# Navigating Cross Cutting Concern 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 Cross Cutting concern 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 Cross Cutting concern design in microservice architecture due to its distributed nature of microservices. It also provides a thorough analysis of several microservice's Cross Cutting concern design 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, Cross Cutting Concern design patten, monolithic, Circuit Breaker, Blue Green deployment, Externalized Configuration**

## 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.Cross Cuttingconcern design patterns, including Externalized Configuration, Service discovery pattern, Circuit breaker pattern and Blue-Green deployment patternare the main topic of this article.

**Cross Cutting Concern design pattern:**

A section of the system that is separated based on functionality is referred to as a concern.There are two types of concerns, namely Core concerns and Crosscutting concerns. Core concerns refer to the system's primary functionality. For instance, Business Logic. Crosscutting concerns refer to the system’s secondary functionality and it affects the entire application. For example, logging, security and data transfer are the concerns which are needed in almost every module of an application, hence they are cross-cutting concerns.

A cross-cutting concern in microservices architecture is an application feature that impacts several components or services but is not the primary focus of the business logic. Logging, authentication, authorization, monitoring, error handling, and security are a few instances of cross-cutting issues.

**a. Externalized Configuration:**

**Challenge:**

Usually, a service makes calls to databases and other services. The endpoint URL or other configuration attributes may vary depending on the environment, such as development, QA, UAT, and/or production. A re-build and a re-deploy of the service may be necessary if any of those properties change. How can configuration changes be made without changing the code?How can a service be made to function in many environments without requiring any configuration changes?

**Solution:**

In microservices architecture, externalized configuration refers to the practice of storing configuration settings outside the application code, typically in external configuration files, centralized repositories, or services. This approach allows for more flexible management of configuration settings across distributed microservices, enabling changes without the need to modify the service code itself.

**Benefits of Externalized Configuration:**

- **Decoupling Configuration from Code:** By keeping configuration and application code apart, microservices can operate in a variety of contexts (development, testing, staging, and production, for example) without being restricted to any one of them.

- **Simple Modification:** Modifications to configuration parameters, including database URLs, API keys, or feature toggles, can be made without requiring a microservice restart or a full program relaunch.

- **Centralized Management:** Externalized configuration makes it simpler to change and manage all microservice settings from one location, which is particularly useful in bigger architectures where there may be hundreds or even thousands of microservices.

- **Dynamic Configuration:** Externalized configuration makes it easier for certain systems to make dynamic configuration adjustments in response to user requests, system load, and other real-time circumstances.

- **Environment-Specific Configuration:** Depending on the environment in which they operate, microservices frequently require distinct setups. Without changing the service code, externalizing configuration enables each environment to have its own set of parameters.

b. **Service Discovery Pattern:**

In a microservices architecture, service discovery refers to the process by which services (microservices) dynamically discover each other within the network.

**Challenge:**

Usually, services have to call each other. Services in a monolithic program call each other using language-level method or procedure calls. Because services in a traditional distributed system deployment operate at fixed, well-known locations (hosts and ports), they can readily call one another using RPC or HTTP/REST. However, a contemporary microservice-based application usually operates in a containerized or virtualized environment where the number of service instances and their locations fluctuate on the fly.How does a service's client, such as an API gateway or another service, find out where a service instance is located?In distributed systems, services often need to communicate with one another, but due to their dynamic nature, especially in cloud-native environments or containerized applications—the locations (e.g., IP addresses and ports) of services can change over time.

**Solution:**

Service discovery becomes essential to ensure that microservices can find and communicate with each other without hardcoding network details.

**Types of ServiceDiscovery:**

Service discovery is typically implemented using two main types: Client-Side Service Discovery and Server-Side Service Discovery.

**Client-Side Service Discovery:**In client-side service discovery, the client (the service requesting communication) is responsible for querying the service registry to discover the available instances of the target service. Then the registry responds with a list of available instances (including their IP addresses and ports).Once the client is aware of the instances that are accessible, it will get in touch with one of them directly, frequently selecting one through load balancing. The Client Side Service Discovery provides fine-grained control over the load balancing process at the client level is an advantageous one.

**Server-Side Service Discovery:**The client does not make a direct query to the service registry while using server-side service discovery. Rather, service discovery is managed by a load balancer or proxy (such an API Gateway or service mesh) on the client's behalf.The load balancer or API Gateway queries the service registry for the available instances of the target service.The load balancer selects an appropriate instance based on a load balancing strategy and forwards the request. The advantages of Server Side Service Discovery is that the client is decoupled from service discovery logic, simplifying the client and the load balancer can abstract the complexity of discovering services and managing failures, often with better integration with features like retries and timeouts.

**Implementation of Service discovery:**

It can be implemented in may patterns like Service registry and Client Query, DNS based registry, Service Mesh.

**Service Registry:**Services register themselves with a service registry (such as Consul, Eureka, or Zookeeper) in this pattern. After then, clients search the registry to find services that are offered.Dynamic discovery with health checks ensures that only healthy instances are discovered.

**DNS-Based Discovery**: In this pattern, services are registered in a DNS (Domain Name System), and clients use DNS queries to discover available service instances.Simple to implement, especially with Kubernetes and cloud-native environments.

**Service Mesh**:The infrastructure layer that controls communication between microservices is called a service mesh. A strong solution for service discovery is offered by service meshes, which handle observability, security, monitoring, and traffic management in addition to service discovery.

**Benefits of Service Discovery Pattern**:

- **Decoupling**: Service discovery decouples the microservices from the specifics of their locations (IP addresses, ports), enabling flexibility in deployment and management.
- **Load Balancing**: Service discovery also plays a crucial role in distributing traffic across multiple instances of a service, helping balance the load and avoid overloading individual instances.
- **Dynamic Scaling**: Microservices are often scaled dynamically, meaning new instances might be added or removed based on traffic or load. Service discovery helps manage this by allowing services to find the new instances automatically.
- **Fault Tolerance**: If one service instance goes down, service discovery allows other healthy instances to be discovered and used, enhancing resilience.
- **Avoiding Hardcoding**: In a dynamic system, it is impractical to hardcode the addresses of services. Service discovery allows services to find one another without knowing their exact location.

c. **Circuit Breaker Pattern:**

The Circuit Breaker pattern is a design pattern used to improve the resilience of a system by preventing it from repeatedly trying to execute a failing operation.

**Challenges:**

Services occasionally work together to process requests. There is always a chance that the other service may be unavailable or displaying such high latency that it is practically unusable when one service synchronously invokes another. While waiting for the other service to react, the caller may use up valuable resources like threads. The calling service can become unable to handle more requests as a result of resource exhaustion. It's possible for the failure of one service to affect other services across the application.How to prevent a network or service failure from cascading to other services?

**Solution:**

The Circuit Breaker pattern helps isolate these failures and provides a mechanism to recover gracefully when a failure occurs, thereby improving the overall stability of the system.

**Working principle of circuit breaker pattern:**

In order to implement the Circuit Breaker pattern, a function call to an external service (or any other possibly unreliable activity) is wrapped in a "circuit breaker" that keeps track of whether it was successful or not. The circuit breaker will "trip" and prevent the service from informing the failing service any more if a predetermined threshold of failures is reached, such as five consecutive failures. It will instead provide an error message, a default value, or a fallback response. The circuit breaker goes into a "half-open" condition after a predetermined amount of time, allowing a restricted number of requests to test if the malfunctioning service has been fixed. The circuit breaker will return to "closed" and permitted to continue regular operations if the requests are successful.

**States of a circuit Breaker:**

**Closed:**

- In the "closed" state, the circuit breaker allows requests to pass through to the service. It monitors the success and failure of each request.
- If the service continues to respond successfully, the circuit breaker remains closed.
- If a certain number of failures occur within a predefined threshold, the circuit breaker will trip and move to the open state.

**Open:**

- In the "open" state, the circuit breaker prevents any further requests from being sent to the failing service. Instead of making the call, it immediately returns a fallback response or an error.
- This state prevents the system from repeatedly trying to call a failing service, which could overload the system and exacerbate the failure.

**Half-Open:**

- After a set period of time (the "reset" period), the circuit breaker enters the "half-open" state. In this state, a limited number of requests are allowed to pass through and attempt to interact with the service.
- If these requests succeed, the circuit breaker is considered to have recovered, and it returns to the closed state.
- If the requests fail again, the circuit breaker returns to the open state.

**Benefits of Circuit Breaker pattern:**

- **Graceful Degradation:** By returning a fallback response when the circuit breaker is open, the system can continue operating, albeit in a degraded state, rather than completely failing.
- **Fault Isolation:** The primary benefit of the Circuit Breaker pattern is that it isolates failures. If one microservice fails, the circuit breaker prevents cascading failures by stopping requests from being made to the failing service.
- **System Stability:** The Circuit Breaker helps prevent overloading a service that is already failing, which can worsen the issue. This improves the overall stability of the microservices ecosystem
- **Automatic Recovery:** The circuit breaker provides a mechanism for automatic recovery. Once the failing service stabilizes, the circuit breaker allows the system to resume normal operations.
- **Improved Reliability:** It prevents cascading failures by isolating failures in one service and avoiding repeated calls to the failing service.
- **Resilience:** Helps the system recover gracefully from partial failures without affecting the entire system.

## d. Blue-Green Deployment Pattern:

The Blue-Green Deployment pattern is a release management strategy that minimizes downtime and risk by running two identical production environments, referred to as Blue and Green.

**Challenge:**

Multiple microservices can be included in a single application using the microservices architecture. There may be a significant outage that affects the business if we suspend all services before launching an upgraded version. Any rollback will be a nightmare as well. Then, how can we minimize or prevent service outages during deployment?

**Solution:**

The Blue-Green Deployment pattern provides an efficient way to release and manage microservices in production environments, ensuring minimal downtime and risk. By maintaining two identical environments (Blue and Green), this pattern enables seamless, controlled rollouts and quick rollbacks, improving the reliability and stability of microservices architectures. However, it requires careful planning, especially regarding infrastructure overhead, database migrations, and traffic management.

**Working nature of Blue Green deployment:**

**Blue Environment (Current Version):** This is the active environment that serves production traffic.It is the current microservices version that is processing real-time user requests at the moment.

**Green Environment (New Version):** The Green Environment (New Version) is a replica of the Blue environment that has been updated to use the latest microservices. Prior to being promoted to production, the Green environment undergoes through testing, validation, and staging.

**Traffic Switching:** Traffic is moved from the Blue environment to the Green environment after the updated version in the Green environment has been thoroughly tested and validated. A load balancer or routing device is frequently used to accomplish this switch.

**Rollback:** To guarantee business continuity while problems are fixed, traffic can be swiftly sent back to the Blue environment, which is still using the previous stable version, in the event that any problems with the new version in the Green environment emerge after the changeover.

**Post-Deployment:** The Green environment becomes the active environment following a successful deployment, while the Blue environment can be utilized for maintenance or to stage the subsequent deployment.

**Benefits of Blue-Green deployment pattern:**

- **Zero Downtime:** The capacity to carry out upgrades with no downtime is the primary benefit. The active environment serves users continuously, and any changes take place in the background without any problems.
- **Rollback Strategy:** It is simple to quickly return to the previous environment (Blue) in the event that there are problems with the new release, causing the least amount of disturbance to users.
- **Improved Release Management:** Releases may be managed more tightly because to the environment separation, which makes it obvious which version is active at any given time.
- **Simplified Testing:** Since the Green and Blue environments are the same, testing can be carried out under production-like circumstances, guaranteeing that the updated version will function as planned.
- **Decreased Risk:** Blue-Green deployment lowers the chance of releasing problematic code into production by testing the updated version in the Green environment before switching traffic.

**Benefits of Cross Cutting:**

- **Consistency:** Across microservices, consistency is guaranteed by centrally managing issues like logging, authentication, and monitoring.
- **Reusability:** By implementing cross-cutting issues as reusable parts, the system can be maintained and developed more easily.
- **Separation of Concerns:** The system is more scalable and manageable when cross-cutting concerns are isolated, allowing the microservices to stay focused on their main business logic.

- **Better Security and Monitoring:** The microservices ecosystem can be better controlled and seen thanks to security features like OAuth2 and centralized monitoring solutions.

## Conclusion:

In summary, Cross-cutting concerns are an important aspect of microservices architecture that must be handled effectively to ensure scalability, maintainability, and security. By leveraging patterns like the API Gateway, service mesh, AOP, and sidecars, organizations can centralize and standardize the handling of concerns such as logging, security, and monitoring across all microservices without cluttering business logic.

## References

[1] Chris Richardson "Pattern:Server-side service discovery"https://microservices.io/patterns/server-side-discovery.html(2019 )

[2] Chris Richardson "Pattern: Client-side service discovery" https://microservices.io/patterns/client-side-discovery.html(2019)

[3] Chris Richardson "Pattern:Service registry" https://microservices.io/patterns/service-registry.html (2019)

[4] Chris Richardson "Pattern: Health Check API" https://microservices.io/patterns/observability/health-check-api.html(2019)

[5] Maryam Naveed "Blue-Green Deployments with Kubernetes: A Comprehensive Guide" https://medium.com/cloud-native-daily/blue-green-deployments-with-kubernetes-a-comprehensive-guide-5d196dad1976(May 07, 2023)

[6] Simone Cusimano "Microservices and Cross-Cutting Concerns" https://www.baeldung.com/cs/microservices-cross-cutting-concerns (Nov 09, 2022)

[7] Microsoft "Front-end client communication" https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/front-end-communication (Apr 06, 2022)

[8] Microsoft "Service-to-service communication" https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/service-to-service-communication(Oct31, 2022)

[9] DZone "Microservices Design Patterns: Essential Architecture and Design Guide" https://dzone.com/articles/design-patterns-for-microservices (Jun 06, 2023)

[10] 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)

[11] Sam Newman "Building Microservices: Designing Fine-Grained Systems" O'Reilly (Feb 9, 2022)

[12] Chris Richardson "Microservice Architecture Pattern" https://microservices.io/patterns/microservices.html (2019)

[13] Ravikiran Butti "Enterprise Microservices Design [Part 5: Cross-Cutting Concerns]" https://ravikiran763.medium.com/enterprise-microservices-design-part-5-cross-cutting-concerns-5bed113323a8 (Oct 19, 2020)