Here’s a well-structured problem statement and justification for your presentation, highlighting the need for distributed caching when transitioning from Kony to a microservices architecture.

Slide 1: Problem Statement

Background

• Current application is built on Kony, leveraging session caching for storing customer and account ownership details.

• Session caching ensures ownership validation is performed locally, reducing repeated calls to the core system.

• Moving to a microservices architecture eliminates session-based caching since stateless services are favored.

Challenges with Microservices (Without Distributed Caching)

1. Increased Core System Dependency:

• Each microservice directly queries the core system for customer and account validation, resulting in:

• Redundant API calls for frequently accessed data.

• Increased load on the core system.

2. Performance Bottlenecks:

• Higher latency due to repeated round trips between microservices and the core system.

• Slower response times for user actions like accessing account details or performing transactions.

3. Scalability Issues:

• As the user base grows, the frequency of API calls to the core system increases, potentially leading to system downtime or degraded performance.

4. Inefficiency Across Microservices:

• Each microservice repeatedly retrieves the same customer and account ownership data independently, leading to duplicated effort and wasted resources.

Slide 2: Impact of No Distributed Caching

Area Without Distributed Caching

Core System Load Increased, as every microservice queries the core system repeatedly.

Latency Higher latency due to frequent API calls to the core system.

Scalability Limited, as core system struggles to handle exponential requests.

Redundancy Duplicate requests for identical data across multiple services.

Cost Higher infrastructure and network costs due to excessive API calls.

Slide 3: Distributed Caching as a Solution

What is Distributed Caching?

• A shared, centralized caching layer (e.g., Redis, Memcached) accessible by all microservices.

• Allows frequently used data (e.g., customer and account details) to be cached close to the application layer.

Slide 4: How Distributed Caching Helps

1. Reduces Core System Dependency

• Customer and account ownership data is cached once and reused across microservices.

• Minimizes redundant API calls to the core system.

2. Improves Performance

• Ownership validation is performed directly using cached data, reducing round-trip latency.

• Faster response times for all customer operations.

3. Enhances Scalability

• Distributed caching can handle high concurrency and traffic spikes without affecting the core system.

4. Centralized Data Access

• Cache acts as a single source of truth for shared data across all microservices.

5. Cost Efficiency

• Decreased infrastructure and network costs due to fewer calls to the core system.

Slide 5: Comparison

Feature Without Distributed Caching With Distributed Caching

Core System Calls High Low

Latency High Low

Scalability Limited High

Data Consistency Managed separately per service Centralized consistency across services

Cost High Lower due to reduced load on core systems

Slide 6: Implementation Approach

1. Step 1: Choose a Distributed Cache:

• Examples: Redis, Memcached, Hazelcast.

2. Step 2: Cache Key Design:

• Use unique identifiers like customerId and accountId as cache keys.

3. Step 3: Data Expiry and Consistency:

• Set appropriate TTL (time-to-live) values to balance freshness and performance.

• Use cache invalidation strategies (e.g., on account updates).

4. Step 4: Integration with Microservices:

• Use caching libraries (e.g., Spring Cache, Redis Client) in microservices to access the distributed cache.

5. Step 5: Monitor and Scale:

• Use monitoring tools (e.g., Prometheus, Grafana) to track cache performance and adjust as needed.

Slide 7: Expected Benefits

1. 95% Reduction in Core System Calls:

• Majority of validation checks served from the cache.

2. 50% Improvement in Latency:

• Faster API responses due to localized data access.

3. Increased Scalability:

• Core system is freed up to handle essential operations.

4. Improved User Experience:

• Seamless navigation across screens with faster ownership validations.

5. Lower Infrastructure Costs:

• Optimized resource utilization through caching.

Slide 8: Risks and Mitigation

Risk Mitigation

Stale Data in Cache Use short TTL and event-driven cache updates.

Cache Downtime Implement fallback to the core system.

High Memory Usage Optimize cache size and eviction policies.

Slide 9: Conclusion

1. Transitioning to microservices provides flexibility but increases core system dependency.

2. Distributed caching is essential to ensure:

• High performance

• Scalability

• Cost efficiency

3. A well-designed caching strategy reduces operational overhead and improves user experience while maintaining system reliability.

Presentation Tips:

• Use diagrams to show:

• Core system interactions without caching vs. with distributed caching.

• Cache as a shared layer across microservices.

• Include performance metrics (e.g., latency reductions, call reduction percentages) to strengthen your case.