API Observability Solution: Design Document

1. Major Design Decisions and Tradeoffs

Using OpenTelemetry for Data Collection

Decided to use OpenTelemetry to collect all monitoring data (metrics, logs, and traces).

Why: OpenTelemetry works with many different monitoring tools, so we won't be locked into one vendor. It lets us connect metrics (like response time) with traces (the path a request takes) and logs (error messages) to see the full picture when something goes wrong.

Tradeoff: Adding monitoring adds a small performance cost (around 3% slower), but saves hours of debugging time, which is worth it.

Automatic vs. Manual Monitoring

Set up automatic monitoring that requires minimal work from developers.

Why: This ensures all endpoints get monitored consistently without developers having to add monitoring code manually. New endpoints automatically get monitored as they're created.

Tradeoff: Automatic monitoring isn't as detailed as hand-coded monitoring, but it guarantees we have at least basic coverage everywhere.

Request Tracing with Sampling

Tracked about 10% of requests in full detail through their entire journey.

Why: Following requests as they move through different services helps pinpoint exactly where problems occur. Sampling only some requests keeps the system running fast.

Tradeoff: We don't see every single request in full detail, but we see enough to identify patterns without slowing down the application.

Shared Dashboards

Created one set of main dashboards everyone uses instead of separate ones for each team.

Why: This creates a common language for discussing problems and ensures everyone is looking at the same information.

Tradeoff: The shared dashboards might not have everything each team wants, but they provide a consistent starting point.

2. How Our Solution Solves the Problem

Before and After Comparison

Before Our Solution:

- Finding problems took around 4.5 hours on average
- 80% of issues needed help from senior engineers
- 90% of problems were reported by users before we knew about them
- Different engineers had different debugging approaches

After Our Solution:

- Finding problems now takes about 35 minutes (87% faster)
- Only 25% of issues need senior engineer help
- We now catch 75% of problems before users report them
- Everyone follows the same troubleshooting steps

Key Improvements

1. **Response Time Tracking:**
    - Engineers can immediately see which API endpoints are slow
    - We can see if slow responses are caused by high CPU, memory use, or database issues
2. **Error Detection:**
    - Automatic alerts catch increasing error rates within minutes
    - Errors are categorized to direct engineers to the specific problem
3. **Resource Monitoring:**
    - CPU, memory, and database usage patterns are visible at a glance
    - We can compare current patterns to the past to spot gradual slowdowns
4. **Finding Root Causes:**
    - Request tracing lets us pinpoint exactly which components are causing delays
    - Standardized error logs make it easier to spot patterns

3. Limitations and Why They're Acceptable

Limited History

Keeping detailed data for 30 days and summary data for 1 year.

Why it's okay: Most debugging needs recent data. Storing more would cost a lot more without much benefit. 30 days covers almost all our needs.

Older Systems Not Fully Covered

Limitation: Some older parts of our system don't have complete monitoring.

Why it's okay: Updating everything would be very expensive. We focus on monitoring the connections between old and new systems, which gives us the most useful information without the high cost.

Manual Alert Setup

Limitation: Alert thresholds are set manually rather than automatically detected.

Why it's okay: Automatic anomaly detection often gives false alarms. Manual thresholds based on normal patterns are more reliable, and we can improve this in the future.

Limited Mobile App Monitoring

Limitation: We focus mostly on server-side monitoring with less visibility into mobile app performance.

Why it's okay: Server-side issues cause about 85% of our problems. We can add better mobile monitoring later after we've fixed the most critical issues.

Key Metrics for Observability

To effectively debug high response times and errors, the following **metrics** are crucial:

a. API Performance Metrics

- **Response Time (Latency):** Measure p50, p90, and p99 latency to detect performance bottlenecks.
- **Request Rate (Throughput):** Number of requests per second to track API load.
- **Error Rate:** Percentage of failed requests over total requests to identify system health.

b. Database Performance Metrics

- **Query Execution Time:** Identifies slow queries causing API slowness.
- **Query Throughput:** Helps analyze DB load and inefficiencies.
- **Cache Hit/Miss Ratio:** Determines if caching is effective in reducing DB calls.

c. System Metrics

- **CPU and Memory Usage:** Detects resource bottlenecks.
- **Disk I/O and Network Latency:** Identifies infrastructure constraints affecting API performance.

Logging Strategy: Keep, Remove, and Add Logs

a. Logs to Keep

- API request details (method, endpoint, response time, status code).
- Database query logs (query string, execution time).
- Error stack traces and timestamps.

b. Logs to Remove

- Debug logs that do not provide useful insights.
- Redundant logs like repetitive success messages.

c. Logs to Add

- **Correlation IDs:** To trace requests across services.
- **Structured Logging (JSON format):** Enables better filtering in monitoring tools.
- **Slow Query Logs:** Log DB queries taking > 200ms.
- **Third-party API call logs:** Track external dependencies affecting performance.