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 (HeadlessChrome/133.0.0.0).

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.

A screenshot of a computer

AI-generated content may be incorrect.

**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

**Timeline Events (Trace Log):**

| **Event** | **Timestamp (ms)** |
| --- | --- |
| fetchStart | 1.1 |
| domainLookupStart | 1.1 |
| connectStart | 1.1 |
| requestStart | 408.5 |
| responseStart | 437.6 |
| responseEnd | 461.9 |

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.

**🖥️ In the Final Presentation (Slide Format)**

**Slide Title: 🔍 *Sample Distributed Trace from Jaeger***

**Slide Content:**

**✅ Trace Details:**

* **Service**: frontend-web
* **URL**: /api/cart
* **Method**: GET
* **Status**: 200 OK
* **Duration**: 488ms
* **User-Agent**: HeadlessChrome

**📊 Span Breakdown (Timeline):**

csharp

CopyEdit

[fetchStart → requestStart: 408ms]

[requestStart → responseEnd: 53.4ms]

*(Visualize this with a simple Gantt-style bar if possible)*

🧠 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

A screenshot of a computer

AI-generated content may be incorrect.

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.

🔍 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

📊 Slide Explanation (For Presentation)

Slide Title:

🚪 Ingress Layer Trace Visibility with Jaeger

Slide Content:

📍 Captured span at the frontend-proxy service

🌐 Shows HTTP GET request routing into system

🧠 Context like URL, response code, and client IP included

🧩 Span kind: server → first receiving point of request

✅ Confirms trace continuity across boundary (client → server)

🧠 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.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure X.X**: 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.

**🖥️ Suggested Slide Explanation (Presentation)**

**Slide Title**: 📤 **Egress Span: Routing to Internal Service**

**Key Points**:

* This span shows the **proxy forwarding traffic** to the frontend service.
* It marks the **transition from edge to core services**.
* The **upstream address and cluster** help in service discovery and latency mapping.
* Duration (~28ms) helps analyze routing overhead.

**🔑 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.

**Figure X.X**: Egress span from frontend-proxy in the OpenTelemetry-instrumented trace, visualized in Jaeger. This client-side span represents the forwarding of an HTTP request to the upstream frontend microservice at IP 10.96.141.13:8080. It confirms successful routing, context propagation, and trace continuity. The presence of detailed tags (e.g., http.status\_code=200, component=proxy, upstream\_cluster=frontend) shows complete observability from edge to core.

**🖥️ Slide Explanation (Final Presentation)**

**Slide Title**:  
📤 **Egress Span – Forwarding to Upstream Microservice**

**Key Highlights**:

* ✅ Span kind: client → proxy making a request to backend
* 🔄 Upstream address: 10.96.141.13:8080
* 🔍 Internal span format: otlp (OpenTelemetry protocol)
* 📦 Namespace: opentelemetry-demo, version: 2.0.2
* 📡 Peer & upstream cluster: Identifies the destination microservice
* 📊 Duration: 28.25ms (routing latency insight)

**🧠 Why It Matters (Short Summary)**

This span bridges the **edge proxy to backend microservice**. It confirms OpenTelemetry’s ability to trace across service boundaries and gives performance data at the routing layer. It’s vital for understanding **where time is spent** and **where failures could occur** in a microservice mesh.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure X.X**: 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.

**🖥️ Slide Explanation (Final Presentation)**

**Slide Title**:  
📦 **Backend Application Span – GET /api/cart**

**Slide Points**:

* ✅ Captured by next.js via OpenTelemetry
* 📍 URL: /api/cart with session and currency query parameters
* 🧠 Executed on pod: 10.244.0.13
* 🛠️ Server handler: BaseServer.handleRequest
* 📡 OpenTelemetry protocol: otlp, version 0.0.1
* ⏱️ Duration: 23.23ms
* 🧩 Span kind: server (final handler of the 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) |

A screenshot of a computer

AI-generated content may be incorrect.

**Figure X.X**: 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.

**🖥️ Slide for Presentation**

**Slide Title**: 🧩 **Internal Span – API Route Execution in Frontend**

**Slide Content**:

* 🛠️ Span Kind: internal (inside the Node.js process)
* 🔄 Route: /api/cart
* 🧠 Handler: Node.runHandler
* 🔍 Host IP: 10.244.0.13, Pod: frontend-f9c7dc6cf-fp9tz
* 🧪 Instrumented with: next.js + OpenTelemetry
* 💡 Confirms execution of core route logic in the backend

**🔍 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 |

A screenshot of a computer

AI-generated content may be incorrect.

Figure X.X: 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.

🖥️ Slide for Final Presentation

Slide Title: 🔁 gRPC Span – Service-to-Service Communication

Slide Points:

📡 Service: frontend ➝ cart

🛠️ RPC Method: GetCart

🧠 RPC System: grpc

✅ Status: rpc.grpc.status\_code = 0 (Success)

🔍 Peer Address: cart:8080

🧪 Instrumented with: @opentelemetry/instrumentation-grpc (v0.57.1)

📦 Span Kind: client – confirms outgoing RPC request

🔍 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

A screenshot of a computer

AI-generated content may be incorrect.

**Figure X.X**: 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.

🖥️ Slide for Presentation

Slide Title: 📥 Cart Service – gRPC Server Span

Slide Content:

📦 Service: cart

🔄 Method: POST /oteldemo.CartService/GetCart

🛠️ RPC Method: GetCart, gRPC Status: 0 (Success)

✅ HTTP Status: 200

🧠 Container ID confirms execution environment

🧪 User-Agent: grpc-node-js/1.12.6

⏱️ Duration: 12.62ms

📡 Span Kind: server – confirms successful backend processing

🔍 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

A screenshot of a computer

AI-generated content may be incorrect.

🧾 Thesis Figure Caption (PDF-ready)

Figure X.X: 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.

🖥️ Slide for Presentation

Slide Title: 🗄️ Redis DB Span – Final Layer of Trace

Slide Content:

📌 Redis Command: HGET

📁 Key Accessed: user session ID

🧠 Hostname: cart-556fd66db4-65m7g, IP: 10.244.0.55

📡 Server Address: valkey-cart:6379

🧪 Instrumentation: StackExchangeRedis, v1.11.0-beta.2

✅ Confirms internal DB call is traced and observable

⏱️ Duration: 841μs

🔍 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)

**🎯 Thesis & Presentation Goal**

**Demonstrate that OpenTelemetry enables full-stack observability by capturing, propagating, and visualizing distributed tracing data across application layers and protocols.**

**✅ What You've Captured (Trace Flow)**

| **Step** | **Layer** | **Span Name** | **Purpose** |
| --- | --- | --- | --- |
| 1 | Client | frontend-web: HTTP GET | Initiates the request |
| 2 | Proxy Ingress | frontend-proxy ingress | First service boundary |
| 3 | Proxy Egress | router frontend egress | Routes to next service |
| 4 | App Server | GET /api/cart | Handles route request |
| 5 | Internal Logic | executing api route | Application code execution |
| 6 | gRPC Call | grpc.oteldemo.CartService/GetCart | Inter-service gRPC call |
| 7 | gRPC Server | POST /oteldemo.CartService/GetCart | Cart service receives the RPC |
| 8 | Redis Query | HGET | Cart service queries Redis |

**📘 How to Present It in Your Thesis Result Section**

**📑 Section Title:**

**5.2 Results: Full-Stack Distributed Tracing with OpenTelemetry**

**📄 Content Example:**

We instrumented our microservice architecture using OpenTelemetry SDKs and auto-instrumentation libraries to trace application-level, inter-service, and database-level operations. The collected traces were exported using the OTLP protocol and visualized in Jaeger.

**Figure 5.2.1–5.2.8** illustrate the end-to-end journey of a user request to /api/cart, traced across the system. Spans include ingress and egress through the proxy, internal execution in the frontend application, a gRPC call to the cart service, and a Redis database query.

This full-chain trace confirms:

* 🧩 **Instrumentation Coverage**: OpenTelemetry captured spans from HTTP, gRPC, and Redis.
* 🛰️ **Context Propagation**: Trace and span context were preserved across 4 services and protocols.
* ⏱️ **Latency Insight**: Each span’s duration enabled micro-level performance profiling.
* ⚙️ **Service Transparency**: We can now track down root causes of latency, failures, or bottlenecks via span tags, service names, and logs.

The following figures show the trace sequence in detail.

* **Figure 5.2.1** – HTTP GET span from frontend-web
* **Figure 5.2.2** – Ingress proxy span
* **Figure 5.2.3** – Egress proxy span
* **Figure 5.2.4** – GET /api/cart in frontend app
* **Figure 5.2.5** – Internal handler logic span
* **Figure 5.2.6** – gRPC client call to CartService
* **Figure 5.2.7** – gRPC server span in CartService
* **Figure 5.2.8** – Redis HGET query span

**🖥️ Final Presentation Strategy**

**🎥 Slide Flow (8–10 slides for this result)**

1. **Title Slide**: “Tracing in Microservices using OpenTelemetry + Jaeger”
2. **Architecture Overview** (diagram showing services)
3. **Slide 1** – Frontend request span
4. **Slide 2** – Proxy ingress/egress
5. **Slide 3** – Frontend route handler
6. **Slide 4** – gRPC client + server
7. **Slide 5** – Redis database query
8. **Slide 6** – Combined timeline view (all spans stacked in trace)
9. **Slide 7** – Key findings: propagation, latency, coverage
10. **Slide 8** – Why this matters (production observability, debugging, scaling)

**🔑 Extra Points to Include (For Thesis & Defense)**

**1. Why This Matters**

* OpenTelemetry is **vendor-neutral** and **cloud-native**
* Jaeger enables **deep root cause analysis**
* Enables **latency budgeting** and **performance tuning**
* Makes it easy to debug **distributed failures** (e.g., timeout in gRPC, Redis cache miss)

**2. Tools Used (brief in methodology/results)**

* OpenTelemetry SDKs: @opentelemetry/api, instrumentation-fetch, instrumentation-grpc, StackExchangeRedis, etc.
* Collector Export Protocol: OTLP
* Visualization: Jaeger UI

**3. Metrics to Highlight (based on your traces)**

| **Component** | **Latency** |
| --- | --- |
| Proxy ingress | ~28.76ms |
| Application handler | ~19.49ms |
| gRPC client call | ~18ms |
| Redis query | ~841μs |

This shows where time is spent → **optimize the longest spans** (proxy + app logic)

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.

These numbers are direct evidence of real-time distributed tracing. They show not only service dependencies but also usage frequency, helping identify critical paths, bottlenecks, and potential scalability concerns. The use of OpenTelemetry instrumentation enables this deep visibility, while Jaeger assembles and visualizes the entire request journey — making it an essential component of modern observability infrastructure.

✅ 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 your 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

**🎯 Slide Title:**

**"Distributed Tracing Architecture with Jaeger"**

**🖼️ Slide Layout (2-Part Design):**

**Left Side: Visual (Graph Screenshot)**

* Insert the **Jaeger System Architecture DAG** screenshot
* Crop or zoom in slightly to make service names and arrows visible
* Add a small caption below:  
  *"Figure: Trace-based service dependency graph generated by Jaeger"*

**Right Side: Summary (Bullet Points)**

Use concise, high-impact bullets like:

✅ **Overview**

* Visualizes real-time service-to-service communication
* Each **arrow** = traced call between two services
* **Number** on arrow = number of spans (requests) traced

📊 **Key Observations**

* frontend-proxy → frontend: **3569 requests**
* frontend → checkout: **1161 requests**
* checkout fan-out:
  + → shipping: **46**
  + → email: **44**
  + → payment, fraud-detection, flagd: **22 each**
* recommendation → product-catalog: **205 requests**
* frontend-web & load-generator both sent traffic to frontend-proxy (681 + 351)

🎯 **Purpose & Impact**

* Shows **real execution path** across microservices
* Helps detect **bottlenecks**, validate **architecture flow**
* Provides **evidence of successful OpenTelemetry tracing**

**📢 Optional Final Line (bottom of slide):**

“Each traced request provides full context, helping developers and ops teams ensure reliability and performance in distributed systems.”