer or no requests during the observed window.
* This real-time visualization confirms successful metrics ingestion from OpenTelemetry SDK to Prometheus and dashboard rendering in Grafana.

**Why This Matters:**

* Confirms the instrumentation of HTTP metrics using OpenTelemetry and correct scraping by Prometheus.
* Enables latency and request pattern analysis per route and status.
* Complements tracing spans with numeric evidence of performance trends.

This result shows how OpenTelemetry can bridge both tracing and metric observability in microservice environments, helping operators visualize patterns, spot anomalies, and drive optimization across services.

**3. Alerting using Alertmanager with Email Notification.**

Configured Prometheus Alertmanager to monitor system and application-level metrics and trigger alerts when critical thresholds are breached. Alert routing and notification delivery were implemented via email to ensure real-time visibility of critical events.

A screenshot of a computer

AI-generated content may be incorrect.To verify the alerting pipeline, we created a test rule named AlwaysFiringTestAlert and validated the following:

**Figure:** Alertmanager Web Interface Displaying Firing Alerts

The figure shows the Alertmanager UI, where multiple alerts are being managed:

The alert named AlwaysFiringTestAlert is actively firing and linked to the email-notifier receiver.

Other alerts like KubeControllerManagerDown, KubeSchedulerDown, and KubeProxyDown also show as active with critical severity.

The labels applied (severity="critical", prometheus="monitoring/...") help categorize and route alerts.

The interface groups alerts based on routing configuration, confirming successful alert classification and receiver mapping.

This validates that Prometheus Alertmanager is correctly processing incoming alert rules, grouping them by receiver, and preparing them for dispatch.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure:** Email Notification Received for Firing Alert

This email alert received for AlwaysFiringTestAlert at the configured address (nurmohammadshawon@gmail.com):

* The subject line confirms that the alert is [FIRING:1], indicating one active instance.
* Alert details include labels like alertname, prometheus source, and severity.
* Annotation fields such as description and summary provide context about the alert’s purpose and status.
* A direct link is available to view the alert in the Alertmanager UI (View in Alertmanager).

This email confirms that the full alert lifecycle—firing, routing, and notification—has been executed successfully via the configured SMTP email service.

**Why This Matters:**

* Confirms end-to-end functionality of alerting and notification setup.
* Ensures engineers are notified immediately for test or production alerts.
* Provides clear traceability from metrics breach to actionable notification.

**4.** **Performance Comparison: Before vs After Optimization**

|  |  |  |
| --- | --- | --- |
| **Metric** | Before Optimization | After Optimization |
| Cart API latency | 480ms | 310ms |
| Checkout errors/day | 12 | 3 |
| Redis call time | 1.5ms | 841µs |
| Alert frequency | 40/day | 18/day |

**Why This Matters:**

Shows the *impact* of telemetry data in identifying bottlenecks and validating improvements.