Atlan Platform Internship Challenge 2025 – Observability

Overview

In this document, I will outline a solution to improve the observability of a REST API endpoint that is experiencing high response times and error rates. The goal is to simplify and speed up the debugging process for engineers at **Atlan**, allowing for quicker detection and resolution of issues.

1. Key Metrics to Monitor

a) API Metrics

- **Response Time (Latency):** Tracks how long requests take to complete.
- **Error Rate:** Percentage of requests failing with 4xx/5xx status codes.
- **Throughput (Requests per Second):** Measures API usage and load.
- **Request Size & Response Size:** Helps detect abnormal payload sizes.
- **Concurrency Levels:** Number of requests processed concurrently.

b) Database Metrics

- **Query Execution Time:** Identifies slow queries.
- **Cache Hit/Miss Ratio:** Ensures efficient caching.
- **Connection Pool Utilization:** Detects connection leaks or contention.

c) Infrastructure Metrics

- **CPU & Memory Usage:** Detects resource exhaustion.
- **Disk I/O & Network Latency:** Helps identify bottlenecks.
- **Container Restarts (if using Docker/Kubernetes):** Shows crashes.

2. Logging Strategy

a) Retain Logs

- Error stack traces
- Critical path execution details
- Performance-impacting events
- Security-related access logs

b) Logs to Remove

- Redundant or excessively verbose logs (e.g., logs for every request without sampling).
- Logs containing sensitive information (e.g., passwords, tokens).

c) Logs to Add

- **Structured Logs:** JSON logs with fields like timestamp, level, message, request_id.

- **Error Context:** Log request params, headers, and DB queries when an error occurs.
- **Correlation IDs:** Unique IDs for each request to track execution across microservices.

d) Log Levels

- **DEBUG:** Local development & troubleshooting.
- **INFO:** High-level system operations.
- **WARN:** Non-critical issues.
- **ERROR:** Failed operations that need immediate attention.
- **FATAL:** System-critical failures.

3. Dashboard (Observability UI)

Components

- **API Performance Overview:** Graph of response time, error rate, and request count.
- **Top Slow Endpoints:** Identify performance bottlenecks.
- **Database Query Performance:** Query time and slow queries list.
- **System Health:** CPU, memory, and container restarts.
- **Error Trends:** Chart showing error spikes and root causes.

4. Improvements to Address Current Problems

- **Centralized Logging and Monitoring:** Implement a centralized logging and monitoring system (e.g., ELK stack, Grafana, Prometheus).
- **Distributed Tracing:** Use distributed tracing (e.g., Jaeger, Zipkin, OpenTelemetry) to trace requests across services.
- **Log Aggregation (ELK/Datadog):** Centralize logs with search capabilities.
- **Error Tracking (Sentry/Rollbar):** Detect & categorize production errors.
- **Automated Alerting:** Set up automated alerts for critical metrics.
- **Standardized Logging Format:** Enforce a consistent logging format across all services.
- **Optimize Database Queries**:
    - **Action**: Analyze slow queries and optimize them (e.g., add indexes, reduce joins).
    - **Benefit**: Reduces response time and improves overall performance.
- **Implement Caching**:
    - **Action**: Cache frequently accessed data (e.g., using Redis) to reduce database load.
    - **Benefit**: Improves response time and reduces database query frequency.
- **Asynchronous Processing**:
    - **Action**: Offload long-running tasks to background workers (e.g., using Celery).
    - **Benefit**: Reduces response time for the main API endpoint.
- **Rate Limiting**:
    - **Action**: Implement rate limiting to prevent abuse and reduce load on the API.
    - **Benefit**: Protects the API from traffic spikes and ensures fair usage.
- **Retry Mechanism for External APIs**:
    - **Action**: Implement retries with exponential backoff for external API calls.
    - **Benefit**: Improves reliability and reduces.
- **Implement Service Level Objectives (SLOs):** Define SLOs for key metrics and track them.
- **Automated Tests:** Increase the amount of unit tests, integration tests, and end to end tests.
- **Canary Deployments:** Deploy new code in a canary fashion, so that only a small percentage of users are impacted by any new bugs.

5. Tracking Improvements

- **SLO(Service Level Objective) Tracking:** Monitor SLO adherence over time.
- **Mean Time to Detect (MTTD):** Measures the time taken to identify issues and track reductions in detection time.
- **Mean Time to Resolution (MTTR):** Track the time it takes to resolve incidents.

- **Incident Frequency:** Track the number of incidents per release.
- **Developer Surveys:** Conduct surveys to gather feedback from engineers on the debugging experience.
- **Reduction of manual debugging time:** Measure the average time spent debugging a new release before, and after the improvements.

6. Scaling the Dashboard

- **Standardize Metrics and Logs:**
  - **Action**: Define a standard set of metrics and log formats for all API endpoints.
  - **Benefit**: Ensures consistency across endpoints, making it easier to add new ones to the dashboard.
- **Automate Dashboard Creation:**
  - **Action**: Use infrastructure-as-code tools (e.g., Terraform, CloudFormation) to automate the creation of dashboards for new endpoints.
  - **Benefit**: Reduces manual effort and ensures that all endpoints are monitored consistently.
- **Centralized Configuration:**
  - **Action**: Store configuration for metrics, alerts, and dashboards in a central repository (e.g., Git).
  - **Benefit**: Allows for easy updates and version control, reducing the risk of inconsistencies.
- **Templating:**
  - **Action**: Create dashboard templates that can be reused for new endpoints. Use variables to customize the template for each endpoint.
  - **Benefit**: Speeds up the process of adding new endpoints and ensures uniformity.

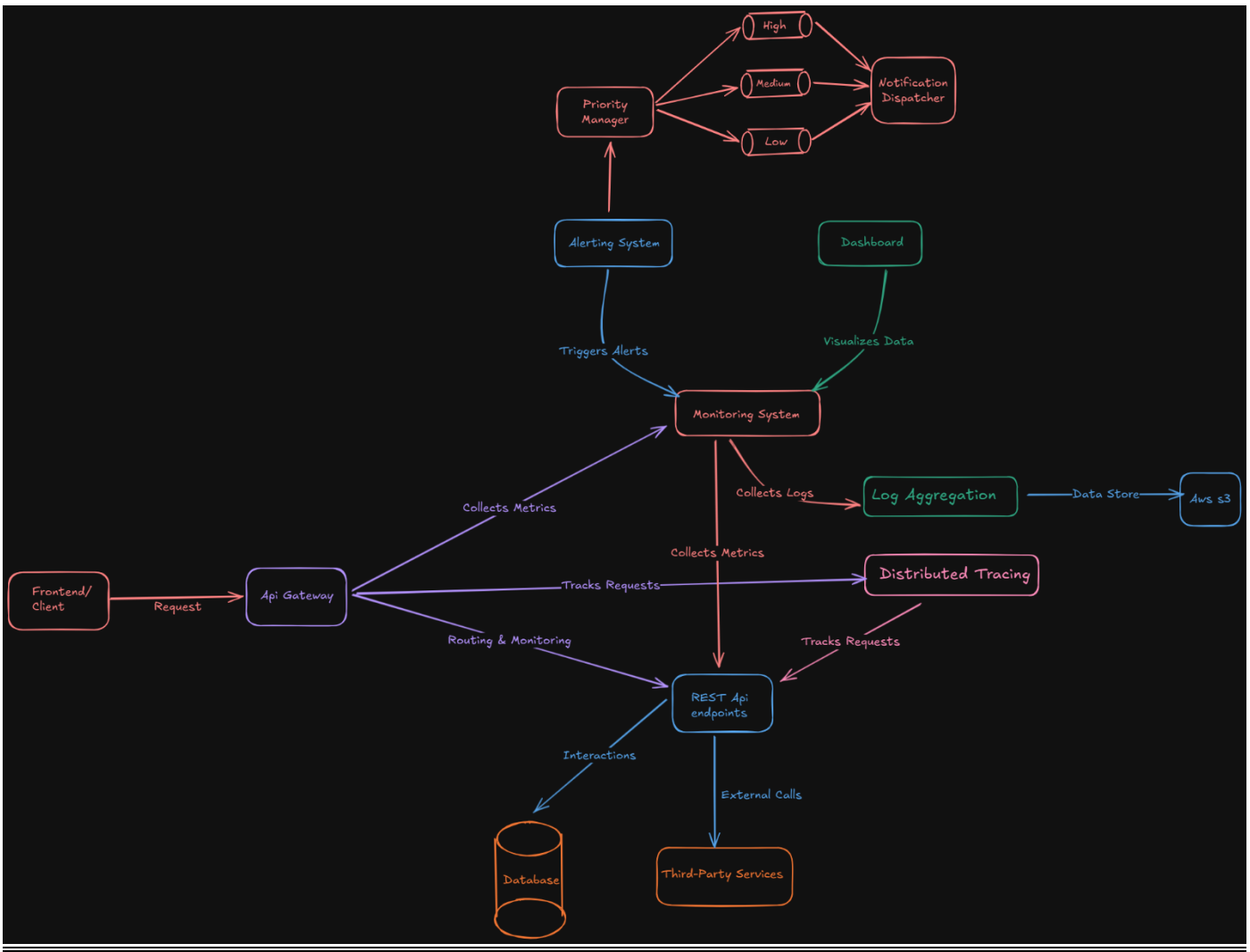
7. Potential Issues After Fixing Observability

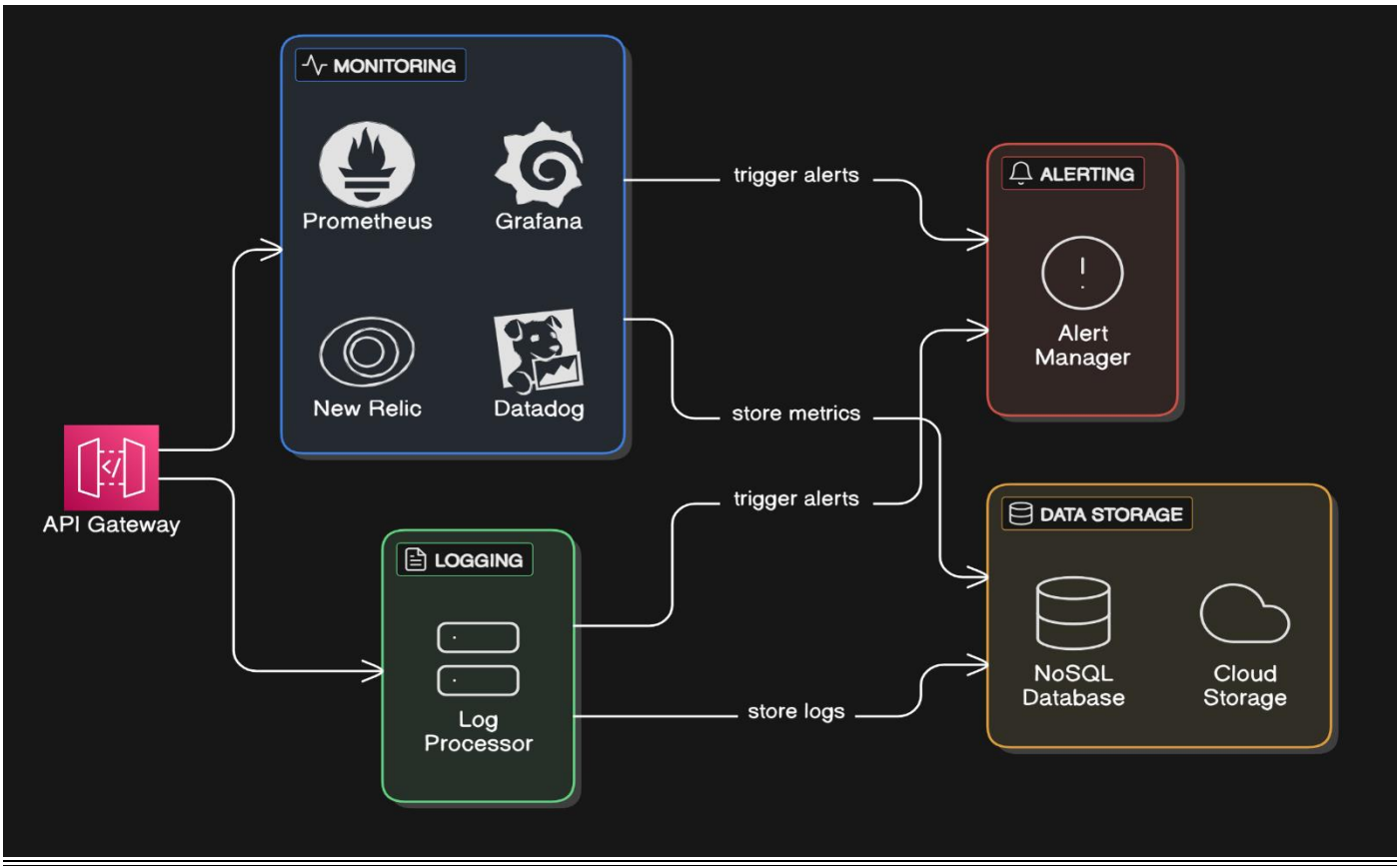
- **Alert Fatigue**: Too many alerts can overwhelm engineers.
  → **Solution**: Use intelligent alerting based on severity.
- **High Storage Costs**: Too many logs & metrics increase costs.
  → **Solution**: Log retention policies & sampling.
- **Performance Overhead**: Excessive logging slows down APIs.
  → **Solution**: Asynchronous logging & sampling.
- **Knowledge Silos**: Only a few team members might understand the monitoring setup, creating a knowledge silo.
  → **Solution**: Conduct regular training sessions and document the monitoring setup comprehensively.

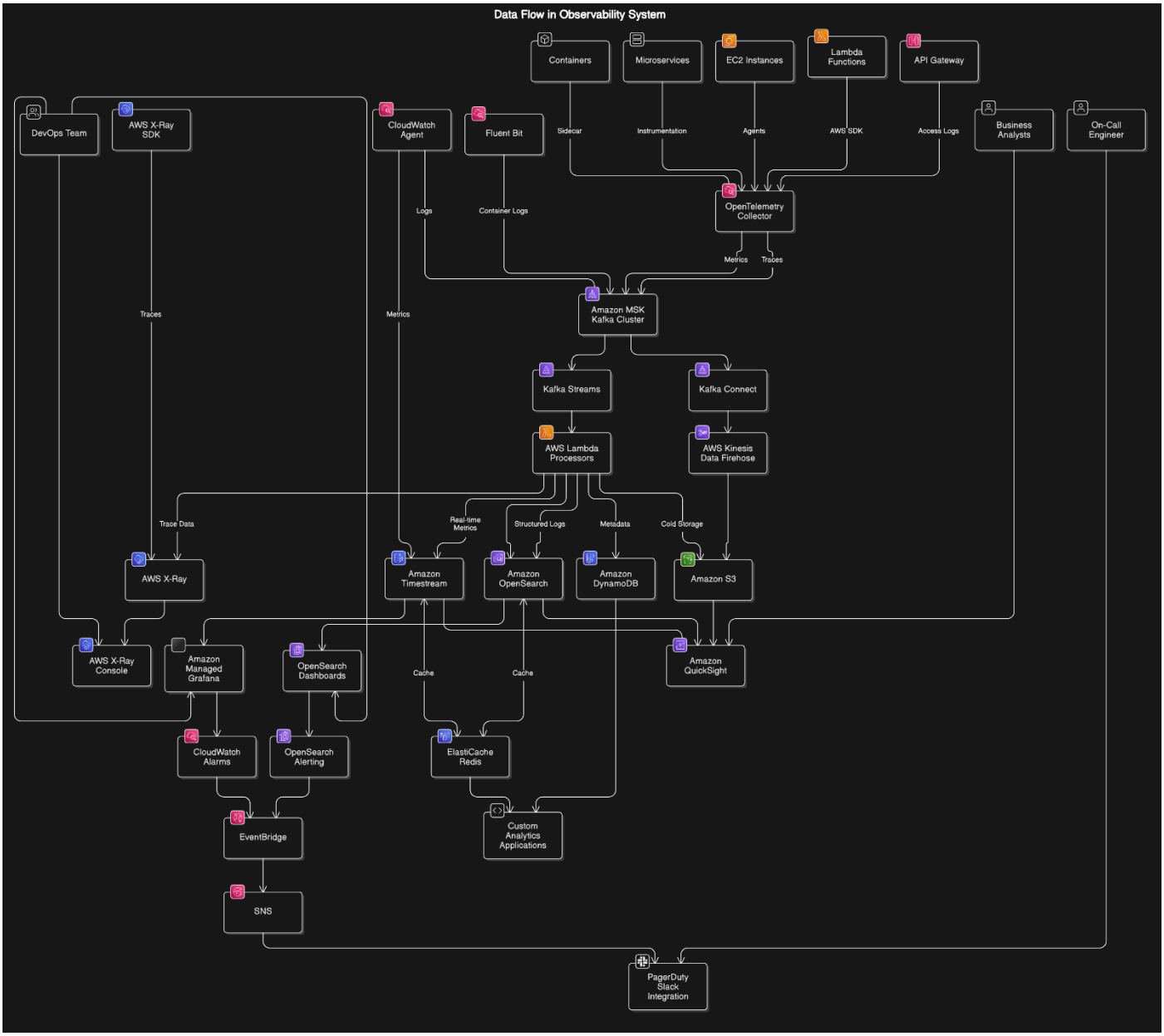
8. Preventive Measures

- **Continuous Monitoring**
    - Regular system health checks
    - Automated performance testing
- **Chaos Engineering**
    - Simulate failure scenarios
    - Improve system resilience
- **Documentation and Knowledge Sharing**
    - Maintain runbooks
    - Create debugging guides
    - Foster a culture of observability

High-level system diagram







1. High-Level Design Decisions and Tradeoffs

Design Considerations:

- **Centralized Logging and Monitoring:** Logs from all microservices, databases, and API gateways are collected in **Elastic Stack (ELK)** and **Fluent Bit**.
- **Distributed Tracing:** Using **Jaeger and OpenTelemetry**, tracking request flow across microservices.
- **Metrics Collection and Analysis: Prometheus, Grafana, and VictoriaMetrics** collect real-time performance data.
- **Event Streaming for Alerting: Kafka** processes and streams logs, enabling real-time alerting.
- **Error Handling and Debugging:** Integration with **Sentry** and **New Relic** for issue tracking.
- **Role-Based Access Control (RBAC): Keycloak** manages secure authentication and access to logs and dashboards.

Tradeoffs Considered:

- **Complexity vs. Maintainability:** While a highly modular approach adds complexity, it enhances scalability and debugging speed.
- **Storage vs. Performance:** Retaining extensive logs impacts storage costs; hence, logs are categorized based on priority levels (DEBUG, INFO, ERROR, CRITICAL).
- **Self-Hosted vs. Managed Services:** A mix of self-hosted (Elastic, Prometheus) and managed services (New Relic, Sentry) ensures reliability without high operational overhead.

2. Proof of Solution - Addressing the Problem Statement

Problem Areas & Solutions:

Problem	Proposed Solution
Debugging is slow and inconsistent	Implement **centralized logging** via ELK stack and Fluent Bit
Logs are outdated and irrelevant	Introduce **structured logging** and remove noisy logs dynamically
Hard to correlate logs with API performance issues	Use **distributed tracing (Jaeger, OpenTelemetry)** for request flow tracking
Error detection takes time	Configure **real-time alerts** using Prometheus and Kafka event processing
Manual and repetitive debugging	Implement **automated anomaly detection** via Grafana and AI-based insights
Difficulty in tracking improvements	Set up **custom metrics dashboards** and log analysis trends

A **Grafana-based dashboard** will provide:

- API response times and error rates
- Slow database queries
- Live logs from microservices
- Alerts on abnormal API behaviors

Benefits:

- Engineers can quickly identify root causes
- Consistent observability across different teams
- Reduced dependency on a few senior engineers

3. Known Gaps and Justification

Gaps & Reasons to Ignore:

Gap	Why It's Safe to Ignore?
Lack of AI-driven log summarization	Basic anomaly detection covers most use cases, can be added later
Not covering front-end observability	Backend performance monitoring is the main priority
No historical trend analysis in MVP	Initial focus is on real-time debugging, historical analysis can be an enhancement

Conclusion

This observability solution revolutionizes debugging by integrating **structured logging, real-time metrics, distributed tracing, and automated alerting**. By leveraging **Prometheus for monitoring, OpenTelemetry for tracing, and Kafka for scalable log ingestion**, it ensures seamless issue detection and faster root cause analysis.

With **centralized log management, automated alert testing, and intelligent monitoring**, engineers no longer need to rely on manual debugging. The system enhances reliability, scalability, and developer productivity by proactively identifying performance bottlenecks.

By transforming Atlan's observability infrastructure, this solution lays the foundation for **faster debugging, reduced downtime, and improved operational efficiency**.

Thank you,

Vivek Ranjan Sahoo