International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Exploring Observability Design Patterns of Microservices: Challenges and Solutions

AzraJabeen Mohamed Ali

Independent researcher, California, USA
Azra.jbn@gmail.com

Abstract:
This paper discusses the thorough exploration of the Observability design patterns associated with the Microservices. Microservices have transformed the software development sector by encouraging modularity, scalability, and maintainability, which enables businesses to react to shifting consumer needs and technology breakthroughs faster. The study's main research question explores the careful consideration of Observability design in microservice architecture due to its distributed nature of microservices. It also provides a thorough analysis of several microservice's Observability patterns and the way it handles its own data for improved scalability, flexibility, and isolation. This paper is therefore meant to be more development-environment centered and infrastructure agnostic. Developers and architects who wish to concentrate on code, patterns, and implementation specifics will find this part most interesting.

Keywords: Micro Services, Design patterns, Observability design patten, monolithic, Log aggregation, Distributed Tracing, Health Check

1. Introduction

Microservice Architecture:

Microservice architecture is a design methodology that divides a large application into smaller, autonomous services, each of which focuses on a distinct business function. Therefore, the back end is the main focus of this method, even though the front end can also use a microservices design. Each service runs independently and communicates with other processes using protocols including HTTP/HTTPS, WebSockets, and AMQP.

Business-critical enterprise applications need to provide updates fast, frequently, and reliably in order to thrive in today's unstable, uncertain, complex, and ambiguous reality. As a result, corporations are divided into small, cross-functional teams with limited connections. Each team uses DevOps methodologies to deploy software. Specifically, it makes use of continuous deployment. An automated deployment pipeline tests the team's stream of frequent, small modifications before they are put into production. The intention is to allow developers to leverage microservices to speed up application releases by allowing teams to deploy each microservice as needed.

Why are Microservices Architectures used by Businesses?

Most firms start by constructing their infrastructures as a collection of closely related monolithic applications or as a single monolith. The monolith does a number of things. All of the programming for those functionalities is included in a single, cohesive piece of application code. Because the code for these

functions is so intertwined, it is difficult to understand. The code of an entire program may break as a result of a single feature addition or alteration in a monolith. This makes any change, no matter how simple, expensive and time-consuming. As upgrades are done, programming becomes more complicated until scaling and upgrading are practically impossible.

Businesses can no longer make additional changes to their coding over time without starting over. Businesses may find themselves stuck with antiquated procedures for a long time after they should have modernized, as the process soon becomes too difficult to handle.

In addition to other pertinent factors, company objectives will determine which pattern (or patterns) is best to use. For microservices, there are numerous design patterns, each with its own advantages and disadvantages. Design patterns are grouped according to their intended use like Decompose patterns, Observability patterns, Integration patterns, Database patterns, Cross-Cutting concern patterns. Observability design patterns, including Log Aggregation, Performance Metrics, Distributed Tracing, Health check are the main topic of this article.

Observability design pattern:

The term "observability design pattern" describes a collection of procedures, resources, and frameworks that offer insights into how a system, application, or infrastructure functions inside. Observability is intended to help teams monitor a system's performance, dependability, and health while also helping them comprehend how the system acts in production. It is crucial for determining the underlying causes of problems, diagnosing them, and enhancing system performance.

a. Log Aggregation:

Logging:

Logging is a critical aspect of software development and operations, providing a detailed record of system events, errors, and other important activities. Logs offer insights into the behavior of applications, helping developers, operators, and security teams to troubleshoot, monitor, and improve systems. Logs can be stored in several ways, and it is important to choose the right storage solution based on the system's requirements. In monolithic application, Logs are written to local files on the machine or server where the application runs. Logging to a flat file on a single computer is much less useful in a cloud setting. Due to containers moving between physical computers, the local disk may be extremely ephemeral or applications generating logs may not have access to it. Sometimes it can be difficult to find the right file-based log file, even when monolithic applications are simply scaled up across numerous nodes.

Challenge:

An integral component of efficient troubleshooting is logging. It is a continuous record of all events that happen in an application. Logging captures information about events, errors, and changes. Logging in a microservices architecture can be challenging since log entries are dispersed over several services, each with its own log file. Even worse, service instances may be ephemeral, meaning that service instance only lasts for a short time. So the log files are lost when they terminate.

Solution:

Log aggregation as a microservices design pattern turns out to be a solution for these problems. It stores logs from various microservices on a centralized platform after normalizing and combining them. A centralized log service that compiles log files from each service instance is used in log aggregation. It easy to retrieve, visualize, and analyze logs through the centralized log service.

Implementation of Log Aggregation:

An application usually logs many types like

Application logs: which capture events within the application (For instance, successful API requests, database queries, errors, or exceptions. Application logs help developers troubleshoot problems, track behavior, and improve performance),

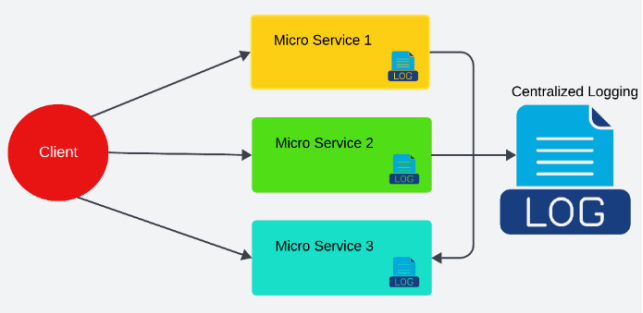
System logs: which are generated by the operating system or server infrastructure and provide insights into the health and performance of the system, such as disk space, memory usage, or hardware errors,

Access logs: which capture HTTP requests in web applications or other types of request-response cycles. They include details like the requested URL, the HTTP method, response status code, IP address, and sometimes user-agent details.,

Security logs: which track authentication, authorization, and access control events. They are important for monitoring suspicious activity, failed login attempts, unauthorized access, and other security-related events.

A good approach of application's design encourages to log frequently, due to which log level cam be increased when logs are gathered. For cloud-native apps, centralized logs are desirable due to the difficulties of utilizing file-based logs. Fig-1 Applications gather logs, which are then sent to a central logging program for indexing and storing. Every day, this type of machine can consume tens of gigabytes of logs. The logs are stored on the log server, which also handles their aggregation, storage, and searchability. Teams may retrieve, view, and analyze logs using the central logging server. They can also set up alerts to be sent out when particular messages or patterns show up in the logs.

Fig-1



b. Performance Metrics:

Performance metrics, together with logs and traces, are one of the core pillars of the Observability Design Pattern. Metrics give teams a numerical assessment of the system's performance and health, enabling them to watch and observe system activity in real time. In order to identify performance problems, guarantee system dependability, and enhance user experiences, these metrics are essential.

Challenge:

Keeping an eye on transactions is essential when the microservice architecture expands the service portfolio. This allows for the monitoring of patterns and the sending of notifications in the event of a problem. How should metrics be gathered to track the performance of an application?

Solution:

The performance metrics pattern allows us to collect and aggregate information about certain processes (such CPU performance and latency). The pattern combines metrics from several services into a single metrics service with the ability to report and modify.

Key Performance Metrics:

Performance metrics are numerical values that capture various aspects of a system's performance. These can include things like latency, throughput, error rates, resource usage, and more. Metrics provide insights into system behavior, performance, and bottlenecks, helping teams make data-driven decisions. The Key Performance Metrics in the Observability Pattern are

Latency: Refers to the time it takes for a system or service to respond to a request. High latency can indicate performance bottlenecks, which can degrade the user experience. Latency can be measured in milliseconds (ms). API response time and Database query latency are the examples of Latency. By tracking latency, teams can identify slow parts of the system and optimize them.

Throughput(Request Rate): Throughput measures how many operations (e.g., requests, transactions) a system can handle in a given period. Throughput helps measure system capacity. A sudden drop in throughput could indicate performance degradation or service overload. Requests Per Second (RPS) and Transactions Per Second (TPS) are examples of Throughput. Monitoring throughput allows teams to detect load spikes or saturation points, providing insight into whether the system is able to handle the traffic load.

Error Rate: Error rate refers to the percentage or count of failed requests or operations compared to the total number of requests or operations. A high error rate can signal system instability, bugs, or operational issues, which can negatively affect the user experience. HTTP 4xx and 5xx error codes and Service Failure Rate are the examples of Error Rate. By tracking error rates, teams can identify failing services, problematic endpoints, or issues with user-facing functionality.

Availability (Uptime): Availability refers to the proportion of time a service or system is operational and able to handle requests. Downtime or service unavailability can significantly impact user experience and business operations. Monitoring availability ensures systems are up and running. Service uptime percentage and Availability per region are the examples of Availability. Availability metrics can trigger alerts when services go down, enabling teams to take quick action to restore service.

CPU and Memory Usage: These metrics measure the amount of computational resources (CPU) and memory (RAM) used by a service or system. High CPU or memory usage can lead to degraded performance, increased latency, or even system crashes. It’s important to monitor these resources to ensure the system operates within acceptable limits. CPU utilization And Memory Consumption are the examples of CPU and Memory usage. These metrics help in capacity planning and performance optimization, as well as identifying resource constraints.

Disk I/O and Network I/O: These metrics capture the rate of data being read from or written to disk (Disk I/O) and the rate of data being sent or received over the network (Network I/O). Disk and network I/O are critical for systems that rely on heavy data storage or communication with external services. Monitoring these metrics helps identify bottlenecks or slowdowns caused by disk or network constraints. Disk read/write operations and Network throughput are the examples of Disk I/O and Network I/O. These metrics are valuable for detecting resource saturation, disk failures, or network congestion.

Queue Length / Backlog: Queue length refers to the number of requests or tasks waiting to be processed by a system, typically in a message queue or job queue. A growing backlog can indicate that the system is overwhelmed and may be unable to keep up with incoming traffic or workload. This can lead to latency spikes and eventual service failure. Pending requests and Job processing Queue Size are the examples of Queue Length / Backlog. Monitoring queue length can alert teams to scaling issues or resource contention.

Saturation: Saturation refers to how close a system is to reaching its maximum capacity for a given

resource (e.g., CPU, memory, disk I/O). If a system becomes saturated, its performance will degrade, and it may fail to process requests. CPU saturation and Memory Saturation are examples of Saturation. Saturation metrics help to predict when the system will reach capacity, allowing teams to take proactive steps to scale the system before it fails.

Implementation of Performance Metrics:

A consolidated view of the performance of microservices architecture is offered by this pattern. In order to gather the necessary data that reveals the system's performance and health, instrumentation must be first set up for different services. There are two models for aggregating metrics from a service:

- **Push:** The service pushes metrics to the metrics service
- **Pull:** The metrics services pull metrics from the service

c. Distributed Tracing:

Challenge:

Requests in microservices architecture frequently span several services. Every service responds to a request by executing one or more tasks across several services. Then, how can we troubleshoot a request by following through from beginning to end?

Solution:

In distributed systems, especially microservices architectures, understanding how requests flow across services is crucial for diagnosing performance issues, identifying bottlenecks, and tracing the root cause of failures. Traditional logging or monitoring might not provide a clear view of how different services interact, but distributed tracing solves this problem by stitching together the request path across all relevant services. Distributed tracing is used to monitor applications that consist of multiple, interconnected services. It helps to track how a request or transaction propagates through different microservices or components in the system. By capturing a trace for each request, distributed tracing provides a clear view of each step and the interactions between services. Distributed Tracing is a key component of the Observability Design Pattern, offering detailed insights into the flow of requests as they travel through different services in a distributed system. It allows teams to track the lifecycle of a request, from its entry point through various services, to understand the latency, performance bottlenecks, and potential failures in microservices architectures.

How Distributed Tracing Works in a Microservices Architecture:

A single user request may go through numerous services and communicate with various elements, such as databases, third-party APIs, or queues, in a distributed microservices system. Distributed tracing operates as follows:

Client Initiates a Request: When a user makes a request, the first service (such as an API gateway) records it. After that, a trace ID is created and attached to the request.

Context Propagation: The trace context (trace ID and span IDs) is spread via the request headers (or any other context-propagating method) as the request moves between services.

Spans are Created: To document its operation, including start and end times and any metadata (such error information), each service that handles the request generates a new span.

Trace Aggregation: A distributed tracing system, like Jaeger, Zipkin, or OpenTelemetry, gathers the trace data, including all spans. The spans are stitched together into a single trace by the system's aggregation and correlation.

Analysis: The trace is stored and visualized in a tracing platform (e.g., Jaeger UI or Zipkin Web Interface),

where developers and operators can view the request journey, identify latency, track errors, and monitor dependencies across services.

d. Health Check:

Challenge:

When using microservices architecture, it is possible for a service to be operational but unable to process transactions. For instance, a recently launched service instance may still be initialized, or a software flaw may have caused a service to freeze without fully crashing. A service that has lost access to its database or whose database is overcrowded and not accepting connections is another example of a service that is operational but not operating correctly. If so, how can we make sure a request doesn't end up in those unsuccessful cases?

Solution:

Implementing health check APIs is the solution to lessen the issues brought on by unhealthy services. In the observability pattern, a Health Check API plays a vital role in ensuring the reliability, availability, and proper functioning of a system. It allows for quick verification of the operational status of services and systems, helping to detect failures early and ensuring that the system is ready to handle requests.

Implementation of Health Check API:

A Health Check API is often a simple endpoint exposed by a service, which returns the health status of that service or system. It is typically used by monitoring tools, load balancers, and orchestrators (like Kubernetes) to perform periodic checks to verify that the service is functioning correctly. By implementing liveness, readiness, and custom health checks, it is ensured that the systems remain resilient, highly available, and responsive to issues.

Purpose of Health Check APIs in Observability:

Service Availability: Ensures that the service is up and running. This is especially critical in distributed systems where multiple services depend on each other.

Proactive Monitoring: Health check endpoints are polled regularly by monitoring systems (e.g., Prometheus, Datadog, AWS CloudWatch) to track the health of a service. If the health check fails, it triggers an alert to the operations team.

Graceful Failover: Health check APIs are used by load balancers and orchestrators to perform graceful failovers. If a service fails the health check, the orchestrator can remove it from the load balancer pool and route traffic to healthy instances.

System Resilience: Health checks are part of creating fault-tolerant and self-healing systems. Services that are deemed unhealthy can be restarted or re-provisioned automatically.

Scaling Decisions: By integrating health checks with auto-scaling tools (such as Kubernetes Horizontal Pod Autoscaler), services can be scaled based on health indicators, such as resource utilization or service response.

Benefits of Observability:

- **Faster Issue Resolution:** By providing real-time insights and detailed context, observability enables faster detection and resolution of issues.
- **Proactive Monitoring:** Instead of waiting for problems to arise, observability allows for proactive management, helping teams detect potential issues before they impact users.
- **Improved User Experience:** By monitoring and maintaining system health, observability helps ensure high availability, low latency, and seamless user experiences.

- **Continuous Improvement:** Continuous observability helps teams identify patterns, optimize performance, and refine their systems over time.

Conclusion:

In summary, the Observability Design Pattern is about creating a system that provides comprehensive visibility into the behavior, health, and performance of applications, particularly in complex distributed systems. It uses metrics, logs, and traces as the core data sources to monitor, troubleshoot, and optimize systems. Observability is a more comprehensive approach that allows teams to deeply understand why and how systems behave, going beyond simple monitoring to support in-depth troubleshooting, system analysis, and performance optimization. Performance metrics are a crucial component of the observability design pattern. They provide essential insights into the health, performance, and scalability of systems and services. By continuously tracking and analyzing metrics like latency, throughput, error rates, and resource usage, teams can detect issues early, optimize performance, ensure availability, and make informed decisions on scaling and resource allocation. The integration of performance metrics with tools for real-time monitoring, alerting, and visualization helps organizations maintain reliable, high-performing systems. By integrating distributed tracing with logging and metrics, teams can create a comprehensive observability strategy, enabling faster root cause analysis, performance optimization, and improved user experiences in modern cloud-native applications. Health Check API ensures that services can be continuously monitored and that any failures are detected early.

References

1. Chris Richardson "Pattern: Log Aggregation" https://microservices.io/patterns/observability/application-logging.html (2019)
2. Chris Richardson "Pattern: Distributed Tracing" https://microservices.io/patterns/observability/distributed-tracing.html (2019)
3. DZone "Microservices Design Patterns: Essential Architecture and Design Guide" https://dzone.com/articles/design-patterns-for-microservices (Jun 06, 2023)
4. Microsoft "Observability patterns" https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/observability-patterns (Apr 02, 2022)
5. Lumigo "What Should You Observe When Deploying Microservices?" https://lumigo.io/microservices-monitoring/microservices-observability/ (2022)
6. Hiren Dhaduk "6 Observability Design Patterns for Microservices Every CTO Should Know" https://www.simform.com/blog/observability-design-patterns-for-microservices/ (Jan 06, 2023)
7. Vinicius Feitosa Pacheco "Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices" Packt publishing (Jan 29, 2018)
8. Sam Newman "Building Microservices: Designing Fine-Grained Systems" O'Reilly (Feb 9, 2022)
9. Angela Davis "An In-Depth Guide to Microservices Design Patterns" https://www.openlegacy.com/blog/microservices-architecture-patterns/ (Dec 23, 2023)
10. Chris Richardson "Microservice Architecture Pattern" https://microservices.io/patterns/microservices.html (2019)