Pragmatic Approach to Refining Microservices Architectures

Engineering

www.99x.io

Pragmatic Approach to Refining Microservices Architectures

Suboptimal microservices architectures can result in performance bottlenecks, systems that struggle to scale, and increased maintenance costs.

Designing a microservices architecture that works flawlessly from the beginning is challenging. With many architectural patterns to choose from, the right balance between modularity and granularity often only becomes clear as our understanding of the product evolves. In our experience, we've seen many product companies face the need to refactor and fine-tune their microservices architecture at various stages of product development.

This paper presents a practical and structured approach to refactoring microservices for enhanced performance and scalability. The strategy is built on a five-step methodology:

1. Identify Bounded Contexts
2. Perform Granularity Analysis
3. Map Service Interactions
4. Identify Common Libraries
5. Refine through Isomorphic Analysis

In addition, the paper offers guidance on implementing the refined architecture using industry best practices to ensure it aligns with business needs while maintaining efficiency and scalability.

Step1: Identify Bounded Contexts

Identifying domain boundaries and critical paths ensures that services stay focused on core business functionality. By defining bounded contexts, unnecessary dependencies are minimized, and scalability can be improved.

How to:

1. **Perform Domain Analysis:** Identify domain boundaries, dependencies, and critical paths within the system.
2. Define Bounded Contexts: Use Domain-Driven Design (DDD) principles to define bounded contexts.
3. **Categorize Microservices:** Group existing microservices into identified bounded contexts to validate the alignment. Update Bounded Contexts if required based on microservices categorization
4. **Create Data Models:** Develop a data model (database schema) for each bounded context based on categorized microservices.

By defining bounded contexts, we can minimize coupling and promote modularity, which is important for long-term scalability.

Step 2: Perform Granularity Analysis

Granularity analysis helps strike a balance between consolidation and separation of services. Over-granular services (Grains of sand anti-pattern) lead to excessive inter-service communication, while under-granular services can hinder maintainability and scalability. Granularity analysis is performed by evaluating the identified microservices against granularity integrators and disintegrators.

Granularity Integrators

Granularity integrators identify instances when services should be merged due to strong dependencies. E.g.,

- **Shared Data Domains:** Services with overlapping data should be consolidated.
- **ACID Transactions:** Transactions should be contained within a single service to avoid distributed transaction complexity.
- **Workflow Choreography:** Workflows that span multiple services often introduce high latency and unreliable execution.

Granularity Disintegrators

Disintegrators indicate when a service should remain independent. E.g.,

- **Single Responsibility:** Services with distinct responsibilities should remain separate.
- **Code Volatility:** Services that experience frequent code changes may require separation to avoid impacting other services.
- **Scalability:** Services with different scaling requirements should remain decoupled.
- **Security and Privacy:** Services with unique security or privacy concerns should remain isolated.

How to:

1. **Group Services by Integrators:** Identify shared data domains and consolidate services accordingly. Mark services that can be merged based on granularity integrator analysis
2. **Apply Disintegrator Criteria:** Apply granularity disintegrator criteria to grouped microservices and recognize microservices that need to be separated. Ensure services remain decoupled where necessary, based on function, volatility, scalability, and security requirements.

This analysis helps to determine the right number of microservices under each bounded context.

Step 3: Map Service Interactions

Defining communication patterns between microservices is critical for reducing latency and ensuring reliable workflows. By selecting the right communication protocols, you can optimize inter-service interactions.

How to:

1. **Map Communication Protocols:** Determine whether services should communicate via REST, gRPC, or GraphQL, depending on their data and performance requirements.
2. **Async Communication:** Use message queues (e.g., AWS SQS) for asynchronous communication between services where appropriate.

Optimizing communication patterns can significantly reduce service-to-service overhead and improve the overall performance of your microservices architecture.

Step 4: Identify Common Libraries

Cross-cutting concerns, such as authentication, logging, and database interactions, should be standardized across microservices to ensure consistency, reduce redundancy, and simplify long-term maintenance.

How to:

1. **Identify Cross-cutting Concerns:** Identify repetitive third-party service calls, DB interactions, logging, telemetry, and authentication services into shared libraries.
2. **Publish Shared Libraries:** Extract the functionality into a shared library.

Having standardized libraries simplifies the development process and ensures that all services adhere to the same security, logging, and monitoring standards.

Step 5: Refine through Isomorphic Analysis

Isomorphic analysis ensures that the refactored architecture satisfies business needs, its key quality attributes and the development team's structure. For example, Conway's Law states that system design mirrors the communication structures of the organization.

One way to mitigate some of the issues found during the analysis is to utilize well known microservices design patterns to enhance the architecture.

Microservices Design Patterns

While there are many established microservices design patterns, below are several that are particularly useful to consider depending on the prioritized quality attributes.

1. **Backends for Frontends/API Composition/Gateway**
   These patterns aggregate data from multiple services. Rather than each service querying separate databases or endpoints, an aggregator is introduced. This component collects and merges responses from multiple microservices and presents them as a unified API response.
2. **Saga pattern**
   Unlike the traditional two-phase commit used in monolithic systems, which isn't straight-forward for distributed microservices, the saga pattern breaks transactions into smaller, custom managed local operations. Each service updates its own data and emits events that trigger corresponding actions in other services. If a failure occurs, compensating transactions are triggered to roll back changes, maintaining overall consistency across services.
3. **Shared Database vs Database per Service**
   Contradictory to 'database per service', shared database pattern allows multiple services to directly access and interact with the same database. This approach enables services to share data easily but can lead to tight coupling between services, making it harder to scale or maintain independently.
4. **Command Query Responsibility Segregation (CQRS)**
   For services struggling with high-performance demands or inconsistencies in read and write operations, CQRS separates the logic for commands (writes) and queries (reads). This optimizes the architecture by ensuring that reads and writes are handled by different models, improving scalability and performance, particularly in high-throughput services.
5. **Circuit Breaker/Bulkhead**
   These patterns improve system resilience, they solve different problems. Circuit breaker focuses on preventing excessive failure requests, while bulkhead focuses on isolating and containing failures.
6. **Event Sourcing/Log Aggregation/Distributed Tracing**
   These patterns help managing data consistency, monitoring, and observability in microservices architectures

In a complex system, these patterns can coexist across the system (ex: Log Aggregation and Distributed Tracing), to one or more selected bounded contexts (ex: Even Sourcing, BFF) or to specific services (ex: Bulkhead, CQRS) as required.

How to:

1. **Align with Quality Attributes:** Ensure the architecture pattern meets business objectives and non-functional attributes.
2. **Align with team's structure:** Ensuring smooth operation and maintenance.
3. **Apply Patterns:** Utilize well known microservices design patterns to mitigate the identified issues and refine the architecture.

By ensuring the architecture reflects both business and organizational needs, teams can achieve higher efficiency and long-term system stability.

Document the Refactored Architecture

After performing the domain analysis, granularity evaluation, and communication mapping, consolidating the refactored architecture provides a comprehensive view of the service interactions, data models, and common libraries.

1. **Visualize Bounded Contexts:** Map microservices to their respective bounded contexts.
2. **Visualize Data Models:** Illustrate the data model associated with each bounded context and its services.
3. **Map Communication Protocols:** Clearly define communication protocols, API gateways, and queues between microservices.
4. **Show Shared Libraries:** Highlight common libraries managing cross-cutting concerns.
5. **Patterns applied:** Document which design pattern is applicable for which bounded context and the microservice.

Consolidating the architecture allows businesses to review the system holistically, ensuring that all elements align for optimal performance.

Architecture Change Management

Implementing the newly refactored microservice architecture must be approached in a phased and pragmatic manner to minimize disruption and risk. The following strategies are recommended for a smooth implementation:

1. **Prioritizing Least Impact Microservices**
   To avoid system-wide disruptions, start by refactoring microservices with the least impact on the core business processes. This allows the team to validate the new architecture on a smaller scale, identify any issues early, and build momentum.

2. **Incremental microservice migration.**
   During the transition, it is essential to maintain both the old and new microservices in parallel and migrate to the newly created microservices gradually to ensure smooth transition. This might result in temporary duplication of code, but it ensures continuity in operations and allows for incremental migration.

3. **Use AI Tools to Generate Test Cases**
   Leverage AI-driven tools to automatically generate test cases for the newly implemented microservices. This ensures comprehensive testing coverage while speeding up the QA process.

4. **Incremental Data Migration**
   For microservices handling persistent data, plan for incremental data migration. We must employ a hybrid approach where the old database and new bounded context-based databases operate concurrently, with the data gradually synchronized until the migration is complete. This approach helps to avoid downtime and potential data integrity issues.

5. **Automate Infrastructure Changes**
   As architecture changes, so will infrastructure needs. Use Infrastructure as Code (IaC) tools to automate the setup of novel resources and services, ensuring consistency and reducing manual effort.

6. **Continuous Monitoring and Feedback Loops**
   Set up monitoring tools that continuously observe the performance and stability of both the newly implemented microservices. Gathering real-time metrics and feedback allows you to identify potential bottlenecks early and adjust, as necessary. Use tools like Prometheus and Grafana to monitor the health of microservices and set up alerts for any performance anomalies.

By using this phased, pragmatic approach, businesses can refactor their microservices architecture while minimizing risks, ensuring smooth transitions, and achieving long-term performance improvements.

Conclusion

Refactoring microservices architecture to maximize performance is important in addressing scalability and maintenance issues in businesses.

This paper discusses a pragmatic five-step approach that covers domain analysis, granularity evaluation, communication mapping, identifying common libraries, and isomorphic analysis as a viable approach to successfully enhance any given microservice architecture. By following these guidelines, along with the implementation strategies, businesses can enhance system performance, reduce complexity, and achieve sustainable scalability.