

Microservices Mastery:

The patterns and the path
to effective system design

Decoding why banks must take a strategic approach
to succeed with microservices-based architecture

Contents



Preface

In today's rapidly evolving banking landscape, agility, scalability, and innovation are more critical than ever. This report, "Microservices Mastery: The patterns and the path to effective system design," aims to equip banking executives with a comprehensive understanding of how microservices architecture, leveraging key design approaches and proven patterns, can revolutionize core banking systems to meet modern demands.

Historically, banking systems have relied on monolithic architectures—large, interconnected applications that are now increasingly seen as impediments to progress. These systems, once effective, now struggle to keep up with the pace of digital transformation and the rising expectations of customers and regulators. There is a clear need for modernization, and microservices-based architecture offers a compelling solution.

Microservices break down banking applications into smaller, modular services, each responsible for a specific functionality. This decomposition enhances agility, scalability, and resilience, allowing banks to adapt quickly to changes. However, the world of microservices is complex and presents several challenges, such as managing distributed systems, countering latency, maintaining data consistency, steering deployments with stability, among others. Overcoming these and several other challenges requires strategic planning and a deep understanding of the intricacies involved.

To navigate these complexities, leveraging established microservice patterns is crucial. These patterns offer proven solutions to common issues in microservices environments, ensuring a coherent and well-organized architecture. By strategically applying these patterns, banks can create robust, scalable, and maintainable systems capable of continuous improvement. The report discusses a handful of patterns, demonstrating their relevance, the approach to adopt them, the benefits proposition, and the caution points to be wary about. Throughout these discussions, several critical system attributes are explored as well. Effective system design requires careful consideration of those attributes and the necessary trade-offs based on the digital banking application context. By adopting a pattern-based microservices approach, banks can build high-performing systems that are responsive to modern business demands.

Domain-driven design (DDD) is another essential approach discussed. DDD aligns microservices with business needs, facilitating effective system design that enhances both agility and maintainability. The report also highlights how Infosys Finacle's cloud-native, true microservices-based digital banking platform exemplifies these design considerations, enabling banks to deliver next-generation services.

"Microservices Mastery" aims to provide you with the insights and strategies needed to navigate the transformative journey of microservices, empowering your institution to achieve excellence in system design and delivery.



01

Monoliths are a passe', the microservices era is here

Historically, banking systems have relied on monolithic architectures, where all banking functions are tightly integrated into a single, monolithic application. While this approach served its purpose in the past, it has become a bottleneck in today's fast-paced digital landscape, hindering banks' ability to innovate and adapt to changing market dynamics. The traditional monolithic architecture that once underpinned core banking systems is facing unprecedented challenges. There's a growing imperative for banks to modernize their architecture, with the rise of digital transformation and the evolving expectations of customers.

Enter microservices-based architecture, a paradigm shift that holds the key to unlocking agility, scalability, and resilience in the banking sector. Microservices offer a compelling alternative to the monolithic model, advocating for the decomposition of banking applications into smaller, modular services, each responsible for specific banking functionalities. This granular approach not only facilitates agility and scalability but also enables banks to respond swiftly to customer demands and regulatory changes.

The promise of micro services

Microservices based architecture provides the right foundations for banking operations, enabling banks to innovate rapidly, enhance customer experiences, and maintain a competitive edge in the digital era. The key propositions include:

Scalability and flexibility

In today's banking world, scalability is paramount. Microservices architecture allows banks to scale individual services independently based on demand, thereby optimizing resource utilization and ensuring optimal performance during peak transaction volumes. This elastic scalability is particularly crucial in handling fluctuating customer traffic and seasonal banking activities.

Agility and innovation

Banking landscapes are constantly evolving, driven by technological advancements and changing consumer expectations. Microservices empower banks to innovate at speed, as changes can be implemented and deployed to individual services without disrupting the bank's IT ecosystem. This

agility enables banks to roll out new products and services swiftly, staying ahead of the competition and meeting the ever-changing needs of customers.

Resilience and security

With cybersecurity threats on the rise, ensuring the resilience and security of banking systems is paramount. Microservices architecture enhances resilience by isolating failures to specific services, preventing system-wide outages and minimizing the impact of security breaches. Additionally, banks can implement robust security measures at the service level, bolstering the overall security posture of their digital banking platforms.

Microservices - the growing imperative

As banking continues its digital transformation journey, the relevance of microservices-based architecture in the banking sector is growing exponentially. From legacy banks to fintech startups, embracing microservices is no longer a choice but a necessity for staying competitive in an increasingly digital banking landscape.





02

The microservices journey is fraught with complexities

Microservices architecture, while offering significant benefits, presents a set of formidable challenges that require careful navigation. As banks and financial institutions embrace this architectural paradigm, the complexities they encounter demand strategic solutions and diligent management.



Following are a few intricate situations, in real-world operations:

The distributed dilemmas

The very essence of microservices architecture, which entails composing applications with independent services, introduces a layer of complexity. Managing intricate dependencies between numerous services necessitates meticulous planning and robust communication protocols. This distributed nature can create hurdles in comprehending overall system behavior and troubleshooting issues.

The management overheads

Deploying, monitoring, and maintaining a multitude of independent services creates a multifaceted challenge. Infrastructure provisioning, configuration management, and service health monitoring become more intricate, demanding additional resources and specialized tooling. This overhead requires ongoing attention to ensure efficient system operation.

Performance optimizations: Countering latency

Microservices often communicate through APIs, introducing network calls that can impact performance. As data traverses the network between services, latency can emerge, potentially leading to sluggish user experiences and reduced responsiveness. Optimizing network communication and minimizing latency are crucial considerations for ensuring a performant microservices landscape.

Data consistencies: A key challenge

Monolithic systems traditionally house data in a centralized location, ensuring consistency. However, microservices often manage their own data stores. This distributed data management introduces the challenge of maintaining data consistency across services, especially during transactions that span multiple microservices. Implementing robust consistency mechanisms is paramount for data integrity.

Deployments: Balancing change and stability

Microservice deployments necessitate a delicate steering – embracing change while maintaining stability. While automation can streamline the process, the risk of introducing regressions or service disruptions during deployments remains a constant concern. Rigorous testing strategies and rollback mechanisms are essential to mitigate these risks.

Resource optimizations: Combating fragmentation

Microservices can lead to resource fragmentation. Each service consumes its own share of processing power, memory, and storage, potentially leading to underutilized resources across the system. Implementing resource optimization techniques and containerization technologies can help ensure efficient utilization.

The complex and plausible real-world scenarios in banking

Poorly designed microservices can lead to a cascade of problems that can cripple applications. The following two scenarios depicts what one might encounter:

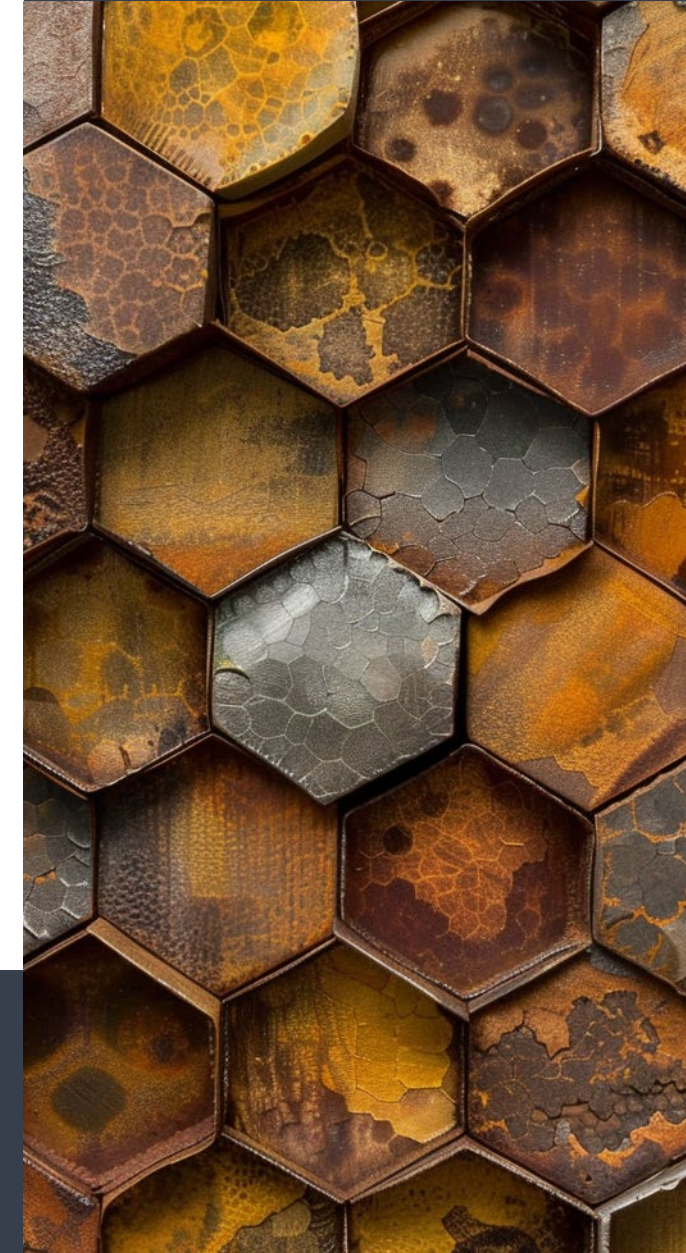
Scenario 1 – In a microservices-based digital lending platform

Consider the following:

- The system fails to handle a surge in loan applications during a promotional period, leading to performance degradation and timeouts. **Scalability**, the key concern.
- Credit score calculation crashes frequently, causing interruptions in loan processing. **Reliability**, the key concern.
- During a server maintenance window, the application experiences unexpected downtime, making the lending services unavailable to users. **Availability**, the key concern.

- Integrating a new third-party credit scoring service requires significant changes across multiple microservices, revealing the system's inflexibility. **Flexibility**, the key concern.
- The system cannot process high volumes of loan applications simultaneously, leading to bottlenecks and delayed approvals. **Throughput**, the key concern.
- A vulnerability exposes sensitive customer data, resulting in a data breach. **Security**, the key concern.
- Incurs high operational costs due to extensive use of cloud resources and third-party services. **Cost-effectiveness**, the key concern.

Each of them illustrates potential pitfalls in the design, implementation, and operation of the digital lending application, emphasizing the need for careful planning and robust practices when adopting microservices based architecture.



Scenario 2 – In a microservices-based digital payments platform

Consider the following:

- During Black Friday or other seasonal sales, the system fails to handle a sudden spike in transactions, causing performance degradation and transaction failures. **Scalability**, the key concern.
- The transactions validation is slow due to inefficient database queries, resulting in delayed payment processing. **Performance**, the key concern.
- Changes in the payment gateway integration service require unexpected modifications. **Poor modularity**, the key concern.
- The application faces issues integrating with a new international payment processor due to incompatible data formats and protocols. **Interoperability**, the key concern.
- The application cannot gracefully handle the failure of a non-critical functionality, causing a complete system outage. **Resilience**, the key concern.
- A vulnerability in payments authentication exposes user credentials, leading to unauthorized access and data breaches. **Security**, the key concern.
- Lack of proper logging and monitoring makes it difficult to diagnose and resolve issues, leading to prolonged outages of payments services. **Observability**, the key concern.

Each of the examples highlights potential challenges and failures in a digital payments application built on a microservices architecture, emphasizing the need for a diligent approach.

Thereby, it's about crafting a system design that excels in every facet!

Creating a robust micro-services architecture demands a meticulous consideration of various system attributes. Scalability ensures seamless handling of increasing loads, while reliability and availability guarantee consistent service delivery. Flexibility allows adaptation to changing requirements, while performance and throughput optimize resource utilization and responsiveness. Security safeguards sensitive data, while maintainability and modularity ease system upkeep and development. Interoperability enables seamless integration, and usability ensures intuitive user experiences. Portability facilitates deployment across diverse environments, while simplicity

reduces complexity! Documentability aids in understanding and troubleshooting, while resilience ensures system continuity. Cost-effectiveness optimizes resource allocation, and adaptability fosters responsiveness. Testability validates functionality, and efficiency maximizes performance. Consistency maintains data integrity, while observability aids in system monitoring. Lastly, feedback loops enable continuous improvement based on user input and operational insights. Integrating these several attributes ensures a well-rounded micro-services architecture that excels in functionality, reliability, security, and adaptability.

But how can one effectively achieve the multitude of system design attributes? That's precisely the focal point of the upcoming chapter's discussion!





03

Perfecting the microservices journeys: The pattern-based strategies

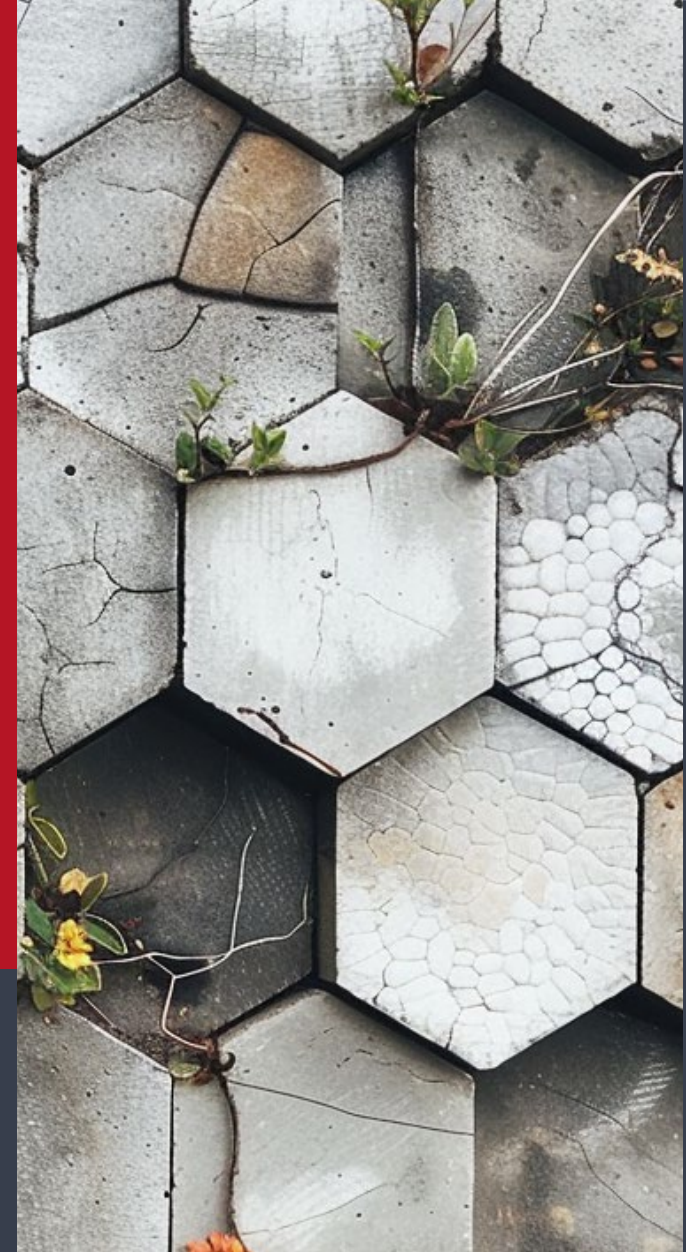
Microservices architectures represent a robust methodology for developing intricate applications by decomposing functionality into discrete, autonomous services. This decomposition endows systems with enhanced scalability, agility, and maintainability. However, the inherently distributed nature of microservices presents significant challenges, including increased complexity in communication, data management, and system orchestration.

To navigate these challenges and construct an optimal microservices architecture, leveraging established microservice design patterns is essential. These patterns, proven through extensive use, address recurrent issues encountered in microservices environments. Implementing a patterns-based approach effectively addresses critical system attributes such as scalability, resilience, maintainability, and many others.

Strategically applying these patterns results in a microservices architecture that is not only coherent and well-organized but also robust and capable of handling growth and change over time. This approach empowers technology teams to accelerate feature delivery and adapt swiftly to evolving requirements, fostering an environment of continuous improvement

and responsiveness. Consequently, a well-patterned microservices architecture facilitates the creation of sophisticated, high-performing banking applications that meet modern business demands.

There are several microservices patterns across various categories. In the rest of this chapter, a few are discussed in detail to highlight their contextual relevance, the approach to implementing them, the benefits they offer, cautionary points to consider, and the scenarios where each pattern is most applicable. Patterns are also referred to as design constructs interchangeably throughout the discussions.





PATTERN

The strangler pattern

The context

The strangler pattern in microservices architecture is a strategy used during the process of migrating from a monolithic application to a microservices-based architecture. The term “strangler” refers to the way in which the new architecture grows around the existing one, gradually replacing it until the old system is completely decommissioned or “strangled.” It allows for the continued delivery of features and improvements to the existing system while gradually transitioning to a more scalable and maintainable microservices architecture. Additionally, it provides flexibility in terms of prioritizing which parts of the monolith to replace first based on business needs and technical feasibility.

The approach

Here's the approach for adopting the strangler pattern in a microservices architecture:

- **Lay the groundwork:** Analyze the monolithic application to identify functionalities that are well-suited for becoming microservices. Prioritize these based on business value and ease of migration. Design clear APIs for the new services and plan how data will be managed across both systems.
- **Build the bridge:** Develop a facade application, the "strangler," that acts as an intermediary. This facade routes requests to either the monolith or the newly developed microservices based on defined criteria. Consider a phased rollout of the facade to gradually introduce the microservices.
- **Strangle incrementally:** Develop and deploy microservices one by one, focusing on the prioritized functionalities. Migrate functionalities from the monolith to the microservices gradually, shifting traffic away from the monolith and towards the new services. Rigorous testing, monitoring is the key.
- **Retirement and simplification:** As microservices take over more functionalities, deprecate the use of the monolith. Once it's no longer critical, consider refactoring or decommissioning it entirely. This will streamline the architecture and free up resources.

The benefits proposition

Adopting the strangler pattern in microservices architecture offers several key benefits:

- **Incremental transition:** Allows for a gradual migration from a monolithic architecture to a microservices architecture. This reduces the risk associated with a big-bang approach.
- **Continuous delivery:** By breaking down the migration into smaller, manageable pieces, the approach supports an agile development process, allowing for more frequent releases and quicker responses to market changes or customer needs.
- **Business continuity:** The existing monolithic application remains operational throughout the migration process, ensuring that business operations are not disrupted.
- **Scalability and flexibility:** As components are migrated to microservices, they can be independently scaled based on demand, improving the overall scalability of the system. The architecture becomes more flexible, allowing for easier updates, maintenance, and integration of new technologies.
- **Technology heterogeneity:** The pattern allows for the use of different technologies and tools for new microservices, without being constrained by the technology stack of the monolithic application.
- **Better alignment with domain-driven design:** The migration process can be guided by domain-driven design, ensuring that microservices are aligned with business domains and processes.
- **Reduced technical debt:** Gradually replacing parts of the monolith allows for addressing and eliminating technical debt in a controlled manner.

The caution points

Adopting the strangler pattern in a microservices-based architecture involves several challenges and potential pitfalls. Here are key points:

- **Complexity of proxy layer:** A proxy layer to route requests can add significant complexity. Ensuring that this layer is performant and doesn't become a bottleneck is crucial.
- **Data consistency and integrity:** Managing data consistency across the monolith and microservices can be challenging. Implementing eventual consistency models and handling data synchronization can add complexity.
- **Inter-service communication:** Differences in protocols, data formats, and communication styles (synchronous vs. asynchronous) can get complicated. Ensuring reliable communication and handling potential latencies and failures is critical in strangler process.
- **Security concerns:** With more services and points of interaction, the attack surface increases. Implementing consistent security policies and practices across the monolith and microservices is necessary.
- **Technical debt:** While the strangler pattern helps manage technical debt by gradually replacing the monolith, there's a risk of accumulating new technical debt if microservices are not properly designed and maintained. Regularly reviewing and refactoring microservices is essential.
- **Testing and quality assurance:** Ensuring comprehensive testing of both the monolith and microservices is crucial. This includes unit tests, integration tests, and end-to-end tests. As microservices are introduced, regression testing becomes more complex but is highly essential to safeguard functionality.

When is this the right option?

The strangler pattern is a recommended approach in the journey towards a microservices architecture, particularly when dealing with legacy monolithic systems that pose challenges in maintenance, scalability, and extensibility. It offers a structured and low-risk strategy to modernize the architecture incrementally, reducing technical debt and improving agility.

The saga pattern

The context

The saga pattern is a way to primarily address the challenges of data consistency in distributed business transactions that spans microservices by breaking them into smaller, manageable local transactions and providing mechanisms for coordination and compensation. This design construct allows microservices to work together to ensure the overall consistency of a business transaction in a distributed environment.



The approach

- **Choreography:** In this approach, each microservice in the saga is responsible for its own local transactions and emits events to signal the success or failure of those transactions. Other microservices listen to these events and react accordingly, advancing the overall saga. It offers flexibility and decentralization, as each microservice can evolve independently without being tightly coupled to a central coordinator.
- **Object based orchestration:** In an object-based orchestrator, often a separate microservice or component explicitly coordinates the flow of the saga. It determines the sequence of steps, communicates with microservices, and handles the overall coordination and compensation logic. It provides a centralized view of the workflow, making it easier to understand and manage. It also offers better control over the entire process.

The benefits proposition

The saga pattern offers several key benefits in the context of distributed transactions within a microservices architecture:

- **Maintaining data consistency:** The primary goal of the saga design is to ensure data consistency in distributed transactions, that spans across microservices.
- **Loose coupling:** The saga design promotes loose coupling between microservices. Each microservice is responsible for its own local transactions, and the interactions are typically based on events or messages. This loose coupling allows individual services to evolve independently without direct dependencies on others.
- **Fault tolerance:** The saga provides mechanisms for handling failures during the execution of a distributed transaction. If a step in the saga fails, compensating transactions can be triggered to undo or compensate for the changes made by the preceding steps, ensuring that the system remains in a consistent state.
- **Scalability:** Microservices implementing the saga approach can scale independently. Each microservice can be scaled horizontally to handle increased load, and the coordination and communication between microservices can still be managed effectively.

- **Flexibility and evolvability:** The saga pattern supports flexibility and evolvability in a microservices architecture. As the business requirements change, individual microservices can be modified or added without significant impact on the overall system, as long as the coordination and compensating mechanisms are appropriately maintained.
- **Event-driven architecture:** The saga design often aligns well with an event-driven architecture. Microservices communicate

through events or messages, facilitating asynchronous communication and decoupling between services. This can improve system responsiveness and scalability.

While the saga offers these benefits, it's essential to carefully consider the specific requirements and characteristics of a system before choosing to implement this design. The choice between choreography and orchestration, for example, depends on factors like system complexity and maintainability requirements.



The caution points

While saga provides benefits in managing distributed transactions that spans microservices, there are some cautionary considerations that should be taken into account.

- **Complexity of compensation logic:** Designing compensating transactions can be challenging, especially when dealing with complex business logic. Ensuring that compensating transactions are capable of reverting changes made by preceding steps requires careful consideration and thorough testing.
- **Increased latency:** Coordinating microservices through events may introduce delays, and this asynchronous nature can impact the overall response time of the system.
- **Monitoring and observability:** Monitoring and debugging distributed systems using the saga design can be challenging. Tools and practices for monitoring events, tracking the state of sagas, and diagnosing issues are crucial for maintaining system health and resolving problems efficiently.
- **Data model consistency:** Ensuring consistent data models across microservices is crucial. Changes in the data model of one microservice might necessitate corresponding adjustments in other services, and managing these dependencies is important to prevent data inconsistencies.
- **Long-running transactions:** Long-running sagas, which involve a large number of steps, can increase the chances of failures occurring during their execution. The longer the saga, the higher the likelihood of partial failures, making it essential to carefully manage and monitor such transactions.

While the saga design addresses challenges in distributed transactions, its successful implementation requires a thorough understanding of the specific system requirements, careful consideration of potential failure scenarios, and robust compensating transaction logic. Thus, developers should carefully weigh the benefits against the complexities and potential pitfalls when deciding to adopt a saga-based design!

When is this the right option?

The adoption of a saga-based design is well-suited for scenarios where microservices architecture demands coordinated, distributed transactions across multiple services. It proves effective in systems that favor an asynchronous communication model, leveraging events to trigger actions and maintaining loose coupling between microservices. The design approach is particularly relevant when compensating transactions can be employed to handle failures, enabling the system to recover to a consistent state. Saga-based designs align with event-driven architectures, support scalability by allowing independent scaling of microservices, and accommodate dynamic business processes, making them adaptable to changing requirements.

The API gateway pattern

The context

It's a design construct or a pattern used in microservices architecture to provide a single point of entry for client applications to access various microservices within the system. It abstracts away the complexities of service-to-service communication, improving security, scalability, and manageability of the system.

The approach

Developing an API gateway for microservices architecture begins with right technology/tools choice. There are three ways -

- **Open-source tools:** Popular choices include Kong, Tyk, and Apigee. These offer flexibility and customization.
- **Cloud-based solutions:** Major cloud providers like AWS (API gateway), Azure (API management), Google Cloud (Apigee) offer managed services for API Gateways. These can be convenient but other factors need to be considered.
- **Develop your own:** For specific needs and complete control, one can build own API gateway using a suitable programming language and framework.

Here's the general approach:

- **Identifying requirements:** Listing the specific requirements including the functionalities the API Gateway has to support, such as authentication, authorization, rate limiting, and protocol translation.
- **Defining the APIs:** Define the APIs that will be exposed by the API Gateway to clients. This includes determining the endpoints, methods, request and response formats, and any additional metadata such as versioning information.
- **Implementing routing logic:** To map incoming requests to the corresponding microservices based on the request URI, headers, or other parameters. This may involve defining routing rules, configuring routes, and handling dynamic routing based on service discovery mechanisms.
- **Integrating with microservices:** Integrate the API Gateway with the underlying microservices to forward requests to the appropriate services and aggregate responses if necessary.
- **Implementing middleware:** To handle common tasks such as request/response transformation, caching, content compression, and error handling.

Handling cross-cutting concerns, implementing monitoring and analytics, managing for gateway reliability and scalability are the other design considerations.

The benefits proposition

Implementing an API Gateway pattern in microservices architecture offers several key benefits:

- **Request routing:** Routing incoming requests to the appropriate microservices based on the request's endpoint or content. It can handle load balancing and distribute requests across multiple instances of a microservice to ensure scalability and availability.
- **Aggregation:** Can aggregate and consolidate data from different microservices to provide a unified response to the client. This reduces the number of requests a client needs to make and can improve performance. Call it the API ergonomics!
- **Authentication and authorization:** Can validate user credentials, generate and check tokens, and enforce access control policies, ensuring that only authorized users can access specific microservices.
- **Security:** Can implement security measures such as SSL termination, request and response validation, and protection against common security threats like SQL injection or cross-site scripting.
- **Monitoring and logging:** Can collect and aggregate logs and metrics from various microservices, offering insights into overall health and performance.
- **Rate limiting and throttling:** To prevent abuse or overuse of resources, the API gateway can enforce rate limiting and request throttling. This helps maintain system stability and prevents individual clients from overwhelming the microservices.
- **Caching:** The API gateway can implement caching strategies to store frequently requested data and reduce the load on microservices. This improves response times and reduces the overall latency of the system.
- **Transformation and protocol translation:** Can handle the transformation of data formats or translate between different communication protocols. This allows microservices to use their preferred data formats or communication protocols while presenting a standardized interface to clients.

The caution points

While implementing the API gateway design in a microservices architecture, there are several cautions and considerations to keep in mind.

- **Single point of failure:** The API gateway becomes a critical component in the system. If it experiences downtime or malfunctions, it can disrupt the entire communication flow between microservices and clients. Implementing redundancy and failover mechanisms is crucial to mitigate this risk.
- **Performance bottleneck:** The API gateway can become a performance bottleneck. Careful consideration must be given to scalability, load balancing, and optimization to handle increasing traffic and ensure low-latency responses.
- **Service discovery:** The API gateway needs to dynamically discover and adapt to changes in the underlying microservices. If new services are added or existing ones are removed, the API gateway must efficiently handle these changes to maintain proper routing and communication.
- **Security concerns:** As a central point for authentication and authorization, the API gateway is a critical security component. It must be robust against various security threats, and its configurations should adhere to best practices.

- **Data consistency:** Aggregating data from multiple microservices may introduce challenges related to data consistency. The API gateway should carefully manage scenarios where one microservice's data is updated while another is still using a cached version, potentially leading to inconsistencies.
- **Protocol and versioning issues:** Microservices may use different communication protocols or have different API versions. The API gateway must handle these differences gracefully and ensure backward compatibility to prevent disruptions when updates or changes occur.
- **Overhead and latency:** While providing various functionalities like authentication, transformation, and aggregation, the API gateway introduces additional processing overhead. This can contribute to increased latency in the communication between clients and microservices. Fine-tuning and optimizing these processes are essential for maintaining acceptable performance.

When is this the right option?

An API gateway design pattern is to be considered when there is a complex microservices architecture that requires centralized management of various concerns, such as security, scalability, and monitoring. It can streamline communication between clients and microservices, making systems more manageable and efficient.

The circuit breaker pattern

The context

In a microservices architecture, a circuit breaker design monitors the interactions between microservices, detect failures, and dynamically adjust its behavior to prevent cascading failures in the system. This service is crucial for enhancing the resilience of the overall architecture by isolating failing services and providing a mechanism for graceful degradation. While the API gateway design pattern focuses on managing the overall communication flow between clients and microservices, circuit breaker service complements to create resilient and efficient applications.

The approach

Integrating a circuit breaker design in a microservices architecture is a proactive step toward creating a resilient and reliable system. The design approach entails the following considerations:

- **Defining service health metrics:** Identifying key metrics along with thresholds that indicate the health of microservices, such as error rates, response times, and availability.
- **Selecting a circuit breaker library:** Choosing a circuit breaker framework that aligns with the technology stack, such as hystrix, polly and others.
- **Configuring circuit breaker parameters:** Defining configurable parameters such as error thresholds, timeout values, and the duration a circuit breaker stays in the open state before transitioning to half-open. It's important to implement adaptive approach basis real-time behaviors!
- **Integrating circuit breaker logic:** Embedding circuit breaker logic within the microservices that require fault tolerance; implementing the closed, open, and half-open states.
- **Implementing fallback mechanisms:** Designing fallback mechanisms to provide alternative responses or default data when a circuit breaker is in the open state.
- **Monitoring microservices health:** Implementing robust monitoring and logging to continuously observe the health and performance of microservices.

The benefits proposition

Enhancing the overall reliability, fault tolerance, and resilience of the application, circuit breakers in a microservices architecture provides several key benefits:

- **Prevents cascading failures:** Helps isolate failing microservices, preventing the propagation of faults to other parts of the system - contains widespread outage!
- **Offers graceful degradation:** By transitioning to a fallback mechanism when a microservice is in the open state, the system can gracefully degrade its functionality instead of completely failing.
- **Enhances user experience:** Users experience more predictable and meaningful responses even when certain microservices are temporarily unavailable.
- **Enables adaptive system behaviors:** Adaptive strategies enable adjusting system behavior dynamically based on changing conditions, in real-time.

Reduced latency and resource consumption, operational insights are other key promises of this pattern.

The caution points

While integrating circuit breaker in a microservices architecture offers numerous benefits, there are key caution points:

- **Overhead and complexity:** Introduces additional complexity to the system architecture. So, carefully consider whether the benefits of fault tolerance outweigh the added overhead and complexity.
- **Proper configuration:** Incorrectly configured parameters such as error thresholds, timeout values, and duration in each state can lead to suboptimal performance or unintended consequences (including false positives, false negatives). It's crucial to thoroughly test and fine-tune these configurations for optimal behavior.
- **Potential resource exhaustion:** In high-throughput systems, the circuit breaker service itself can become a point of contention or a bottleneck, especially during periods of heavy load or when multiple microservices are experiencing issues simultaneously. Implement mechanisms to prevent resource exhaustion or contention within the circuit breaker service.
- **Vendor lock-in:** Depending on the chosen circuit breaker library or framework, organizations may face vendor lock-in, limiting flexibility and portability. So, evaluate the long-term implications and consider alternatives that offer greater flexibility and interoperability.

When is this the right option?

The decision to adopt the circuit breaker design in a microservices architecture hinges on specific needs and system characteristics. In cases where the microservices architecture entails intricate interactions and dependencies, this technique proves valuable by simplifying fault management and preventing cascading failures. Additionally, if the microservices experience unpredictable workloads, the design becomes crucial in maintaining system stability during periods

of high traffic. Moreover, for those seeking to enhance operational efficiency by proactively handling faults and minimizing manual interventions would find the technique useful. These considerations collectively underscore the importance of evaluating the unique aspects of a microservices ecosystem to determine whether the adoption of the circuit breaker aligns with the system's requirements and goals.



The sidecar pattern

The context

The sidecar pattern in a microservices architecture involves deploying a separate service alongside a primary application. The sidecar service provides additional functionalities or capabilities to the primary application without modifying its core logic. This design pattern is commonly used and typically provides various functionalities such as monitoring, logging, security, and communication with other services. Often, the sidecar runs in the same container or in close proximity to the main application, acting as an extension of it. The sidecar enhances the flexibility, scalability, and maintainability of microservices architectures by offloading cross-cutting concerns into separate, modular components.



The approach

Designing a sidecar pattern effectively involves careful consideration of several factors to ensure that it integrates seamlessly with the main application and provides the necessary functionalities without introducing unnecessary complexity. Here's a step-by-step approach to designing the sidecar pattern:

- **Identifying cross-cutting concerns:** Analyzing the main application to identify cross-cutting concerns, such as logging, monitoring, security, service discovery, or communication protocols, that can be offloaded to a sidecar.
- **Defining sidecar responsibilities:** Clearly defining the responsibilities of the sidecar based on the identified cross-cutting concerns. Each sidecar should have a well-defined purpose and set of functionalities.
- **Choosing deployment strategy:** Deciding on the deployment strategy for the sidecar - as a separate container alongside the main application, as a separate process running on the same host, or even as a library linked directly into the main application.

- **Establishing communication mechanism:** Defining the communication mechanism between the main application and the sidecar - can include inter-process communication mechanisms such as local network sockets, shared memory, or RPC (remote procedure call).
- **Handling failure scenarios:** Implementing appropriate error handling and recovery mechanisms in both the main application and the sidecar – examples include implementing retry logic, circuit breakers, or graceful degradation to handle failures gracefully.

Managing security considerations, and implementing monitoring and metrics collection in the sidecar pattern are also important aspects.

The benefits proposition

The sidecar pattern offers several benefits in the context of microservices architectures:

- **Modularity and separation of concerns:** By separating cross-cutting concerns into sidecar modules, the main application's codebase remains focused on its core functionality. This improves code maintainability, readability, and testability by isolating different concerns into separate components.
- **Isolation and independence:** Since sidecar can be developed, deployed, and managed independently of the main application, it greatly reduces complexity and potential conflicts between different concerns and allows teams to work on different functionalities in parallel.
- **Scalability and performance optimization:** Sidecar instances can be scaled independently from the main application, allowing for fine-grained control over resources and performance optimization.
- **Flexibility and agility:** The pattern enables the addition of new functionalities or changes to existing ones without modifying the main application - thus promotes flexibility and agility in development and deployment.
- **Enhanced security:** By centralizing security functionalities in sidecar modules, teams can ensure consistent enforcement of security policies across all services without the need for duplication or manual configuration.
- **Improved observability and monitoring:** Sidecars can handle tasks such as logging, monitoring, and metrics collection, providing real-time visibility into the performance and health of the microservices architecture.
- **Service discovery and load balancing:** By maintaining a registry of available services and distributing incoming requests across multiple instances, sidecars improve fault tolerance, resilience, and scalability of the overall system.
- **Cross-language compatibility:** The sidecar pattern allows teams to implement functionalities in different programming languages or frameworks than the main application, thereby enabling teams to leverage existing libraries, tools, and expertise without being constrained by the technology stack of the main application.

The caution points

While designing sidecar constructs in the microservices architecture, it's important to consider several cautions to ensure the effectiveness, maintainability, and scalability:

- **Resource overhead:** Adding a sidecar to each microservice can increase resource consumption, such as CPU, memory, and network bandwidth. Carefully consider the resource requirements of each sidecar and monitor resource utilization.
- **Complexity and dependency management:** Introducing multiple sidecar instances can increase complexity in deployment, configuration, and dependency management. Ensure that dependencies between the main application and sidecar modules are well-defined and properly managed.
- **Latency and network overhead:** Inter-process communication between the main application and sidecar can introduce latency and network overhead, especially in distributed environments. Optimize communication protocols, data serialization formats, and network configurations to minimize latency and maximize throughput.
- **Failure isolation and resilience:** Sidecar failures should be isolated from the main application to prevent cascading failures and ensure the resilience of the system. Implement

fault tolerance mechanisms such as circuit breakers, retries, and graceful degradation to handle sidecar failures.

- **Security risks:** Sidecars can introduce security risks if not properly configured or secured. Implement security best practices such as least privilege access, secure communication protocols, and regular security audits to mitigate security risks.
- **Versioning and compatibility:** It's important to ensure that sidecar modules are designed with backward and forward compatibility in mind to support rolling upgrades, version transitions, and heterogeneous environments.

When is this the right option?

The sidecar pattern is a strong choice when there is a need to offload non-core functionalities, standardize practices across services, achieve dynamic scaling for specific tasks, or enable independent deployments in a polyglot environment.

The service mesh pattern

The context

The service mesh pattern is a design approach used in microservices architectures to handle the complexities of service-to-service communications. It involves deploying a dedicated infrastructure layer of lightweight network proxies, also known as sidecars, alongside each microservice instance. These sidecars manage and facilitate communication between services, providing a centralized control plane for routing, security, monitoring, and other cross-cutting concerns. Service mesh provides a way to reliably and efficiently connect, manage, and secure microservices across distributed applications.

The approach

Setting up a service mesh pattern involves several steps to deploy the necessary infrastructure, configure communication between microservices, and enable various features such as traffic management, security, and observability. Here are few key considerations:

- **Selecting a service mesh platform:** Entails choosing a service mesh platform basis the factors such as features, compatibility with existing infrastructure, community support, and ease of integration. Popular options include Istio, Linkerd, and Consul Connect.
- **Preparing the infrastructure:** Ensuring that the microservices are containerized and deployed in a container orchestration platform such as Kubernetes or Docker Swarm.
- **Deploying sidecar proxies:** Involves installing the sidecar proxies alongside each microservice instance. This can be done manually or automatically using tools provided by the chosen service mesh platform.

- **Configuring service mesh control plane:** Setting up the control plane components such as control plane APIs, service discovery, and configuration management to define routing rules, traffic policies, security policies, and observability settings.
- **Enabling traffic management:** Defining traffic management policies such as routing rules, traffic splitting, and load balancing, along with configuring canary deployments, blue-green deployments, or other deployment strategies as needed.
- **Setting up observability:** Configuring monitoring, logging, and distributed tracing to gain insights into the behavior and performance of microservices using built-in or third-party observability tools.

Implementing the right security measures, monitoring the performance and scalability of service mesh deployments are the other key design considerations.





The benefits proposition

The service mesh pattern offers several benefits for managing communication between microservices in a distributed application:

- **Centralized control and management:** Service mesh provides a centralized control plane for managing and configuring communication between microservices. Thus, it simplifies administration, reduces complexity, and enables consistent policies across the entire microservices architecture.
- **Traffic management and load balancing:** Service mesh facilitates optimal distribution of traffic across microservice instances, improving performance, and resource utilization.
- **Enhanced security:** Service mesh enhances security by providing encryption, authentication, and authorization mechanisms for service-to-service communication.
- **Observability and monitoring:** Service mesh provides insights into service behavior, performance, and dependencies, facilitating debugging, optimization, and monitoring of the microservices architecture.
- **Resilience and fault tolerance:** Service mesh implements resilience patterns such as circuit breaking, retries, and timeouts to handle failures gracefully, thereby improving application reliability and fault tolerance.
- **Multi-platform support:** Service mesh can be deployed across different infrastructure platforms, including Kubernetes, VMs, and bare-metal servers, thus enabling organizations adopt microservices architectures across diverse environments and technology stacks.
- **Simplified development and deployment:** Service mesh abstracts away many of the complexities associated with microservices communication, allowing developers to focus on building and deploying individual services.

The caution points

Here are some caution points to consider when designing a service mesh:

- **Increased complexity:** While service mesh simplifies development for individual services, it introduces an additional layer of complexity to overall system architecture. Management and troubleshooting becomes intricate due to the distributed nature of the service mesh.
- **Operational overhead:** Setting up, configuring, and maintaining a service mesh requires additional operational overhead. This includes tasks like monitoring the health of the service mesh itself, managing sidecar deployments, and handling potential configuration issues.
- **Performance impact:** Adding a sidecar proxy to each microservice can introduce some overhead in terms of resource consumption and potential latency. It's important to carefully evaluate the performance impact on specific workloads.
- **Learning curve:** There's a learning curve associated with understanding and effectively utilizing service mesh pattern. The development and operations teams will need to invest time in learning the chosen service mesh implementation and best practices.

- **Security considerations:** While service mesh can enhance security, it introduces new attack surfaces. Proper configuration of authentication, authorization, and encryption policies is crucial to avoid security vulnerabilities.
- **Not a silver bullet!** Service mesh is a powerful tool, but it's not a one-size-fits-all solution. It's best suited for complex microservices architectures where managing communication becomes a challenge. For simpler architectures, the overhead of a service mesh might outweigh the benefits.
- **Vendor lock-in:** While the service mesh pattern itself is vendor-neutral, specific implementations might have dependencies on certain tools or platforms.

When is this the right option?

A service mesh is an ideal addition to a microservices architecture in scenarios where there's a complex landscape of microservices with intricate communication patterns, necessitating centralized management. Where there is a need for advanced traffic management capabilities, enhanced security and compliance, comprehensive observability, scalability, and systems resilience, service mesh offers compelling propositions.



The health check APIs

The context

Health check APIs, also known as health monitoring endpoints or health probes, are a type of API endpoint or interface that allows applications, underlying services to report their operational status or health to external entities, typically monitoring tools or other software components within the same ecosystem. Health check APIs are commonly used in distributed systems, microservices architectures, cloud computing environments, and containerized applications where multiple components need to communicate with each other. They enable continuous monitoring and help in detecting and responding to issues or failures promptly. Considered as a design pattern within the microservices architecture, these APIs offer a quick and automated way to assess the overall health and availability of a system.

The approach

Setting up health check APIs involves several steps to ensure that they accurately reflect the operational status of the services they represent. Below are some of the key considerations:

- **Defining health check endpoints:** Entails deciding on the endpoint(s) where health status will be exposed. Typically, this is an HTTP endpoint reachable by monitoring systems and other services within the ecosystem. Common endpoints include `/health`, `/healthcheck`, or similar.
- **Choosing health check response format:** Determining the format of the response that the health check endpoint will return. This could be a simple JSON response, plaintext, or any other suitable format.
- **Defining health check criteria:** Establishing the criteria that may include checking database connectivity, external service dependencies, CPU and memory usage, disk space, or any other relevant metrics.
- **Implementing health check logic:** Developing the logic within service that evaluates the health check criteria and generates the appropriate response. This logic may involve querying dependencies, performing self-checks, or evaluating system metrics.
- **Handling dependency checks:** If the service depends on other services or resources, make sure to include checks for these dependencies in the health check logic.

Configuring health check frequency, handling unhealthy states, securing health check endpoints are some of the other key considerations.

The benefits proposition

As a construct or a pattern in microservices-based architectures, health check APIs offer several key benefits and play a role in contributing to the reliability, scalability, and manageability of the system. Some of them include:

- **Improved reliability and fault tolerance:** By promptly detecting failures or issues at granular level, health check APIs facilitate quick response mechanisms such as automatic failover, rerouting of traffic, or restarting of failed services. This helps in minimizing downtime and ensuring high availability of the overall system.
- **Autoscaling and load balancing:** Health check APIs provide real-time information about the capacity and availability of microservices. This can be used to dynamically adjust resource allocation and distribute incoming traffic among healthy instances, optimizing performance and resource utilization.
- **Decentralized architecture:** In a microservices architecture, services are designed to be loosely coupled and independently deployable. Health check APIs align with this decentralized approach by allowing each microservice to manage and report its health status independently. This promotes autonomy and resilience!
- **Enhanced development and deployment practices:** By integrating health checks into the development and deployment pipelines, teams can ensure that new releases or updates are thoroughly tested for compatibility and stability before being deployed to production environments.
- **Scalability and elasticity:** Health check APIs support the scalability and elasticity requirements of modern applications. As the workload fluctuates, services can dynamically scale up or down based on demand, while health checks ensure that only healthy instances receive traffic, maintaining performance and reliability.



The caution points

While health check APIs offer numerous benefits, there are several caution points to consider to ensure they are implemented effectively and do not introduce unintended complexities or risks into the system. Some of them are:

- **Overhead and performance impact:** May introduce additional overhead, especially if it involves resource-intensive operations or queries. It's essential to carefully design and optimize health check logic to minimize performance impact, particularly in high-throughput or latency-sensitive systems.
- **False positives and negatives:** Health checks may sometimes produce false positive or false negative results, incorrectly indicating the health status of a service. False positives can lead to unnecessary alerts or actions, while false negatives may result in delayed detection of issues. It's crucial to tune health check criteria and thresholds appropriately.
- **Dependency management:** If the dependencies themselves are unhealthy or experiencing issues, it may lead to cascading failures or incorrect health assessments. Carefully manage dependencies and consider fallback mechanisms or alternative health check strategies to mitigate such scenarios.
- **Security considerations:** Health check endpoints expose operational information about the system, which could be exploited by malicious actors to gather intelligence or launch attacks. Ensure that health check APIs are appropriately secured!

- **Monitoring tool compatibility:** Different monitoring tools or frameworks may have varying requirements or expectations for health check APIs. Ensure that health check endpoints are compatible with the monitoring systems used.
- **Continuous maintenance:** Health check APIs require ongoing maintenance and monitoring to remain effective. Regularly review and update health check criteria, response formats, and monitoring configurations as the system evolves.

When is this the right option?

Health check APIs are a recommended design pattern for microservices architectures. They act like checkups for the services, monitoring their health and dependencies. By implementing health checks, one can proactively identify issues within individual microservices and prevent them from cascading across the entire system. While there are some cautionary points like overhead and security concerns, the advantages outweigh the drawbacks in most cases. If unsure about using health checks, it's generally better to implement them for the increased visibility and maintainability they offer.

The database per service pattern

The context

The database per microservice pattern or construct is a design approach within microservices architecture where each microservice has its own dedicated database. In this pattern, typically, each microservice is responsible for managing its own data storage and schema. This stands in contrast to the traditional monolithic architecture where a single database serves multiple components or services. The approach facilitates decentralized data management, enabling efficient scaling, flexibility in database technologies, and independent evolution of microservices.

The approach

Implementing a database per service approach in a microservices architecture involves several key considerations:

- **Domain-driven design:** Begin by identifying the domain boundaries and business contexts of the application. Align database design with these bounded contexts to ensure that each microservice has a database schema that reflects its specific domain responsibilities!
- **Database technology selection:** Choose appropriate database technologies for each microservice based on its requirements, such as relational database, NoSQL, or specialized databases. Consider factors like data volume, access patterns, scalability requirements, and consistency models.
- **Data consistency strategies:** Implement consistency strategies such as eventual consistency, distributed transactions, or compensating transactions based on the needs of individual microservices and their interactions.
- **Service contracts:** Clearly define the contract between each microservice and its database, including data access patterns, API endpoints, and data formats. Use techniques such as API versioning and documentation.

- **Data access patterns:** Design data access patterns tailored to the specific requirements of each microservice. Consider factors such as read vs. write operations, data volume, latency, and consistency.
- **Inter-service communication:** Implement communication mechanisms such as RESTful APIs, messaging queues, or other techniques for inter-service communication. Define clear boundaries and responsibilities between services to minimize dependencies and facilitate loose coupling.
- **Monitoring and observability:** Implement monitoring and observability solutions to track the health, performance, and behavior of each microservice and its associated database. Use metrics, logs, and distributed tracing to identify and troubleshoot issues effectively.

Determining data partitioning strategies for each microservice's database, planning for data migration and evolution as microservices and their schemas evolve are some of the other key considerations.

The benefits proposition

The database per microservice pattern offers several key advantages:

- **Isolation and encapsulation:** Since each microservice has its own dedicated database, it promotes better separation of concerns and reduces the risk of unintended data access or modification by other services, enhancing overall system reliability and security.
- **Autonomy and independence:** Microservices are autonomous entities, and having a dedicated database for each service aligns well. It promotes polyglot persistence!
- **Scalability:** With each microservice having its own database, it becomes easier to scale each component independently based on its unique workload and performance requirements.

- **Flexibility and agility:** Since each microservice manages its own database schema, it can evolve independently from other services - The flexibility allows for easier modifications, updates, and optimizations without impacting the entire system!
- **Reduced complexity and coupling:** Having a database per service reduces the complexity and coupling between services since each service operates independently with its own data store.
- **Improved fault isolation:** In case of failures or errors within a microservice, having a dedicated database ensures that the impact is limited to that specific service. This improves fault isolation and resilience.

Enhanced security and compliance, performance optimization are other key benefits.



The caution points

While the database per microservice pattern offers numerous benefits, there are also several caution points and challenges to be wary of:

- **Data consistency:** Ensuring data consistency across microservices with separate databases can be challenging. Need to carefully design and implement consistency models, distributed transactions, or eventual consistency strategies to maintain data integrity.
- **Increased operational complexity:** Managing multiple databases adds operational overhead, including provisioning, deployment, monitoring, backups, and maintenance tasks. Resource consumption overhead is yet another concern.
- **Data duplication:** With each microservice having its own database, there may be instances of data duplication or redundancy across services. It's important to avoid inconsistencies and unnecessary storage costs.
- **Cross-service joins and queries:** Performing joins and queries that span multiple microservices' databases can be inefficient and complex. The design should minimize cross-service dependencies and optimize data access patterns to reduce latency and improve performance.
- **Synchronization and versioning:** Synchronizing schema changes and version upgrades across multiple databases can be challenging.

Managing backup and disaster recovery processes, security and access controls, evolving data schemas and migrating data across microservices' databases, performance bottlenecks due to inefficient database queries, resource contention, or hotspots are some of the other key issues.

When is this the right option?

The database per microservice design is a suitable choice in scenarios where each microservice demands a high degree of autonomy over its data and business logic. This design approach proves advantageous when there are diverse data storage needs among microservices, allowing each service to choose a database technology that aligns with its specific requirements. Additionally, it is well-suited for situations where microservices exhibit varying scalability demands, enabling independent scaling based on individual needs without affecting other services. The design's ability to accommodate these considerations makes it a valuable option in microservices architectures where autonomy, flexibility, and scalability are key priorities.

The command query responsibility segregation (CQRS) pattern

The context

CQRS is a design pattern commonly used in microservices architecture to separate the responsibilities of handling read (query) and write (command) operations. In a traditional CRUD (create, read, update, delete) architecture, a single data model handles both read and write operations. However, in complex systems with evolving requirements and scalability needs, this approach can become cumbersome. CQRS addresses this by segregating the responsibilities of handling commands and queries into separate components. By separating concerns and enabling independent optimization of read and write operations, CQRS enhances scalability, flexibility, and maintainability in complex systems.

The approach

Achieving the CQRS pattern in a microservices architecture involves several key steps and considerations:

- **Identifying domain boundaries:** Here, different domains represent a distinct area of functionality with its own set of commands and queries.
- **Defining command and query contracts:** Commands represent actions that modify the state of the system, while queries represent operations that retrieve data without modifying the state. These contracts should specify the inputs, outputs, and behavior of each command and query.
- **Implementing command-side services:** Creating microservices responsible for handling commands within each domain. These services should encapsulate the business logic associated with processing commands and updating the state of the system accordingly.
- **Implementing query-side services:** Creating microservices responsible for handling queries within each domain. Consider using denormalized views, caching strategies, and other optimization techniques to improve query performance.
- **Deciding on data storage:** In a CQRS architecture, it's common to use different data storage mechanisms - a relational database for the command side to ensure transactional consistency, and a NoSQL database or specialized data stores for the query side to optimize read operations.
- **Synchronizing data between command and query sides:** Since the command and query sides operate independently, it will need mechanisms to synchronize data between them. This can be achieved through event sourcing, where changes to the system's state are captured as a series of immutable events. Command-side services publish events, which are then consumed by query-side services to update their data stores.
- **Handling asynchronous communications:** Using messaging systems or event-driven architectures to enable asynchronous communication between command and query services. Consider using technologies like Apache Kafka, RabbitMQ for reliable message delivery.
- **Ensuring consistency and resilience:** Entails implementing mechanisms such as idempotent command processing, eventual consistency, and error handling strategies. Use compensating transactions or other mechanisms to handle failures and maintain system resilience.

The benefits proposition

The CQRS pattern offers several benefits in the context of software architecture, particularly in microservices environments. Here are some key advantages:

- **Improved scalability:** Since commands and queries have different characteristics (write-heavy vs. read-heavy), they can scale each side independently to handle fluctuations in traffic without impacting the other. This enables efficient resource utilization and better overall system scalability.
- **Optimized performance:** The query side can be optimized for fast data retrieval by using denormalized views, caching, or specialized data stores, while the command side can focus on ensuring transactional integrity and business logic enforcement.
- **Flexibility and maintainability:** CQRS promotes a clear separation of concerns between commands and queries, which simplifies the design and maintenance of the system. Each side can be developed, tested, and deployed independently, allowing for greater flexibility.
- **Better domain modeling:** CQRS encourages a domain-driven design approach, where the domain model is based on the business requirements and concepts. Defining commands and queries within each domain allows to create expressive and focused model that closely aligns with requirements.
- **Support for event sourcing:** CQRS is often used in conjunction with event sourcing, where changes to the system's state are captured as a series of immutable events. This approach enhances system reliability, resilience, and traceability, making it easier to diagnose and recover from errors.
- **Enhanced security and compliance:** Separating read and write operations can improve security by limiting access to sensitive data and operations. Additionally, event sourcing provides a tamper-evident log of all state changes, which can be valuable for auditing and compliance purposes.



The caution points

While the CQRS pattern offers various benefits, there are also some caution points and challenges to consider when adopting it:

- **Increased complexity:** Implementing CQRS introduces additional complexity to the system architecture. It will require separate models, data stores, and communication channels for commands and queries. This complexity can make the system harder to understand, develop, and maintain.
- **Consistency challenges:** Maintaining consistency between the command and query sides can be challenging, especially in distributed systems. Since commands and queries operate independently, it will need mechanisms to synchronize data between them. Ensuring eventual consistency and handling concurrency issues requires careful design and may involve trade-offs in terms of performance and scalability.
- **Data synchronization overhead:** Keeping the command and query sides synchronized can introduce overhead, especially in scenarios with high write throughput or complex data transformations. Event sourcing, which is often used with CQRS, adds to additional complexity and infrastructure overhead.
- **Operational complexity:** CQRS can increase operational complexity, particularly in terms of deployment, monitoring, and debugging. With multiple services and data stores involved,

it will need effective tooling and processes for managing deployments, monitoring system health, and diagnosing issues. Adopting CQRS may require additional investments in infrastructure, automation, and operational expertise – costs overhead too!

- **Performance considerations:** CQRS can introduce performance overhead due to data synchronization, event handling, and increased network communication. It will require optimizations to minimize latency and maximize throughput, especially in high-volume or latency-sensitive applications.

When is this the right option?

CQRS is a valuable pattern for microservices architectures handling a significant disparity between reads and writes, or expecting high scalability. It separates read and write operations, allowing independent optimization and scaling for each workload. This is ideal for applications with complex queries or needing different data storage options for read and write models. However, the added complexity of managing separate models and eventual consistency considerations make it best suited for applications that can benefit from these trade-offs.

The backend for frontend pattern

The context

The backend for frontend (BFF) pattern enhances the efficiency and flexibility of microservices based architectures, especially when dealing with diverse frontend applications. It allows for a focused and tailored interaction between the frontend and its dedicated backend service, contributing to a better user experience and improved development agility.

The approach

The BFF's primary objective is clear separation of concerns. The design approach entails the following key considerations:

- **Defining the frontend needs:** Identifying the specific data and functionalities each frontend application requires from the backend is the first step. This analysis helps designing the APIs tailored for each frontend's consumption patterns.
- **Designing the APIs:** It includes planning the BFF APIs - defining endpoints, data formats, authentication mechanisms, and error handling specific to each frontend's needs.
- **Backend integrations:** Mainly focuses on developing the BFF service logic to interact with various backend microservices. The BFF acts as an orchestrator, fetching data from relevant services and potentially aggregating or transforming it for the frontend's consumption.
- **Frontend integrations:** This is about integrating the BFF APIs into frontend applications. This involves making API calls from the frontend code to fetch and utilize the processed data provided by the BFF.

The benefits proposition

The BFF in a microservices architecture offers the following key benefits:

- **Decoupling frontend and backend:** The frontend and backend can evolve independently, promoting flexibility and maintainability.
- **Specialized APIs:** This allows the frontend to request only the data and functionality it requires, reducing over-fetching and improving performance.
- **User experience optimization:** BFFs are designed to optimize the user experience by tailoring responses to the frontend's specific requirements.
- **Performance and responsiveness:** With a backend for a frontend design, the communication between them can be optimized for performance, reducing unnecessary data transfer and processing.
- **Cross-cutting concerns:** BFFs may handle cross-cutting concerns such as authentication, authorization, logging, and caching, which are specific to the needs of the associated frontend.
- **Development autonomy:** Different teams can work on frontend and backend components independently, leading to faster development cycles and easier maintenance.



The caution points

While BFF offers advantages for microservices architecture, there are some potential concerns to be wary about:

- **Avoid overly complex BFFs:** While it's important to tailor the BFF pattern adoption to the needs of the frontend, avoid making it overly complex. Strive for simplicity and clarity in BFF service design to ease maintenance and troubleshooting.
- **Versioning and compatibility concerns:** Changes to the BFF should not break existing frontend implementations! Implement proper versioning mechanisms for BFF APIs to ensure backward compatibility.
- **Granularity of APIs:** APIs that are too fine-grained might lead to over-fetching, while overly coarse-grained APIs could result in under-fetching and reduced efficiency. Consider the granularity of APIs provided by the BFF. Striking the right balance is crucial.
- **Caching strategies:** While caching can enhance performance, be cautious about the data being cached. Ensure that the cache is appropriately invalidated or refreshed to reflect changes in the underlying microservices.
- **Failure handling:** Implement robust error-handling mechanisms in the BFF to gracefully handle failures. Provide meaningful error messages and consider fallback strategies to maintain a good user experience even in the face of service failures.
- **Scalability:** Plan for the scalability of the BFF design. Ensure that it can handle increased loads from the frontend without compromising performance. Implement strategies for horizontal scaling if necessary.

When is this the right option?

Adopting the Backend for Frontend (BFF) microservices pattern proves to be a strategic decision when specific priorities shape the application design. This choice is particularly beneficial in scenarios where multiple frontends are integral to the application, and the desire is to craft specialized backends meticulously tailored to meet the unique requirements of each frontend. It becomes a compelling option when separate development teams govern various frontends, seeking autonomy in the design and evolution of their respective applications. Furthermore, BFF shines when the frontend necessitates precise data and functionality spanning multiple microservices, preventing issues associated with over-fetching or under-fetching. Also, in applications with performance-critical demands, where optimizing communication between the frontend and backend is paramount, the implementation of BFF emerges as a crucial strategy to ensure efficiency and responsiveness. If one or more of these considerations are key priorities for the application, then adopting the BFF pattern and related design approach becomes imperative.



04

Architecting microservices with domain driven design (DDD)

DDD is an approach to software development that emphasizes understanding the domain (the problem space) and using that understanding to inform the design and implementation of software. When applied to microservices architecture, DDD helps in creating cohesive, well-organized microservices that closely align with the business domain they serve. Albeit considered as a design pattern, it is more accurately described as a set of practices and approach rather than a specific implementation pattern. DDD guides developers in structuring software systems in a way that reflects the domain they're working with. While DDD is more abstract than traditional design patterns, it still provides a structured approach to solving problems related to software design and architecture, especially in domains with complex business rules and interactions.



Essential steps and strategies for adopting domain driven design

When implementing Domain-Driven Design (DDD) in a microservices architecture, the focus is on creating a system of loosely coupled microservices that reflect the bounded contexts, entities, and business processes of the domain. The approach entails:

- **Identifying bounded contexts:** A bounded context represents a specific area of the business domain with its own language, rules, and models. Each bounded context will likely correspond to a separate microservice in the architecture.
- **Defining ubiquitous language:** Establishing a common language that is shared between the stakeholders. This language should accurately represent the concepts and terms within each bounded context.
- **Decomposing the monolith:** If migrating from a monolithic architecture, decompose the monolith into smaller, more manageable microservices, based on bounded contexts.
- **Designing aggregates:** Within each microservice, identify aggregates, which are clusters of domain objects (entities) that are treated as a single unit for data changes. Design aggregates to enforce consistency boundaries and encapsulate business rules within the microservice.
- **Defining service interfaces:** Determine how microservices will communicate with each other. Define clear service interfaces (API contracts) for each microservice, specifying the data formats and protocols used for communication.
- **Establishing context mapping:** Define the relationships and interactions between microservices. Use context mapping techniques to handle communication between bounded contexts, such as shared kernel, customer-supplier, or anti-corruption layer patterns.
- **Implementing business logic:** Encapsulate domain-specific behavior within the microservice, using domain events, aggregates, and domain services.
- **Ensuring data consistency:** Using patterns such as eventual consistency, distributed transactions, or saga patterns to manage data updates and ensure data integrity across microservices.
- **Deploying and scaling microservices:** Entails using tools and enablers for containerization (e.g., Docker) and orchestration (e.g., Kubernetes).



Domain driven design offers several benefits propositions

Applying DDD in a microservices architecture can yield several key benefits:

- **Alignment with business goals:** By aligning microservices with bounded contexts and ubiquitous language, the architecture closely reflects the business domain leading to software that better meets business requirements.
- **Modularity and scalability:** DDD encourages breaking down complex systems into cohesive components (microservices) based on bounded contexts.
- **Flexibility and agility:** DDD based microservices architecture promotes flexibility and agility in software development. Each microservice can be developed, deployed, and scaled independently, enabling faster iterations.
- **Improved team collaboration:** By using a common ubiquitous language and modeling the domain explicitly, teams can communicate effectively and ensure a shared understanding of the system's behavior and requirements.
- **Domain-driven modeling:** DDD provides patterns and techniques for modeling complex domain concepts, such as aggregates, entities, value objects, and domain events. This improves the maintainability and extensibility of the system over time.
- **Reduced complexity and coupling:** By defining clear boundaries between bounded contexts and microservices, teams can minimize dependencies and coupling, making it easier to understand and modify individual parts of the system without impacting others.
- **Scalability and resilience:** Encourages the use of patterns like event sourcing and eventual consistency, which can improve fault tolerance and resilience.
- **Technology flexibility:** Microservices architecture with DDD enables technology heterogeneity within the system. Different microservices can be implemented using different technologies and programming languages, chosen based on the specific requirements of each bounded context.

Navigating domain driven design requires careful deliberations

Here are some cautionary points to consider when adopting DDD patterns in a microservices architecture:

- **Increased complexity:** For large or evolving domains, the process of defining bounded contexts, identifying entities and aggregates, and establishing a ubiquitous language can become complex. It requires careful planning and collaboration.
- **Overengineering:** While DDD provides valuable tools, it's easy to get caught up in the details of domain modeling. Models should be clear and maintainable.
- **Distributed transactions:** DDD doesn't inherently solve challenges with distributed transactions across microservices. It will require additional mechanisms like saga patterns to meet data consistency requirements.

- **Bounded context drifts:** As the system evolves, bounded contexts can drift over time. The key is to revisit and refine bounded contexts often.
- **Communication overhead:** While DDD promotes a ubiquitous language, communication overhead can still arise between microservice teams, especially during initial development stages. Invest in clear documentation and communication channels to foster collaboration.

Synergizing domain driven design and microservices

When dealing with intricate domain logic, large applications, or evolving requirements where data consistency is crucial, DDD can be a powerful tool for building robust and scalable microservices. DDD offers a structured approach to decompose domain into manageable chunks, aligning perfectly with the microservices philosophy. For maintainable, and scalable microservice architectures, DDD offers a powerful approach.



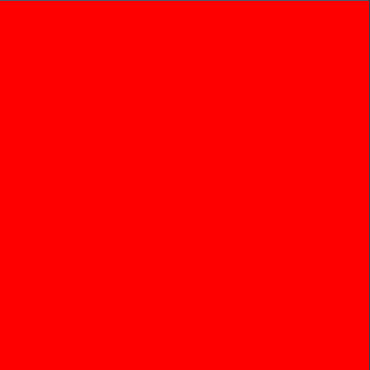


05

Unlock with Finacle: True microservices and cloud native powered digital banking

Finacle is an industry leader in digital banking solutions. We partner with emerging and established financial institutions to inspire better banking. Our cloud-native solution suite and SaaS services help banks engage, innovate, operate, and transform better to scale digital transformation with confidence.

Finacle solutions address the core banking, lending, digital engagement, payments, cash management, wealth management, treasury, analytics, AI, and blockchain requirements of financial institutions globally. Finacle's componentized structure allows banks to deploy and upgrade solutions flexibly as per their business priorities. Our solutions run in a containerized environment orchestrated by Kubernetes and can be deployed on a private, public, or hybrid cloud.



Finacle's composable banking platform is built on the foundations of a 100% open architecture, embracing true microservices architectural thinking. Fueled by domain driven design constructs, the platform offers right grained microservices tailored to the business domains they support.

Rooted in pattern language, the platform's microservices design ensures the delivery of efficient, precisely-tailored components perfectly suited to the unique requirements of each business domain.

- A host of data strategy patterns ensure consistency and seamless querying across services, pivotal for system integrity, scalability and performance.
- A variety of integration, messaging and communication patterns seamlessly connect services, manage communications, and drive integrations to leverage the full potential of microservices.
- The deployment automation and scalability patterns provide optimal microservices deployment, addressing resource utilization, isolation, cross-cutting concerns, population-scale performance, and operational complexities.
- The observability and performance management patterns deliver on optimal behavior, performance and resilience of the microservices.

Through persistent R&D investments, adoption of modern technology components, and continuous innovation, Finacle's composable architecture has stood the test of time throughout our existence. Consequently, Finacle has consistently earned recognition as the most advanced cloud-native banking platform by multiple analyst firms. Finacle's cloud-native, microservices based architecture empowers banks and financial institutions worldwide to future-proof their technology investments and deliver next-gen banking services to their customers.

Authors



Sudhindra Murthy
Product Marketing Lead,
Strategic Initiatives, Infosys Finacle



Diwakar Mandal
Product Marketing Manager,
Infosys Finacle



finacle@edgeverve.com



www.finacle.com



www.linkedin.com/company/finacle



twitter.com/finacle



For more information, contact finacle@edgeverve.com

www.finacle.com

© 2024 EdgeVerve Systems Limited, a wholly owned subsidiary of Infosys, Bangalore, India. All Rights Reserved. This documentation is the sole property of EdgeVerve Systems Limited ("EdgeVerve"). EdgeVerve believes the information in this document or page is accurate as of its publication date; such information is subject to change without notice. EdgeVerve acknowledges the proprietary rights of other companies to the trademarks, product names and such other intellectual property rights mentioned in this document. This document is not for general distribution and is meant for use solely by the **person or entity that it has been specifically issued to and can be used for the sole purpose it is intended to be used for as communicated by EdgeVerve** in writing. Except as expressly permitted by EdgeVerve in writing, neither this documentation nor any part of it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, printing, photocopying, recording or otherwise, without the prior written permission of EdgeVerve and/ or any named intellectual property rights holders under this document.