

# Traces & Anomaly Monitoring — A Practical Guide

A compact, Medium-style walkthrough that explains the tracing decisions we made and the design/implementation of the time-aware + adaptive anomaly detection system.

---

A compact, Medium-style walkthrough that explains the tracing decisions we made and the design/implementation of the time-aware + adaptive anomaly detection system. This is written to help you replicate the setup, run it in your environment, and extend it safely.

---

## TL;DR 🚀

- We fixed span parent/child propagation across browser → gateway → API → RabbitMQ → worker by explicitly setting and passing contexts and by injecting a known `x-parent-traceparent` header through the queue.
  - We built a monitoring pipeline that computes **time-aware baselines** (1-hour buckets, 30-day window) and derives **adaptive thresholds** (percentiles) nightly, mapping deviations to **SEV1–SEV5**.
  - Ollama LLM is integrated to analyze traces + correlated Prometheus metrics; use the smaller `llama3.2:1b` for better latency in production-like environments.
- 

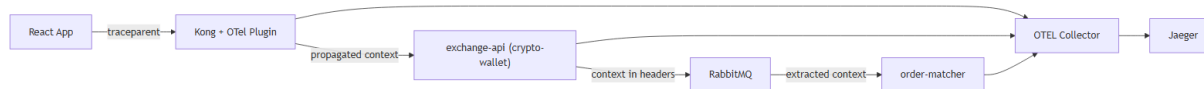
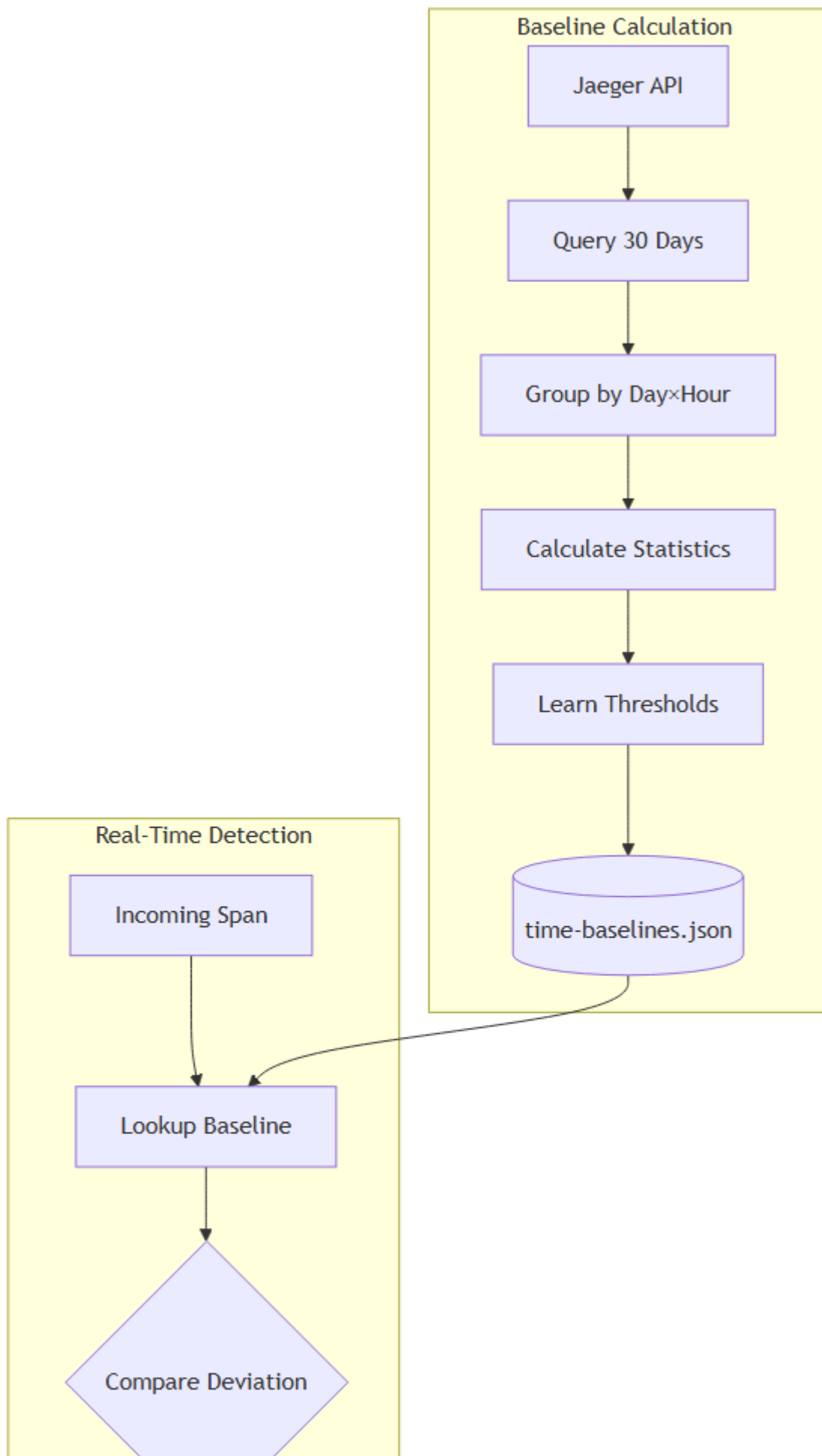


Figure: Trace flow and context propagation (Browser → Gateway → API → RabbitMQ → Worker).



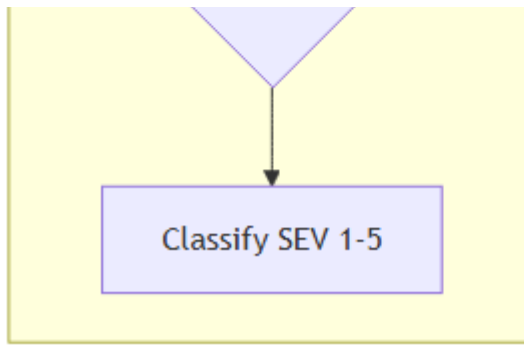


Figure: Nightly baseline calculation and real-time detection pipeline.

## Key achievements

- **End-to-end tracing corrected and unified:**
  - Proper parent/child span propagation for browser → `api-gateway` → `exchange-api` → `RabbitMQ` → `order-matcher` (fixed in `trade-form.tsx`, `rabbitmq-client.ts`, `index.ts`).
- **Trace hierarchy & message correlation:**
  - Injected and passed original parent context via message headers ( `x-parent-traceparent` ) so response processing appears as a sibling span.
- **Monitoring system built:**
  - Baseline profiler, anomaly detector, history store and manual endpoint to `POST /api/monitor/recalculate` .
  - Time-aware baselines (1-hour buckets, 30-day history) and adaptive, percentile-based thresholds turned into five severity levels (SEV1–SEV5).
- **LLM-augmented analysis:**
  - Ollama integrated for AI-assisted trace+metrics analysis; prompts include correlated Prometheus metrics (cpu, mem, P99, etc.).
  - Switched to `llama3.2:1b` for faster responses.
- **Metrics + traces correlation:**
  - Prometheus metrics collection and a metrics correlator service to fetch metrics at anomaly timestamps and include them in AI prompts.

- **UI & UX improvements:**
    - Monitor dashboard: severity filter dropdown, SEV badges and colors, AI Analysis moved above baseline table, visible header buttons, accessibility/contrast and layout refinements.
  - **Infra & ops cleanup:**
    - Docker compose fixes (port conflicts resolved), removed unused Tempo/Grafana, and added Prometheus scrape for `/metrics`.
- 

## Impact / measurable results

- Baselines collected (example): ~22 baselines from ~2.9k spans during initial run.
  - Tracing now shows deeper, correct hierarchy (e.g., `exchange-api:POST` → `publish orders` and sibling `payment_response process`).
  - AI responses became data-driven after including real metrics (reduced hallucination).
- 

## Lessons learned & best practices

- **Explicit context propagation matters:** always set parent context explicitly around async operations (use `context.with` / `trace.setSpan`), especially across await boundaries.
- **Message brokers require intentional header propagation:** to maintain meaningful trace relationships across queue-based patterns, send the original parent trace context explicitly.
- **Time-awareness is essential:** 1-hour time buckets with 30-day history dramatically reduce false positives compared to single global baselines.
- **Adaptive thresholds beat fixed  $\sigma$ :** calculating percentile breakpoints (p95/p99/p99.9) nightly yields meaningful SEV boundaries per-span.
- **LLMs must be fed precise, validated context:** include raw metrics in prompts and log the prompt payload to detect hallucination quickly.
- **UX affects adoption:** place analysis near selection (AI Analysis above baselines), show severity badges, and ensure visible controls and contrast.

- **Incremental & test-first rollout:** start with polling-based detection, a manual recalculation endpoint, and iterate to nightly automation and streaming later.
  - **Performance tradeoffs:** smaller LLM models and quantized variants give big latency/throughput wins; keep a tiered/streaming plan for heavy usage.
  - **Keep debug logs brief during validation:** remove verbose logs after verifying behavior.
  - **Document tracing touchpoints & config:** include clear documentation so others can reproduce the same trace semantics.
- 

## Part 1 — How to replicate the tracing behavior ✨

Goal: achieve the same, clear trace hierarchy and reliable sibling/child relationships for operations that cross HTTP and message queues.

### Key concepts (short)

- Context is not magically preserved across async boundaries — explicitly preserve and use it.
- When sending messages across message brokers, **inject** a trace context into message headers so the consumer can link spans correctly.
- For response messages that should appear as siblings (not children) of an original request, **carry the original parent span context** and use that as the parent when creating the response consumer span.

### Touchpoints and where the logic lives in the repo

- Client-side: `client/src/lib/tracing.ts` and components like `client/src/components/trade-form.tsx` — create client spans and preserve context across await boundaries when processing responses.
- RabbitMQ client (producer): `server/services/rabbitmq-client.ts` — create publish spans with the correct parent context and inject trace headers into message properties (e.g. `headers['traceparent']` or a custom `x-parent-traceparent`).

- Worker / matcher (consumer): `payment-processor/index.ts` — extract the parent context from message headers; when responding, inject the original POST context into the response so the consumer's processing span becomes a sibling of `publish orders`.
- Auto-instrumentation: be mindful that `amqplib`/auto-instrumentation will create spans from extracted trace headers; decide whether to rely on it or implement explicit spans.

## Minimal code patterns (Node / OpenTelemetry)

- Create a span and run code in its context:

```
const parentCtx = context.active();
const span = tracer.startSpan('order.submit');
await context.with(trace.setSpan(parentCtx, span), async () =>
  // your async code here (fetch, publish, etc.)
  // inject context for outgoing messages
  propagation.inject(context.active(), carrier, defaultTextMapSetter);
span.end();
```

- Inject parent trace into message headers (producer):

```
const carrier = { headers: {} };
propagation.inject(context.active(), carrier.headers, defaultTextMapSetter);
// Also include original POST context explicitly as x-parent-traceparent
carrier.headers['x-parent-traceparent'] = getTraceparentFromContext();
// publish message with 'carrier.headers'
```

- Extract context on consumer and use the original POST context in responses:

```
const headers = msg.properties.headers || {};
const parentCtx = propagation.extract(ROOT_CONTEXT, headers, defaultTextMapGetter);
const originalParent = headers['x-parent-traceparent'];
// For response: set the original parent context so response spans are siblings
```

```
const respCtx = reconstructContextFromTraceparent(originalParent)
const responseSpan = tracer.startSpan('payment_response process')
```

Tip: use consistent header naming ( `x-parent-traceparent` ) so you can find and debug propagation easily in logs.

## Verification steps

1. Start the app and Jaeger: UI at `http://localhost:5173` and Jaeger at `http://localhost:16686`.
2. Submit a BUY order via UI or curl:

```
curl -X POST http://localhost:8000/api/orders -H 'Content-Type: application/json'
```

3. In Jaeger you should see:

- `exchange-api: POST` as root
- Child `exchange-api: publish orders` → down to `order-matcher: order.match`
- A sibling `exchange-api: payment_response process` that is a child of the original POST.

4. If a span is missing or orphaned, check the message headers printed in logs (we added debug logs in `server/services/rabbitmq-client.ts` and `payment-processor/index.ts`).

---

## Part 2 — Anomaly Detection Design (Time-aware + Adaptive thresholds) 🌀

This section explains the architecture we implemented, why we chose it, and how to reproduce or adapt it to your data.

### Motivation

- Latency and span durations vary by time-of-day and day-of-week.

- A single global mean/std-dev produces many false positives.
- We need a system that (1) captures time patterns, (2) learns what constitutes “rare” deviations, and (3) exposes severity tiers for alerting.

## High-level architecture

1. TraceProfiler polls Jaeger and stores span duration samples (per span operation) for the last 30 days.
2. BaselineCalculator groups samples into **time buckets** (dayOfWeek × hourOfDay) and computes incremental stats (count, sum, sum-of-squares → mean/std).
3. Nightly job computes percentiles (p95, p99, p99.9) of historical deviations and derives severity thresholds (SEV1–SEV5).
4. AnomalyDetector uses the best-available baseline (specific bucket → hourly fallback → global) and computes deviation in  $\sigma$  units:  $\text{deviation} = (\text{value} - \text{mean}) / \sigma$ .
5. If deviation exceeds severity thresholds, create an anomaly with associated metadata and store it in HistoryStore.
6. On anomaly selection, the AnalysisService fetches correlated Prometheus metrics and calls Ollama for insights.

## Storage & shapes (SQLite-style)

```
TimeBaseline { spanKey, dayOfWeek, hourOfDay, count, mean, std
Thresholds { spanKey, dayOfWeek, hourOfDay, p95, p99, p999, sev
Anomaly { id, spanKey, duration, deviationSigma, sev, traceId,
```

## Key details & decisions

- Granularity: 1-hour buckets ( $24 \times 7 = 168$  per span)
- History depth: 30 days (configurable)
- Recalc mode: nightly job + manual endpoint `POST /api/monitor/recalculate` for validation and ad-hoc re-runs
- Adaptive severity mapping: By default we map percentiles to SEV levels like this:
  - SEV1: > p999 (extreme, top 0.1%)
  - SEV2: > p99 (critical)



- SEV3: > p95 (major)
- SEV4: > p90 (minor)
- SEV5: > p80 (low)

This mapping is configurable and done during the nightly baseline recalculation.

## Algorithm (nightly recalculation)

1. For each spanKey and each hour bucket:
  - Query last 30 days of duration samples for that span & bucket
  - Compute mean  $\mu$ , std  $\sigma$ , and percentiles of deviations
  - Store (mean,  $\sigma$ , p95, p99, p999)
2. Create SEV threshold map from percentile results, persist for AnomalyDetector to use.

## Real-time detection (per incoming span sample)

1. Identify bucket: dayOfWeek + hourOfDay of sample timestamp
2. Lookup baseline (if sampleCount >= MIN\_SAMPLES, use it; else fallback to hourly or global baseline)
3. Compute deviation:  $z = (\text{duration} - \mu) / \sigma$
4. Map  $z$  to severity using stored thresholds
5. Create anomaly if  $z$  crosses configured SEV threshold

## Example code sketch (detection)

```
const baseline = getBaseline(spanKey, day, hour) || fallbackBa
const z = (observed - baseline.mean) / baseline.std;
const sev = classifyByThresholds(z, thresholdsFor(spanKey, day,
if (sev <= configuredAlertLevel) createAnomaly(...);
```

## Severity classification (SEV1–SEV5)

Make severity explicit in UI and storage and show the percentile and  $\sigma$  that triggered it. This allows users to filter and triage efficiently (we added a severity dropdown to the UI).

## Why not ML-first? (and when to use ML)

- The incremental stats approach (mean/std/percentiles) is simple, explainable, and works with limited historical data.
  - ML methods (one-class SVM, isolation forest, autoencoder, or LoRA-finetuned models that summarize traces) can be applied later when you have labeled anomalies and richer feature vectors.
  - For now, prefer deterministic, inspectable thresholds with nightly recalculation and anomaly history.
- 

## Part 3 — LLM integration (operational tips)

- Include correlated metrics (Prometheus) in the prompt when analyzing an anomaly — CPU, memory, request rate, error rate, P95/P99 latency.
- Keep prompts bounded (summarize long traces) and log prompt payloads during testing to avoid hallucinations.
- Use a smaller model (we switched to `llama3.2:1b`) for production-like responsiveness; keep the larger model for ad-hoc deep analysis.
- Add timeouts and retries to Ollama calls (we added both) and a circuit-breaker for model overload.

Sample prompt structure:

```
- Short summary of anomaly (service, span, duration, deviation)
- Recent trace (key spans and durations, up to a length limit)
- Correlated metrics snapshot (CPU, memory, P99, request-rate,
- Ask the model for top-3 probable causes and recommended next
```

---

## Part 4 — Tests & operational checklist

- ☐ Start services: `docker-compose up -d` and `npm run dev` (server + vite)

- ☐ Verify Jaeger traces: Visit `http://localhost:16686` and search traces for `exchange-api` service
- ☐ Submit order via UI or curl and verify: publish spans exist and response spans are siblings
- ☐ Run baseline recalculation: `POST /api/monitor/recalculate` and check SEV thresholds stored
- ☐ Trigger anomaly (simulate slow handler) and confirm UI receives an alert with SEV badge
- ☐ Select anomaly → click `Analyze` → check model response mentions metrics and suggests actions

## Part 5 — Next steps & roadmap

- Automate the nightly baseline job via cron/worker (server side) and alert on job failures.
- Add a small finite buffer/summary extractor on trace content before sending to LLM (reduce prompt size and cost).
- Create a labeled dataset and start LoRA fine-tuning for domain-specific insights (200–500 examples to start).
- Add metric historical charts for anomalies and a service dependency map for root-cause suggestions.

## Appendix — Quick commands and endpoints

- Submit an order (example):

```
curl -X POST http://localhost:8000/api/orders -H 'Conter
```

- Recalculate baselines (manual):

```
curl -X POST http://localhost:5000/api/monitor/recalculate
```

- Analyze an anomaly (trigger model):

```
curl -X POST http://localhost:5000/api/monitor/analyze -
```

---

## Export to HTML / PDF (one-command)

To generate the rendered HTML and a printable PDF of this document (including diagrams):

```
# Renders diagrams and builds HTML + PDF
npm run docs:build
```

Outputs:

- `docs/traces-and-anomaly.html` (HTML)
  - `docs/traces-and-anomaly.pdf` (PDF)
- 

If you'd like, I can now:

- Generate a short `docs/README_TRACING.md` version suitable for on-call staff,
- Add a checklist `MONITORING_RUNBOOK.md` for incident responders,
- Or create the Pull Request with these docs and update the main `README.md`.

Which would you like me to do next? 🙏

---

Generated from `docs/TRACES-AND-ANOMALY-MONITORING.md`